

Библиотека Keras – инструмент глубокого обучения

Реализация нейронных сетей с помощью
библиотек Theano и TensorFlow

Deep Learning with Keras

Implement neural networks with Keras on Theano and TensorFlow

Antonio Gulli

Sujit Pal

Packt>

BIRMINGHAM – MUMBAI

Библиотека Keras – инструмент глубокого обучения

Реализация нейронных сетей с помощью
библиотек Theano и TensorFlow

Антонио Джулли

Суджит Пал



Москва, 2018

УДК 004.85Keras
ББК 32.971.3
P51

P51 Антонио Джулли, Суджит Пал

Библиотека Keras – инструмент глубокого обучения. Реализация нейронных сетей с помощью библиотек Theano и TensorFlow / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2018. – 294 с.: ил.

ISBN 978-5-97060-573-8

Книга представляет собой краткое, но обстоятельное введение в современные нейронные сети, искусственный интеллект и технологии глубокого обучения. В ней представлено более 20 работоспособных нейронных сетей, написанных на языке Python с использованием модульной библиотеки Keras, работающей поверх библиотек TensorFlow от Google или Theano от компании Lisa Lab. Описан функциональный API библиотеки Keras и возможности его расширения. Рассмотрены алгоритмы обучения с учителем (простая линейная регрессия, классический многослойный перцептрон, глубокие сверточные сети), а также алгоритмы обучения без учителя – автокодировщики и порождающие сети. Дано введение в технологию глубокого обучения с подкреплением и ее применение к построению игр со встроенным искусственным интеллектом.

Издание предназначено для программистов и специалистов по анализу и обработке данных.

УДК 004.85Keras
ББК 32.971.3

Authorized Russian translation of the English edition of Deep Learning with Keras, ISBN 978-1-78712-842-2. Copyright ©Packt Publishing 2017. First published in the English language under the title 'Deep Learning with Keras – (9781787128422)'. This translation is published and sold by permission of Published by Packt Publishing Ltd., which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78712-842-2 (англ.)
ISBN 978-5-97060-573-8 (рус.)

© Packt Publishing, 2017.
© Оформление, перевод на русский язык,
издание, ДМК Пресс, 2018

Оглавление

Об авторах	9
О рецензенте	12
Предисловие.....	13
Назначение	13
Чем глубокое обучение отличается от машинного обучения и искусственного интеллекта	14
Краткое содержание книги	16
Что необходимо для чтения книги	17
На кого рассчитана эта книга	17
Графические выделения.....	17
Отзывы.....	18
Поддержка клиентов	19
Загрузка кода примеров.....	19
Загрузка цветных иллюстраций.....	20
Опечатки.....	20
Нарушение авторских прав.....	20
Вопросы.....	21
Глава 1. Основы нейронных сетей	22
Перцептрон.....	24
Первый пример кода с использованием Keras.....	24
Многослойный перцептрон – первый пример нейросети	25
Проблемы обучения перцептрона и их решение	26
Сигмоида	27
Блок линейной ректификации	28
Функции активации	28
Реальный пример – распознавание рукописных цифр	29
Унитарное кодирование	30
Определение простой нейронной сети в Keras	30
Прогон простой сети Keras и создание эталона для сравнения	34
Улучшение простой сети в Keras посредством добавления скрытых слоев	35
Дальнейшее улучшение простой сети Keras с помощью прореживания.....	38
Тестирование различных оптимизаторов в Keras	41
Увеличение числа периодов	46
Управление скоростью обучения оптимизатора	46
Увеличение числа нейронов в скрытых слоях	47
Увеличение размера пакета.....	48
Подведение итогов экспериментов по распознаванию рукописных цифр.....	49

Применение регуляризации для предотвращения переобучения	50
Настройка гиперпараметров	52
Предсказание выхода	52
Практическое изложение алгоритма обратного распространения ...	52
В направлении глубокого обучения	54
Резюме	55
Глава 2. Установка Keras и описание API	56
Установка Keras	56
Шаг 1 – установка зависимостей	56
Шаг 2 – установка Theano	57
Шаг 3 – установка TensorFlow	57
Шаг 4 – установка Keras	58
Шаг 5 – проверка работоспособности Theano, TensorFlow и Keras	58
Настройка Keras	59
Установка Keras в контейнер Docker	60
Установка Keras в Google Cloud ML	62
Установка Keras в Amazon AWS	64
Установка Keras в Microsoft Azure	65
Keras API	67
Введение в архитектуру Keras	68
Обзор готовых слоев нейронных сетей	69
Обзор готовых функций активации	72
Обзор функций потерь	72
Обзор показателей качества	73
Обзор оптимизаторов	73
Некоторые полезные операции	73
Резюме	77
Глава 3. Глубокое обучение с применением сверточных сетей	79
Глубокая сверточная нейронная сеть	80
Локальные рецептивные поля	80
Разделяемые веса и смещения	81
Пулинговые слои	82
Промежуточные итоги	83
Пример ГСНС – LeNet	83
Код LeNet в Keras	83
О силе глубокого обучения	89
Распознавание изображений из набора CIFAR-10 с помощью глубокого обучения	90
Повышение качества распознавания набора CIFAR-10 путем углубления сети	95
Повышение качества распознавания набора CIFAR-10 путем пополнения данных	97
Предсказание на основе результатов обучения на наборе CIFAR-10	100

Очень глубокие сверточные сети для распознавания больших изображений.....	101
Распознавание кошек с помощью сети VGG-16.....	102
Использование встроенного в Keras модуля VGG-16.....	103
Использование готовых моделей глубокого обучения для выделения признаков.....	104
Очень глубокая сеть inception-v3, применяемая для переноса обучения.....	105
Резюме.....	108
Глава 4. Порождающие состязательные сети и WaveNet.....	109
Что такое ПСС?.....	109
Некоторые приложения ПСС.....	111
Глубокие сверточные порождающие состязательные сети.....	114
Применение Keras adversarial для создания ПСС, подделывающей MNIST.....	118
Применение Keras adversarial для создания ПСС, подделывающей CIFAR.....	124
WaveNet – порождающая модель для обучения генерации звука ...	132
Резюме.....	141
Глава 5. Погружения слов.....	143
Распределенные представления.....	144
word2vec.....	145
Модель skip-грамм.....	146
Модель CBOW.....	150
Извлечение погружений word2vec из модели.....	151
Сторонние реализации word2vec.....	154
Введение в GloVe.....	158
Использование предобученных погружений.....	159
Обучение погружений с нуля.....	161
Настройка погружений на основе предобученной модели word2vec.....	165
Настройка погружений на основе предобученной модели GloVe.....	169
Поиск погружений.....	170
Резюме.....	174
Глава 6. Рекуррентная нейронная сеть – РНС.....	176
Простые ячейки РНС.....	177
Простая РНС с применением Keras – порождение текста.....	179
Топологии РНС.....	184
Проблема исчезающего и взрывного градиента.....	186
Долгая краткосрочная память – LSTM.....	188
Пример LSTM – анализ эмоциональной окраски.....	191
Вентильный рекуррентный блок – GRU.....	197
Пример GRU – частеречная разметка.....	198
Двунаправленные РНС.....	205

РНС с запоминанием состояния	206
Пример LSTM с запоминанием состояния – предсказание потребления электричества	206
Другие варианты РНС	212
Резюме	213
Глава 7. Дополнительные модели машинного обучения	214
Функциональный API Keras	215
Регрессионные сети	218
Пример регрессии – предсказание содержания бензола в воздухе	218
Обучение без учителя – автокодировщики	223
Пример автокодировщика – векторы предложений	225
Композиция глубоких сетей	234
Пример – сеть с памятью для ответов на вопросы	235
Расширение Keras	242
Пример – использование слоя lambda	242
Пример – построение пользовательского слоя нормировки	243
Порождающие модели	247
Пример – глубокие сновидения	248
Пример – перенос стиля	255
Резюме	260
Глава 8. Искусственный интеллект играет в игры	262
Обучение с подкреплением	263
Максимизация будущих вознаграждений	264
Q-обучение	265
Глубокая Q-сеть как Q-функция	267
Баланс между исследованием и использованием	268
Воспроизведение опыта	269
Пример – глубокая Q-сеть для поимки мяча	269
Что дальше?	282
Резюме	283
Заключение	285
Keras 2.0 – что нового	286
Установка Keras 2.0	287
Изменения API	287

Об авторах

Антонио Джулли – директор по программному обеспечению и предприниматель с тягой к созданию и управлению глобальными инновационными технологическими компаниями. Специализируется в области поисковых систем, онлайн-сервисов, машинного обучения, информационного поиска, аналитики и облачных вычислений. Профессиональный опыт приобретал в шести странах Европы и Америки. Антонио работал в должности исполнительного директора, генерального директора, технического директора, вице-президента и руководителя группы в различных отраслях: от издательского бизнеса (Elsevier) до интернет-технологий для конечного пользователя (Ask.com и Tiscali) и НИОКР в сфере высоких технологий (Microsoft и Google).

Выражаю благодарность своему талантливому соавтору, Суджиту Палу, за неизменное стремление помочь, не требуя ничего взамен. Я очень ценю его преданность командной работе, благодаря чему эта книга и смогла стать чем-то стоящим.

Благодарю также Франсуа Шолле и многих людей, внесших вклад в Keras, за то, что они тратили свое время и силы на создание впечатляющего инструментария для глубокого обучения, который прост в использовании и не требует сверхъестественных усилий.

Спасибо также нашим редакторам из издательства Packt, Дивия Пуджари, Черил Дса и Динешу Павару, и рецензентам из Packt и Google за поддержку и ценные предложения. Без вас эта книга не состоялась бы.

Я также благодарен своему начальнику, Брэду, и коллегам Майку и Коррадо из Google, которые подвигли меня написать эту книгу, читали ее черновые варианты и высказывали свое мнение.

Еще я признателен кофейне Same Fusu в Варшаве, где у меня впервые появилась мысль написать эту книгу, когда я наслаждался чашечкой чая, выбранной из весьма обширного меню.

Это место обладает особой магией, и я горячо рекомендую его всем, ищущим, где бы подстегнуть свое воображение (<http://www.samefussy.pl/>).

Далее я хочу поблагодарить отдел кадров в Google, пошедший навстречу моему пожеланию отдать все отчисления от продажи этой книги на стипендии представителям этнических меньшинств.

Спасибо моим друзьям, Эрику, Лауре, Франческо, Этторе и Антонелле, которые поддерживали меня, когда я в том нуждался. Дружба – большая ценность, и вы – мои настоящие друзья.

Спасибо моему сыну Лоренцо, который побудил меня устроиться в Google, моему сыну Леонардо за постоянное стремление открывать что-то новое и моей дочери Авроре, благодаря которой я встречаю каждый день с улыбкой. И наконец, спасибо моему отцу Элио и матери Марии за их любовь.

Суджит Пал – руководитель отдела технологических исследований в Elsevier Labs, работает над созданием интеллектуальных систем поиска по содержимому и метаданным. В область его интересов входят информационный поиск, онтологии, обработка естественных языков, машинное обучение и распределенная обработка. В настоящее время занимается классификацией и установлением сходства изображений с применением моделей глубокого обучения. До этого работал в промышленности безрецептурных медицинских препаратов, где участвовал в построении онтологической системы семантического поиска, организации контекстной рекламы и платформ обработки данных. Ведет посвященный технологиям блог *Salmon Run*.

Выражаю благодарность своему соавтору, Антонио Джулли, пригласившему меня принять участие в написании книги. Это редкая возможность, благодаря которой я многому научился. К тому же, если бы не он, меня бы здесь в буквальном смысле не было.

Хочу поблагодарить Рона Дэниэла, директора Elsevier Labs, и Брэдли П. Аллена, главного архитектора в Elsevier, которые познакомили меня с глубоким обучением и заставили поверить в возможности этой технологии.

Благодарю также Франсуа Шолле и многих людей, внесших вклад в Keras, за то, что они тратили свое время и силы на создание впечатляющего инструментария для глубокого обучения, который прост в использовании и не требует сверхъестественных усилий.

Спасибо также нашим редакторам из издательства Packt, Дивия Пуджари, Черил Дса и Динешу Павару, и рецензентам из Packt и Google за поддержку и ценные предложения. Без вас эта книга не состоялась бы.

Я также признателен коллегам и начальникам, с которыми работал на протяжении своей жизни, а особенно тем, кто верил в меня и помогал мне строить свою извилистую профессиональную карьеру.

Наконец, я благодарен своей семье, которая на протяжении нескольких месяцев мирилась с тем, как я разрывался между работой, этой книгой и семьей – именно в таком порядке. Надеюсь, вы согласитесь, что дело того стоило.

О рецензенте

Ник Макклюр в настоящее время работает старшим специалистом по анализу данных в компании PayScale Inc., Сиэтл, штат Вашингтон, США. До этого работал в компании Zillow and Caesars Entertainment. Защитил диссертации по прикладной математике в Университете штата Монтана, колледже Святого Бенедикта и Университете Святого Иоанна. Ник – автор книги «TensorFlow Machine Learning Cookbook», вышедшей в издательстве Packt Publishing.

Его страсть – изучать и делиться знаниями об аналитике, машинном обучении и искусственном интеллекте. Плоды своих размышлений Ник публикует в блоге на сайте fromdata.org и в своем аккаунте в Твиттере по адресу [@nfmccclure](https://twitter.com/nfmccclure).

Предисловие

Книга, которую вы держите в руках, – краткое, но обстоятельное введение в современные нейронные сети, искусственный интеллект и технологии глубокого обучения. Она написана специально для программистов и специалистов по анализу и обработке данных.

Назначение

В книге представлено более 20 работоспособных нейронных сетей, написанных на языке Python с использованием модульной библиотеки Keras, работающей поверх библиотек TensorFlow от Google или Theano от компании Lisa Lab.

Читатель шаг за шагом познакомится с алгоритмами обучения с учителем, начиная с простой линейной регрессии и классического многослойного перцептрона и кончая более сложными глубокими сверточными сетями и порождающими состязательными сетями. В книге также рассматриваются алгоритмы обучения без учителя: автокодировщики и порождающие сети. Подробно объясняется, что такое рекуррентные сети и сети с долгой краткосрочной памятью (long short-term memory, LSTM). Описывается функциональный API библиотеки Keras и обсуждается, как расширить Keras, если встретится задача, для которой в ней нет готового решения. Также рассматриваются более крупные и сложные системы, состоящие из описанных ранее структурных блоков. В заключение дается введение в технологию глубокого обучения с подкреплением и ее применение к построению игр со встроенным искусственным интеллектом.

Если говорить о практических приложениях, то в книгу включен код программ для классификации новостей по заранее заданным категориям, для синтаксического анализа текста, для анализа эмоциональной окраски текста, для синтеза текстов и частеречной разметки. Не оставлена без внимания также обработка изображений: распознавание рукописных цифр, классификация изображений по категориям и распознавание объектов с последующим аннотированием изображений. Из области анализа зву-

ковых сигналов взят пример распознавания слов, произносимых несколькими лицами. Техника обучения с подкреплением применяется для построения глубокой сети Q-обучения, способной автономно играть в игры.

Суть книги составляют эксперименты. Каждая сеть представлена несколькими вариантами, качество которых постепенно улучшается путем изменения входных параметров, формы сети, вида функции потерь и применяемых алгоритмов оптимизации. В ряде случаев приводятся сравнительные результаты обучения на CPU и GPU.

Чем глубокое обучение отличается от машинного обучения и искусственного интеллекта

Искусственный интеллект (ИИ) – очень широкая область исследований, посвященная *когнитивным* способностям машин: обучение определенному поведению, упреждающее взаимодействие с окружающей средой, способность к логическому выводу и дедукции, компьютерное зрение, распознавание речи, решение задач, представление знаний, восприятие действительности и многое другое (за подробностями отсылаем к книге S. Russell, P. Norvig «Artificial Intelligence: A Modern Approach», Prentice Hall, 2003). Менее формально под ИИ понимается любая ситуация, в которой машины имитируют *интеллектуальное* поведение, считающееся присущим человеку. Искусственный интеллект заимствует методы исследования из информатики, математики и статистики.

Машинное обучение (МО) – отрасль ИИ, посвященная тому, как обучать компьютеры решению конкретных задач без программирования (см. книгу С. М. Bishop «Pattern Recognition and Machine Learning», Springer, 2006). Основная идея МО заключается в том, что можно создавать алгоритмы, способные обучаться на данных и впоследствии давать предсказания. Существует три основных вида МО. В случае обучения с учителем машине предъявляются данные и правильные результаты, а цель состоит в том, чтобы машина обучилась на этих примерах и смогла выдавать осмысленные результаты для данных, которые раньше не видела. В случае обучения без учителя машине предъявляются только сами данные, а она должна выявить структуру без постороннего вмешательства.

В случае обучения с подкреплением машина ведет себя как агент, который взаимодействует с окружающей средой и обучается находить варианты поведения, приносящие вознаграждение.

Глубокое обучение (ГО) – подмножество методов МО, в которых применяются искусственные нейронные сети (ИНС), построенные на базе аналогии со структурой нейронов человеческого мозга (см. статью Y. Bengio «Learning Deep Architectures for AI», Found. Trends, vol. 2, 2009). Неформально говоря, слово «глубокий» подразумевает наличие большого числа слоев в ИНС, но его интерпретация со временем менялась. Если еще четыре года назад считалось, что 10 слоев достаточно, чтобы называть сеть *глубокой*, то теперь глубокой обычно называется сеть, содержащая сотни слоев.



Глубокое обучение – это настоящее цунами (см. статью C. D. Manning «Computational Linguistics and Deep Learning» в журнале «Computational Linguistics», vol. 41, 2015) в области машинного обучения в том смысле, что сравнительно небольшое число хитроумных методов с огромным успехом применяется в самых разных областях (обработка изображений, текста, видео и речи, компьютерное зрение), что позволило добиться значительного прогресса по сравнению с результатами, достигнутыми за предшествующие десятки лет. Своими успехами ГО обязано также наличию больших объемов обучающих данных (например, набора ImageNet в области обработки изображений) и относительно дешевых графических процессоров (GPU), позволяющих построить очень эффективную процедуру вычислений. В компаниях Google, Microsoft, Amazon, Apple, Facebook и многих других методы глубокого обучения постоянно используются для анализа больших

массивов данных. Теперь эти знания и навыки вышли за рамки чисто академических исследований и стали достоянием крупных промышленных компаний. Они стали неотъемлемой составной частью современной программной продукции, и владение ими обязательно для программиста. В этой книге не предполагается наличие у читателя специальной математической подготовки. Однако же знакомство с языком Python является необходимым условием.

Краткое содержание книги

В главе 1 «Основы нейронных сетей» излагаются основные сведения о нейронных сетях.

В главе 2 «Установка Keras и описание API» описано, как установить Keras в облаке AWS, Microsoft Azure, Google Cloud или на вашу собственную машину, а также дается краткий обзор различных API библиотеки Keras.

Глава 3 «Глубокое обучение с применением сверточных сетей» знакомит с понятием сверточной сети. Это фундаментальное новшество стало причиной успеха глубокого обучения в применении к различным предметным областям, от видео до речи, выйдя далеко за пределы обработки изображений, где эта идея первоначально зародилась.

Глава 4 «Порождающие состязательные сети и WaveNet» содержит введение в порождающие состязательные сети, используемые для синтеза данных, похожих на порождаемые людьми. Мы представляем глубокую нейронную сеть WaveNet, предназначенную для высококачественной имитации человеческого голоса и звучания музыкальных инструментов.

В главе 5 «Погружения слов» обсуждаются методы глубокого обучения, служащие для выявления связей между словами и группировки похожих слов.

В главе 6 «Рекуррентные нейронные сети» рассматривается класс нейронных сетей, оптимизированных для обработки последовательных данных, в т. ч. текста.

Глава 7 «Дополнительные модели глубокого обучения» содержит краткий обзор функционального API Keras, регрессионных сетей, автокодировщиков и т. д.

В главе 8 «Искусственный интеллект играет в игры» вы узнаете о глубоком обучении с подкреплением и о том, как Keras позволяет

его использовать для построения глубоких сетей, умеющих играть в аркадные игры.

Приложение содержит сводку обсуждаемых в книге тем и информацию о нововведениях в версии Keras 2.0.

Что необходимо для чтения книги

Вам понадобится следующее программное обеспечение:

- TensorFlow версии 1.0.0 или выше;
- Keras версии 2.0.2 или выше;
- Matplotlib версии 1.5.3 или выше;
- Scikit-learn версии 0.18.1 или выше;
- NumPy версии 1.12.1 или выше.

К оборудованию предъявляются следующие требования:

- 32- или 64-разрядная архитектура;
- CPU с таковой частотой не ниже 2 ГГц;
- оперативная память объемом не меньше 4 ГБ;
- не менее 10 ГБ свободного места на диске.

На кого рассчитана эта книга

Если вы – специалист по анализу и обработке данных со знанием машинного обучения или занимаетесь программированием ИИ и знакомы с нейронными сетями, то эта книга станет неплохой отправной точкой для овладения методами глубокого обучения с применением библиотеки Keras. Знание Python – обязательное условие.

Графические выделения

В этой книге тип информации обозначается шрифтом. Ниже приведено несколько примеров с пояснениями.

Фрагменты кода внутри абзаца, имена таблиц базы данных, папок и файлов, URL-адреса, данные, которые вводит пользователь, и адреса в Твиттере выделяются следующим образом: «Кроме того, мы загружаем истинные метки соответственно в `Y_train` и `Y_test` и применяем к ним унитарное кодирование».

Кусок кода выглядит так:

```
from keras.models import Sequential
model = Sequential()
model.add(Dense(12, input_dim=8, kernel_initializer='random_uniform'))
```

Желая привлечь внимание к части кода, мы выделяем ее полужирным шрифтом:

```
# 10 outputs
# final stage is softmax
model = Sequential()
model.add(Dense(NB_CLASSES, input_shape=(RESHAPED,)))
model.add(Activation('softmax'))
model.summary()
```

Входная и выходная информация командных утилит выглядит так:

```
pip install quiver_engine
```

Новые термины и важные фрагменты выделяются полужирным шрифтом. Например, элементы графического интерфейса в меню или диалоговых окнах выглядят в книге так: «Первоначально наша простая сеть имеет верность **92.22 %**, т. е. примерно 8 из 100 рукописных символов распознаются неправильно».



Таким значком обозначаются предупреждения и важные примечания.



Таким значком обозначаются советы и рекомендации.

Отзывы

Мы всегда рады отзывам читателей. Расскажите нам, что вы думаете об этой книге – что вам понравилось или, быть может, не понравилось. Читательские отзывы важны для нас, так как помогают выпускать книги, из которых вы черпаете максимум полезного для себя.

Чтобы отправить обычный отзыв, просто пошлите письмо на адрес feedback@packtpub.com, указав название книги в качестве темы. Если вы являетесь специалистом в некоторой области и хотели

бы стать автором или соавтором книги, познакомьтесь с инструкциями для авторов по адресу www.packtpub.com/authors.

Поддержка клиентов

Счастливым обладателям книг Packt мы можем предложить ряд услуг, которые позволят извлечь из своего приобретения максимум пользы.

Загрузка кода примеров

Вы можете скачать код примеров к этой книге из своей учетной записи на сайте <http://www.packtpub.com>. Если книга была куплена в другом месте, зайдите на страницу <http://www.packtpub.com/support>, зарегистрируйтесь, и мы отправим файлы по электронной почте.

Для скачивания файлов с кодом выполните следующие действия:

1. Зарегистрируйтесь или зайдите на наш сайт, указав свой адрес электронной почты и пароль.
2. Наведите мышь на вкладку **SUPPORT** в верхней части страницы.
3. Щелкните по ссылке **Code Downloads & Errata**.
4. Введите имя книги в поле **Search**.
5. Выберите интересующую вас книгу.
6. С помощью выпадающего меню укажите, где вы приобрели книгу.
7. Нажмите **Code Download**.

Загруженный файл можно распаковать, воспользовавшись последними версиями программ:

- WinRAR / 7-Zip для Windows;
- Zipeg / iZip / UnRarX для Mac;
- 7-Zip / PeaZip для Linux.

Код к этой книге имеется также на странице сайта GitHub по адресу <https://github.com/PacktPublishing/Deep-Learning-with-Keras>. По адресу <https://github.com/PacktPublishing/> размещен также код и видео к другим книгам из нашего обширного каталога. Полюбопытствуйте!

Загрузка цветных иллюстраций

Мы также предлагаем PDF-файл, содержащий цветные изображения, встречающиеся в книге. Цвет поможет лучше понять, как изменяются результаты. Этот файл можно скачать по адресу https://www.packtpub.com/sites/default/files/downloads/DeepLearningwithKeras_ColorImages.pdf.

Опечатки

Мы проверяли содержимое книги со всем тщанием, но какие-то ошибки все же могли проскользнуть. Если вы найдете в нашей книге ошибку, в тексте или в коде, пожалуйста, сообщите нам о ней. Так вы избавите других читателей от разочарования и поможете нам сделать следующие издания книги лучше. При обнаружении опечатки просьба зайти на страницу <http://www.packtpub.com/support>, выбрать книгу, щелкнуть по ссылке **Errata Submission Form** и ввести информацию об опечатке. Проверив ваше сообщение, мы поместим информацию об опечатке на нашем сайте или добавим ее в список замеченных опечаток в разделе Errata для данной книги.

Список ранее отправленных опечаток можно просмотреть, выбрав название книги на странице <http://www.packtpub.com/books/content/support>. Запрошенная информация появится в разделе **Errata**.

Нарушение авторских прав

Незаконное размещение защищенного авторским правом материала в Интернете – проблема для всех носителей информации. В издательстве Packt мы относимся к защите прав интеллектуальной собственности и лицензированию очень серьезно. Если вы обнаружите незаконные копии наших изданий в любой форме в Интернете, пожалуйста, незамедлительно сообщите нам адрес или название веб-сайта, чтобы мы могли предпринять соответствующие меры.

Просим отправить ссылку на вызывающий подозрение в пиратстве материал по адресу copyright@packtpub.com.

Мы будем признательны за помощь в защите прав наших авторов и содействие в наших стараниях предоставлять читателям полезные сведения.

Вопросы

Если вас смущает что-то в этой книге, вы можете связаться с нами по адресу questions@packtpub.com, и мы сделаем все возможное для решения проблемы.

Глава 1

Основы нейронных сетей

Искусственные нейронные сети (для краткости *нейросети* или просто *сети*) – это класс моделей машинного обучения, в основе которых лежат исследования центральной нервной системы млекопитающих. Нейросеть состоит из нескольких взаимосвязанных *нейронов*, организованных в *слои*, которые обмениваются между собой сообщениями (как говорят, *возбуждаются*) при выполнении определенных условий. Первые исследования относятся к 1950-м годам, когда было введено понятие перцептрона (см. статью F. Rosenblatt «The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain», *Psychological Review*, vol. 65, pp. 386–408, 1958), двуслойной сети для выполнения простых операций. Затем в конце 1960-х годов был предложен *алгоритм обратного распространения* для эффективного обучения многослойных сетей (см. статьи P. J. Werbos «Backpropagation through Time: What It Does and How to Do It», *Proceedings of the IEEE*, vol. 78, pp. 1550–1560, 1990 и G. E. Hinton, S. Osindero, Y. W. Teh «A Fast Learning Algorithm for Deep Belief Nets, *Neural Computing*, vol. 18, pp. 1527–1554, 2006). В некоторых работах утверждается, что эти методы корнями уходят глубже, чем принято считать (см. статью J. Schmidhuber «Deep Learning in Neural Networks: An Overview», by, vol. 61, pp. 85–117, 2015). Нейронные сети были предметом интенсивных научных исследований до 1980-х годов, когда на первый план вышли другие, более простые, подходы. Однако с середины 2000-х годов отмечается возрождение интереса к этой теме в связи с прорывным алгоритмом быстрого обучения, предложенным Дж. Хинтоном (см. S. Leven «The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting», *Neural Networks*, vol. 9, 1996, и D. E. Rumelhart, G. E. Hinton, R. J. Williams «Learning Representations by Backpropagating Errors», vol. 323, 1986), и появлением (примерно в 2011 году) графических процессоров для массово-параллельных численных расчетов.

Эти достижения проложили дорогу современному *глубокому обучению*, классу нейронных сетей, для которых характерно большое число слоев и которые способны обучать весьма изощренные модели на основе иерархии уровней абстрагирования. Несколько лет назад *глубокой* считалась сеть с 3–5 слоями, теперь же их число возросло до 100–200.

Обучение путем последовательного абстрагирования напоминает модели зрения в мозге человека, эволюционировавшие миллионы лет. Человеческая система зрения действительно состоит из нескольких уровней. Глаз соединен с областью мозга, которая называется **первичной зрительной корой**, или **зрительной корой V1** и занимает задний полюс затылочной доли каждого полушария. Эта область имеется у многих млекопитающих и играет важную роль в распознавании простых образов и обработке информации об изменении ориентации, пространственной частоте и цвете. Согласно некоторым оценкам, первичная зрительная кора содержит примерно 140 миллионов нейронов и 10 миллиардов связей между ними. Кора V1 соединяется с областями V2, V3, V4, V5 и V6, отвечающими за всю последующую обработку изображений и распознавание более сложных объектов: фигур, лиц, животных и многого другого. Такая многослойная организация явилась результатом множества попыток, которые природа предпринимала на протяжении сотен миллионов лет. Согласно оценкам, кора головного мозга человека насчитывает порядка 16 миллиардов нейронов, и примерно 10–25 % из них относятся к зрению (см. статью S. Herculano-Houzel «The Human Brain in Numbers: A Linearly Scaled-up Primate Brain», vol. 3, 2009). Глубокое обучение заимствовало идею многослойной организации у зрительной системы человека: наружные слои искусственных нейронов обучаются базовым свойствам изображений, а более глубокие обрабатывают более сложные концепции.

В этой книге рассмотрено несколько важных аспектов нейронных сетей на примере кода с использованием минималистской и весьма эффективной библиотеки глубокого обучения Keras, написанной на языке Python и работающей поверх библиотеки TensorFlow от Google (см. <https://www.tensorflow.org/>) или библиотеки Theano, разработанной в Монреальском университете (см. <http://deeplearning.net/software/theano/>). Итак, приступим.

В этой главе будут рассмотрены следующие темы:

- перцептрон;
- многослойный перцептрон;
- функции активации;
- градиентный спуск;
- стохастический градиентный спуск;
- алгоритм обратного распространения.

Перцептрон

Перцептрон – это простой алгоритм, который получает входной вектор x , содержащий n значений $(x_1, x_2, \dots, x_n)$, которые часто называются входными признаками, или просто признаками, и выдает на выходе 1 (да) или 0 (нет). Формально говоря, мы определяем функцию:

$$f(x) = \begin{cases} 1, & \text{если } wx + b > 0 \\ 0 & \text{в противном случае} \end{cases}$$

Здесь w – вектор весов, wx – скалярное произведение $\sum_{j=1}^m w_j x_j$, а b – смещение. Как мы знаем из геометрии, уравнение $wx + b = 0$ определяет граничную гиперплоскость, положение которой изменяется в зависимости от значений w и b . Если x лежит выше этой гиперплоскости (в двумерном случае – прямой), то ответ положительный, иначе отрицательный. Очень простой алгоритм! С помощью перцептрона нельзя выразить ответ «*может быть*». Он может ответить *да* (1) или *нет* (0), если мы знаем, как определить w и b , а это и есть процесс обучения, который обсуждается в следующих разделах.

Первый пример кода с использованием Keras

Исходным строительным блоком Keras является модель, а простейшая модель называется **последовательной**. В Keras последовательная модель представляет собой линейный конвейер (стек) слоев нейронной сети. В следующем фрагменте определен один слой с 12 нейронами, который ожидает получить 8 входных переменных (признаков):


```
from keras.models import Sequential
model = Sequential()
model.add(Dense(12, input_dim=8, kernel_initializer='random_uniform'))
```

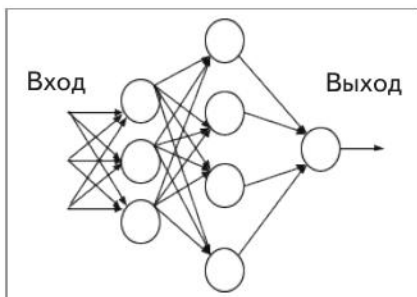
На этапе инициализации каждому нейрону можно назначить вес. Keras предлагает несколько вариантов, перечислим наиболее употребительные:

- `random_uniform`: веса инициализируются равномерно распределенными случайными значениями из диапазона $(-0.05, 0.05)$. Иными словами, любое значение из этого интервала выбирается с одинаковой вероятностью;
- `random_normal`: веса инициализируются нормально распределенными случайными значениями со средним 0 и стандартным отклонением 0.05. Те, кто не знает, что такое нормальное распределение, могут представлять себе симметричную колоколообразную кривую;
- `zero`: все веса инициализируются нулями.

Полное описание параметров имеется на странице <https://keras.io/initializations/>.

Многослойный перцептрон – первый пример нейросети

В этой главе мы определим первый пример сети с несколькими линейными слоями. Исторически перцептроном называлась модель с единственным линейным слоем, поэтому модель с несколькими слоями логично назвать **многослойным перцептроном** (МСП). На рисунке ниже показана нейронная сеть общего вида с одним входным, одним промежуточным и одним выходным слоем.



Каждый узел первого слоя получает входной сигнал и возбуждается в соответствии с заранее определенными локальными граничными условиями. Выход первого слоя подается на вход второго, а выход второго – последнему слою, состоящему всего из одного нейрона. Интересно, что такая многослойная организация отдаленно напоминает работу человеческой системы зрения, о чем мы уже говорили.

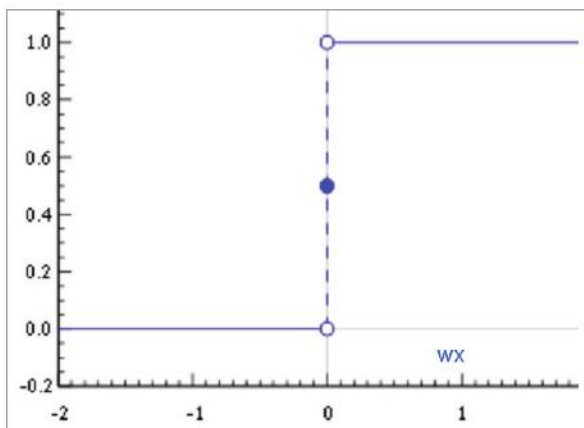


Эта *сеть* плотная в том смысле, что каждый нейрон одного слоя связан со всеми нейронами предыдущего слоя и всеми нейронами следующего слоя.

Проблемы обучения перцептрона и их решение

Рассмотрим один нейрон; каков оптимальный выбор веса w и смещения b ? В идеале мы хотели бы предъявить набор обучающих примеров и поручить компьютеру выбрать вес и смещение, так чтобы ошибка при вычислении результата была минимальна. Переходя к конкретике, предположим, что имеется набор изображений кошек и другой набор изображений, на которых кошек нет. Для простоты будем считать, что каждый нейрон анализирует только один входной пиксель. Мы хотели бы, чтобы в процессе обработки изображений компьютером наш нейрон изменял свой вес и смещение таким образом, чтобы число изображений, ошибочно распознанных как не кошки, со временем уменьшалось. Этот подход кажется интуитивно очевидным, но для него требуется, чтобы малое изменение веса (и/или смещения) приводило к малому изменению результата.

Если имеется большой скачок на выходе, то *прогрессивное* обучение невозможно (разве что пробовать все возможные направления – это называется исчерпывающим поиском – не зная, достигаем ли мы какого-нибудь улучшения). В конце концов, дети ведь учатся постепенно. К сожалению, для перцептрона такое «постепенное» поведение не характерно. Перцептрон выдает значение 0 или 1 , разница между ними велика, и это никак не способствует его обучению, что видно из следующего рисунка:



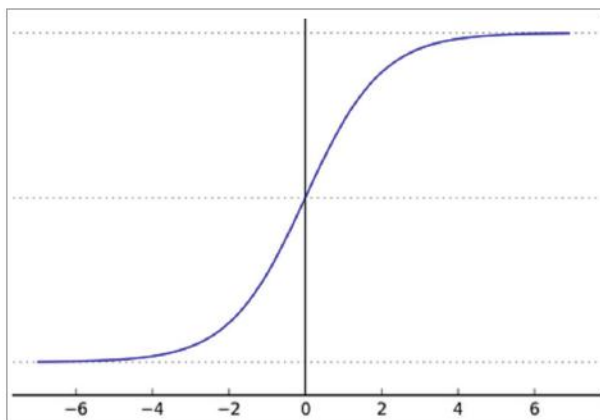
Нам нужно что-то более гладкое – непрерывная, дифференцируемая функция, монотонно возрастающая от 0 до 1.

Сигмоида

Сигмоида определяется следующим образом:

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

Как видно из графика ниже, она непрерывна и изменяется от 0 до 1, когда аргумент пробегает область определения $(-\infty, \infty)$:



Нейрон может воспользоваться сигмоидой для вычисления нелинейной функции $\sigma(z = wx + b)$. Отметим, что когда величина $z = wx + b$ очень велика и положительна, $e^{-z} \rightarrow 0$, так что $\sigma(z) \rightarrow 1$, а когда эта величина велика по модулю и отрицательна, то $e^{-z} \rightarrow \infty$, так что $\sigma(z) \rightarrow 0$. Иными словами, нейрон с сигмоидной функцией активации ведет себя подобно перцептрону, но изменяется плавно и может порождать такие значения, как 0.5539 или 0.123191. В некотором смысле сигмоидный нейрон умеет давать ответ «может быть».

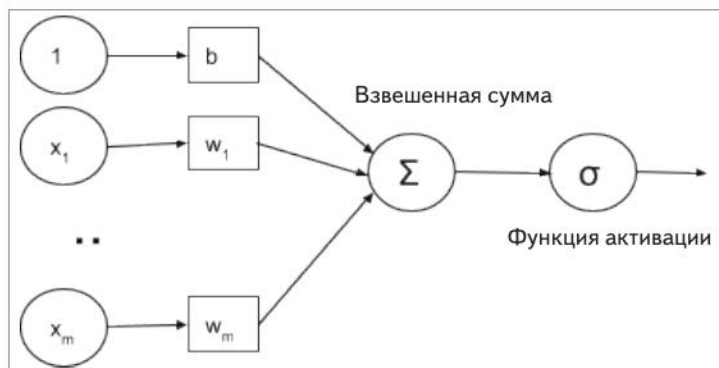
Блок линейной ректификации

Сигмоида – не единственная гладкая функция, применяемая в нейронных сетях. В последнее время стала очень популярна совсем простая функция, называемая блоком линейной ректификации (rectified linear unit, ReLU), поскольку в экспериментах она дает замечательные результаты. ReLU определяется формулой $f(x) = \max(0, x)$, а ее график показан на рисунке ниже. Как видим, она равна нулю для отрицательных значений x и линейно возрастает для положительных.



Функции активации

Сигмоида и ReLU называются *функциями активации*. В разделе «Тестирование различных оптимизаторов в Keras» мы увидим, что непрерывное изменение, характерное для этих функций, крайне важно для разработки алгоритмов обучения, которые адаптируются постепенно, стремясь уменьшить ошибку сети. На рисунке ниже показана схема применения функции активации σ к входному вектору $(x_1, x_2, \dots, x_m)$, вектору весов $(w_1, w_2, \dots, w_m)$, смещению b и сумматору Σ :



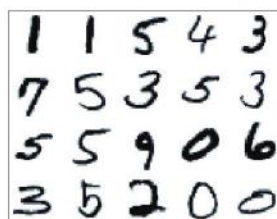
Keras поддерживает несколько функций активации, их полный перечень приведен на странице <https://keras.io/activations/>.

Реальный пример – распознавание рукописных цифр

В этом разделе мы построим сеть, умеющую распознавать рукописные цифры. Для этого будем использовать набор данных MNIST (см. <http://yann.lecun.com/exdb/mnist/>), включающий 60 000 обучающих и 10 000 тестовых примеров. В обучающих примерах человеком проставлены правильные ответы. Например, с изображением рукописной цифры три ассоциирована метка 3.

Когда имеется набор данных, содержащий правильные ответы, говорят об *обучении с учителем*. Обучающие примеры используются для обучения сети. С тестовыми примерами также ассоциированы правильные ответы, но идея в том, чтобы притвориться, будто они неизвестны, дать сети возможность сделать предсказание, а затем, сравнив его с правильным ответом, оценить, насколько хорошо сеть научилась распознавать цифры. Так что тестовые примеры применяются только для проверки сети – что и не удивительно.

Все изображения в наборе MNIST полутоновые, размера 28×28 пикселей. На рисунке приведено несколько примеров.



Унитарное кодирование

Во многих приложениях удобно преобразовывать категориальные (нечисловые) признаки в числовые. Например, категориальный признак – цифру, принимающую значение d от 0 до 9, – можно представить бинарным вектором длины 10, в котором d -ый элемент равен 1, а все остальные – нулю. Такое представление называется *унитарным кодированием* (one-hot encoding) и очень часто применяется в добыче данных, когда алгоритм обучения рассчитан на работу с числами.

Определение простой нейронной сети в Keras

Воспользуемся библиотекой Keras, чтобы определить сеть, распознающую рукописные цифры из набора MNIST. Начнем с очень простой нейросети и постепенно будем ее улучшать.

Keras предоставляет средства для загрузки набора данных и разбиения его на обучающий, `x_train`, и тестовый, `x_test`. Для поддержки вычислений на GPU данные преобразуются к типу `float32` и нормируются на интервал `[0, 1]`. Кроме того, в `y_train` и `y_test` мы загружаем правильные метки и применяем к ним унитарное кодирование. Вот как выглядит код:

```
from __future__ import print_function
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD
from keras.utils import np_utils
np.random.seed(1671) # для воспроизводимости результатов

# сеть и ее обучение
NB_EPOCH = 200
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10 # количество результатов = числу цифр
OPTIMIZER = SGD() # СГС-оптимизатор, обсуждается ниже в этой главе
N_HIDDEN = 128
VALIDATION_SPLIT=0.2 # какая часть обучающего набора зарезервирована для контроля

# данные: случайно перетасованы и разбиты на обучающий и тестовый набор
#
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# X_train содержит 60000 изображений размера 28x28 -->
# преобразуем в массив 60000 x 784 RESHAPED = 784
```



```
#
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# нормировать
#
X_train /= 255
X_test /= 255
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

# преобразовать векторы классов в бинарные матрицы классов
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)
```

Во входном слое с каждым пикселем изображения ассоциирован один нейрон, т. е. всего получается $28 \times 28 = 784$ нейрона.

Обычно значения, ассоциированные с пикселями, нормируются с целью привести их к диапазону $[0, 1]$ (это значит, что яркость каждого пикселя делится на максимально возможную яркость 255). На выходе получается 10 классов, по одному для каждой цифры.

Последний слой состоит из единственного нейрона с функцией активации *softmax*, являющейся обобщением сигмоиды. *Softmax сплюсчивает* k -мерный вектор, содержащий произвольные вещественные числа, в k -мерный вектор вещественных чисел из интервала $(0, 1)$. В нашем случае она агрегирует 10 ответов, выданных предыдущим слоем из 10 нейронов:

```
# 10 выходов
# на последнем этапе softmax
model = Sequential()
model.add(Dense(NB_CLASSES, input_shape=(RESHAPED,)))
model.add(Activation('softmax'))
model.summary()
```

Определенную таким образом модель необходимо откомпилировать, т. е. привести к виду, допускающему исполнение базовой библиотекой (Theano или TensorFlow). Перед компиляцией необходимо принять несколько решений:

- выбрать *оптимизатор*, т. е. конкретный алгоритм, который будет обновлять веса в процессе обучения модели;
- выбрать целевую функцию, которую оптимизатор использует для навигации по пространству весов (часто целевая

функция называется также *функцией потерь*, а процесс оптимизации – *минимизацией* потерь);

- оценить качество обученной модели.

Перечислим несколько распространенных целевых функций (полный перечень целевых функций, поддерживаемых Keras, приведен на странице <https://keras.io/objectives/>):

- **Среднеквадратическая ошибка (СКО):** это усредненная сумма разностей квадратов между предсказанными и истинными значениями. Если обозначить $\hat{Y}$ вектор n предсказаний, а Y – вектор n наблюдаемых значений, то среднеквадратическая ошибка равна

$$СКО = \frac{1}{n} \sum_{i=1}^n (\hat{Y} - Y)^2$$



Если предсказание сильно отличается от истинного значения, то возведение в квадрат делает отличие еще более явственным.

- **Бинарная перекрестная энтропия:** если модель предсказывает значение p , тогда как истинное значение равно t , то бинарная перекрестная энтропия равна:

$$-t \log(p) - (1 - t) \log(1 - p)$$



Эта целевая функция подходит для предсказания бинарных меток.

- **Категориальная перекрестная энтропия:** это логарифмическая потеря в случае нескольких классов. Если модель предсказывает значения $p_{i,j}$, тогда как истинные значения равны $t_{i,j}$, то категориальная перекрестная энтропия равна:

$$L_i = -\sum_j t_{i,j} \log(p_{i,j})$$



Эта целевая функция подходит для предсказания многоклассовых меток. Она же по умолчанию используется совместно с функцией активации softmax.

Ниже перечислено несколько популярных показателей качества (полный список см. на странице <https://keras.io/metrics/>):

- **верность**: отношение числа правильных предсказаний к общему числу меток;
- **точность**: доля правильных ответов модели;
- **полнота**: доля обнаруженных истинных событий.

Показатели качества похожи на целевые функции, различаются они только тем, что показатели используются не для обучения модели, а для оценки ее качества. Компиляция модели в Keras производится просто:

```
model.compile(loss='categorical_crossentropy',
              optimizer=OPTIMIZER,
              metrics=['accuracy'])
```

Для обучения откомпилированной модели служит функция `fit()`, принимающая в частности следующие параметры:

- `epochs`: число периодов – сколько раз обучающий набор предъявляется модели. На каждой итерации оптимизатор пытается подкорректировать веса, стремясь минимизировать целевую функцию;
- `batch_size`: сколько обучающих примеров должен увидеть оптимизатор, прежде чем он обновит веса.

Обучить модель в Keras очень просто. Допустим, что выполняет `NB_EPOCH` итераций:

```
history = model.fit(X_train, Y_train,
                   batch_size=BATCH_SIZE, epochs=NB_EPOCH,
                   verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
```



Мы зарезервировали часть обучающего набора для контроля. Идея в том, чтобы отложить часть обучающих данных для контроля качества в процессе обучения. Эта рекомендуемая практика для всех задач машинного обучения, и далее мы будем ее придерживаться.

После того как модель обучена, ее следует проверить на тестовом наборе, который содержит ранее не предъявлявшиеся примеры. Таким образом, мы сможем получить минимальное значение, достигаемое целевой функцией, и наилучшее значение показателя качества.

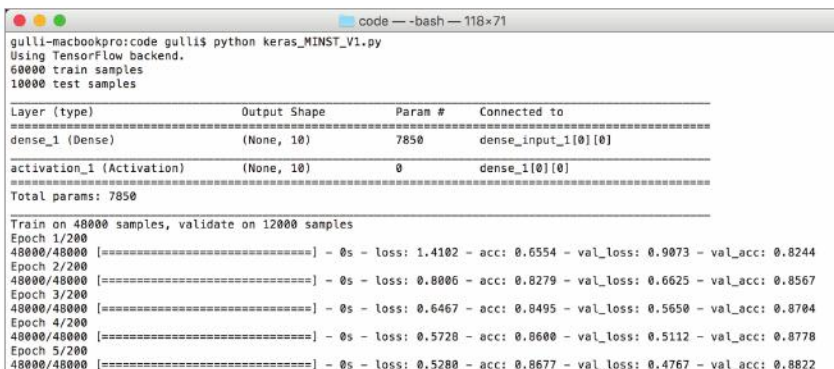
Подчеркнем, что обучающий и тестовый набор не должны пересекаться. Не имеет никакого смысла оценивать модель на примере, который она видела во время обучения. Смысл обучения в том, чтобы модель могла обобщаться на ранее не встречавшиеся данные, а не в том, чтобы она запоминала то, что уже известно.

```
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

Вы только что определили свою первую нейронную сеть на Keras, можете принимать поздравления. Всего несколько строк кода – и компьютер умеет распознавать рукописные цифры. Теперь выполним этот код и оценим качество.

Прогон простой сети Keras и создание эталона для сравнения

На рисунке ниже показано, что происходит во время прогона программы:



```
gulli-macbookpro:code gulli$ python keras_MINST_V1.py
Using TensorFlow backend.
60000 train samples
10000 test samples
```

Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 10)	7850	dense_input_1[0][0]
activation_1 (Activation)	(None, 10)	0	dense_1[0][0]

```
Total params: 7850

Train on 48000 samples, validate on 12000 samples
Epoch 1/200 [=====] - 0s - loss: 1.4102 - acc: 0.6554 - val_loss: 0.9073 - val_acc: 0.8244
Epoch 2/200 [=====] - 0s - loss: 0.8005 - acc: 0.8279 - val_loss: 0.6625 - val_acc: 0.8567
Epoch 3/200 [=====] - 0s - loss: 0.6467 - acc: 0.8495 - val_loss: 0.5650 - val_acc: 0.8704
Epoch 4/200 [=====] - 0s - loss: 0.5728 - acc: 0.8600 - val_loss: 0.5112 - val_acc: 0.8778
Epoch 5/200 [=====] - 0s - loss: 0.5280 - acc: 0.8677 - val_loss: 0.4767 - val_acc: 0.8822
```

Сначала распечатывается архитектура сети, мы видим типы слоев, форму выхода, количество оптимизируемых параметров и характер связей между слоями. Затем сеть обучается на 48 000 примерах, а 12 000 примеров зарезервировано для контроля. Построенная нейронная сеть тестируется на 10 000 примерах. Как видим, Keras использует для вычислений базовую библиотеку TensorFlow. Пока что не будем вдаваться в детали обучения, но отметим, что программа выполнила 200 итераций и с каждым разом верность улучшалась.

По завершении обучения мы проверяем модель на тестовом наборе и видим, что на обучающем наборе получена верность 92.36%, на контрольном – 92.27%, а на тестовом – 92.22%.

Это значит, что доля неправильно распознанных рукописных символов составляет чуть менее одного на десять. Безусловно, этот результат можно улучшить.

```
Epoch 198/200
48000/48000 [=====] - 0s - loss: 0.2761 - acc: 0.9230 - val_loss: 0.2762 - val_acc: 0.9224
Epoch 199/200
48000/48000 [=====] - 0s - loss: 0.2760 - acc: 0.9231 - val_loss: 0.2762 - val_acc: 0.9223
Epoch 200/200
48000/48000 [=====] - 0s - loss: 0.2758 - acc: 0.9236 - val_loss: 0.2761 - val_acc: 0.9227
9808/10000 [=====] - ETA: 0s
Test score: 0.27792117235
Test accuracy: 0.9222
gulli-macbookpro:code gulli$
```

Улучшение простой сети в Keras посредством добавления скрытых слоев

Мы достигли верности 92.36% на обучающем наборе, 92.27% – на контрольном и 92.22% – на тестовом. Для начала неплохо, но есть возможность добиться большего. Посмотрим, как.

Первое улучшение – включить в сеть дополнительные слои. После входного слоя поместим первый плотный слой с `N_HIDDEN` нейронами и функцией активации `relu`. Этот слой называется *скрытым*, потому что он напрямую не соединен ни с входом, ни с выходом. После первого скрытого слоя добавим еще один, также содержащий `N_HIDDEN` нейронов, а уже за ним будет расположен выходной слой с 10 нейронами, которые возбуждаются, если распознана соответствующая цифра. Вот код, определяющий новую сеть:

```
from __future__ import print_function
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD
from keras.utils import np_utils
np.random.seed(1671) # для воспроизводимости результатов

# сеть и ее обучение
NB_EPOCH = 20
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10 # количество результатов = числу цифр
```



```

OPTIMIZER = SGD() # СГС-оптимизатор, обсуждается ниже в этой главе
N_HIDDEN = 128
VALIDATION_SPLIT=0.2 # какая часть обучающего набора зарезервирована для контроля

# данные: случайно перетасованы и разбиты на обучающий и тестовый набор
#
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# X_train содержит 60000 изображений размера 28x28 --> преобразуем в массив
60000 x 784 RESHAPED = 784
#
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# нормировать
X_train /= 255
X_test /= 255
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

# преобразовать векторы классов в бинарные матрицы классов
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)

# M_HIDDEN скрытых слоев
# 10 выходов
# на последнем этапе softmax
model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dense(NB_CLASSES))
model.add(Activation('softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',
              optimizer=OPTIMIZER,
              metrics=['accuracy'])
history = model.fit(X_train, Y_train,
                   batch_size=BATCH_SIZE, epochs=NB_EPOCH,
                   verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print("Test accuracy:", score[1])

```

Выполним этот код и посмотрим, какие результаты дает такая многослойная сеть. Неплохо. Добавив два скрытых слоя, мы достигли верности 94.50% на обучающем наборе, 94.63% – на кон-

трольном и 94.41% – на тестовом, т. е. получили прирост 2.2% на тестовом наборе по сравнению с предыдущей сетью. И при этом число итераций резко уменьшилось – с 200 до 20. Хорошо, но мы хотим большего.

Если хотите, можете посмотреть, что будет, если добавить только один скрытый слой вместо двух или если добавить больше двух слоев. Оставляю это в качестве упражнения. На рисунке ниже показан результат работы последней программы.

```
gulli-macbookpro:code gulli$ python keras_MINST_V2.py
Using TensorFlow backend.
60000 train samples
10000 test samples
```

Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 128)	100480	dense_input_1[0][0]
activation_1 (Activation)	(None, 128)	0	dense_1[0][0]
dense_2 (Dense)	(None, 128)	16512	activation_1[0][0]
activation_2 (Activation)	(None, 128)	0	dense_2[0][0]
dense_3 (Dense)	(None, 10)	1290	activation_2[0][0]
activation_3 (Activation)	(None, 10)	0	dense_3[0][0]

```

Total params: 118202

Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [=====] - 1s - loss: 1.5266 - acc: 0.6101 - val_loss: 0.7839 - val_acc: 0.8296
Epoch 2/20
48000/48000 [=====] - 1s - loss: 0.6108 - acc: 0.8464 - val_loss: 0.4603 - val_acc: 0.8796
Epoch 3/20
48000/48000 [=====] - 1s - loss: 0.4422 - acc: 0.8794 - val_loss: 0.3765 - val_acc: 0.8963
Epoch 4/20
48000/48000 [=====] - 1s - loss: 0.3796 - acc: 0.8946 - val_loss: 0.3374 - val_acc: 0.9065
Epoch 5/20
48000/48000 [=====] - 1s - loss: 0.3458 - acc: 0.9027 - val_loss: 0.3119 - val_acc: 0.9116
Epoch 6/20
48000/48000 [=====] - 1s - loss: 0.3214 - acc: 0.9090 - val_loss: 0.2940 - val_acc: 0.9165
Epoch 7/20
48000/48000 [=====] - 1s - loss: 0.3033 - acc: 0.9148 - val_loss: 0.2794 - val_acc: 0.9213
Epoch 8/20
48000/48000 [=====] - 1s - loss: 0.2885 - acc: 0.9181 - val_loss: 0.2668 - val_acc: 0.9251
Epoch 9/20
48000/48000 [=====] - 1s - loss: 0.2763 - acc: 0.9220 - val_loss: 0.2569 - val_acc: 0.9287
Epoch 10/20
48000/48000 [=====] - 1s - loss: 0.2654 - acc: 0.9245 - val_loss: 0.2491 - val_acc: 0.9304
Epoch 11/20
48000/48000 [=====] - 1s - loss: 0.2556 - acc: 0.9274 - val_loss: 0.2400 - val_acc: 0.9335
Epoch 12/20
48000/48000 [=====] - 1s - loss: 0.2464 - acc: 0.9299 - val_loss: 0.2329 - val_acc: 0.9355
Epoch 13/20
48000/48000 [=====] - 1s - loss: 0.2382 - acc: 0.9321 - val_loss: 0.2279 - val_acc: 0.9369
Epoch 14/20
48000/48000 [=====] - 1s - loss: 0.2309 - acc: 0.9342 - val_loss: 0.2208 - val_acc: 0.9388
Epoch 15/20
48000/48000 [=====] - 1s - loss: 0.2237 - acc: 0.9365 - val_loss: 0.2140 - val_acc: 0.9413
Epoch 16/20
48000/48000 [=====] - 1s - loss: 0.2172 - acc: 0.9380 - val_loss: 0.2085 - val_acc: 0.9423
Epoch 17/20
48000/48000 [=====] - 1s - loss: 0.2110 - acc: 0.9397 - val_loss: 0.2035 - val_acc: 0.9435
Epoch 18/20
48000/48000 [=====] - 1s - loss: 0.2051 - acc: 0.9415 - val_loss: 0.1993 - val_acc: 0.9445
Epoch 19/20
48000/48000 [=====] - 1s - loss: 0.1997 - acc: 0.9427 - val_loss: 0.1954 - val_acc: 0.9461
Epoch 20/20
48000/48000 [=====] - 1s - loss: 0.1947 - acc: 0.9450 - val_loss: 0.1914 - val_acc: 0.9463
9696/10000 [=====] - ETA: 0s
Test score: 0.19105276902
Test accuracy: 0.9441
gulli-macbookpro:code gulli$
```

Дальнейшее улучшение простой сети Keras с помощью прореживания

Наше последнее достижение – верность 94.50% на обучающем наборе, 94.63% – на контрольном и 94.41% – на тестовом. Второе улучшение совсем простое. Мы применим прореживание – с вероятностью `dropout` будем случайным образом отбрасывать некоторые значения, распространяющиеся внутри сети, состоящей из плотных скрытых слоев. Это хорошо известная форма регуляризации в машинном обучении. Как ни странно, отбрасывание некоторых значений приводит к повышению качества:

```
from __future__ import print_function
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD
from keras.utils import np_utils
np.random.seed(1671) # для воспроизводимости результатов

# сеть и ее обучение
NB_EPOCH = 20
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10 # количество результатов = числу цифр
OPTIMIZER = SGD() # СГС-оптимизатор, обсуждается ниже в этой главе
N_HIDDEN = 128
VALIDATION_SPLIT=0.2 # какая часть обучающего набора зарезервирована
для контроля DROPOUT = 0.3

# данные: случайно перетасованы и разбиты на обучающий и тестовый набор
#
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# X_train содержит 60000 изображений размера 28x28 --> преобразуем
в массив 60000 x 784 RESHAPED = 784
```

```

#
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# нормировать
X_train /= 255
X_test /= 255
# преобразовать векторы классов в бинарные матрицы классов
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)

# M_HIDDEN скрытых слоев
model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(NB_CLASSES))
model.add(Activation('softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',
              optimizer=OPTIMIZER,
              metrics=['accuracy'])
history = model.fit(X_train, Y_train,
                    batch_size=BATCH_SIZE, epochs=NB_EPOCH,
                    verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])

```

Выполнив те же 20 итераций, что и раньше, мы увидим, что сеть достигла верности 91.54% на обучающем наборе, 94.48% – на контрольном и 94.25% – на тестовом:

```

gulli-macbookpro:code gulli$ python keras_MINST_V3_1.py
Using TensorFlow backend.
60000 train samples
10000 test samples

Layer (type)                 Output Shape          Param #          Connected to
=====
dense_1 (Dense)              (None, 128)           100480           dense_input_1[0][0]
activation_1 (Activation)    (None, 128)           0               dense_1[0][0]
dropout_1 (Dropout)         (None, 128)           0               activation_1[0][0]
dense_2 (Dense)              (None, 128)           16512            dropout_1[0][0]
activation_2 (Activation)    (None, 128)           0               dense_2[0][0]
dropout_2 (Dropout)         (None, 128)           0               activation_2[0][0]
dense_3 (Dense)              (None, 10)            1290             dropout_2[0][0]
activation_3 (Activation)    (None, 10)            0               dense_3[0][0]
=====
Total params: 118282

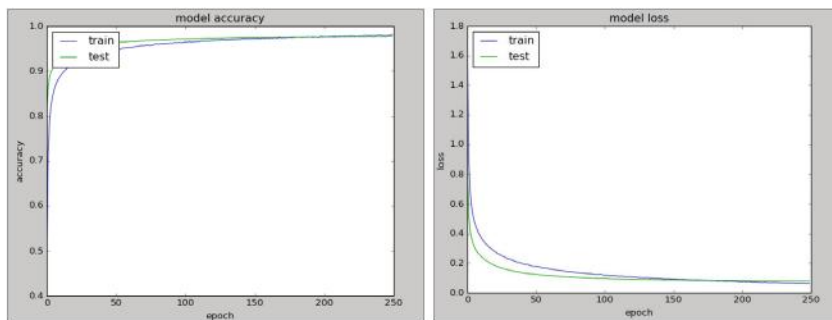
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [=====] - 1s - loss: 1.7206 - acc: 0.4625 - val_loss: 0.9125 - val_acc: 0.8036
Epoch 2/20
48000/48000 [=====] - 1s - loss: 0.9254 - acc: 0.7149 - val_loss: 0.5374 - val_acc: 0.8621
Epoch 3/20
48000/48000 [=====] - 1s - loss: 0.6938 - acc: 0.7883 - val_loss: 0.4240 - val_acc: 0.8872
Epoch 4/20
48000/48000 [=====] - 1s - loss: 0.5917 - acc: 0.8205 - val_loss: 0.3724 - val_acc: 0.8958
Epoch 5/20
48000/48000 [=====] - 1s - loss: 0.5307 - acc: 0.8398 - val_loss: 0.3370 - val_acc: 0.9038
Epoch 6/20
48000/48000 [=====] - 1s - loss: 0.4868 - acc: 0.8546 - val_loss: 0.3126 - val_acc: 0.9084
Epoch 7/20
48000/48000 [=====] - 1s - loss: 0.4563 - acc: 0.8654 - val_loss: 0.2939 - val_acc: 0.9126
Epoch 8/20
48000/48000 [=====] - 1s - loss: 0.4322 - acc: 0.8726 - val_loss: 0.2789 - val_acc: 0.9173
Epoch 9/20
48000/48000 [=====] - 1s - loss: 0.4061 - acc: 0.8799 - val_loss: 0.2666 - val_acc: 0.9196
Epoch 10/20
48000/48000 [=====] - 1s - loss: 0.3908 - acc: 0.8848 - val_loss: 0.2556 - val_acc: 0.9236
Epoch 11/20
48000/48000 [=====] - 1s - loss: 0.3758 - acc: 0.8893 - val_loss: 0.2463 - val_acc: 0.9263
Epoch 12/20
48000/48000 [=====] - 1s - loss: 0.3592 - acc: 0.8938 - val_loss: 0.2372 - val_acc: 0.9297
Epoch 13/20
48000/48000 [=====] - 1s - loss: 0.3491 - acc: 0.8970 - val_loss: 0.2294 - val_acc: 0.9323
Epoch 14/20
48000/48000 [=====] - 1s - loss: 0.3361 - acc: 0.9009 - val_loss: 0.2224 - val_acc: 0.9344
Epoch 15/20
48000/48000 [=====] - 1s - loss: 0.3266 - acc: 0.9036 - val_loss: 0.2165 - val_acc: 0.9348
Epoch 16/20
48000/48000 [=====] - 1s - loss: 0.3182 - acc: 0.9064 - val_loss: 0.2102 - val_acc: 0.9371
Epoch 17/20
48000/48000 [=====] - 1s - loss: 0.3073 - acc: 0.9103 - val_loss: 0.2035 - val_acc: 0.9395
Epoch 18/20
48000/48000 [=====] - 1s - loss: 0.2998 - acc: 0.9109 - val_loss: 0.1987 - val_acc: 0.9418
Epoch 19/20
48000/48000 [=====] - 1s - loss: 0.2930 - acc: 0.9131 - val_loss: 0.1930 - val_acc: 0.9423
Epoch 20/20
48000/48000 [=====] - 1s - loss: 0.2855 - acc: 0.9154 - val_loss: 0.1893 - val_acc: 0.9448
9689/10000 [=====] - ETA: 0s
Test score: 0.191873697177
Test accuracy: 0.9425
gulli-macbookpro:code gulli$

```

Отметим, что верность на обучающем наборе должна быть выше, чем на тестовом, в противном случае мы прервали обучение слишком рано. Увеличив число периодов до 250, мы получим верность 98.1% на обучающем наборе, 97.73% – на контрольном и 97.7% – на тестовом:


```
Epoch 248/250
48000/48000 [=====] - 1s - loss: 0.0630 - acc: 0.9804 - val_loss: 0.0785 - val_acc: 0.9769
Epoch 249/250
48000/48000 [=====] - 1s - loss: 0.0634 - acc: 0.9799 - val_loss: 0.0789 - val_acc: 0.9775
Epoch 250/250
48000/48000 [=====] - 1s - loss: 0.0616 - acc: 0.9810 - val_loss: 0.0787 - val_acc: 0.9773
9696/10000 [=====] - ETA: 0s
Test score: 0.0726828922328
Test accuracy: 0.9777
gulli-macbookpro:code gulli$
```

Интересно понаблюдать за тем, как возрастает верность на обучающем и тестовом наборе при увеличении числа периодов. Как видно из приведенного ниже графика, эти две кривые сходятся, когда число периодов приблизительно равно 250, так что последующее обучение ничего не даст.



Замечено, что сети со случайным прореживанием внутренних слоев часто лучше обобщаются на новые примеры из тестового набора. Интуитивно это можно объяснить тем, что каждый нейрон становится «умнее», потому что знает, что нельзя полагаться на соседей. В процессе тестирования прореживание не производится, т. е. используются все тщательно настроенные нейроны. Короче говоря, в общем случае рекомендуется проверять, как будет работать сеть, если применена та или иная форма прореживания.

Тестирование различных оптимизаторов в Keras

Мы определили и использовали сеть, теперь будет полезно на интуитивном уровне объяснить, как происходит обучение сети. Остановимся на одном популярном методе обучения – **градиентном спуске (ГС)**. Представим себе общую функцию стоимости $C(w)$ от одной переменной w с графиком такого вида:



Градиентный спуск можно уподобить альпинисту, спускающемуся с горы в долину. Гора представлена функцией C , а долина – ее минимальным значением C_{min} . Альпинист находится в начальной точке w_0 и перемещается небольшими шажками. На каждом шаге r градиент дает направление максимального роста. Математически это направление определяется частной производной $\frac{ac}{\partial w}$ в точке w_r , в которой альпинист оказался на шаге r . Поэтому, двигаясь противоположном направлении $-\frac{ac}{\partial w}(w_r)$, альпинист будет направляться в сторону долины. На каждом шаге альпинист может учитывать длину своей ноги пред следующим шагом. В терминологии метода градиентного спуска это называется *скоростью обучения* $\eta \geq 0$. Если она слишком мала, то альпинист будет двигаться медленно, а если слишком велика, то есть шанс проскочить мимо долины.

Вспомним, что сигмоида – гладкая функция, так что мы можем вычислить ее производную. Можно доказать, что производная сигмоиды

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

имеет вид

$$\frac{d\sigma(x)}{d(x)} = \sigma(x)(1 - \sigma(x))$$

Функция ReLU не дифференцируема в точке 0. Однако можно доопределить ее производную в этой точке, выбрав в качестве значения 0 или 1. Тогда производной блока линейной ректификации $y = \max(0, x)$ будет такая кусочно-постоянная функция:

$$\frac{dy}{dx} = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

Зная производную, мы можем оптимизировать сеть методом градиентного спуска. Keras пользуется для вычисления производной базовой библиотекой (TensorFlow или Theano), так что нам думать о реализации не нужно. Мы просто выбираем функцию активации, а Keras вычисляет ее производную.

Нейронная сеть представляет собой композицию нескольких функций с тысячами, а иногда и миллионами параметров. Каждый слой вычисляет функцию, ошибку которой необходимо минимизировать, чтобы улучшить верность на этапе обучения. При обсуждении обратного распространения мы поймем, что минимизация – не такое простое дело, как в нашем иллюстративном примере. Но все равно она основана на тех же идеях, что спуск альпиниста в долину.

В Keras реализован быстрый вариант градиентного спуска – **стохастический градиентный спуск (СГС)** и два более продвинутых метода оптимизации: **RMSprop** и **Adam**. В обоих участвует понятие импульса в дополнение к ускорению, используемому в СГС. В результате достигается более быстрая сходимость, правда, ценой увеличения объема вычислений. Полный перечень оптимизаторов, поддерживаемых Keras, приведен на странице <https://keras.io/optimizers/>. До сих пор мы по умолчанию выбирали оптимизатор СГС. Теперь попробуем два других. Для этого всего-то и надо что изменить пару строк:

```
from keras.optimizers import RMSprop, Adam
...
OPTIMIZER = RMSprop() # оптимизатор
```

Вот и все. Результаты тестирования показаны на рисунке ниже:

```
code — python keras_MINST_V4.py — 118x71
gulli-macbookpro:code gulli$ python keras_MINST_V4.py
Using TensorFlow backend.
60000 train samples
10000 test samples
```

Layer (type)	Output Shape	Param #	Connected to
dense_1 (Dense)	(None, 128)	100480	dense_input_1[0][0]
activation_1 (Activation)	(None, 128)	0	dense_1[0][0]
dropout_1 (Dropout)	(None, 128)	0	activation_1[0][0]
dense_2 (Dense)	(None, 128)	16512	dropout_1[0][0]
activation_2 (Activation)	(None, 128)	0	dense_2[0][0]
dropout_2 (Dropout)	(None, 128)	0	activation_2[0][0]
dense_3 (Dense)	(None, 10)	1290	dropout_2[0][0]
activation_3 (Activation)	(None, 10)	0	dense_3[0][0]

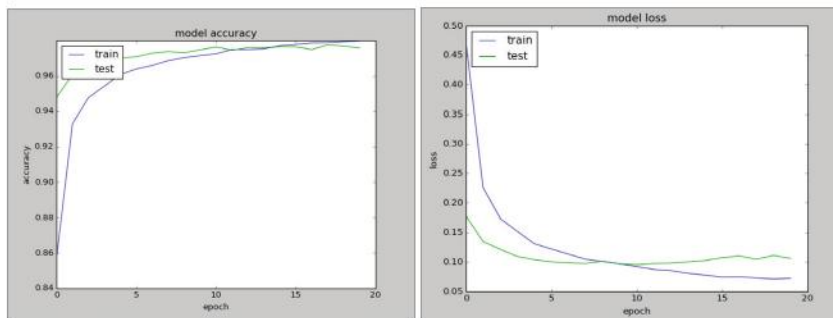
```

Total params: 118282

Train on 48000 samples, validate on 12000 samples
Epoch 1/20 [=====] - 2s - loss: 0.4714 - acc: 0.8571 - val_loss: 0.1780 - val_acc: 0.9478
Epoch 2/20 [=====] - 1s - loss: 0.2257 - acc: 0.9328 - val_loss: 0.1350 - val_acc: 0.9608
Epoch 3/20 [=====] - 1s - loss: 0.1737 - acc: 0.9477 - val_loss: 0.1217 - val_acc: 0.9643
Epoch 4/20 [=====] - 1s - loss: 0.1522 - acc: 0.9542 - val_loss: 0.1095 - val_acc: 0.9687
Epoch 5/20 [=====] - 1s - loss: 0.1312 - acc: 0.9609 - val_loss: 0.1039 - val_acc: 0.9703
Epoch 6/20 [=====] - 1s - loss: 0.1222 - acc: 0.9640 - val_loss: 0.1004 - val_acc: 0.9710
Epoch 7/20 [=====] - 1s - loss: 0.1134 - acc: 0.9660 - val_loss: 0.0985 - val_acc: 0.9730
Epoch 8/20 [=====] - 1s - loss: 0.1046 - acc: 0.9688 - val_loss: 0.0975 - val_acc: 0.9739
Epoch 9/20 [=====] - 1s - loss: 0.1009 - acc: 0.9705 - val_loss: 0.1014 - val_acc: 0.9732
Epoch 10/20 [=====] - 1s - loss: 0.0970 - acc: 0.9717 - val_loss: 0.0967 - val_acc: 0.9748
Epoch 11/20 [=====] - 1s - loss: 0.0922 - acc: 0.9726 - val_loss: 0.0956 - val_acc: 0.9764
Epoch 12/20 [=====] - 1s - loss: 0.0874 - acc: 0.9751 - val_loss: 0.0975 - val_acc: 0.9747
Epoch 13/20 [=====] - 1s - loss: 0.0853 - acc: 0.9750 - val_loss: 0.0980 - val_acc: 0.9760
Epoch 14/20 [=====] - 1s - loss: 0.0807 - acc: 0.9754 - val_loss: 0.1003 - val_acc: 0.9760
Epoch 15/20 [=====] - 1s - loss: 0.0777 - acc: 0.9771 - val_loss: 0.1025 - val_acc: 0.9766
Epoch 16/20 [=====] - 1s - loss: 0.0742 - acc: 0.9778 - val_loss: 0.1074 - val_acc: 0.9765
Epoch 17/20 [=====] - 1s - loss: 0.0746 - acc: 0.9786 - val_loss: 0.1104 - val_acc: 0.9750
Epoch 18/20 [=====] - 1s - loss: 0.0730 - acc: 0.9788 - val_loss: 0.1046 - val_acc: 0.9776
Epoch 19/20 [=====] - 1s - loss: 0.0711 - acc: 0.9793 - val_loss: 0.1112 - val_acc: 0.9769
Epoch 20/20 [=====] - 1s - loss: 0.0725 - acc: 0.9797 - val_loss: 0.1060 - val_acc: 0.9759
9888/10000 [=====] - ETA: 0s
Test score: 0.0902571567255
Test accuracy: 0.9784
['acc', 'loss', 'val_acc', 'val_loss']

```

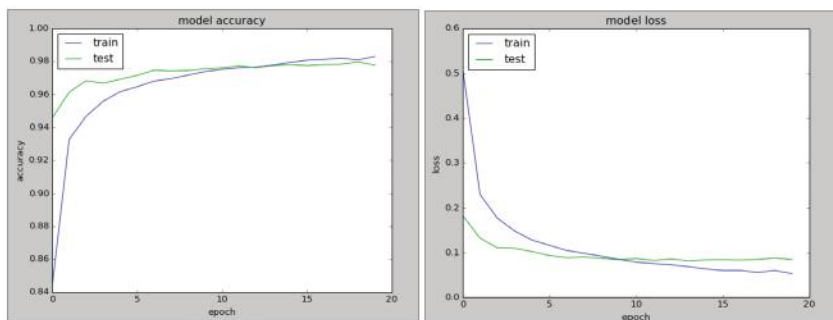
Как видим, RMSprop быстрее SGD, так что для достижения верности 97.97% на обучающем наборе, 97.59% на контрольном и 97.84% на тестовом понадобилось всего 20 итераций. Для полноты посмотрим, как изменяется верность и потеря при увеличении числа периодов:



Теперь попробуем оптимизатор Adam. Для этого нужно изменить одну строку:

```
OPTIMIZER = Adam() # оптимизатор
```

Как видно, Adam чуть лучше. За 20 итераций достигается верность 98.28% на обучающем наборе, 98.03% на контрольном и 97.93% на тестовом.



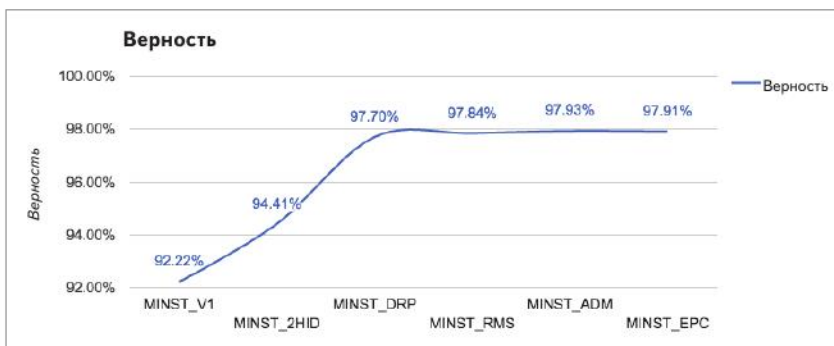
Это пятый вариант, а эталонная верность, напомним, была равна 92.36%.

Мы последовательно улучшаем модель, но с каждым разом прирост дается все труднее. Сейчас оптимизация производится с прореживанием 30%. Для полноты картины покажем верность на тестовых данных для других значений прореживания, когда в качестве оптимизатора используется Adam:



Увеличение числа периодов

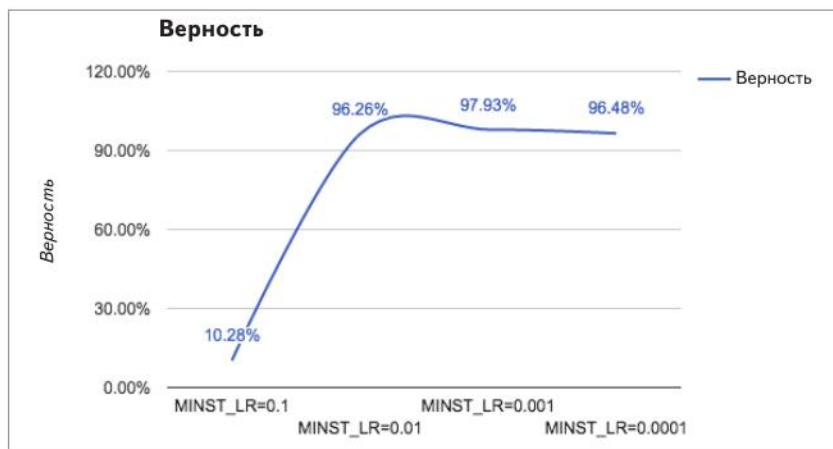
Предпримем еще одну попытку: увеличим число периодов обучения с 20 до 200. Увы, время вычислений при этом увеличивается в 10 раз, но никакого выигрыша мы не получаем. Эксперимент оказался неудачным, но мы узнали, что увеличение времени обучения необязательно приводит к улучшению. Успех обучения обусловлен скорее применением удачных методов, а не временем, потраченным на расчеты. Результаты шестого варианта представлены на следующем графике:



Управление скоростью обучения оптимизатора

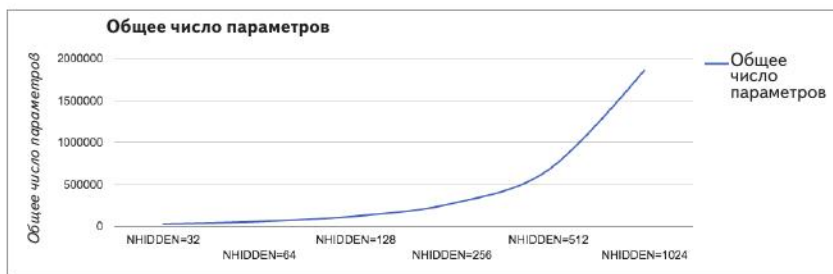
Мы можем еще попробовать изменить скорость обучения оптимизатора. Из следующего графика видно, что оптимальное значение близко к 0.001, а это как раз и есть значение, подразумеваемое

по умолчанию. Прекрасно! Adam работает, не требуя никакой настройки.



Увеличение числа нейронов в скрытых слоях

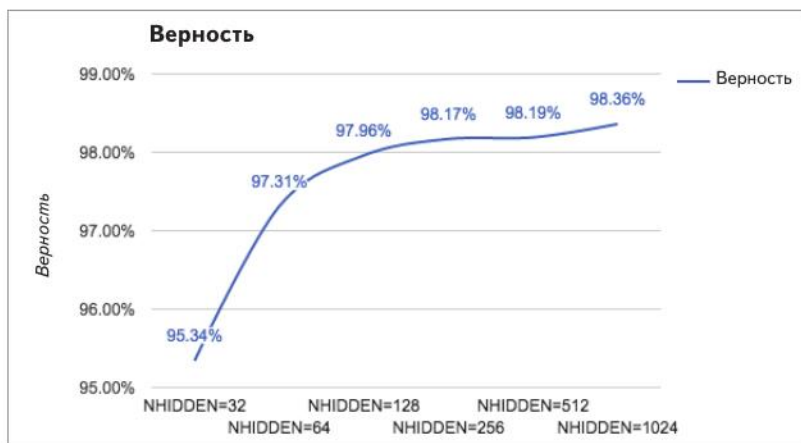
Еще одна возможность – изменить число нейронов во внутренних скрытых слоях. На следующем графике показаны результаты, получаемые при увеличении числа нейронов, т. е. сложности модели. Как видно, время вычислений быстро растет, поскольку приходится оптимизировать все больше параметров. Но достигаемый выигрыш при этом становится все меньше и меньше.



На следующем графике показано, как изменяется время одной итерации при росте числа скрытых нейронов.



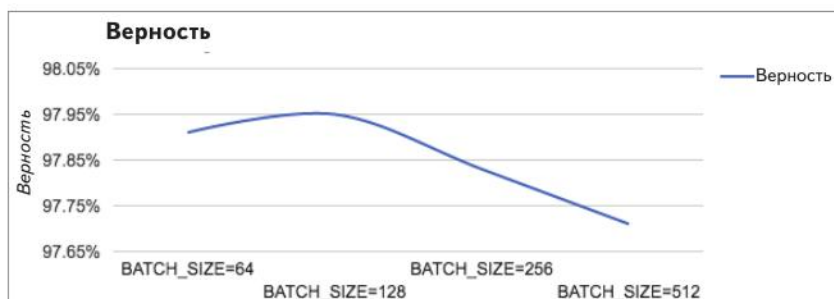
А на этом графике мы видим изменение верности при росте числа нейронов.



Увеличение размера пакета

Алгоритм градиентного спуска пытается минимизировать функцию стоимости одновременно на всех примерах из обучающего набора и для всех представленных в нем признаков. Алгоритм стохастического градиентного спуска обходится гораздо дешевле, потому что в нем рассматривается только `BATCH_SIZE`

примеров. Посмотрим, как зависит поведение модели от этого параметра. Как видим, оптимальная верность достигается, когда `BATCH_SIZE=128`:



Подведение итогов экспериментов по распознаванию рукописных цифр

Итак, опробовав пять вариантов, мы смогли улучшить выбранный показатель качества с 92.36% до 97.93%. Сначала мы определили простую однослойную сеть средствами Keras. Затем мы улучшили качество, добавив скрытые слои. Дальнейшего улучшения удалось добиться путем включения случайного прореживания сети и выбора подходящего оптимизатора. Полученные результаты сведены в таблицу ниже.

Модель/Верность	На обучающем наборе	На контрольном наборе	На тестовом наборе
Простая	92.36%	92.37%	92.22%
С двумя скрытыми слоями (128)	94.50%	94.63%	94.41%
С прореживанием (30%)	98.10%	97.73%	97.7% (200 периодов)
RMSprop	97.97%	97.59%	97.84% (20 периодов)
Adam	98.28%	98.03%	97.93% (20 периодов)

Однако следующие два эксперимента не принесли существенного улучшения. Увеличение числа внутренних нейронов влечет

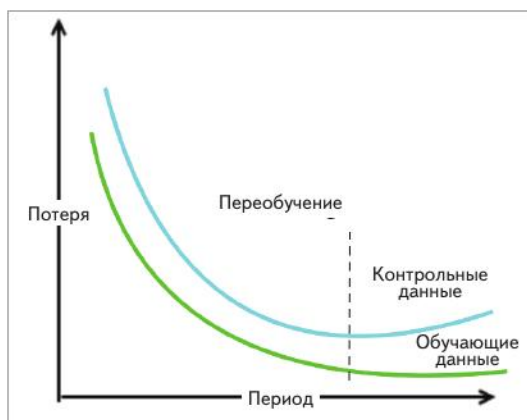
за собой усложнение модели и рост объема вычислений, но дает едва осязаемый выигрыш. То же самое относится к увеличению числа периодов обучения. Наш последний эксперимент состоял в изменении параметра оптимизатора `BATCH_SIZE`.

Применение регуляризации для предотвращения переобучения

Интуитивно представляется, что хорошая модель машинного обучения должна давать малую ошибку на обучающих данных. Математически это равносильно минимизации построенной моделью функции потерь на обучающих данных и выражается следующей формулой:

$$\min: \{loss(Training\ Data \mid Model)\}$$

Однако этого может оказаться недостаточно. Модель может стать избыточно сложной, стремясь уловить все связи, присущие обучающим данным. У такого увеличения сложности есть два нежелательных последствия. Во-первых, для выполнения сложной модели нужно много времени. Во-вторых, сложная модель может показывать великолепное качество на обучающих данных – поскольку она запомнила все присутствующие в них связи, но гораздо худшее на контрольных – поскольку модель не обобщается на новые данные. Таким образом, обучение свелось не к способности к обобщению, а к запоминанию. На следующем графике показана типичная функция потерь, которая убывает как на обучающем, так и на контрольном наборе. Однако в какой-то момент потеря на контрольных данных начинает расти из-за переобучения.



Эвристическое правило состоит в том, что если в процессе обучения мы наблюдаем возрастание потери на контрольном наборе после первоначального убывания, значит, модель слишком сложна и слишком близко подогнана к обучающим данным. В машинном обучении этот феномен называется *переобучением*.

Для решения проблемы переобучения необходимо как-то выразить сложность модели и управлять ею. И как это сделать? Но ведь модель по существу – всего лишь вектор весов. Поэтому ее сложность можно представить в виде количества ненулевых весов. Иными словами, если две модели $M1$ и $M2$ дают примерно одинаковое качество в терминах функции потерь, то следует предпочесть ту, в которой количество ненулевых весов меньше. Для управления важностью выбора более простой модели можно завести гиперпараметр $\lambda \geq 0$ и минимизировать следующую функцию:

$$\min: \{\text{loss}(\text{Training Data} \mid \text{Model}) + \lambda * \text{complexity}(\text{Model})\}$$

В машинном обучении применяются три способа регуляризации.

- **Регуляризация по норме L1** (известная также под названием **lasso**): сложность модели выражается в виде суммы модулей весов.
- **Регуляризация по норме L2 (гребневая)**: сложность модели выражается в виде суммы квадратов весов.
- **Эластичная сеть**: для выражения сложности модели применяется комбинация двух предыдущих способов.

Отметим, что идею регуляризации можно применить к весам, к модели и к активации.

Таким образом, регуляризация может способствовать повышению качества сети, особенно если налицо очевидное переобучение. Оставляем эксперименты в качестве упражнения для интересующихся читателей.

Keras поддерживает все три формы регуляризации. Добавить регуляризацию просто, ниже показано задание L2-регуляризатора ядра (вектора весов W):

```
from keras import regularizers model.add(Dense(64, input_dim=64,
    kernel_regularizer=regularizers.l2(0.01)))
```

Полное описание параметров имеется на странице <https://keras.io/regularizers/>.

Настройка гиперпараметров

Описанные выше эксперименты помогли составить представление о том, какие имеются способы настройки нейросети. Однако то, что годится для данного примера, может не подойти в других случаях. Для каждой сети имеется много допускающих оптимизацию параметров (количество скрытых нейронов, `BATCH_SIZE`, количество периодов и ряд других, зависящих от сложности сети).

Настройкой гиперпараметров называется процесс поиска оптимального сочетания этих параметров, при котором достигается минимум функции стоимости. Если имеется n параметров, то можно считать, что они определяют n -мерное пространство, а наша цель – найти в этом пространстве точку, в которой функция стоимости принимает минимальное значение. Для достижения этой цели можно, например, создать координатную сетку в пространстве и для каждого ее узла вычислить значение функции стоимости. Иными словами, выполнить полный перебор всех комбинаций параметров.

Предсказание выхода

Обученную сеть естественно использовать для предсказания. В Keras это очень просто:

```
# вычислить предсказание
predictions = model.predict(X)
```

Для заданного входного вектора можно вычислить несколько значений:

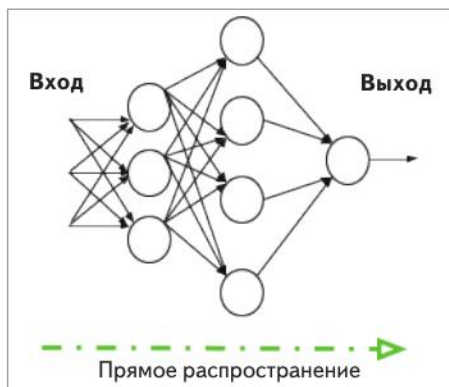
- `model.evaluate()`: вычисляет потерю;
- `model.predict_classes()`: вычисляет категориальные выходы;
- `model.predict_proba()`: вычисляет вероятности классов.

Практическое изложение алгоритма обратного распространения

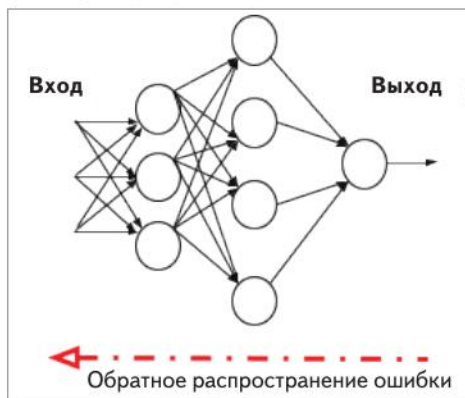
Многослойный перцептрон обучается на данных с помощью процесса, называемого обратным распространением. Его можно описать как постоянное исправление ошибок по мере их обнаружения. Посмотрим, как он работает.

Напомним, что с любой нейронной сетью ассоциирован набор весов, которые служат для вычисления выходных значений по входным. Кроме того, в нейронной сети может быть несколько скрытых слоев.

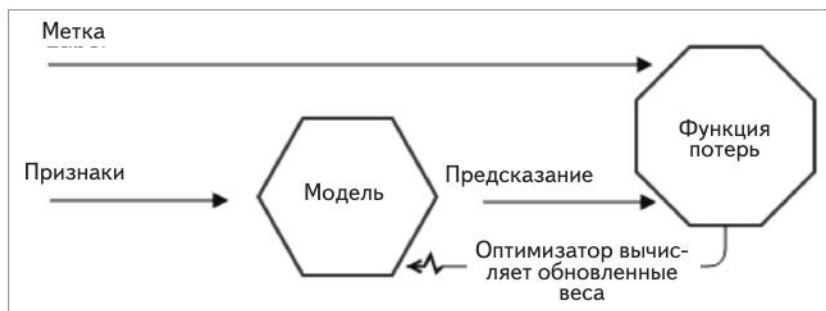
Первоначально всем весам присваиваются случайные значения. Затем сеть активируется для каждого входного значения из обучающего набора: значения распространяются в *прямом* направлении от входного слоя через скрытые к выходному, который и выдает предсказание:



Поскольку истинное наблюдаемое значение для обучающего набора известно, мы можем вычислить ошибку предсказания. Идея заключается в том, чтобы выполнить обратное распространение ошибки и с помощью подходящего алгоритма оптимизации, например градиентного спуска, подправить веса нейронной сети с целью уменьшения ошибки:



Процесс прямого распространения сигнала от входного слоя к выходному и обратного распространения ошибки повторяется несколько раз, пока ошибка не станет ниже заранее заданного порогового значения. Весь процесс изображен на следующем рисунке:



Признаки – это входные данные, а метки служат для управления процессом обучения. Модель обновляется таким образом, что функция потерь на каждом шаге минимизируется. В нейронной сети важен не столько отклик отдельно взятого нейрона, сколько весь набор корректируемых весов в каждом слое. Поэтому сеть постепенно изменяет внутренние веса, так чтобы увеличить количество правильно предсказанных меток. Конечно, для минимизации расхождения в процессе обучения принципиально важно, чтобы были выбраны подходящие признаки, а данные были размечены правильно.

В направлении глубокого обучения

Экспериментируя с распознаванием рукописных цифр, мы пришли к выводу, что чем ближе к 99 % достигнута верность, тем труднее ее улучшить. Если мы хотим продвинуться дальше, то нужна какая-то новая идея. Чего нам не хватает?

Важнейшее наблюдение состоит в том, что до сих пор мы не учитывали информацию о расположении изображения в пространстве. Так, приведенный ниже код преобразует растровое изображение, представляющее все цифры, в плоский вектор, в котором вся пространственная информация потеряна:

```
# X_train содержит 60000 изображений размера 28x28 --> преобразуем
в массив 60000 x 784
```



```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
```

Однако же наш мозг работает иначе. Напомним, что в основе зрения лежит несколько областей коры, каждая из которых распознает все более крупные структуры, сохраняя при этом информацию о локализации в пространстве. Сначала мы видим отдельные пиксели, затем распознаем среди них простые геометрические формы, а затем все более сложные элементы: предметы, лица, тела людей, животных и т. д.

В главе 3 мы познакомимся со специальным типом сети глубокого обучения, **сверточной нейронной сетью (СНС)**, разработанной так, чтобы можно было одновременно сохранить пространственную локализацию в изображении и обучаться все более высоким уровням абстракции: ближний к входным данным слой обучается распознаванию простых образов, а чем слой дальше, тем более сложные образы он распознает. Но прежде чем переходить к обсуждению СНС, нам предстоит рассмотреть некоторые особенности архитектуры Keras и дополнительные концепции машинного обучения.

Резюме

В этой главе мы познакомились с основами нейронных сетей: что такое перцептрон и многослойный перцептрон, как определяют нейронные сети в Keras, как постепенно улучшить качество сети по сравнению с эталоном и как настраивать гиперпараметры. Кроме того, мы теперь знаем о некоторых полезных функциях активации (сигмоиде и блоке линейной активации ReLU), о том, как обучать сеть с помощью алгоритма обратного распространения, основанного на методе градиентного спуска или стохастического градиентного спуска, или на более сложных методах типа Adam и RMSprop.

В следующей главе мы увидим, как установить Keras в облаке AWS, Microsoft Azure, Google Cloud или на свою локальную машину. Мы также дадим обзор API Keras.

Глава 2

Установка Keras и описание API

В предыдущей главе мы обсудили основные принципы нейронных сетей и привели несколько примеров сетей, умеющих распознавать рукописные цифры из набора данных MNIST.

В этой главе мы обсудим установку Keras, Theano и TensorFlow. Мы увидим, как настроить рабочее окружение и за короткое время перейти от смутной идеи к работоспособной нейросети. Затем мы поговорим о том, как произвести установку в инфраструктуре, основанной на контейнерах Docker, и в облаках Google GCP, Amazon AWS и Microsoft Azure. Попутно мы представим обзор API Keras и опишем некоторые полезные операции, в т. ч. загрузку и сохранение архитектуры и весов нейронной сети, раннюю остановку, сохранение истории, контрольные точки и взаимодействие с TensorBoard и Quiver.

Установка Keras

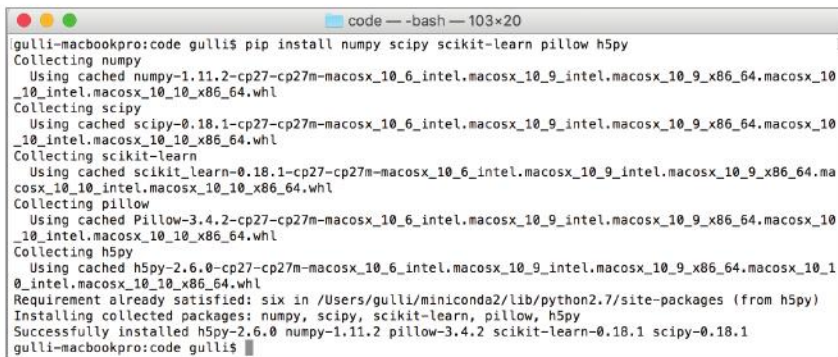
В следующих разделах мы покажем, как установить Keras на различные платформы.

Шаг 1 – установка зависимостей

Первым делом установим пакет `numpy`, поддерживающий работу с многомерными массивами и матрицами, а также с математическими функциями. Затем установим библиотеку для научных расчетов `scipy`. После этого имеет смысл установить пакет `scikit-learn`, считающийся в машинном обучении на Python универсальным средством – ножом швейцарской армии. Кроме того, полез-

но будет установить библиотеку обработки изображений `pillow` и библиотеку сериализации данных `h5py`, которой Keras пользуется для сохранения моделей. Для установки всего необходимого достаточно одной команды.

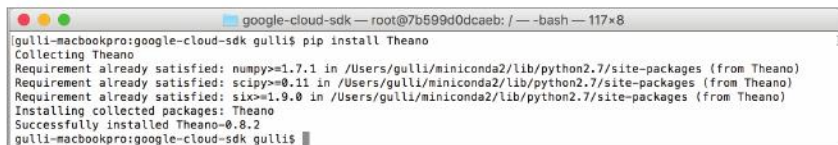
Можно вместо этого установить дистрибутив Anaconda Python, который уже содержит `numpy`, `scipy`, `scikit-learn`, `h5py`, `pillow` и множество других библиотек, используемых в научных расчетах (см. S. Ioffe, C. Szegedy «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift», [arXiv.org/abs/1502.03167](https://arxiv.org/abs/1502.03167), 2015). Список пакетов, входящих в состав Anaconda Python, см. на странице <https://docs.continuum.io/anaconda/pkg-docs>. На снимке экрана ниже показана процедура установки необходимых пакетов.



```
code --bash-- 103x20
gulli-macbookpro:code gulli$ pip install numpy scipy scikit-learn pillow h5py
Collecting numpy
  Using cached numpy-1.11.2-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting scipy
  Using cached scipy-0.18.1-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting scikit-learn
  Using cached scikit_learn-0.18.1-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting pillow
  Using cached Pillow-3.4.2-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting h5py
  Using cached h5py-2.6.0-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Requirement already satisfied: six in /Users/gulli/miniconda2/lib/python2.7/site-packages (from h5py)
Installing collected packages: numpy, scipy, scikit-learn, pillow, h5py
Successfully installed h5py-2.6.0 numpy-1.11.2 pillow-3.4.2 scikit-learn-0.18.1 scipy-0.18.1
gulli-macbookpro:code gulli$
```

Шаг 2 – установка Theano

Установить Theano поможет `pip`:



```
google-cloud-sdk --root@7b599d0dcaeb: / --bash-- 117x8
gulli-macbookpro:google-cloud-sdk gulli$ pip install Theano
Collecting Theano
Requirement already satisfied: numpy>=1.7.1 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from Theano)
Requirement already satisfied: scipy>=0.11 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from Theano)
Requirement already satisfied: six>=1.9.0 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from Theano)
Installing collected packages: Theano
Successfully installed Theano-0.8.2
gulli-macbookpro:google-cloud-sdk gulli$
```

Шаг 3 – установка TensorFlow

Теперь можно установить TensorFlow, следуя инструкциям на сайте TensorFlow по адресу https://www.tensorflow.org/versions/r0.11/get_started/os_setup.html#pip-installation. И на этот раз для установки нужного пакета используется `pip`, как показано на рисунке ниже.

```

gulli-macbookpro:code gulli$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-0.11.0-py2-none-any.whl
gulli-macbookpro:code gulli$ sudo pip install --upgrade $TF_BINARY_URL --ignore-installed
Collecting tensorflow==0.11.0 from https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-0.11.0-py2-none-any.whl
  Using cached https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-0.11.0-py2-none-any.whl
Collecting mock==2.0.0 (from tensorflow==0.11.0)
  Using cached mock-2.0.0-py2.py3-none-any.whl
Collecting protobuf==3.0.0 (from tensorflow==0.11.0)
  Using cached protobuf-3.0.0-py2.py3-none-any.whl
Collecting numpy==1.11.0 (from tensorflow==0.11.0)
  Using cached numpy-1.11.2-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting wheel (from tensorflow==0.11.0)
  Using cached wheel-0.29.0-py2.py3-none-any.whl
Collecting six==1.10.0 (from tensorflow==0.11.0)
  Using cached six-1.10.0-py2.py3-none-any.whl
Collecting funcsigs>=1; python_version < "3.3" (from mock==2.0.0->tensorflow==0.11.0)
  Using cached funcsigs-1.0.2-py2.py3-none-any.whl
Collecting pbr==0.11 (from mock==2.0.0->tensorflow==0.11.0)
  Using cached pbr-1.10.0-py2.py3-none-any.whl
Collecting setuptools (from protobuf==3.0.0->tensorflow==0.11.0)
  Using cached setuptools-28.8.0-py2.py3-none-any.whl
Installing collected packages: six, funcsigs, pbr, mock, setuptools, protobuf, numpy, wheel, tensorflow
Successfully installed funcsigs-1.0.2 mock-2.0.0 numpy-1.11.2 pbr-1.10.0 protobuf-3.0.0 setuptools-28.8.0 six-1.10.0 tensorflow-0.11.0 wheel-0.29.0
gulli-macbookpro:code gulli$

```

Шаг 4 – установка Keras

Теперь можно установить Keras:

```

gulli-macbookpro:code gulli$ pip install keras
Collecting keras
Requirement already satisfied: theano in /Users/gulli/miniconda2/lib/python2.7/site-packages (from keras)
Requirement already satisfied: pyyaml in /Users/gulli/miniconda2/lib/python2.7/site-packages (from keras)
Requirement already satisfied: six in /Users/gulli/miniconda2/lib/python2.7/site-packages (from keras)
Requirement already satisfied: numpy>=1.9.1 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from theano->keras)
Requirement already satisfied: scipy>=0.14 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from theano->keras)
Installing collected packages: keras
Successfully installed keras-1.1.1
gulli-macbookpro:code gulli$

```

Шаг 5 – проверка работоспособности Theano, TensorFlow и Keras

Проверим созданное окружение. Сначала попробуем определить сигмоиду в Theano. Как видим, это очень просто: нужно просто записать математическую формулу и применить ее ко всем элементам матрицы. Запустите оболочку Python Shell и введите показанный ниже код:

```

>>> import theano
>>> import theano.tensor as T
>>> x = T.dmatrix('x')
>>> s = 1 / (1 + T.exp(-x))
>>> logistic = theano.function([x], s)
>>> logistic([[0, 1], [-1, -2]])
array([[ 0.5, 0.73105858],
       [ 0.26894142, 0.11920292]])
>>>

```

Итак, Theano работает. Для проверки TensorFlow просто импортируем набор данных MNIST, как показано на рисунке ниже. В главе 1 мы уже видели несколько примеров нейросетей, созданных в Keras:

```
code — python — 103x17
gulli-macbookpro:code gulli$ python
Python 2.7.12 |Continuum Analytics, Inc.| (default, Jul 2 2016, 17:43:17)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting MNIST_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
>>>
```

Настройка Keras

Конфигурационный файл Keras очень прост. Загрузите его в редактор vi. Вот список параметров:

Параметр	Значения
image_dim_ordering	Определяет представление изображений: <code>tf</code> – принятое в TensorFlow, <code>th</code> – в Theano
epsilon	Значение константы <code>epsilon</code> в вычислениях
floatx	Может принимать значение <code>float32</code> или <code>float64</code>
backend	Может принимать значение <code>tensorflow</code> или <code>theano</code>

Значение `th` параметра `image_dim_ordering` определяет интуитивно неочевидный порядок измерений изображения: (глубина, ширина, высота) вместо (ширина, высота, глубина), как в случае `tf`. Ниже приведены значения параметров на моей машине:

```
code — vi ~/.keras/keras.json — 103x8
{
  "image_dim_ordering": "th",
  "epsilon": 1e-07,
  "floatx": "float32",
  "backend": "tensorflow"
}

~/.keras/keras.json [noeol] 6L, 113C
```




При установке версии TensorFlow с поддержкой GPU Keras автоматически будет использовать GPU, если в качестве базовой библиотеки выбрана TensorFlow.

Установка Keras в контейнер Docker

Один из самых простых способов начать работу с TensorFlow и Keras – установить их в контейнер Docker. Удобно воспользоваться готовым образом Docker для глубокого обучения, созданным сообществом; он содержит все популярные библиотеки ГО (TensorFlow, Theano, Torch, Caffe и т. д.). Необходимые файлы имеются в репозитории на GitHub по адресу <https://github.com/saiprashanth/dl-docker>. В предположении, что Docker уже установлен и работает (см. <https://www.docker.com/products/overview>), установка не вызывает никаких трудностей:

```
gulli-macbookpro:dl-docker gulli$ git clone https://github.com/saiprashanth/dl-docker
.git
Cloning into 'dl-docker'...
remote: Counting objects: 89, done.
remote: Total 89 (delta 0), reused 0 (delta 0), pack-reused 89
Unpacking objects: 100% (89/89), done.
gulli-macbookpro:dl-docker gulli$
```

На следующем снимке экрана показано, как после получения образа из Git строится контейнер Docker:

```
gulli-macbookpro:dl-docker gulli$ cd dl-docker/
gulli-macbookpro:dl-docker gulli$ docker build -t floydhub/dl-docker:cpu -f Dockerfile
.cpu .
Sending build context to Docker daemon 284.2 kB
Step 1 : FROM ubuntu:14.04
----> 3f755ca42730
Step 2 : MAINTAINER Sai Soundararaj <saip@outlook.com>
----> Using cache
----> af02b42bde1c
Step 3 : ARG THEANO_VERSION=rel-0.8.2
----> Using cache
----> c8d03ba70cff
Step 4 : ARG TENSORFLOW_VERSION=0.8.0
----> Using cache
----> de0ed51e5732
Step 5 : ARG TENSORFLOW_ARCH=cpu
----> Using cache
----> 270d4bfbccaa
Step 6 : ARG KERAS_VERSION=1.0.3
----> Using cache
----> 61219a95474f
Step 7 : ARG LASAGNE_VERSION=v0.1
----> Using cache
----> 585e125f1e76
Step 8 : ARG TORCH_VERSION=latest
----> Using cache
----> fa5c4246c2ec
Step 9 : ARG CAFFE_VERSION=master
----> Using cache
----> 989ad8491f04
Step 10 : RUN apt-get update && apt-get install -y
```

bc

build-

А здесь мы видим, как этот контейнер запускается:

```
gulli-macbookpro:dl-docker gulli$ docker run -it -p 8888:8888 -p 6006:6006 floydhub/dl
-docker:cpu bash
root@780e0d54bfc0:~# ls
caffe iTorch run_jupyter.sh torch
root@780e0d54bfc0:~#
```

Из контейнера можно активировать поддержку сервера Jupyter Notebooks (см. <http://jupyter.org/>):

```
root@780e0d54bfc0:~# sh run_jupyter.sh
[I 10:51:17.489 NotebookApp] Copying /root/.ipython/kernels -> /root/.local/share/jupyter/kernels
[I 10:51:17.498 NotebookApp] Writing notebook server cookie secret to /root/.local/share/jupyter/runtime/notebook_cookie_secret
[W 10:51:17.520 NotebookApp] WARNING: The notebook server is listening on all IP addresses and not using encryption. This is not recommended.
[I 10:51:17.536 NotebookApp] Serving notebooks from local directory: /root
[I 10:51:17.536 NotebookApp] 0 active kernels
[I 10:51:17.537 NotebookApp] The Jupyter Notebook is running at: http://[all ip addresses on your system]:8888/?token=503b59dc969d43f588638e3bd153dd1525837ff46d7b1eb9
[I 10:51:17.537 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 10:51:17.539 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=503b59dc969d43f588638e3bd153dd1525837ff46d7b1eb9
[I 10:51:32.547 NotebookApp] 302 GET / (172.17.0.1) 0.60ms
[I 10:51:32.553 NotebookApp] 302 GET /tree? (172.17.0.1) 0.86ms
[I 10:51:40.207 NotebookApp] 302 GET /?token=503b59dc969d43f588638e3bd153dd1525837ff46d7b1eb9 (172.17.0.1) 0.36ms
```

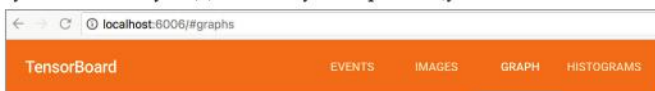
Система будет работать прямо на локальной машине:



Есть также возможность получить доступ к TensorBoard (см. https://www.tensorflow.org/how_tos/summaries_and_tensorboard/), для чего нужно выполнить показанную ниже команду:

```
root@7b599d0dcaeb:~# tensorboard --logdir .
```

В результате вы увидите такую страницу:



Установка Keras в Google Cloud ML

Установить Keras в облако Google Cloud очень просто. Сначала нужно установить командный интерфейс (файл можно скачать по адресу <https://cloud.google.com/sdk/>), к платформе Google Cloud; после этого мы сможем использовать CloudML, управляемую службу, которая позволяет без труда строить модели машинного обучения средствами TensorFlow. Прежде чем переходить к Keras, воспользуемся Google Cloud в сочетании с TensorFlow, чтобы обучить модель на наборе данных MNIST, который имеется на GitHub. Код будет находиться на локальной машине, а обучение происходить в облаке.

```
gulli-macbookpro:google-cloud-sdk gullis$ git clone https://github.com/GoogleCloudPlatform/cloudml-samples/
Cloning into 'cloudml-samples'...
remote: Counting objects: 118, done.
remote: Total 118 (delta 0), reused 0 (delta 0), pack-reused 118
Receiving objects: 100% (118/118), 84.40 KiB | 0 bytes/s, done.
Resolving deltas: 100% (49/49), done.
gulli-macbookpro:google-cloud-sdk gullis$
```

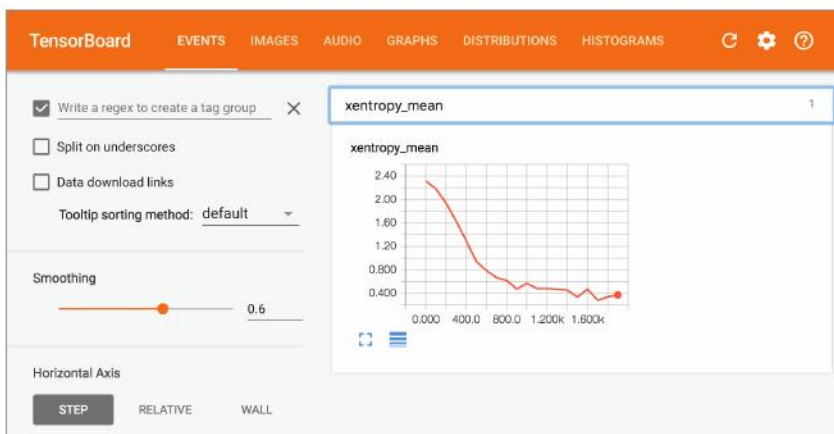
На следующем снимке экрана показан протокол обучения:

```
gulli-macbookpro:codeBook gullis$ cd cloudml-samples/mnist/trainable/
gulli-macbookpro:trainable gullis$ ls
trainer
gulli-macbookpro:trainable gullis$ gcloud beta ml local train --package-path=trainer --module-name=trainer.task
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /var/folders/dx/ssb40l192sz_sl5b6tf35mj_r00cn0L/T/tmpCARkfj/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /var/folders/dx/ssb40l192sz_sl5b6tf35mj_r00cn0L/T/tmpCARkfj/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /var/folders/dx/ssb40l192sz_sl5b6tf35mj_r00cn0L/T/tmpCARkfj/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /var/folders/dx/ssb40l192sz_sl5b6tf35mj_r00cn0L/T/tmpCARkfj/t10k-labels-idx1-ubyte.gz
Step 0: loss = 2.32 (0.018 sec)
Step 100: loss = 2.19 (0.002 sec)
Step 200: loss = 1.94 (0.002 sec)
Step 300: loss = 1.64 (0.002 sec)
Step 400: loss = 1.30 (0.002 sec)
Step 500: loss = 0.95 (0.002 sec)
Step 600: loss = 0.80 (0.002 sec)
Step 700: loss = 0.67 (0.002 sec)
Step 800: loss = 0.62 (0.002 sec)
Step 900: loss = 0.48 (0.002 sec)
Training Data Eval:
  Num examples: 55000  Num correct: 47295  Precision @ 1: 0.8599
Validation Data Eval:
  Num examples: 5000  Num correct: 4347  Precision @ 1: 0.8694
Test Data Eval:
  Num examples: 10000  Num correct: 8649  Precision @ 1: 0.8649
Step 1000: loss = 0.58 (0.018 sec)
Step 1100: loss = 0.49 (0.115 sec)
Step 1200: loss = 0.49 (0.002 sec)
Step 1300: loss = 0.48 (0.002 sec)
Step 1400: loss = 0.46 (0.002 sec)
Step 1500: loss = 0.34 (0.002 sec)
Step 1600: loss = 0.49 (0.002 sec)
Step 1700: loss = 0.29 (0.002 sec)
Step 1800: loss = 0.35 (0.002 sec)
Step 1900: loss = 0.39 (0.002 sec)
Training Data Eval:
  Num examples: 55000  Num correct: 49243  Precision @ 1: 0.8953
Validation Data Eval:
  Num examples: 5000  Num correct: 4519  Precision @ 1: 0.9038
Test Data Eval:
  Num examples: 10000  Num correct: 9000  Precision @ 1: 0.9000
gulli-macbookpro:trainable gullis$
```

Чтобы наблюдать за последовательным уменьшением перекрестной энтропии, можно воспользоваться TensorBoard:

```
guli-macbookpro:trainable gullis tensorboard --logdir=data/ --port=8080
Starting TensorBoard 29 on port 8080
```

График перекрестной энтропии показан на рисунке ниже.



Чтобы воспользоваться Keras поверх TensorFlow, нужно просто скачать исходный код Keras с сайта PyPI (<https://pypi.python.org/pypi/Keras/1.2.0> или более позднюю версию), а затем использовать Keras как пакетное решение для CloudML:

```
guli-macbookpro:trainable gullis gcloud beta ml local train --package-path=trainer --package-path=../../CloudML/fchol
let-keras-1.2.0-g12d868f.tar.gz --module-name=trainer.task2
Using TensorFlow backend.
(0, 'input_1', (None, 224, 224, 3))
(1, 'block1_conv1', (None, 224, 224, 64))
(2, 'block1_conv2', (None, 224, 224, 64))
(3, 'block1_pool', (None, 112, 112, 64))
(4, 'block2_conv1', (None, 112, 112, 128))
(5, 'block2_conv2', (None, 112, 112, 128))
(6, 'block2_pool', (None, 56, 56, 128))
(7, 'block3_conv1', (None, 56, 56, 256))
(8, 'block3_conv2', (None, 56, 56, 256))
(9, 'block3_conv3', (None, 56, 56, 256))
(10, 'block3_pool', (None, 28, 28, 256))
(11, 'block4_conv1', (None, 28, 28, 512))
(12, 'block4_conv2', (None, 28, 28, 512))
(13, 'block4_conv3', (None, 28, 28, 512))
(14, 'block4_pool', (None, 14, 14, 512))
(15, 'block5_conv1', (None, 14, 14, 512))
(16, 'block5_conv2', (None, 14, 14, 512))
(17, 'block5_conv3', (None, 14, 14, 512))
(18, 'block5_pool', (None, 7, 7, 512))
(19, 'flatten', (None, 25088))
(20, 'fc1', (None, 4096))
(21, 'fc2', (None, 4096))
(22, 'predictions', (None, 1000))
guli-macbookpro:trainable gullis ls
data trainer
guli-macbookpro:trainable gullis █
```

Ниже в качестве примера приведен скрипт `trainer.task2.py`:

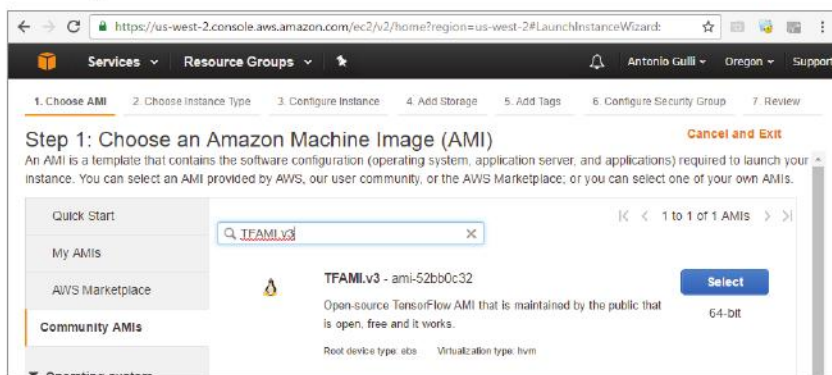
```
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.preprocessing import image
```

```
from keras.applications.vgg16 import preprocess_input
import numpy as np

# готовая, уже обученная модель глубокого обучения VGG-16
base_model = VGG16(weights='imagenet', include_top=True)
for i, layer in enumerate(base_model.layers):
    print(i, layer.name, layer.output_shape)
```

Установка Keras в Amazon AWS

Установить TensorFlow и Keras в облако Amazon очень просто. Можно даже использовать готовый образ машины `TFAMI.v3`, свободный и бесплатный (см. <https://github.com/ritchieng/tensorflow-aws-ami>):



Этот образ устанавливается меньше чем за пять минут и поддерживает TensorFlow, Keras, OpenAI Gym и все необходимые зависимости. По состоянию на январь 2017 году поддерживались следующие версии:

- TensorFlow 0.12
- Keras 1.1.0
- TensorLayer 1.2.7
- CUDA 8.0
- CuDNN 5.1
- Python 2.7
- Ubuntu 16.04

Кроме того, образ `TFAMI.v3` работает на вычислительных инстансах P2 (см. <https://aws.amazon.com/ec2/instance-types/#p2>), как показано на следующем снимке экрана:

Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. Learn more about instance types and how they can meet your computing needs.

Filter by: GPU compute Current generation Show/Hide Columns

Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GB memory, EBS only)

	Family	Type	vCPUs (1)	Memory (GiB)	Instance Storage (GB) (1)	EBS-Optimized Available (1)	Network Performance (1)
☐	GPU compute	p2.xlarge	4	61	EBS only	Yes	High
☐	GPU compute	p2.6xlarge	32	486	EBS only	Yes	10 GigaBit
☐	GPU compute	p2.16xlarge	64	732	EBS only	Yes	20 GigaBit

Ниже перечислены некоторые характеристики инстансов P2:

- процессоры Intel Xeon E5-2686v4 (Broadwell);
- графические процессоры NVIDIA K80 с 2496 параллельными ядрами и 12 ГБ памяти;
- поддержка прямого обмена данными между GPU;
- расширенные сетевые возможности (см. https://aws.amazon.com/ec2/faqs/#What_networking_capabilities_are_included_in_this_feature), агрегированная пропускная способность сети 20 Гб/с.

Образ `tfami.v3` работает также на вычислительных инстансах G2 (см. <https://aws.amazon.com/ec2/instance-types/#g2>), обладающих, в частности, такими свойствами:

- процессоры Intel Xeon E5-2670 (Sandy Bridge);
- графические процессоры NVIDIA с 1536 ядрами CUDA и 4 ГБ видеопамати.

Установка Keras в Microsoft Azure

Один из способов установить Keras в облако Azure – сначала установить поддержку Docker, а затем скачать контейнерную версию TensorFlow плюс Keras. В сети можно найти подробные инструкции по установке Keras и TensorFlow в сочетании с Docker, но по существу это то, что мы уже видели в предыдущем разделе (см. https://blogs.msdn.microsoft.com/uk_faculty_connection/2016/09/26/tensorflow-on-docker-with-microsoft-azure/).

Если вы используете Theano в качестве базовой библиотеки, то для запуска Keras достаточно всего лишь загрузить готовый пакет, имеющийся в коллекции Cortana Intelligence Gallery (см. <https://gallery.cortanaintelligence.com/Experiment/Theano-Keras-1>). В следующем примере показано, как можно импортировать Theano и Keras в Azure ML непосредственно в виде ZIP-файла и использовать их в модуле Execute Python Script. Этим примером мы обязаны Хай Ниню (см. <https://goo.gl/VLR25o>), по существу здесь Keras выполняется внутри метода `azureml_main()`:

```
# Скрипт ДОЛЖЕН содержать функцию azureml_main, являющуюся
# точкой входа в этот модуль.

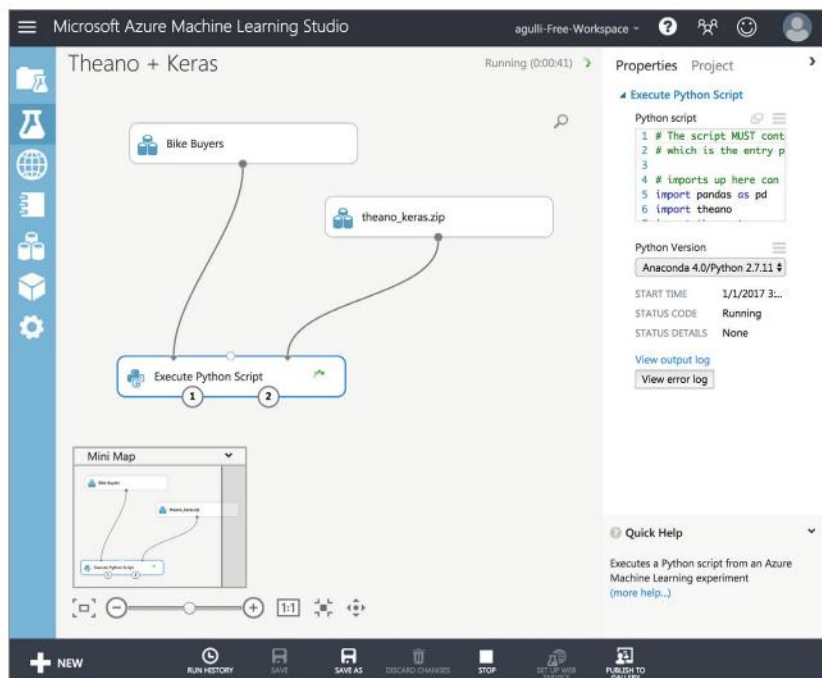
import pandas as pd
import theano
import theano.tensor as T
from theano import function
from keras.models import Sequential
from keras.layers import Dense, Activation
import numpy as np

# Эта функция принимает два необязательных аргумента:
# Param<dataframe1>: a pandas.DataFrame
# Param<dataframe2>: a pandas.DataFrame
def azureml_main(dataframe1 = None, dataframe2 = None):
    # Здесь должна быть логика программы
    # print('Input pandas.DataFrame #1:\n\n{0}'.format(dataframe1))

    # Если zip-файл подключен к стороннему входному порту,
    # то он распаковывается в каталог ".Script Bundle-", который
    # добавляется в sys.path. Поэтому, если zip-файл содержит
    # Python-файл mymodule.py, то его можно импортировать так:
    # import mymodule
    model = Sequential()
    model.add(Dense(1, input_dim=784, activation="relu"))
    model.compile(optimizer='rmsprop', loss='binary_crossentropy',
                  metrics=['accuracy'])
    data = np.random.random((1000,784))
    labels = np.random.randint(2, size=(1000,1))
    model.fit(data, labels, nb_epoch=10, batch_size=32)
    model.evaluate(data, labels)

    return dataframe1
```

На рисунке ниже показан пример использования Microsoft Azure ML для выполнения Theano и Keras:



Keras API

Keras обладает модульной минималистской и легко расширяемой архитектурой. Франсуа Шолле, автор Keras, пишет:

При разработке библиотеки основное внимание уделялось поддержке быстрых экспериментов. Сокращение пути от идеи к результату – ключ к успешной исследовательской работе.

С помощью Keras определяются высокоуровневые нейронные сети, работающие поверх библиотеки TensorFlow (см. <https://github.com/tensorflow/tensorflow>) или Theano (см. <https://github.com/Theano/Theano>). Дадим некоторые пояснения.

- **Модульность.** Модель представляет собой последовательность или граф автономных модулей, которые соединяются между собой, как детали конструктора LEGO, образуя нейросеть. В библиотеке имеется множество готовых модулей, реализующих различные типы слоев, функций стоимости,

оптимизаторов, схем инициализации, функций активации и методов регуляризации.

- **Минимализм.** Библиотека написана на Python, все модули короткие и самодокументированные.
- **Расширяемость.** В библиотеку можно добавлять новую функциональность. Этой теме посвящена глава 7.

Введение в архитектуру Keras

В этом разделе мы рассмотрим самые важные компоненты Keras, применяемые для определения нейронных сетей. Сначала определим, что такое тензор, затем обсудим различные способы соединения готовых модулей и в заключение опишем наиболее употребительные модули.

Что такое тензор?

Keras пользуется библиотекой Theano или TensorFlow для эффективных вычислений с тензорами. Но что такое тензор? Да просто многомерный массив или матрица. Обе библиотеки умеют эффективно выполнять символические вычисления с тензорами, а это основной строительный блок для создания нейронных сетей.

Соединение моделей Keras

В Keras есть два способа соединения моделей:

- последовательная композиция;
- функциональная композиция.

Рассмотрим их подробнее.

Последовательная композиция

В этом случае готовые модели соединяются в линейный конвейер слоев, напоминающий стек или очередь. В главе 1 мы встречались с такими последовательными конвейерами, например:

```
model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(nb_classes))
```

```
model.add(Activation('softmax'))
model.summary()
```

Функциональная композиция

Функциональный API позволяет определять более сложные модели, например, ациклические графы, модели с разделяемыми слоями или с несколькими выходами. Примеры будут приведены в главе 7.

Обзор готовых слоев нейронных сетей

Keras предоставляет несколько готовых слоев. Мы рассмотрим наиболее употребительные и отметим, в каких главах эти слои используются.

Обычный плотный слой

Плотная модель – это полносвязный слой нейронной сети. Примеры мы уже видели в главе 1. Ниже приведен прототип модели со всеми параметрами:

```
keras.layers.core.Dense(units, activation=None, use_bias=True,
    kernel_initializer='glorot_uniform', bias_initializer='zeros',
    kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, bias_constraint=None)
```

Рекуррентные нейронные сети – простая, LSTM и GRU

Рекуррентные нейронные сети – это класс нейронных сетей, в которых используется последовательная природа входных данных. Входными данными может быть текст, речь, временные ряды и вообще любой объект, в котором появление элемента последовательности зависит от предшествующих элементов. В главе 6 мы будем обсуждать рекуррентные сети трех видов: простые, LSTM и GRU. Ниже приведены прототипы моделей со всеми параметрами:

```
keras.layers.recurrent.Recurrent(return_sequences=False,
    go_backwards=False, stateful=False, unroll=False, implementation=0)
keras.layers.recurrent.SimpleRNN(units, activation='tanh', use_bias=True,
    kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
    bias_initializer='zeros', kernel_regularizer=None,
    recurrent_regularizer=None, bias_regularizer=None,
    activity_regularizer=None, kernel_constraint=None,
    recurrent_constraint=None, bias_constraint=None, dropout=0.0,
```



```

recurrent_dropout=0.0)

keras.layers.recurrent.GRU(units, activation='tanh',
    recurrent_activation='hard_sigmoid', use_bias=True,
    kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
    bias_initializer='zeros', kernel_regularizer=None,
    recurrent_regularizer=None, bias_regularizer=None,
    activity_regularizer=None, kernel_constraint=None,
    recurrent_constraint=None, bias_constraint=None, dropout=0.0,
    recurrent_dropout=0.0)

keras.layers.recurrent.LSTM(units, activation='tanh',
    recurrent_activation='hard_sigmoid', use_bias=True,
    kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
    bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,
    recurrent_regularizer=None, bias_regularizer=None,
    activity_regularizer=None, kernel_constraint=None,
    recurrent_constraint=None, bias_constraint=None, dropout=0.0,
    recurrent_dropout=0.0)

```

Сверточные и пулинговые слои

Сверточные сети – класс нейронных сетей, в которых сверточные и пулинговые операции используются для постепенного обучения довольно сложных моделей с повышающимся уровнем абстракции. Такой способ обучения напоминает модель человеческого зрения, сложившуюся в результате миллионов лет эволюции. Сверточные сети обсуждаются в главе 3. Ниже приведены прототипы моделей со всеми параметрами:

```

keras.layers.convolutional.Conv1D(filters, kernel_size, strides=1,
    padding='valid', dilation_rate=1, activation=None, use_bias=True,
    kernel_initializer='glorot_uniform', bias_initializer='zeros',
    kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, bias_constraint=None)

keras.layers.convolutional.Conv2D(filters, kernel_size, strides=(1, 1),
    padding='valid', data_format=None, dilation_rate=(1, 1), activation=None,
    use_bias=True, kernel_initializer='glorot_uniform',
    bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None,
    activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

keras.layers.pooling.MaxPooling1D(pool_size=2, strides=None,
    padding='valid')

keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=None,
    padding='valid', data_format=None)

```


Регуляризация

Цель регуляризации – предотвратить переобучение. Мы уже видели примеры использования в главе 1. В слоях различных типов имеются параметры регуляризации. Ниже приведен список параметров регуляризации, часто используемых в плотных и сверточных модулях.

- `kernel_regularizer`: функция регуляризации, применяемая к матрице весов;
- `bias_regularizer`: функция регуляризации, применяемая к вектору смещений;
- `activity_regularizer`: функция регуляризации, применяемая к выходу слоя (его функции активации).

Кроме того, для регуляризации можно использовать прореживание и зачастую это дает весомый эффект:

```
keras.layers.core.Dropout(rate, noise_shape=None, seed=None)
```

где:

- `rate` – вещественное число от 0 до 1, показывающее, сколько входных блоков отбрасывать;
- `noise_shape` – одномерный целочисленный тензор, задающий форму двоичной маски прореживания, которая умножается на входной сигнал;
- `seed` – целое число, служащее для инициализации генератора случайных чисел.

Пакетная нормировка

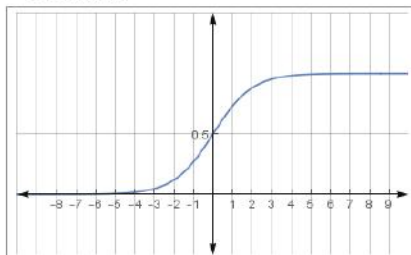
Пакетная нормировка (см. <https://www.colwiz.com/cite-in-google-docs/cid=f20f9683aaf69ce>) позволяет ускорить обучение и в общем случае получить большую верность. Примеры будут рассмотрены в главе 4 при обсуждении порождающих состязательных сетей. Ниже приведен прототип с параметрами:

```
keras.layers.normalization.BatchNormalization(axis=-1, momentum=0.99,
epsilon=0.001, center=True, scale=True, beta_initializer='zeros',
gamma_initializer='ones', moving_mean_initializer='zeros',
moving_variance_initializer='ones', beta_regularizer=None,
gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)
```

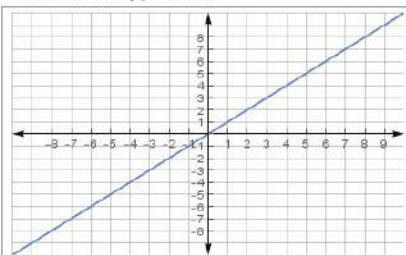
Обзор готовых функций активации

К числу готовых функций активации относятся, в частности, сигмоида, линейная функция, гиперболический тангенс и блок линейной ректификации (ReLU). Несколько примеров мы уже видели в главе 1, а в последующих главах встретим и другие. На рисунке ниже приведены графики вышеперечисленных функций.

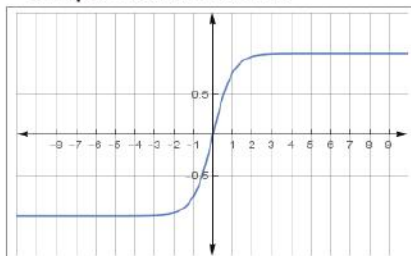
Сигмоида



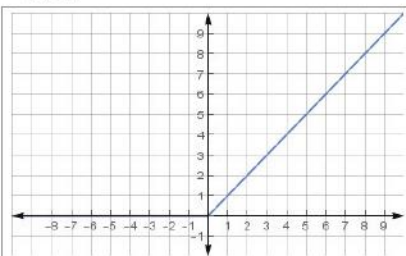
Линейная функция



Гиперболический тангенс



ReLU



Обзор функций потерь

Функции потерь (или целевые функции) (см. <https://keras.io/losses/>) можно отнести к четырем категориям:

- Верность, используемая в задачах классификации. Таких функций четыре: `binary_accuracy` (средняя верность по всем предсказаниям в задачах бинарной классификации), `categorical_accuracy` (средняя верность по всем предсказаниям в задачах многоклассовой классификации), `sparse_categorical_accuracy` (используется, когда метки разреженные) и `top_k_categorical_accuracy` (успехом считается случай, когда истинный целевой класс находится среди первых `top_k` предсказаний).
- Ошибка, измеряющая различие между предсказанными и фактическими значениями. Варианты таковы: `mse` (средне-

квадратическая ошибка), `rmse` (квадратный корень из среднеквадратической ошибки), `mae` (средняя абсолютная ошибка), `mape` (средняя ошибка в процентах), `msle` (средняя квадратично-логарифмическая ошибка).

- Кусочно-линейная функция потерь, которая обычно применяется для обучения классификаторов. Существует два варианта: *кусочно-линейная*, определяемая как $\max(1 - y_{true} * y_{pred}, 0)$ и *квадратичная кусочно-линейная*, равная квадрату кусочно-линейной.
- Классовая потеря используется для вычисления перекрестной энтропии в задачах классификации. Существует несколько вариантов, включая бинарную перекрестную энтропию (см. https://en.wikipedia.org/wiki/Cross_entropy) и категориальную перекрестную энтропию.

Несколько примеров целевых функций мы видели в главе 1, а дополнительные будут приведены в следующих главах.

Обзор показателей качества

Функции показателей качества (см. <https://keras.io/metrics/>) аналогичны целевым функциям. Единственное различие между ними состоит в том, что результаты вычисления показателей не используются на этапе обучения модели. Примеры мы видели в главе 1, а дополнительные будут приведены ниже.

Обзор оптимизаторов

К числу оптимизаторов относятся SGD, RMSprop и Adam. Несколько примеров мы видели в главе 1, а дополнительные (Adagrad и Adadelta, см. <https://keras.io/optimizers/>) будут приведены в следующих главах.

Некоторые полезные операции

Ниже перечислены некоторые вспомогательные операции, включенные в Keras API. Их цель – упростить создание сетей, процесс обучения и сохранение промежуточных результатов.

Сохранение и загрузка весов и архитектуры модели

Для сохранения и загрузки архитектуры модели служат следующие функции:

```
# сохранить в формате JSON
json_string = model.to_json()
# сохранить в формате YAML
yaml_string = model.to_yaml()
# восстановить модель из JSON-файла
from keras.models import model_from_json
model = model_from_json(json_string)
# восстановить модель из YAML-файла
model = model_from_yaml(yaml_string)
```

Для сохранения и загрузки параметров модели служат следующие функции:

```
from keras.models import load_model
# создать HDF5-файл 'my_model.h5'
model.save('my_model.h5')
# удалить существующую модель
del model
# вернуть откомпилированную модель, идентичную исходной
model = load_model('my_model.h5')
```

Обратные вызовы для управления процессом обучения

Процесс обучения можно остановить, когда показатель качества перестает улучшаться. Для этого служит следующая функция обратного вызова:

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0,
                             patience=0, verbose=0, mode='auto')
```

Историю потерь можно сохранить, определив такие обратные вызовы:

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
        model = Sequential()
        model.add(Dense(10, input_dim=784, init='uniform'))
        model.add(Activation('softmax'))
        model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
        history = LossHistory()
        model.fit(X_train, Y_train, batch_size=128, nb_epoch=20,
                verbose=0, callbacks=[history])
        print history.losses
```


Контрольные точки

Контрольная точка – это процесс периодического сохранения мгновенного снимка состояния приложения, так чтобы приложение можно было перезапустить с последнего сохраненного состояния в случае отказа. Это бывает полезно при обучении глубоких моделей, которое часто занимает длительное время. Состоянием глубокой модели обучения в любой момент времени являются веса, вычисленные к этому моменту. Keras сохраняет веса в формате HDF5 (см. <https://www.hdfgroup.org/>) и предоставляет средства сохранения контрольной точки с помощью API обратных вызовов.

Приведем несколько ситуаций, когда контрольная точка полезна.

- Если требуется перезапустить программу с последней контрольной точки после того, как спотовый инстанс AWS Spot (см. <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/how-spot-instances-work.html>) или вытесняемая виртуальная машина Google (см. <https://cloud.google.com/compute/docs/instances/preemptible>) неожиданно остановилась.
- Если требуется остановить обучение, например, для того чтобы проверить модель на тестовых данных, а затем продолжить с последней контрольной точки.
- Если требуется сохранять бета-версию (с наилучшим показателем качества, например, потерей на контрольном наборе) модели, обучаемой на протяжении нескольких периодов.

В первом и втором случае можно сохранять контрольную точку после каждого периода, для чего достаточно стандартного использования обратного вызова `ModelCheckpoint`. Приведенный ниже код показывает, как сохранить контрольную точку в процессе обучения глубокой модели в Keras:

```
from __future__ import division, print_function
from keras.callbacks import ModelCheckpoint
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout
from keras.utils import np_utils
import numpy as np
import os
```

```
BATCH_SIZE = 128
NUM_EPOCHS = 20
```



```

MODEL_DIR = "/tmp"

(Xtrain, ytrain), (Xtest, ytest) = mnist.load_data()
Xtrain = Xtrain.reshape(60000, 784).astype("float32") / 255
Xtest = Xtest.reshape(10000, 784).astype("float32") / 255
Ytrain = np_utils.to_categorical(ytrain, 10)
Ytest = np_utils.to_categorical(ytest, 10)
print(Xtrain.shape, Xtest.shape, Ytrain.shape, Ytest.shape)

model = Sequential()
model.add(Dense(512, input_shape=(784,), activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(512, activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(10, activation="softmax"))

model.compile(optimizer="rmsprop", loss="categorical_crossentropy",
              metrics=["accuracy"])

# сохранить модель
checkpoint = ModelCheckpoint(filepath=os.path.join(MODEL_DIR,
            "model-{epoch:02d}.h5"))
model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE, nb_epoch=NUM_EPOCHS,
        validation_split=0.1, callbacks=[checkpoint])

```

В третьем случае нужно следить за показателем качества, например верностью или потерей, и сохранять контрольную точку, только если текущий показатель лучше, чем у предыдущей сохраненной версии. В Keras имеется дополнительный параметр объекта контрольной точки, `save_best_only`, которому следует присвоить значение `true`, если указанная функциональность необходима.

Использование TensorBoard совместно с Keras

Keras предлагает обратный вызов для сохранения показателей качества на обучающем и тестовом наборе, а также гистограмм активации для различных слоев модели:

```

keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0,
                             write_graph=True, write_images=False)

```

Сохраненные данные можно затем визуализировать с помощью программы TensorBoard, запущенной из командной строки:

```

tensorboard --logdir=/full_path_to_your_logs

```

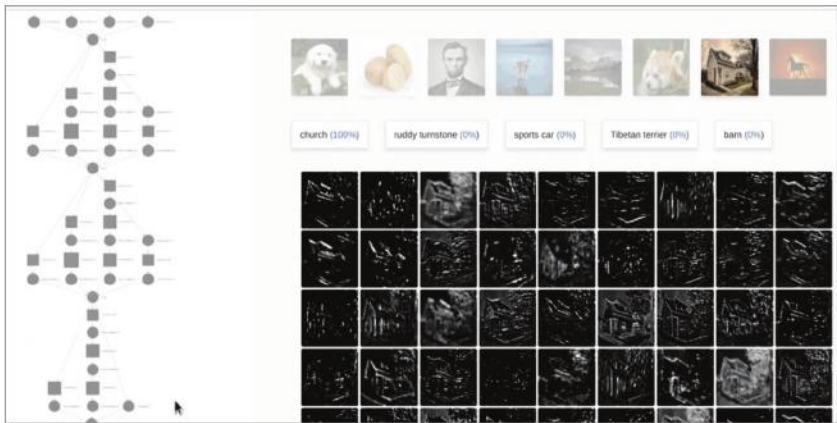
Использование Quiver совместно с Keras

В главе 3 мы будем обсуждать сверточные сети, специально предназначенные для обработки изображений. А сейчас дадим краткий обзор приложения Quiver (см. <https://github.com/jakebian/quiver>), полезного для интерактивной визуализации признаков сверточных сетей. После простой установки для его использования достаточно одной строки:

```
pip install quiver_engine

from quiver_engine import server server.launch(model)
```

Эта команда запускает сервер визуализации на порту localhost:5000. Quiver позволяет визуально исследовать нейронную сеть, как показано в следующем примере:



Резюме

В этой главе мы обсудили, как установить Theano, TensorFlow и Keras:

- на локальную машину;
- в контейнер Docker;
- в облако Google GCP, Amazon AWS и Microsoft Azure.

Помимо этого мы рассмотрели несколько модулей Keras и такие распространенные операции, как загрузка и сохранение архитек-

тур и весов нейронных сетей, ранняя остановка, сохранение истории, контрольные точки, взаимодействие с TensorBoard и Quiver.

В следующей главе мы познакомимся со сверточными сетями, фундаментальной новацией в глубоком обучении, которая успешно применяется в таких разных предметных областях, как обработка текста, видео и речи, а не только для обработки изображений, как первоначально задумывалось.

Глава 3

Глубокое обучение с применением сверточных сетей

В предыдущих главах мы обсуждали плотные сети, где каждый нейрон связан со всеми нейронами соседних слоев. Мы применили плотные сети к классификации рукописных цифр из набора данных MNIST. В этом контексте каждому пикселю входного изображения сопоставляется отдельный нейрон, так что всего получается 784 (28×28 пикселей) входных нейронов. Однако при такой стратегии игнорируется пространственная структура и связи внутри изображения. Так, следующий фрагмент кода преобразует растровые изображения всех цифр в плоский вектор, что приводит к потере информации о пространственной локализации:

```
# X_train содержит 60000 изображений размера 28x28 --> преобразуем в  
# массив 60000 x 784  
X_train = X_train.reshape(60000, 784)  
X_test = X_test.reshape(10000, 784)
```

Сверточные сети задействуют пространственную информацию и потому хорошо подходят для классификации изображений. В них используется специальная архитектура, инспирированная данными, полученными в физиологических экспериментах со зрительной корой. Как уже отмечалось, наша зрительная система состоит из нескольких уровней коры, причем каждый последующий распознает все более крупные структуры в поступающей информации. Сначала мы видим отдельные пиксели, затем различаем в них простые геометрические формы, а затем – все более сложные элементы: предметы, лица, тела людей и животных и т. п.

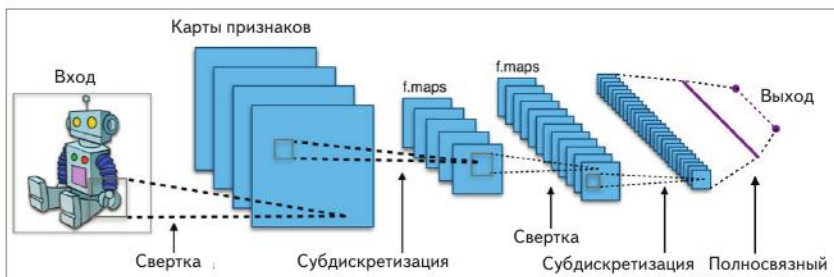
Сверточные сети завораживают. На протяжении короткого времени они стали *революционной* технологией, перевернувшей представления о возможном в таких областях, как обработка текста, видео и речи, а не только изображений.

В этой главе мы рассмотрим следующие темы:

- глубокие сверточные нейронные сети;
- классификация изображений.

Глубокая сверточная нейронная сеть

Глубокая сверточная нейронная сеть (ГСНС) состоит из большого числа слоев. Обычно в ней чередуются слои двух типов – сверточные и пулинговые. Глубина фильтра возрастает слева направо. На последних этапах обычно используется один или несколько полносвязных слоев.



В основе сверточных сетей лежат три идеи:

- локальное рецептивное поле;
- разделяемые веса;
- пулинг.

Рассмотрим их поочередно.

Локальные рецептивные поля

Для сохранения пространственной информации удобно представлять каждое изображение матрицей пикселей. Тогда для кодирования локальной структуры можно просто соединить подматрицу соседних входных нейронов с одним скрытым нейроном следующего слоя, который и представляет одно локальное рецептивное поле. Эта операция, называемая сверткой, и дала название типу сетей.

Используя перекрывающиеся подматрицы, мы сможем закодировать больше информации. Предположим, к примеру, что размер каждой подматрицы равен 5×5 и что эти подматрицы используются для обработки изображений размера 28×28 из набора MNIST. Тогда мы сумеем создать 23×23 нейронов локального рецептивного поля в следующем скрытом слое. Действительно, подматрицу можно сдвинуть только на 23 позиции, а затем она уйдет за границу изображения. В Keras размер одной подматрицы, называемый **длиной шага** (stride length), является гиперпараметром, который можно настроить в процессе конструирования сетей.

Определим карту признаков при переходе от одного слоя к другому. Конечно, можно завести несколько карт признаков, которые обучаются независимо. Например, для обработки изображений из набора MNIST можно начать с 28×28 входных нейронов, а затем организовать k карт признаков размера 23×23 (с шагом 5×5) в следующем скрытом слое.

Разделяемые веса и смещения

Допустим, мы хотим отойти от строкового представления пикселей и получить возможность обнаруживать один и тот же признак независимо от того, в каком месте изображения он находится. На ум сразу приходит мысль воспользоваться общим набором весов и смещений для всех нейронов в скрытых слоях. Тогда каждый слой обучится распознавать множество позиционно-независимых признаков в изображении.

Если входное изображение имеет размер (256, 256) с тремя каналами в порядке *tf* (TensorFlow), то его можно представить тензором (256, 256, 3). Отметим, что в режиме *th* (Theano) индекс канала глубины равен 1, а в режиме *tf* (TensorFlow) – 3.

В Keras, чтобы добавить сверточный слой с 32 выходами и фильтром размера 3×3 , мы пишем:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(256, 256, 3)))
```

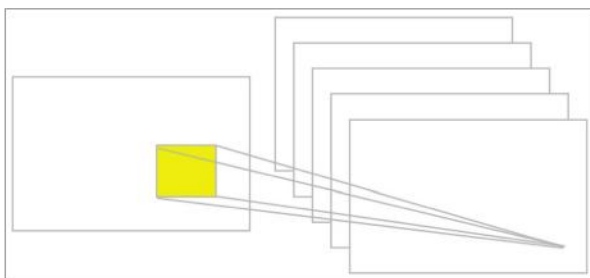
То же самое можно записать и по-другому:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=3, input_shape=(256, 256, 3)))
```

Это значит, что свертка с ядром 3×3 применяется к изображению размера 256×256 с тремя входными каналами (входными

фильтрами), и в результате получается 32 выходных канала (выходных фильтра).

Пример свертки приведен на следующем рисунке.



Пулинговые слои

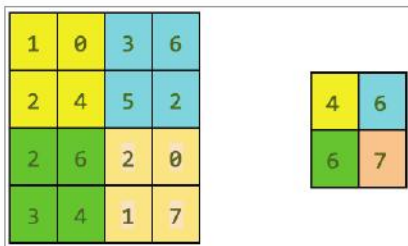
Допустим, мы хотим агрегировать выход карты признаков. И в этом случае можно воспользоваться пространственной смежностью выходов, порожденных из одной карты признаков, и агрегировать значения подматрицы в одно выходное значение, которое дает сводное описание *смысла*, ассоциированного с данной физической областью.

Мак-пулинг

Часто применяется *max-пулинг*, когда просто берется максимальный отклик в области. В Keras, чтобы определить слой *max-пулинга* размера 2×2 , мы пишем

```
model.add(MaxPooling2D(pool_size = (2, 2)))
```

На следующем рисунке приведен пример *max-пулинга*:



Усредненный пулинг

Другой вариант – *усредненный пулинг*, когда берется среднее арифметическое откликов в некоторой области.

В Keras реализовано еще много пулинговых слоев, их полный перечень приведен на странице <https://keras.io/layers/pooling/>. Все операции пулинга сводятся к тому или иному способу агрегирования значений в заданной области.

Промежуточные итоги

Мы изложили основные понятия сверточных сетей. В СНС операции свертки и пулинга применяются в одном направлении (время) для звуковых и текстовых данных, в двух направлениях (ширина и высота) для изображений и в трех направлениях (ширина, высота, время) для видео. В случае изображений перемещение фильтра по входной матрице порождает карту, дающую отклики фильтра для каждого положения в пространстве. Иначе говоря, сверточная сеть состоит из нескольких собранных в стопку фильтров, которые обучаются распознавать конкретные визуальные признаки независимо от того, в каком месте изображения они находятся. В начальных слоях сети признаки простые, а затем становятся все более сложными.

Пример ГСНС – LeNet

Ян Лекун (Yann le Cun) предложил (см. статью Y. LeCun, Y. Bengio «Convolutional Networks for Images, Speech, and Time-Series», Brain Theory Neural Networks, vol. 3361, 1995) семейство сверточных сетей, получившее название LeNet, обученных распознаванию рукописных цифр из набора MNIST и устойчивых к простым геометрическим преобразованиям и искажению. Основная идея состоит в наличии чередующихся слоев, реализующих операции свертки и max-пулинга. Операции свертки основаны на тщательно подобранных локальных рецептивных полях с весами, разделяемыми между несколькими картами признаков. Последние слои полносвязные – как в традиционном МСП со скрытыми слоями и функцией активации softmax в выходном слое.

Код LeNet в Keras

Для определения сети LeNet используется модуль двумерной сверточной сети:

```
keras.layers.convolutional.Conv2D(filters, kernel_size, padding='valid')
```

Здесь `filters` – число сверточных ядер (например, размерность выхода), `kernel_size` – одно целое число или кортеж (либо список) из двух целых чисел, задающих ширину и высоту двумерного окна свертки (если указано одно число, то ширина и высота одинаковы), а `padding='same'` означает, что используется дополнение. Существует два режима: `padding='valid'` означает, что свертка вычисляется только там, где фильтр целиком помещается в области входа, поэтому выход оказывается меньше входа, а `padding='same'` – что размер выхода такой же (`same`), как размер входа, для чего входная область дополняется нулями по краям.

Мы также используем модуль `MaxPooling2D`:

```
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=(2, 2))
```

Здесь `pool_size=(2, 2)` – кортеж из двух целых чисел, определяющих коэффициенты уменьшения изображения по вертикали и по горизонтали. Таким образом, `(2, 2)` означает, что изображение уменьшается вдвое в обоих направлениях. Наконец, параметр `strides=(2, 2)` определяет шаг обработки.

Теперь перейдем к коду. Сначала импортируется ряд модулей:

```
from keras import backend as K
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras.datasets import mnist
from keras.utils import np_utils
from keras.optimizers import SGD, RMSprop, Adam
import numpy as np
import matplotlib.pyplot as plt
```

Затем определяется сеть `LeNet`:

```
#define the ConvNet
class LeNet:
    @staticmethod
    def build(input_shape, classes):
        model = Sequential()
        # CONV => RELU => POOL
```

Первый слой – сверточный с функцией активации `ReLU`, за ним следует слой `max-пулинга`. В нашей сети будет 20 сверточных фильтров размера 5×5 . Размер выхода такой же, как размер входа – 28×28 . Поскольку первым элементом конвейера является модуль

Convolution2D, необходимо определить его форму, `input_shape`. Операция max-пулинга реализует скользящее окно, перемещающееся по слою, и вычисляет максимальное значение в области. Шаг перемещения по горизонтали и по вертикали равен 2.

```
model.add(Convolution2D(20, kernel_size=5, padding="same",
    input_shape=input_shape))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# CONV => RELU => POOL
```

Затем добавляется второй сверточный слой с функцией активации ReLU, а за ним еще один слой max-пулинга. Но теперь мы увеличиваем число сверточных фильтров с 20 до 50. Увеличение числа фильтров в более глубоких слоях – стандартный прием глубокого обучения.

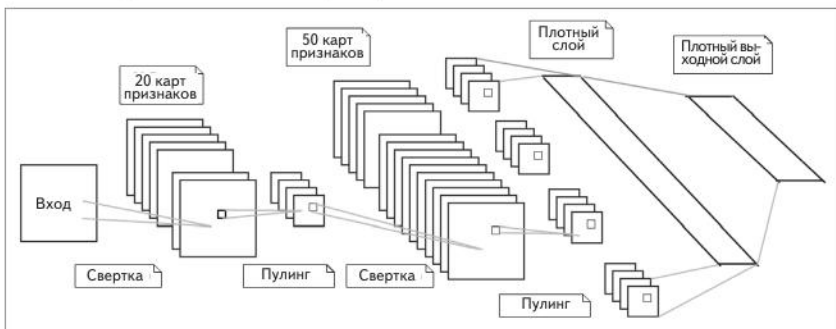
```
model.add(Conv2D(50, kernel_size=5, border_mode="same"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
```

Затем идет довольно стандартный слой упрощения, плотный слой с 500 нейронами и softmax-классификатор с 10 классами:

```
# слой Flatten => RELU
model.add(Flatten())
model.add(Dense(500))
model.add(Activation("relu"))

# softmax-классификатор
model.add(Dense(classes))
model.add(Activation("softmax"))
return model
```

Ну, вот и всё. Мы только что определили свою первую сеть глубокого обучения! Посмотрим, как она выглядит.



Дальше нужен код обучения сети, но он очень похож на тот, что мы видели в главе 1. На этот раз показан также код печати потери:

```
# сеть и ее обучение
NB_EPOCH = 20
BATCH_SIZE = 128
VERBOSE = 1
OPTIMIZER = Adam()
VALIDATION_SPLIT=0.2
IMG_ROWS, IMG_COLS = 28, 28 # размеры входного изображения
NB_CLASSES = 10 # число выходов = число цифр
INPUT_SHAPE = (1, IMG_ROWS, IMG_COLS)

# данные: перетасованы и разбиты на обучающий и тестовый набор
(X_train, y_train), (X_test, y_test) = mnist.load_data()
k.set_image_dim_ordering("th")

# рассматриваем как числа с плавающей точкой и нормируем
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

# нам нужна форма 60K x [1 x 28 x 28], подаваемая на вход сверточной сети
X_train = X_train[:, np.newaxis, :, :]
X_test = X_test[:, np.newaxis, :, :]
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

# преобразуем векторы классов в бинарные матрицы классов
y_train = np_utils.to_categorical(y_train, NB_CLASSES)
y_test = np_utils.to_categorical(y_test, NB_CLASSES)

# инициализировать оптимизатор и модель
model = LeNet.build(input_shape=INPUT_SHAPE, classes=NB_CLASSES)
model.compile(loss="categorical_crossentropy", optimizer=OPTIMIZER,
              metrics=["accuracy"])
history = model.fit(X_train, y_train,
                   batch_size=BATCH_SIZE, epochs=NB_EPOCH,
                   verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
score = model.evaluate(X_test, y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])

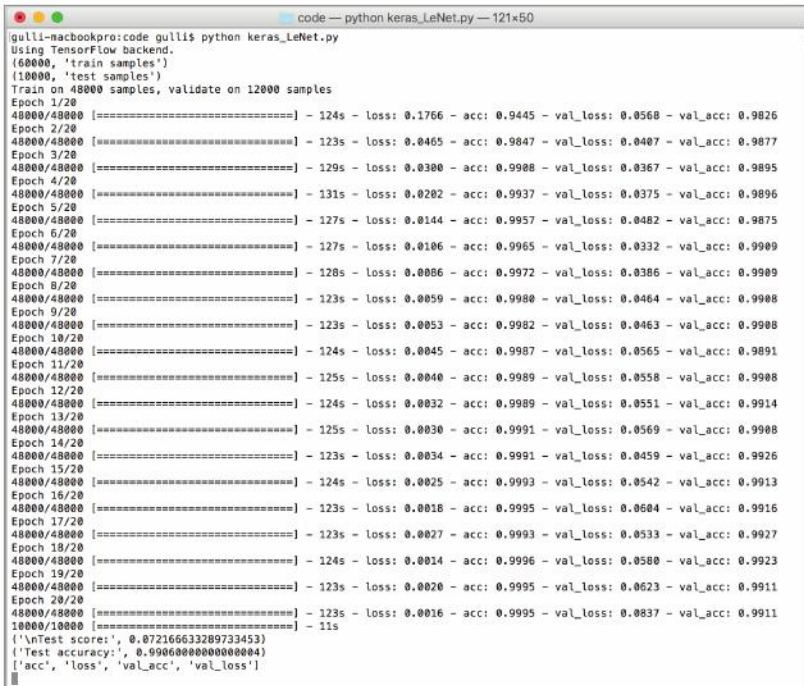
# перечислить все данные в истории
print(history.history.keys())

# построить график изменения верности
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
```

```
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

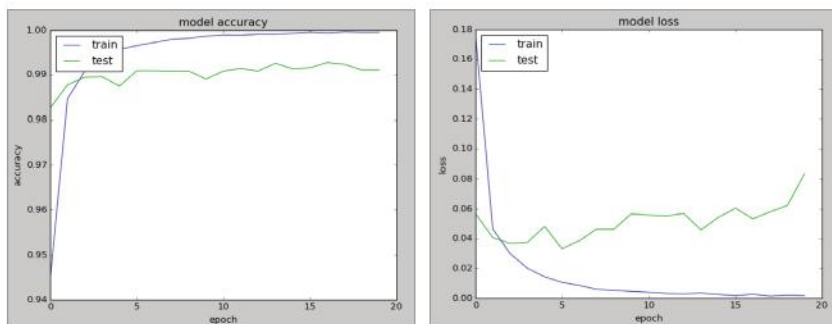
# построить график изменения потери
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Теперь выполним код. Как видим, время заметно выросло, теперь на каждую итерацию обучения сети уходит ~134 секунды вместо ~1–2 секунд для сети из главы 1. Однако и максимальная верность теперь равна 99.96%.



```
code — python keras_LeNet.py — 121x50
gulli-macbookpro:code gulli$ python keras_LeNet.py
Using TensorFlow backend.
(60000, 'train samples')
(10000, 'test samples')
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
40000/40000 [=====] - 124s - loss: 0.1766 - acc: 0.9445 - val_loss: 0.0568 - val_acc: 0.9826
Epoch 2/20
40000/40000 [=====] - 123s - loss: 0.0465 - acc: 0.9847 - val_loss: 0.0407 - val_acc: 0.9877
Epoch 3/20
40000/40000 [=====] - 129s - loss: 0.0300 - acc: 0.9908 - val_loss: 0.0367 - val_acc: 0.9895
Epoch 4/20
40000/40000 [=====] - 131s - loss: 0.0202 - acc: 0.9937 - val_loss: 0.0375 - val_acc: 0.9896
Epoch 5/20
40000/40000 [=====] - 127s - loss: 0.0144 - acc: 0.9957 - val_loss: 0.0482 - val_acc: 0.9875
Epoch 6/20
40000/40000 [=====] - 127s - loss: 0.0106 - acc: 0.9965 - val_loss: 0.0332 - val_acc: 0.9909
Epoch 7/20
40000/40000 [=====] - 128s - loss: 0.0086 - acc: 0.9972 - val_loss: 0.0386 - val_acc: 0.9909
Epoch 8/20
40000/40000 [=====] - 123s - loss: 0.0059 - acc: 0.9980 - val_loss: 0.0464 - val_acc: 0.9908
Epoch 9/20
40000/40000 [=====] - 123s - loss: 0.0053 - acc: 0.9982 - val_loss: 0.0463 - val_acc: 0.9908
Epoch 10/20
40000/40000 [=====] - 124s - loss: 0.0045 - acc: 0.9987 - val_loss: 0.0565 - val_acc: 0.9891
Epoch 11/20
40000/40000 [=====] - 125s - loss: 0.0040 - acc: 0.9989 - val_loss: 0.0558 - val_acc: 0.9908
Epoch 12/20
40000/40000 [=====] - 124s - loss: 0.0032 - acc: 0.9989 - val_loss: 0.0551 - val_acc: 0.9914
Epoch 13/20
40000/40000 [=====] - 125s - loss: 0.0030 - acc: 0.9991 - val_loss: 0.0569 - val_acc: 0.9908
Epoch 14/20
40000/40000 [=====] - 123s - loss: 0.0034 - acc: 0.9991 - val_loss: 0.0459 - val_acc: 0.9926
Epoch 15/20
40000/40000 [=====] - 124s - loss: 0.0025 - acc: 0.9993 - val_loss: 0.0542 - val_acc: 0.9913
Epoch 16/20
40000/40000 [=====] - 123s - loss: 0.0018 - acc: 0.9995 - val_loss: 0.0604 - val_acc: 0.9916
Epoch 17/20
40000/40000 [=====] - 123s - loss: 0.0027 - acc: 0.9993 - val_loss: 0.0533 - val_acc: 0.9927
Epoch 18/20
40000/40000 [=====] - 124s - loss: 0.0014 - acc: 0.9996 - val_loss: 0.0580 - val_acc: 0.9923
Epoch 19/20
40000/40000 [=====] - 123s - loss: 0.0020 - acc: 0.9995 - val_loss: 0.0623 - val_acc: 0.9911
Epoch 20/20
40000/40000 [=====] - 123s - loss: 0.0016 - acc: 0.9995 - val_loss: 0.0837 - val_acc: 0.9911
10000/10000 [=====] - 11s
('nTest score:', 0.072166633289733453)
('Test accuracy:', 0.9996000000000004)
['acc', 'loss', 'val_acc', 'val_loss']
```

Из графиков верности и потери видно, что для достижения верности 99.2% было бы достаточно всего 4–5 итераций.



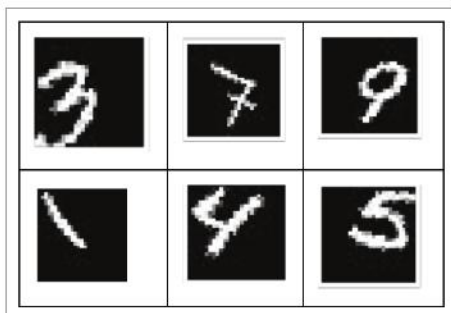
На следующем рисунке показана окончательная верность модели:

```

gulli-macbookpro:code gulli$ python keras_LeNet.py
Using TensorFlow backend.
(50000, 'train samples')
(10000, 'test samples')
Train on 48000 samples, validate on 12000 samples
Epoch 1/4
48000/48000 [=====] - 139s - loss: 0.1758 - acc: 0.9450 - val_loss: 0.0618 - val_acc: 0.9806
Epoch 2/4
48000/48000 [=====] - 136s - loss: 0.0461 - acc: 0.9849 - val_loss: 0.0408 - val_acc: 0.9878
Epoch 3/4
48000/48000 [=====] - 130s - loss: 0.0294 - acc: 0.9905 - val_loss: 0.0413 - val_acc: 0.9889
Epoch 4/4
48000/48000 [=====] - 129s - loss: 0.0199 - acc: 0.9936 - val_loss: 0.0373 - val_acc: 0.9900
10000/10000 [=====] - 12s
('Test score:', 0.027107118735135736)
('Test accuracy:', 0.9920999999999998)
['acc', 'loss', 'val_acc', 'val_loss']

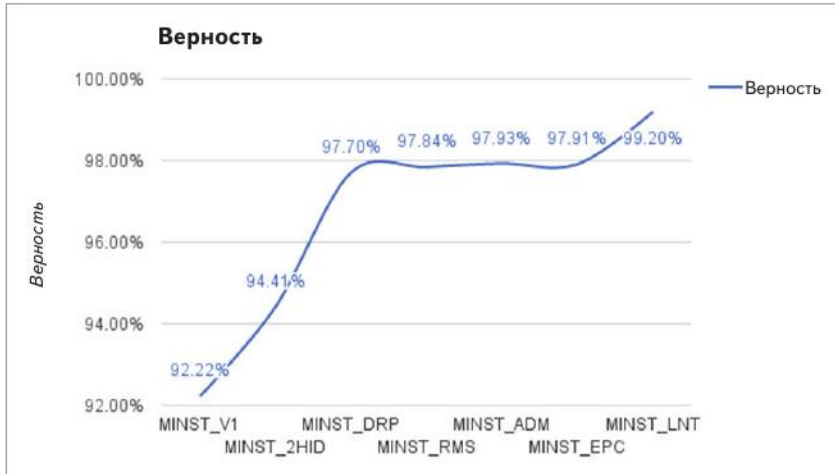
```

Рассмотрим несколько изображений из набора MNIST, просто чтобы понять, насколько хорошо значение 99.2%. Например, есть много способов написания цифры 9, один из них показан на рисунке ниже. То же самое можно сказать о цифрах 3, 7, 4 и 5. Число 1 на этом рисунке распознать так трудно, что, наверное, и у человека возникли бы проблемы.



На следующем графике мы подвели итог своим успехам. Мы начали с простой модели, достигшей верности 92.22%, т. е. из 100

цифр примерно 8 распознавались неправильно. А использование глубокой архитектуры позволило добиться прироста 7% и получить верность **99.20%** – неправильно распознается лишь одна цифра из 100.



О силе глубокого обучения

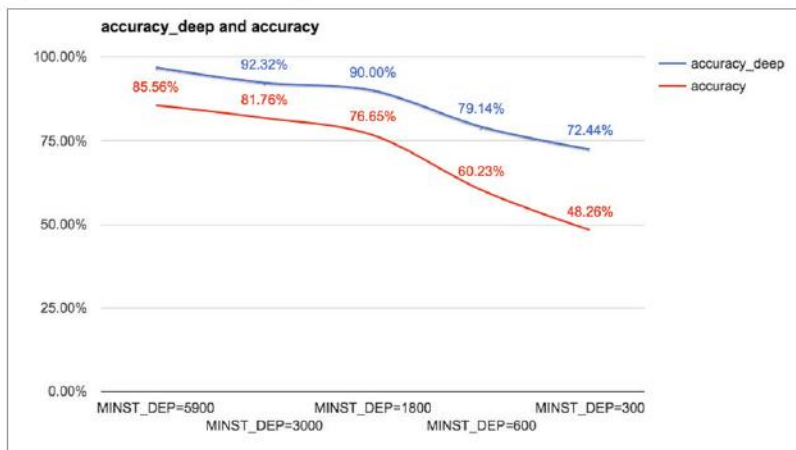
Чтобы лучше понять силу глубокого обучения и сверточных сетей, мы можем поставить еще один эксперимент: уменьшить размер обучающего набора и понаблюдать за снижением качества. Для этого разобьем набор из 50 000 примеров на два набора:

- размер собственно обучающего набора будет последовательно уменьшаться и составлять 5900, 3000, 1800, 600 и 300 примеров;
- остальные примеры будут входить в контрольный набор, используемый для оценки хода обучения.

Тестовый набор остается неизменным и содержит 10 000 примеров.

При такой конфигурации сравним только что определенную сверточную сеть глубокого обучения с первой нейронной сетью, определенной в главе 1. На следующем графике видно, что глубокая сеть всегда превосходит простую и разрыв тем больше, чем меньше обучающих примеров. При 5900 примерах верность глубокой сети равна 96.68% против 85.56% у простой. Но важнее, что при жалких 300

примерах верность глубокой сети все еще составляет 72.44%, тогда как у простой сети она снизилась до 48.26%. Все эксперименты проводились для четырех итераций обучения. Тем самым подтверждается грандиозный прогресс, достигнутый в результате изобретения глубокого обучения. На первый взгляд, это может показаться удивительным с математической точки зрения, потому что в глубокой сети гораздо больше неизвестных (весов), а, следовательно, и экспериментальных точек вроде бы должно быть больше.



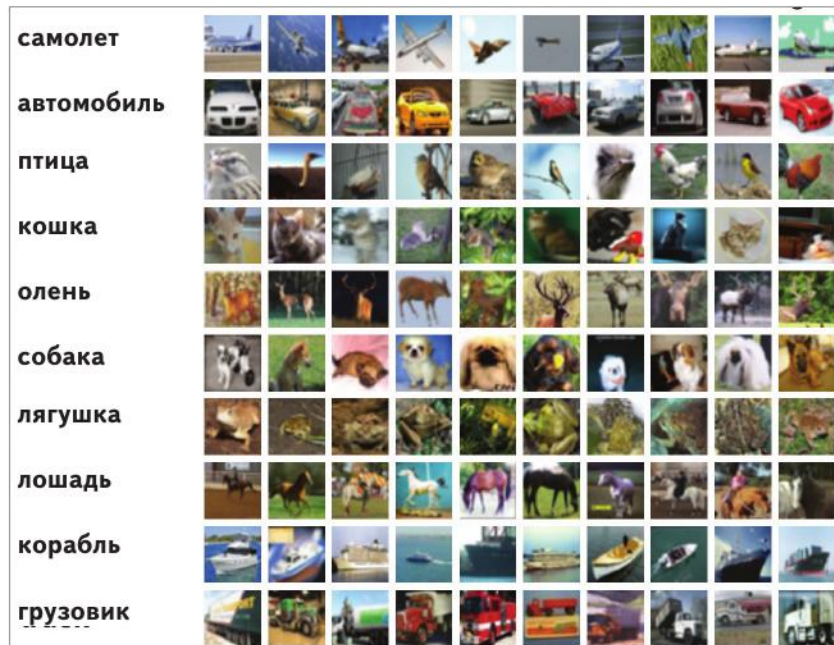
Однако сверточные сети выигрывают от сохранения пространственной информации, добавления свертки, пулинга и карт признаков, а эти механизмы совершенствовались в ходе миллионлетней эволюции (ведь такая организация подсмотрена у зрительной коры головного мозга).

Обзор современных результатов для набора данных MNIST опубликован на странице http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html. По состоянию на январь 2017 года лучшим достижением была частота ошибок 0.21%.

Распознавание изображений из набора CIFAR-10 с помощью глубокого обучения

Набор данных CIFAR-10 содержит 60 000 цветных изображений размера 32×32 пикселя с 3 каналами, разбитых на 10 классов.

В обучающем наборе 50 000 изображений, в тестовом – 10 000. На следующем рисунке, взятом из репозитория CIFAR (<https://www.cs.toronto.edu/~kriz/cifar.html>), представлены случайно выбранные примеры из каждого класса:



Задача состоит в том, чтобы распознать не предъявлявшиеся ранее изображения и отнести их к одному из 10 классов.

Прежде всего импортируем ряд модулей, определим некоторые константы и загрузим набор данных:

```
from keras.datasets import cifar10
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.optimizers import SGD, Adam, RMSprop
import matplotlib.pyplot as plt

# набор CIFAR_10 содержит 60K изображений 32x32 с 3 каналами
IMG_CHANNELS = 3
IMG_ROWS = 32
IMG_COLS = 32

# константы
```

```

BATCH_SIZE = 128
NB_EPOCH = 20
NB_CLASSES = 10
VERBOSE = 1
VALIDATION_SPLIT = 0.2
OPTIM = RMSprop()

# загрузить набор данных
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

```

Теперь применим унитарное кодирование и нормируем изображения:

```

# преобразовать к категориальному виду
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)

# преобразовать к формату с плавающей точкой и нормировать
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

```

В нашей сети будет 32 сверточных фильтра размера 3×3 . Размер выхода такой же, как размер входа, т. е. 32×32 , а в качестве функции активации используется ReLU, вносящая нелинейность. Далее следует операция max-пулинга с размером блока 2×2 и прореживание с коэффициентом 25%:

```

# сеть
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=(IMG_ROWS, IMG_COLS, IMG_CHANNELS)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

```

Следующий элемент конвейера – плотный слой с 512 нейронами и функцией активации ReLU, а за ним прореживание с коэффици-

ентом 50% и выходной слой softmax-классификации с 10 классами, по одному на категорию:

```
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(NB_CLASSES))
model.add(Activation('softmax'))
model.summary()
```

Определив нейросеть, мы можем перейти к обучению модели. В данном случае мы выделяем контрольный набор, помимо обучающего и тестового. Обучающий набор нужен для обучения модели, контрольный – для выбора наилучшего подхода к обучению, а тестовый – для проверки обученной модели на новых данных.

```
# обучение
model.compile(loss='categorical_crossentropy', optimizer=OPTIM,
              metrics=['accuracy'])
model.fit(X_train, Y_train, batch_size=BATCH_SIZE,
          epochs=NB_EPOCH, validation_split=VALIDATION_SPLIT,
          verbose=VERBOSE)
score = model.evaluate(X_test, Y_test,
                       batch_size=BATCH_SIZE, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

Сохраним еще архитектуру глубокой сети:

```
# сохранить модель
model_json = model.to_json()
open('cifar10_architecture.json', 'w').write(model_json)
# и веса, вычисленные в результате обучения сети
model.save_weights('cifar10_weights.h5', overwrite=True)
```

Выполним программу. Сеть достигает верности на тестовом наборе 66.4% при 20 итерациях. Мы также построили графики верности и потери и сохранили сеть методом `model.summary()`:

```
code — python keras_CIFAR10_simple.py — 121x77
```

```
gulli-macbookpro:code gulli$ python keras_CIFAR10_simple.py
Using TensorFlow backend.
('X_train shape:', (50000, 3, 32, 32))
(50000, 'train samples')
(10000, 'test samples')
```

Layer (type)	Output Shape	Param #	Connected to
convolution2d_1 (Convolution2D)	(None, 32, 32, 32)	896	convolution2d_input_1[0][0]
activation_1 (Activation)	(None, 32, 32, 32)	0	convolution2d_1[0][0]
maxpooling2d_1 (MaxPooling2D)	(None, 32, 16, 16)	0	activation_1[0][0]
dropout_1 (Dropout)	(None, 32, 16, 16)	0	maxpooling2d_1[0][0]
flatten_1 (Flatten)	(None, 8192)	0	dropout_1[0][0]
dense_1 (Dense)	(None, 512)	4194816	flatten_1[0][0]
activation_2 (Activation)	(None, 512)	0	dense_1[0][0]
dropout_2 (Dropout)	(None, 512)	0	activation_2[0][0]
dense_2 (Dense)	(None, 10)	5130	dropout_2[0][0]
activation_3 (Activation)	(None, 10)	0	dense_2[0][0]

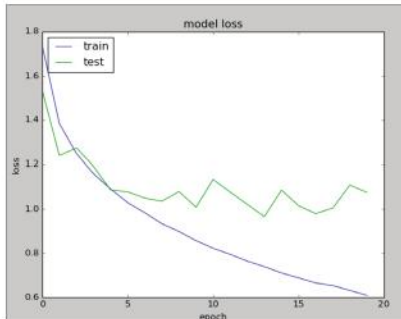
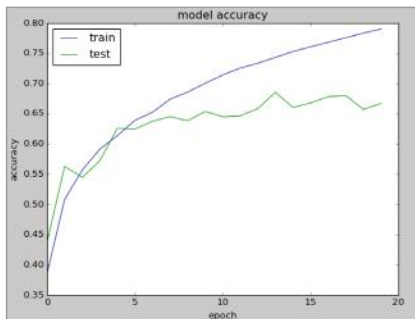
```

Total params: 4208842

Train on 40000 samples, validate on 10000 samples
Epoch 1/20
40000/40000 [=====] - 114s - loss: 1.7380 - acc: 0.3855 - val_loss: 1.5353 - val_acc: 0.4376
Epoch 2/20
40000/40000 [=====] - 114s - loss: 1.3847 - acc: 0.5001 - val_loss: 1.2392 - val_acc: 0.5629
Epoch 3/20
40000/40000 [=====] - 116s - loss: 1.2481 - acc: 0.5566 - val_loss: 1.2737 - val_acc: 0.5446
Epoch 4/20
40000/40000 [=====] - 114s - loss: 1.1590 - acc: 0.5913 - val_loss: 1.1919 - val_acc: 0.5722
Epoch 5/20
40000/40000 [=====] - 116s - loss: 1.0904 - acc: 0.6138 - val_loss: 1.0860 - val_acc: 0.6257
Epoch 6/20
40000/40000 [=====] - 115s - loss: 1.0282 - acc: 0.6391 - val_loss: 1.0771 - val_acc: 0.6245
Epoch 7/20
40000/40000 [=====] - 115s - loss: 0.9828 - acc: 0.6523 - val_loss: 1.0491 - val_acc: 0.6375
Epoch 8/20
40000/40000 [=====] - 114s - loss: 0.9328 - acc: 0.6739 - val_loss: 1.0344 - val_acc: 0.6453
Epoch 9/20
40000/40000 [=====] - 114s - loss: 0.8978 - acc: 0.6858 - val_loss: 1.0709 - val_acc: 0.6384
Epoch 10/20
40000/40000 [=====] - 115s - loss: 0.8556 - acc: 0.7004 - val_loss: 1.0072 - val_acc: 0.6538
Epoch 11/20
40000/40000 [=====] - 114s - loss: 0.8215 - acc: 0.7142 - val_loss: 1.1334 - val_acc: 0.6450
Epoch 12/20
40000/40000 [=====] - 115s - loss: 0.7938 - acc: 0.7256 - val_loss: 1.0761 - val_acc: 0.6464
Epoch 13/20
40000/40000 [=====] - 118s - loss: 0.7631 - acc: 0.7337 - val_loss: 1.0204 - val_acc: 0.6587
Epoch 14/20
40000/40000 [=====] - 121s - loss: 0.7381 - acc: 0.7433 - val_loss: 0.9647 - val_acc: 0.6853
Epoch 15/20
40000/40000 [=====] - 114s - loss: 0.7094 - acc: 0.7529 - val_loss: 1.0852 - val_acc: 0.6604
Epoch 16/20
40000/40000 [=====] - 114s - loss: 0.6872 - acc: 0.7608 - val_loss: 1.0144 - val_acc: 0.6600
Epoch 17/20
40000/40000 [=====] - 115s - loss: 0.6642 - acc: 0.7682 - val_loss: 0.9787 - val_acc: 0.6781
Epoch 18/20
40000/40000 [=====] - 114s - loss: 0.6524 - acc: 0.7758 - val_loss: 1.0035 - val_acc: 0.6803
Epoch 19/20
40000/40000 [=====] - 114s - loss: 0.6302 - acc: 0.7834 - val_loss: 1.1080 - val_acc: 0.6571
Epoch 20/20
40000/40000 [=====] - 113s - loss: 0.6081 - acc: 0.7902 - val_loss: 1.0744 - val_acc: 0.6672
Testing...
10000/10000 [=====] - 13s
('Test score:', 1.0762448620796203)
('Test accuracy:', 0.66490000000000005)
['acc', 'loss', 'val_acc', 'val_loss']

```

На следующих графиках показано, как изменяются верность и потеря на обучающем и тестовом наборе в зависимости от номера итерации:



Повышение качества распознавания набора CIFAR-10 путем углубления сети

Один из способов повысить качество распознавания – определить более глубокую сеть с несколькими операциями свертки. В данном случае мы возьмем такую последовательность модулей:

conv+conv+maxpool+dropout+conv+conv+maxpool

И в конце – стандартная последовательность *dense+dropout+dense*. Функцией активации всегда будет ReLU. Вот как выглядит код определения новой сети:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
    input_shape=(IMG_ROWS, IMG_COLS, IMG_CHANNELS)))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(NB_CLASSES))
model.add(Activation('softmax'))
```


Теперь прогоним программу. Сначала сохраним сеть, а затем выполним 40 итераций.

```
code — python keras_CIFAR10_V2.py — 121x77
~/Keras/codeBook/code — python keras_CIFAR10_V2.py — bash
```

```
gulli-macbookpro:code gulli$ python keras_CIFAR10_V1.py
Using TensorFlow backend.
('X_train shape:', (50000, 3, 32, 32))
(50000, 'train samples')
(10000, 'test samples')
```

Layer (type)	Output Shape	Param #	Connected to
convolution2d_1 (Convolution2D)	(None, 32, 32, 32)	896	convolution2d_input_1[0][0]
activation_1 (Activation)	(None, 32, 32, 32)	0	convolution2d_1[0][0]
convolution2d_2 (Convolution2D)	(None, 32, 32, 32)	9248	activation_1[0][0]
activation_2 (Activation)	(None, 32, 32, 32)	0	convolution2d_2[0][0]
maxpooling2d_1 (MaxPooling2D)	(None, 32, 16, 16)	0	activation_2[0][0]
dropout_1 (Dropout)	(None, 32, 16, 16)	0	maxpooling2d_1[0][0]
convolution2d_3 (Convolution2D)	(None, 64, 16, 16)	18496	dropout_1[0][0]
activation_3 (Activation)	(None, 64, 16, 16)	0	convolution2d_3[0][0]
convolution2d_4 (Convolution2D)	(None, 64, 14, 14)	36928	activation_3[0][0]
activation_4 (Activation)	(None, 64, 14, 14)	0	convolution2d_4[0][0]
maxpooling2d_2 (MaxPooling2D)	(None, 64, 7, 7)	0	activation_4[0][0]
dropout_2 (Dropout)	(None, 64, 7, 7)	0	maxpooling2d_2[0][0]
flatten_1 (Flatten)	(None, 3136)	0	dropout_2[0][0]
dense_1 (Dense)	(None, 512)	1606144	flatten_1[0][0]
activation_5 (Activation)	(None, 512)	0	dense_1[0][0]
dropout_3 (Dropout)	(None, 512)	0	activation_5[0][0]
dense_2 (Dense)	(None, 10)	5130	dropout_3[0][0]
activation_6 (Activation)	(None, 10)	0	dense_2[0][0]

```

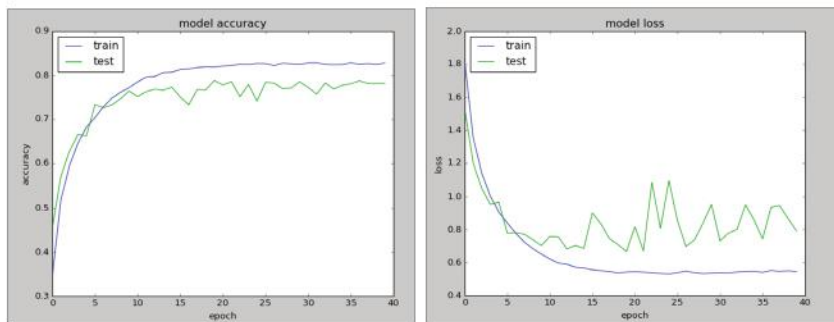
Total params: 1676842
Train on 40000 samples, validate on 10000 samples
Epoch 1/40
40000/40000 [=====] - 430s - loss: 1.8179 - acc: 0.3443 - val_loss: 1.5250 - val_acc: 0.4551
Epoch 2/40
40000/40000 [=====] - 382s - loss: 1.3506 - acc: 0.5182 - val_loss: 1.1998 - val_acc: 0.5714

```

Как показано на следующем снимке экрана, достигнута верность 76.9%.

```
Epoch 39/40
40000/40000 [=====] - 348s - loss: 0.5497 - acc: 0.8246 - val_loss: 0.8669 - val_acc: 0.7811
Epoch 40/40
40000/40000 [=====] - 346s - loss: 0.5447 - acc: 0.8280 - val_loss: 0.7910 - val_acc: 0.7816
Testing...
10000/10000 [=====] - 41s
('Test score:', 0.79934534568786619)
('Test accuracy:', 0.76929999999999998)
['acc', 'loss', 'val_acc', 'val_loss']
```

Таким образом, мы улучшили предыдущий результат на 10.5%. Для полноты картины построим еще графики зависимости верности и потери от числа итераций:



Повышение качества распознавания набора CIFAR-10 путем пополнения данных

Еще один способ повысить качество – сгенерировать дополнительные обучающие изображения. Идея состоит в том, чтобы взять стандартный набор данных CIFAR и пополнить его, подвергнув изображения различным преобразованиям: вращению, параллельному переносу, масштабированию, отражению относительно горизонтальной и вертикальной оси, перестановке каналов и т. д. Приведем соответствующий код:

```
from keras.preprocessing.image import ImageDataGenerator
from keras.datasets import cifar10
import numpy as np
NUM_TO_AUGMENT=5

# загрузить набор данных
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# пополнение
print("Augmenting training set images...")
datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```

Аргумент `rotation_range` – это диапазон углов в градусах (0–180), на которые можно поворачивать изображения (случайным обра-

зом). Аргументы `width_shift` и `height_shift` – диапазоны случайного параллельного переноса по горизонтали и по вертикали. Аргумент `zoom_range` задает диапазон случайного масштабирования изображений, `horizontal_flip` говорит, что случайным образом отобранную половину изображений нужно отразить относительно вертикальной оси, а `fill_mode` определяет стратегию вычисления новых пикселей, образующихся при повороте или параллельном переносе:

```
xtas, ytas = [], []
for i in range(X_train.shape[0]):
    num_aug = 0
    x = X_train[i] # (3, 32, 32)
    x = x.reshape((1,) + x.shape) # (1, 3, 32, 32)
    for x_aug in datagen.flow(x, batch_size=1,
        save_to_dir='preview', save_prefix='cifar', save_format='jpeg'):
        if num_aug >= NUM_TO_AUGMENT:
            break
        xtas.append(x_aug[0])
        num_aug += 1
```

В результате пополнения мы получим много новых изображений, сгенерированных на основе стандартного набора CIFAR-10:



Теперь посмотрим, что это нам дает. Мы генерируем новые изображения, а затем обучаем ту же самую сверточную сеть, что и раньше, на пополненном наборе данных. Эффективности ради генератор работает параллельно обучению модели. Это позволяет пополнять набор на CPU и одновременно обучать сеть на GPU. Код показан ниже:

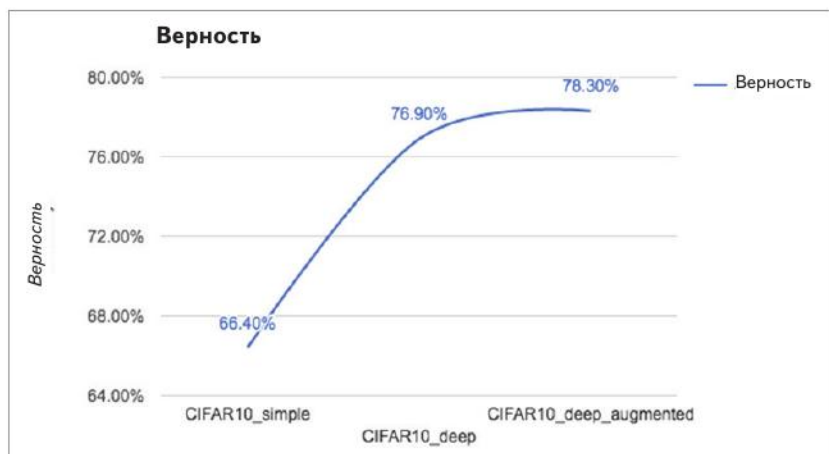
```
# инициализировать генератор
datagen.fit(X_train)

# обучить
history = model.fit_generator(datagen.flow(X_train, Y_train,
    batch_size=BATCH_SIZE), samples_per_epoch=X_train.shape[0],
    epochs=NB_EPOCH, verbose=VERBOSE)
score = model.evaluate(X_test, Y_test,
    batch_size=BATCH_SIZE, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

Каждая итерация теперь обходится дороже, поскольку обучающих данных стало больше. Выполнив всего 50 итераций, мы достигли верности 78.3%:

```
Epoch 46/50
50000/50000 [=====] - 405s - loss: 0.8288 - acc: 0.7297
Epoch 47/50
50000/50000 [=====] - 424s - loss: 0.8349 - acc: 0.7303
Epoch 48/50
50000/50000 [=====] - 408s - loss: 0.8319 - acc: 0.7295
Epoch 49/50
50000/50000 [=====] - 403s - loss: 0.8386 - acc: 0.7281
Epoch 50/50
50000/50000 [=====] - 398s - loss: 0.8394 - acc: 0.7267
Testing...
10000/10000 [=====] - 42s
('Test score:', 0.73110332846641546)
('Test accuracy:', 0.7836999999999995)
['acc', 'loss']
```

Полученные в ходе экспериментов результаты отражены на следующем графике:



Обзор современных результатов для набора данных CIFAR-10 опубликован на странице http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html. По состоянию на январь 2017 года лучшим достижением была верность 96.53%.

Предсказание на основе результатов обучения на наборе CIFAR-10

Пусть теперь мы хотим использовать обученную на наборе CIFAR-10 модель для массовой обработки изображений. Поскольку мы сохранили модель вместе с весами, то обучать ее каждый раз не нужно.

```
import numpy as np
import scipy.misc
from keras.models import model_from_json
from keras.optimizers import SGD

# загрузить модель
model_architecture = 'cifar10_architecture.json'
model_weights = 'cifar10_weights.h5'
model = model_from_json(open(model_architecture).read())
model.load_weights(model_weights)

# загрузить изображения
img_names = ['cat-standing.jpg', 'dog.jpg']
imgs = [np.transpose(scipy.misc.imread(scipy.misc.imread(img_name), (32,32)),
    (1, 0, 2)).astype('float32') for img_name in img_names]
imgs = np.array(imgs) / 255

# обучить
optim = SGD()
model.compile(loss='categorical_crossentropy', optimizer=optim,
    metrics=['accuracy'])

# предсказать
predictions = model.predict_classes(imgs)
print(predictions)
```

Давайте посмотрим, что предсказывает модель для изображе-

ний  и . Как и ожидалось, мы получаем категории 3 (кошка) и 5 (собака):

```
gulli-macbookpro:code gulli$ python keras_EvaluateCIFAR10.py
Using TensorFlow backend.
2/2 [=====] - 0s
[3 5]
gulli-macbookpro:code gulli$
```


Очень глубокие сверточные сети для распознавания больших изображений

В 2014 году был внесен интересный вклад в распознавание изображений (см. K. Simonyan, A. Zisserman «Very Deep Convolutional Networks for Large-Scale Image Recognition», 2014). В этой работе показано, что, увеличив число весовых слоев до 16–19, можно добиться значительного улучшения по сравнению с предшествующими конфигурациями. В одной из рассматриваемых моделей (D или VGG-16) было 16 слоев. Для обучения модели на наборе данных ImageNet ILSVRC-2012 (<http://image-net.org/challenges/LSVRC/2012/>) была написана программа на Java с использованием библиотеки Caffe (<http://caffe.berkeleyvision.org/>). Этот набор содержит изображения из 1000 классов, разбитые на три набора: обучающий (1.3 миллиона изображений), контрольный (50 000 изображений) и тестовый (100 000 изображений). Все изображения трехканальные размера 224×224 . Для этой модели ошибка непопадания в первые 5 классов составила 7.5% на наборе ILSVRC-2012-val и 7.4% на наборе ILSVRC-2012-test.

Приведем цитату с сайта ImageNet:

Цель соревнования – оценить содержание фотографий для целей поиска и автоматического аннотирования с применением подмножества большого размеченного вручную набора ImageNet (10 миллионов помеченных изображений объектов из 10 с лишним тысяч категорий) для обучения. Тестовые изображения не содержат никаких аннотаций – ни сегментации, ни меток, а алгоритм должен вывести метки изображенных объектов.

Веса, полученные в результате обучения модели, реализованной на Caffe, были преобразованы к виду, понятному Keras (см. <https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3>), так что их можно загрузить в модель, которая ниже определена так же, как в оригинальной статье:

```
from keras.models import Sequential
from keras.layers.core import Flatten, Dense, Dropout
from keras.layers.convolutional import Conv2D, MaxPooling2D, ZeroPadding2D
from keras.optimizers import SGD
import cv2, numpy as np
```

```
# определить сеть VGG16
```

```
def VGG_16(weights_path=None):
    model = Sequential()
    model.add(ZeroPadding2D((1,1),input_shape=(3,224,224)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(256, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(256, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(256, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(512, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(512, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(512, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(512, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(512, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(512, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))
    model.add(Flatten())

    # верхние слои сети VGG
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1000, activation='softmax'))

    if weights_path:
        model.load_weights(weights_path)

    return model
```

Распознавание кошек с помощью сети VGG-16

Теперь протестируем сеть на изображении



```

im = cv2.resize(cv2.imread('cat.jpg'), (224, 224)).astype(np.float32)
im = im.transpose((2,0,1))
im = np.expand_dims(im, axis=0)

# Тестировать предобученную модель
model = VGG_16('/Users/gulli/Keras/codeBook/code/data/vgg16_weights.h5')
optimizer = SGD()
model.compile(optimizer=optimizer, loss='categorical_crossentropy')
out = model.predict(im)
print np.argmax(out)

```

Программа возвращает класс 285 – кошку породы сфинкс (см. <https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>):



```

code — -bash — 108x5
~/Keras/codeBook/code — -bash  ...  ~/Keras/codeBook/code — -bash +
set properly.
gulli-macbookpro:code gulli$ python keras_VGG16.py
Using TensorFlow backend.
285
gulli-macbookpro:code gulli$

```

Использование встроенного в Keras модуля VGG-16

Приложения Keras – это предварительно построенные и обученные глубокие модели. Веса автоматически загружаются при создании экземпляра модели и хранятся в каталоге `~/.keras/models/`. Использовать встроенный код очень просто:

```

from keras.models import Model
from keras.preprocessing import image
from keras.optimizers import SGD
from keras.applications.vgg16 import VGG16
import matplotlib.pyplot as plt
import numpy as np
import cv2

# готовая модель с предобученными на наборе imagenet весами
model = VGG16(weights='imagenet', include_top=True)
sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd, loss='categorical_crossentropy')

# сделать размер таким же, как у изображений, на которых обучалась модель VGG16
im = cv2.resize(cv2.imread('steam-locomotive.jpg'), (224, 224))
im = np.expand_dims(im, axis=0)

# предсказание
out = model.predict(im)
plt.plot(out.ravel())

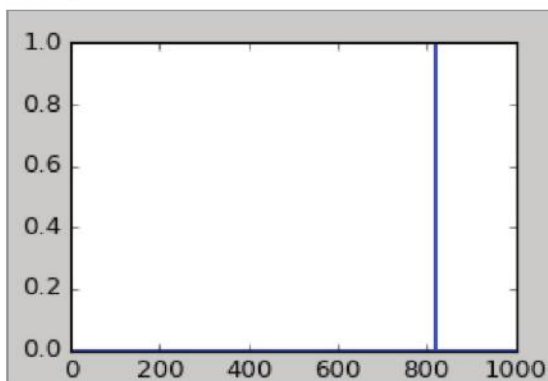
```

```
plt.show()
print np.argmax(out)
```

должна быть напечатана категория 820 – паровоз

Теперь возьмем изображение паровоза:

На таком ездил мой дедушка. Выполнив программу, мы получим категорию 820, которой в наборе ImageNet обозначается *паровоз*. Важно также, что вероятность всех остальных классов очень мала, как видно из следующего графика:



В заключение этого раздела отметим, что VGG-16 – лишь одна из моделей, встроенных в Keras. Полный перечень предобученных моделей приведен на странице <https://keras.io/applications/>.

Использование готовых моделей глубокого обучения для выделения признаков

Модель VGG-16 и вообще любую ГСНС можно очень просто использовать для выделения признаков. Ниже показана реализация этой идеи для выделения признаков в конкретном слое.

```
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np
```

```
# предварительно построенная и обученная модель глубокого обучения VGG16
base_model = VGG16(weights='imagenet', include_top=True)
```



```

for i, layer in enumerate(base_model.layers):
    print(i, layer.name, layer.output_shape)

# выделить признаки из слоя block4_pool
model = Model(input=base_model.input,
              output=base_model.get_layer('block4_pool').output)
img_path = 'cat.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

# получить признаки
features = model.predict(x)

```

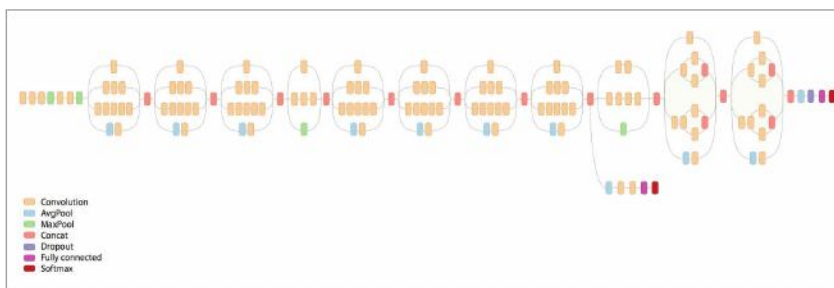
Возникает вопрос, а зачем может понадобиться выделять признаки из промежуточного слоя ГСНС? Дело в том, что если сеть обучилась классифицировать изображения, то каждый ее слой научился находить признаки, необходимые для окончательной классификации. Нижние слои идентифицируют такие низкоуровневые признаки, как цвета и границы, а верхние составляют из них признаки более высокого уровня, например, геометрические фигуры или объекты. Следовательно, промежуточный слой способен выделить из изображения важные признаки, которые могут оказаться полезны для других видов классификации. Преимуществ тут несколько. Во-первых, можно опереться на находящиеся в открытом доступе крупномасштабные обученные модели и перенести результаты их обучения на другие предметные области. Во-вторых, можно сэкономить время на дорогостоящем обучении большой модели. В-третьих, можно получить разумное решение даже тогда, когда для некоторой предметной области недостаточно обучающих примеров. Кроме того, мы получаем хорошее начальное приближение для решения имеющейся задачи вместо случайной гипотезы.

Очень глубокая сеть inception-v3, применяемая для переноса обучения

Перенос обучения – весьма эффективная техника глубокого обучения, имеющая приложения в разных областях. Идея очень проста, для ее объяснения воспользуемся аналогией. Предположим, вы хотите изучить новый язык, скажем, испанский. Тогда полезно начать с того, что вы уже знаете о каком-то другом языке, например, английском.

В русле этой идеи специалисты по компьютерному зрению активно используют предобученные СНС для порождения представлений в новых задачах, где набор данных не настолько велик, чтобы обучить СНС с нуля. Еще одна часто встречающаяся тактика – взять предобученную на наборе ImageNet сеть и настроить ее под новую задачу.

Inception-v3 – очень глубокая сверточная сеть, разработанная Google. Keras реализует полную сеть, показанную на рисунке ниже, и модель, предобученная на наборе ImageNet, включена в дистрибутив. По умолчанию в этой модели используются трехканальные изображения размера 299×299 :



В основу этого схематического примера положено приложение, имеющееся на странице <https://keras.io/applications/>. Предполагается, что есть обучающий набор данных D в предметной области, отличной от ImageNet. В D имеется 1024 входных признака и 200 выходных категорий. Рассмотрим следующий фрагмент кода:

```
from keras.applications.inception_v3 import InceptionV3
from keras.preprocessing import image
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D
from keras import backend as K

# создать базовую предобученную модель
base_model = InceptionV3(weights='imagenet', include_top=False)
```

Мы используем обученную сеть inception-v3 и не включаем верхние слои, потому что хотим адаптировать ее к D . В нашей модели выходным будет плотный слой softmax-классификации с 200 классами. Для преобразования входных данных к форме, пригодной для этого плотного слоя, применяется модуль `GlobalAveragePooling2D`. В действительности тензор `base_model.output` имеет форму (*samples, channels, rows, cols*), если `dim_ordering="th"`, или форму (*samples, rows,*

cols, channels), если `dim_ordering="tf"`, тогда как плотному слою нужна форма (*samples, channels*). Поэтому `GlobalAveragePooling2D` производит усреднение по строкам *rows* и столбам *cols*. Взглянув на последние четыре слоя (при `include_top=True`), мы увидим такие формы:

```
# layer.name, layer.input_shape, layer.output_shape
('mixed10', [(None, 8, 8, 320), (None, 8, 8, 768), (None, 8, 8, 768),
              (None, 8, 8, 192)], (None, 8, 8, 2048))
('avg_pool', (None, 8, 8, 2048), (None, 1, 1, 2048))
('flatten', (None, 1, 1, 2048), (None, 2048))
('predictions', (None, 2048), (None, 1000))
```

Если же положить `include_top=False`, то три верхних слоя удаляются, а сверху остается слой `mixed10`, так что модуль `GlobalAveragePooling2D` преобразует *(None, 8, 8, 2048)* в *(None, 2048)*, где каждый элемент тензора *(None, 2048)* – результат усреднения по соответствующему подтензору размера *(8, 8)* тензора *(None, 8, 8, 2048)*:

```
# добавить глобальный слой пулинга, выполняющего пространственное усреднение
x = base_model.output
x = GlobalAveragePooling2D()(x) # первым добавляем полносвязный слой
x = Dense(1024, activation='relu')(x) # а последним ReLU-слоем с 200 классами
predictions = Dense(200, activation='softmax')(x) # обучаемая модель
model = Model(input=base_model.input, output=predictions)
```

Все сверточные слои предобучены, поэтому замораживаем их на время обучения модели в целом:

```
# заморозить все сверточные слои сети InceptionV3
for layer in base_model.layers: layer.trainable = False
```

Затем модель компилируется и обучается в течение нескольких периодов, чтобы обучить верхние слои:

```
# откомпилировать модель (это нужно делать ПОСЛЕ того, как некоторые слои
# помечены как необучаемые)
compile the model (should be done *after* setting layers to nontrainable)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# обучать модель на новых данных в течение нескольких периодов
model.fit_generator(...)
```

Затем мы замораживаем верхние слои модели и настраиваем нижние. В данном случае замораживаются первые 172 слоя (гиперпараметр настройки):

```
# мы решили обучить 2 слоя inception, так что первые 172 слоя замораживаются,
# а остальные размораживаются
for layer in
model.layers[:172]: layer.trainable = False
```

```
for layer in
model.layers[172:]: layer.trainable = True
```

Затем модель перекомпилируется, чтобы изменения вступили в силу:

```
# используем CGC с малой скоростью обучения
from keras.optimizers
import SGD
model.compile(optimizer=SGD(lr=0.0001, momentum=0.9),
              loss='categorical_crossentropy')

# снова обучаем модель (на этот раз настраиваем 2 верхних слоя inception)
# и верхние слои Dense
model.fit_generator(...)
```

Теперь мы имеем новую глубокую сеть, в которой повторно используется часть стандартной сети Inception-v3, но обучена она на данных из другой предметной области посредством переноса обучения. Разумеется, для достижения приемлемой верности можно настроить много параметров. Но в качестве отправной точки мы теперь используем очень большую предобученную сеть, и, следовательно, можем отказаться от полного обучения на наших машинах и воспользоваться тем, что уже есть в Keras.

Резюме

В этой главе мы научились использовать глубокие сверточные сети для распознавания рукописных цифр из набора MNIST с высокой верностью. Затем мы воспользовались набором данных CIFAR 10, чтобы построить глубокий классификатор с 10 категориями, и набором ImageNet для построения точного классификатора с 1000 категорий. Кроме того, мы узнали, как можно использовать большие предобученные сети типа VGG16 и очень глубокие сети типа InceptionV3. В заключение мы обсудили технику переноса обучения, позволяющую адаптировать готовые модели, обученные на больших наборах данных, к новым предметным областям.

В следующей главе мы познакомимся с применением порождающих состязательных сетей к задаче синтеза данных, похожих на порождаемые людьми. Мы также представим глубокую нейронную сеть WaveNet для высококачественного воспроизведения человеческого голоса и звучания музыкальных инструментов.

Глава 4

Порождающие состязательные сети и WaveNet

В этой главе мы обсудим **порождающие состязательные сети (ПСС)** (generative adversarial network – GAN) и сеть WaveNet. Ян Лекун, один из отцов глубокого обучения, считает ПСС самой интересной идеей за последние 10 лет развития МО (<https://www.quora.com/What-are-some-recent-and-potentially-upcoming-breakthroughs-in-deep-learning>). ПСС способны обучаться порождению синтетических данных, которые выглядят в точности как настоящие. Например, компьютер можно научить рисовать и создавать реалистичные изображения. Идея была предложена Яном Гудфеллоу (см. I. Goodfellow «NIPS 2016 Tutorial: Generative Adversarial Networks», 2016), который работал в Монреальском университете, в компании Google Brain и, в последнее время, в OpenAI (<https://openai.com/>). WaveNet – глубокая порождающая сеть, предложенная компанией Google DeepMind для обучения компьютеров высококачественному воспроизведению человеческого голоса и звучания музыкальных инструментов.

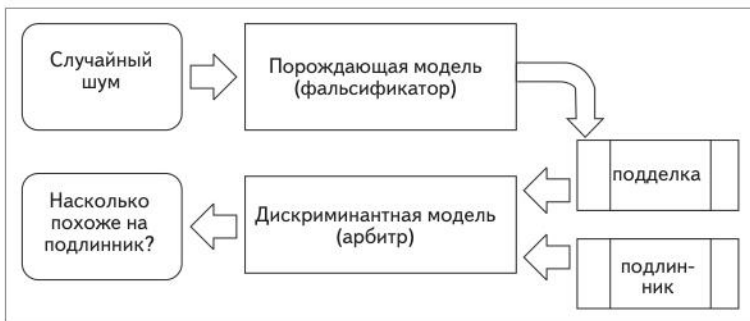
Мы обсудим следующие темы:

- что такое ПСС;
- глубокие сверточные ПСС;
- приложения ПСС.

Что такое ПСС?

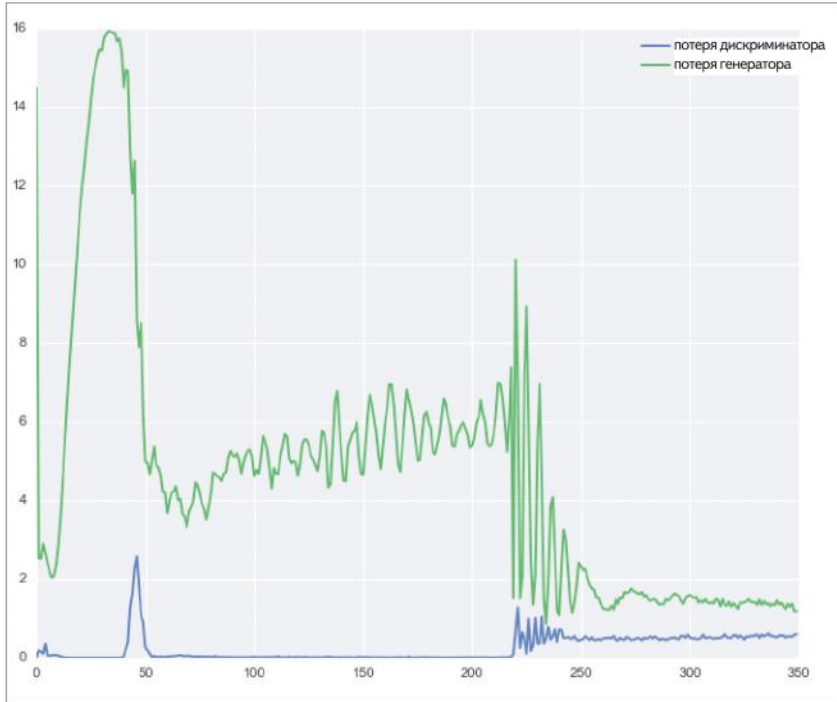
Основная идея ПСС аналогична *подделке произведений искусства*, т. е. созданию работ (<https://en.wikipedia.org/wiki/Art>), ошибочно

приписываемых другим, обычно более известным, авторам. ПСС обучает две нейронные сети одновременно, как показано на рисунке ниже. Генератор $G(Z)$ порождает подделку, а дискриминатор $D(Y)$ судит о том, насколько репродукция реалистична, основываясь на наблюдениях аутентичных произведений и копий. $D(Y)$ принимает вход Y (например, изображение) и выражает свое суждение о его подлинности – в общем случае значение, близкое к 0, означает *подлинный*, а близкое к единице – *подделка*. $G(Z)$ принимает на входе случайный шум Z и обучается обманывать D , заставляя его думать, что результат работы $G(Z)$ – подлинное произведение. Таким образом, цель обучения дискриминатора – максимизировать $D(Y)$ для всех изображений из истинного распределения данных и минимизировать для изображений, не выбранных из истинного распределения. Следовательно, G и D ведут себя как противники в некоторой игре, отсюда и название *состязательное обучение*. Отметим, что G и D обучаются попеременно, а в качестве целевой функции выступает функция потерь, оптимизируемая методом градиентного спуска. Порождающая модель обучается подделывать, а дискриминантная распознавать подделки. Дискриминантная сеть (обычно стандартная сверточная нейронная сеть) пытается классифицировать изображение как настоящее или сгенерированное. Важная новая идея – обратное распространение через дискриминатор и генератор с целью корректировать параметры генератора таким образом, чтобы генератор мог обучиться, как успешнее обманывать дискриминатор. В конечном итоге генератор научится порождать поддельные изображения, неотличимые от настоящих.



Разумеется, от ПСС требуется найти точку равновесия в игре двух игроков. Чтобы обучение оказалось эффективным, необходимо, чтобы обновление, в результате которого один игрок опуска-

ется вниз, одновременно приводило к опусканию другого игрока. Задумайтесь об этом! Если фальсификатор научится обманывать арбитра в каждом случае, то самому фальсификатору больше нечему учиться. Иногда оба игрока достигают равновесия, но это не гарантируется, и игра может продолжаться долго. На следующем графике показан пример обучения обоих игроков.



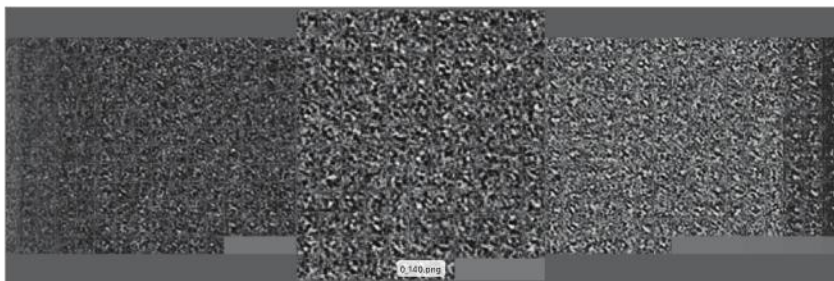
Некоторые приложения ПСС

Мы видели, что генератор обучается подделывать данные. Это значит, что он обучается создавать новые синтетические данные, которые выглядят так, будто созданы человеком. Прежде чем переходить к коду, хочу продемонстрировать результаты из недавней статьи Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaolei Huang, Xiaogang Wang, Dimitris Metaxas «StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks» (код доступен по адресу <https://github.com/hanzhanggit/StackGAN>).

Здесь ПСС используется для синтеза изображений по текстовому описанию. Результаты впечатляют. В первом столбце мы видим реальные изображения из тестового набора, а во всех остальных – изображения, сгенерированные на стадии I и II сети StackGAN. Дополнительные примеры можно найти на YouTube (<https://www.youtube.com/watch?v=SuRyL5vhCIM&feature=youtu.be>):



Теперь посмотрим, как можно научить ПСС *подделывать* набор данных MNIST. В этом случае в качестве генератора и дискриминатора ПСС используются сверточные сети (см. A. Radford, L. Metz, and S. Chintala «Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks», arXiv: 1511.06434, 2015). В начале генератор порождает нечто неразборчивое, но после нескольких итераций синтетические цифры становятся все более отчетливыми. На следующем рисунке панели упорядочены по номеру итерации и, как видите, качество постепенно повышается.



Далее показаны имитации рукописных цифр на последующих итерациях:



А вот как выглядят поддельные цифры в конце обучения. Они практически неотличимы от оригинала.



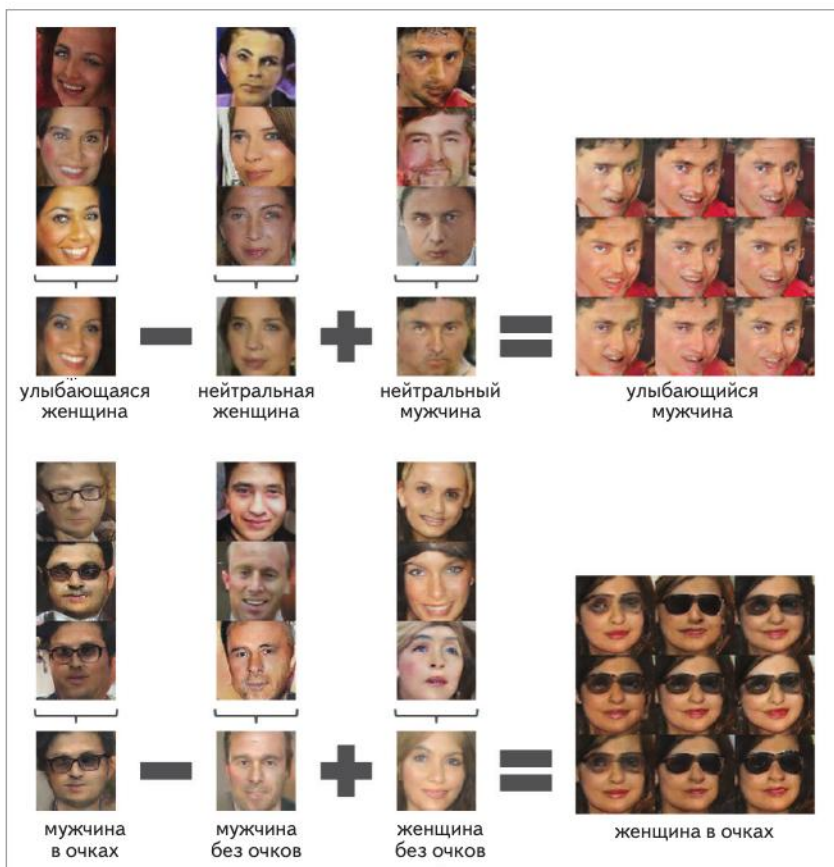
Одно из самых потрясающих применений ПСС – арифметические операции с лицами в векторе генератора Z . Иными словами, оставаясь в пространстве синтетических изображений, мы можем производить такие действия:

[улыбающаяся женщина] – [нейтральная женщина] + [нейтральный мужчина] = [улыбающийся мужчина]

Или такие:

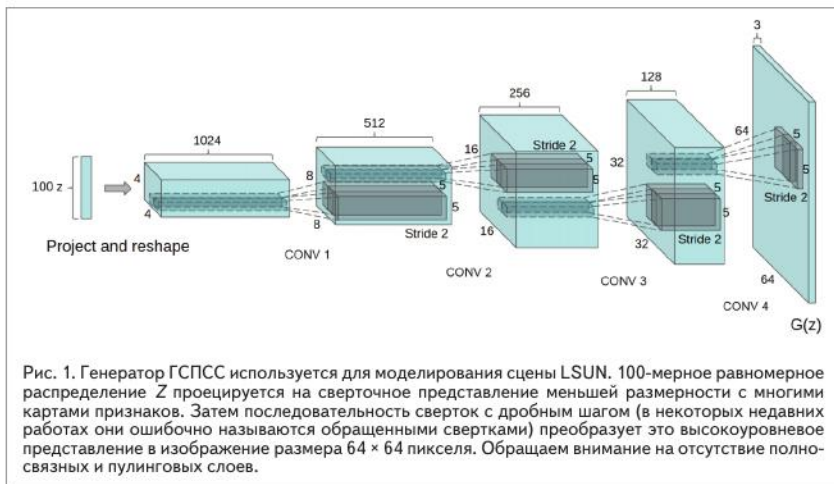
[мужчина в очках] – [мужчина без очков] + [женщина без очков] = [женщина в очках]

Следующее изображение взято из вышеупомянутой статьи «Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks»:



Глубокие сверточные порождающие состязательные сети

Глубокие сверточные порождающие состязательные сети (ГСПСС, англ. DCGAN) впервые описаны в статье A. Radford, L. Metz, and S. Chintala «Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks», arXiv: 1511.06434, 2015. В генераторе используется 100-мерное пространство с равномерным распределением Z , которое проецируется на пространство меньшей размерности с помощью последовательности парных операций свертки. Пример показан на рисунке ниже.



Генератор ГСПСС можно описать следующим кодом на Keras; имеется также другая реализация по адресу <https://github.com/jacobgil/keras-dcgan>.

```
def generator_model():
    model = Sequential()
    model.add(Dense(input_dim=100, output_dim=1024))
    model.add(Activation('tanh'))
    model.add(Dense(128*7*7))
    model.add(BatchNormalization())
    model.add(Activation('tanh'))
    model.add(Reshape((128, 7, 7), input_shape=(128*7*7)))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Convolution2D(64, 5, 5, border_mode='same'))
    model.add(Activation('tanh'))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Convolution2D(1, 5, 5, border_mode='same'))
    model.add(Activation('tanh'))
    return model
```

Отметим, что здесь используется синтаксис Keras 1.x. Но благодаря унаследованным интерфейсам код будет работать и в версии Keras 2.0. Правда, выдаются предупреждения, показанные на следующем рисунке:


```

gullis-macbookpro:keras-dcgan gullis python dcgan.py --mode train
Using TensorFlow backend.
dcgan.py:40: UserWarning: Update your 'Conv2D' call to the Keras 2 API: 'Conv2D(64, (5, 5), padding="same", input_shape=(1, 28, 28...))'
dcgan.py:43: UserWarning: Update your 'Conv2D' call to the Keras 2 API: 'Conv2D(128, (5, 5))'
dcgan.py:20: UserWarning: Update your 'Dense' call to the Keras 2 API: 'Dense(units=1024, input_dim=100)'
dcgan.py:27: UserWarning: Update your 'Conv2D' call to the Keras 2 API: 'Conv2D(64, (5, 5), padding="same")'
dcgan.py:30: UserWarning: Update your 'Conv2D' call to the Keras 2 API: 'Conv2D(1, (5, 5), padding="same")'
('Epoch 15', 0)
('Number of batches', 408)

```

Теперь рассмотрим код. Первый плотный слой принимает 100-мерный входной вектор и порождает 1024 выхода, в качестве функции активации используется *tanh*. Предполагается, что входные данные выбираются из равномерного распределения на отрезке $[-1, 1]$. Следующий плотный слой порождает на выходе тензор формы $128 \times 7 \times 7$, применяя пакетную нормировку (см. S. Ioffe, C. Szegedy «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift», arXiv: 1502.03167, 2014) – технику, помогающую стабилизировать обучение путем нормировки входных данных, так чтобы их среднее было равно нулю, а дисперсия – единице. Эмпирически установлено, что пакетная нормировка во многих случаях ускоряет обучение, смягчает проблемы, вызванные неудачной инициализацией, и вообще приводит к более точным результатам. В конвейер вставляется также модуль `Reshape()`, который порождает данные формы $127 \times 7 \times 7$ (127 каналов, ширина 7, высота 7) с параметром `dim_ordering` равным `tf`, и модуль повышающей передискретизации `UpSampling()`, который повторяет каждый пиксель в квадрате 2×2 . После этого идет сверточный слой, порождающий 64 фильтра с ядром размера 5×5 и функцией активации *tanh*, а за ним еще один модуль `UpSampling()` и последняя свертка с одним выходным фильтром, ядром размера 5×5 и функцией активации *tanh*. Отметим, что в этой сверточной сети нет пулинговых операций. Дискриминатор описывается следующим кодом:

```

def discriminator_model():
    model = Sequential()
    model.add(Convolution2D(64, 5, 5, border_mode='same', input_shape=(1, 28, 28)))
    model.add(Activation('tanh'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Convolution2D(128, 5, 5))
    model.add(Activation('tanh'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(1024))
    model.add(Activation('tanh'))

```

```
model.add(Dense(1))
model.add(Activation('sigmoid'))
return model
```

Мы берем изображение из стандартного набора MNIST, имеющее форму (1, 28, 28), применяем свертку с 64 фильтрами размера 5×5 и функцию активации $\tanh$. Далее следует операция тах-пулинга по области размера 2×2 и еще одна свертка и операция тах-пулинга. Последние два слоя плотные, и самый верхний, дающий предсказание о подделке, состоит всего из одного нейрона с сигмоидной функцией активации. На протяжении выбранного количества периодов генератор и дискриминатор по очереди обучаются с использованием `binary_crossentropy` в качестве функции потерь. В каждом периоде генератор делает ряд предсказаний (например, порождает поддельные изображения рукописных цифр), а дискриминатор пытается обучиться после смешения предсказания с настоящими изображениями из набора MNIST. Через 32 периода генератор обучается подделывать этот набор данных. Никто не писал программу для вывода цифр, однако машина научилась порождать цифры, неотличимые от написанных человеком. Отметим, что обучение ПСС может оказаться очень трудной задачей из-за необходимости поддерживать равновесие между обоими игроками. Если эта тема вас заинтересовала, рекомендую познакомиться с практическими приемами, описанными по адресу <https://github.com/soumith/ganhacks>.



Применение Keras adversarial для создания ПСС, подделывающей MNIST

Keras adversarial (<https://github.com/bstriner/keras-adversarial>) – написанный на Python пакет с открытым исходным кодом, предназначенный для построения ПСС. Его автор – Бен Страйнер (Ben Striner) (<https://github.com/bstriner> и <https://github.com/bstriner/keras-adversarial/blob/master/LICENSE.txt>). Поскольку версия Keras 2.0 появилась совсем недавно, я рекомендую скачать самую последнюю версию этого пакета:

```
git clone --depth=50 --branch=master
https://github.com/bstriner/keras-adversarial.git
```

И установить его:

```
python setup.py install
```

Совместимость с Keras 2.0 обсуждается по адресу <https://github.com/bstriner/keras-adversarial/issues/11>.

Если генератор G и дискриминатор D основаны на одной и той же модели M , то их можно объединить в состязательную модель; она получает тот же вход, что и M , но цели и показатели качества различны для G и D . В библиотеке определена следующая функция создания модели:

```
adversarial_model = AdversarialModel(base_model=M,
    player_params=[generator.trainable_weights, discriminator.trainable_weights],
    player_names=["generator", "discriminator"])
```

Если генератор G и дискриминатор D основаны на разных моделях, то можно воспользоваться такой функцией:

```
adversarial_model = AdversarialModel(player_models=[gan_g, gan_d],
    player_params=[generator.trainable_weights, discriminator.trainable_weights],
    player_names=["generator", "discriminator"])
```

Рассмотрим пример вычислений для MNIST:

```
import matplotlib as mpl
# Эта строка позволяет использовать mpl без определения DISPLAY
mpl.use('Agg')
```

Ниже рассматривается открытый исходный код (https://github.com/bstriner/keras-adversarial/blob/master/examples/example_gan_convolutional.py). В нем используется синтаксис Keras 1.x, но код работает

и с Keras 2.x благодаря набору вспомогательных функций в файле `legacy.py`. Содержимое файла `legacy.py` приведено в приложении, а также по адресу https://github.com/bstriner/keras-adversarial/blob/master/keras_adversarial/legacy.py.

Сначала импортируется ряд модулей. Мы уже встречались со всеми, кроме `LeakyReLU`, специальной версии `ReLU`, которая допускает малый градиент, когда нейрон не активен. Экспериментально показано, что в ряде случаев функция `LeakyReLU` может улучшить качество ПСС (см. В. Xu, N. Wang, T. Chen, M. Li «Empirical Evaluation of Rectified Activations in Convolutional Network», arXiv:1505.00853, 2014).

```
from keras.layers import Dense, Reshape, Flatten, Dropout, LeakyReLU,
    Input, Activation, BatchNormalization
from keras.models import Sequential, Model
from keras.layers.convolutional import Convolution2D, UpSampling2D
from keras.optimizers import Adam
from keras.regularizers import l1, l1l2
from keras.datasets import mnist
```

```
import pandas as pd
import numpy as np
```

Затем импортируются специальные модули для ПСС:

```
from keras_adversarial import AdversarialModel, ImageGridCallback,
    simple_gan, gan_targets
from keras_adversarial import AdversarialOptimizerSimultaneous,
    normal_latent_sampling, AdversarialOptimizerAlternating
from image_utils import dim_ordering_fix, dim_ordering_input,
    dim_ordering_reshape, dim_ordering_unfix
```

Состязательные модели обучаются в ходе игры с несколькими игроками. Если дана базовая модель с n целями и k игроками, то создается модель с $n*k$ целями, в которой каждый игрок оптимизирует потерю на своих целях. Кроме того, функция `simple_gan` порождает ПСС с заданными целями `gan_targets`. Отметим, что в библиотеке цели для генератора и дискриминатора противоположны, это стандартная практика для ПСС:

```
def gan_targets(n):
    """
    Стандартные цели обучения
    [generator_fake, generator_real, discriminator_fake, discriminator_real] = [1, 0, 0, 1]
    :param n: число примеров
    :return: массив целей
    """
    generator_fake = np.ones((n, 1))
```



```

generator_real = np.zeros((n, 1))
discriminator_fake = np.zeros((n, 1))
discriminator_real = np.ones((n, 1))
return [generator_fake, generator_real, discriminator_fake, discriminator_real]

```

Генератор в этом примере определяется так же, как мы видели раньше. Но теперь мы используем функциональный синтаксис – каждый модуль в конвейере просто передается в качестве параметра следующему модулю. Первый слой сети плотный, инициализирован в режиме `glorot_normal`. В этом режиме используется гауссов шум, масштабированный на сумму входов и выходов из узла. Аналогично инициализированы все остальные модули. Параметр `mode=2` функции `BatchNormalization` определяет попризнаковую нормировку на основе статистики каждого пакета. Экспериментально показано, что так получаются более качественные результаты:

```

def model_generator():
    nch = 256
    g_input = Input(shape=[100])
    H = Dense(nch * 14 * 14, init='glorot_normal')(g_input)
    H = BatchNormalization(mode=2)(H)
    H = Activation('relu')(H)
    H = dim_ordering_reshape(nch, 14)(H)
    H = UpSampling2D(size=(2, 2))(H)
    H = Convolution2D(int(nch / 2), 3, 3, border_mode='same',
        init='glorot_uniform')(H)
    H = BatchNormalization(mode=2, axis=1)(H)
    H = Activation('relu')(H)
    H = Convolution2D(int(nch / 4), 3, 3, border_mode='same',
        init='glorot_uniform')(H)
    H = BatchNormalization(mode=2, axis=1)(H)
    H = Activation('relu')(H)
    H = Convolution2D(1, 1, 1, border_mode='same',
        init='glorot_uniform')(H)
    g_V = Activation('sigmoid')(H)
    return Model(g_input, g_V)

```

Дискриминатор очень похож на определенный нами выше. Единственное различие – модуль `LeakyReLU`:

```

def model_discriminator(input_shape=(1, 28, 28), dropout_rate=0.5):
    d_input = dim_ordering_input(input_shape, name="input_x")
    nch = 512
    H = Convolution2D(int(nch / 2), 5, 5, subsample=(2, 2),
        border_mode='same', activation='relu')(d_input)
    H = LeakyReLU(0.2)(H)

```



```

H = Dropout(dropout_rate) (H)
H = Convolution2D(nch, 5, 5, subsample=(2, 2),
    border_mode='same', activation='relu') (H)
H = LeakyReLU(0.2) (H)
H = Dropout(dropout_rate) (H)
H = Flatten() (H)
H = Dense(int(nch / 2)) (H)
H = LeakyReLU(0.2) (H)
H = Dropout(dropout_rate) (H)
d_V = Dense(1, activation='sigmoid') (H)
return Model(d_input, d_V)

```

Далее следуют две простые функции для загрузки и нормировки набора данных MNIST:

```

def mnist_process(x):
    x = x.astype(np.float32) / 255.0
    return x

def mnist_data():
    (xtrain, ytrain), (xtest, ytest) = mnist.load_data()
    return mnist_process(xtrain), mnist_process(xtest)

```

На следующем шаге определяется совместная модель ПСС в виде комбинации генератора и дискриминатора. Заметим, что веса инициализируются функцией `normal_latent_sampling`, которая производит выборку из нормального распределения:

```

if __name__ == "__main__":
    # z принадлежит R^100
    latent_dim = 100
    # x принадлежит R^{28x28}
    input_shape = (1, 28, 28)
    # генератор (z -> x)
    generator = model_generator()
    # дискриминатор (x -> y)
    discriminator = model_discriminator(input_shape=input_shape)
    # ПСС (x -> yfake, yreal), z генерируется на GPU
    gan = simple_gan(generator, discriminator, normal_latent_sampling((latent_dim,)))
    # печатать общие сведения о моделях
    generator.summary()
    discriminator.summary()
    gan.summary()

```

Затем в примере создается ПСС и компилируется модель, обученная с использованием оптимизатора Adam и функции потерь `binary_crossentropy`:

```

# построить состязательную модель
model = AdversarialModel(base_model=gan,

```

```

player_params=[generator.trainable_weights, discriminator.trainable_weights],
player_names=["generator", "discriminator"])
model.adversarial_compile(adversarial_optimizer=AdversarialOptimizerSimultaneous(),
player_optimizers=[Adam(1e-4, decay=1e-4), Adam(1e-3, decay=1e-4)],
loss='binary_crossentropy')

```

Определяется генератор, который создает новые изображения, похожие на настоящие. В каждом периоде обучения генерируется новое поддельное изображение:

```

def generator_sampler():
    zsamples = np.random.normal(size=(10 * 10, latent_dim))
    gen = dim_ordering_unfix(generator.predict(zsamples))
    return gen.reshape((10, 10, 28, 28))

generator_cb = ImageGridCallback(
    "output/gan_convolutional/epoch-{:03d}.png", generator_sampler)
xtrain, xtest = mnist_data()
xtrain = dim_ordering_fix(xtrain.reshape((-1, 1, 28, 28)))
xtest = dim_ordering_fix(xtest.reshape((-1, 1, 28, 28)))
y = gan_targets(xtrain.shape[0])
ytest = gan_targets(xtest.shape[0])
history = model.fit(x=xtrain, y=y,
    validation_data=(xtest, ytest), callbacks=[generator_cb], nb_epoch=100,
    batch_size=32)
df = pd.DataFrame(history.history)
df.to_csv("output/gan_convolutional/history.csv")
generator.save("output/gan_convolutional/generator.h5")
discriminator.save("output/gan_convolutional/discriminator.h5")

```

Здесь `dim_ordering_unfix` – вспомогательная функция для поддержки различных форматов изображений. Она определена в файле `image_utils.py` и выглядит следующим образом:

```

def dim_ordering_fix(x):
    if K.image_dim_ordering() == 'th':
        return x
    else:
        return np.transpose(x, (0, 2, 3, 1))

```

Теперь запустим программу и посмотрим, как изменяется потеря генератора и дискриминатора. На следующем снимке экрана распечатаны характеристики обеих сетей:

```

examples — python example_gan_convolutional.py — 140x75
~/Keras/codeBook/keras-dogan — bash
~/examples — python example_gan_convolutional.py
gull1:~/examples gull1$ python example_gan_convolutional.py
Using TensorFlow backend.
/Users/gull1/miniconda2/lib/python2.7/site-packages/keras_adversarial/legacy.py:48: UserWarning: Update yo
ur 'Conv2D' call to the Keras 2 API: 'Conv2D(256, (5, 5), padding="same", strides=(2, 2), activation="relu", kernel_initializer="glorot_unif
orm", kernel_regularizer=None)'
/Users/gull1/miniconda2/lib/python2.7/site-packages/keras_adversarial/legacy.py:48: UserWarning: Update yo
ur 'Conv2D' call to the Keras 2 API: 'Conv2D(512, (5, 5), padding="same", strides=(2, 2), activation="relu", kernel_initializer="glorot_unif
orm", kernel_regularizer=None)'

Layer (type)                 Output Shape                 Param #
=====
input_1 (InputLayer)         (None, 100)                  0
dense_1 (Dense)              (None, 50176)                5067776
batch_normalization_1 (Batch Normalization) (None, 50176)                200784
activation_1 (Activation)     (None, 50176)                0
reshape_1 (Reshape)          (None, 14, 14, 256)         0
up_sampling2d_1 (UpSampling2D) (None, 28, 28, 256)         0
conv2d_1 (Conv2D)            (None, 28, 28, 128)         295040
batch_normalization_2 (Batch Normalization) (None, 28, 28, 128)         112
activation_2 (Activation)     (None, 28, 28, 128)         0
conv2d_2 (Conv2D)            (None, 28, 28, 64)          73792
batch_normalization_3 (Batch Normalization) (None, 28, 28, 64)          112
activation_3 (Activation)     (None, 28, 28, 64)          0
conv2d_3 (Conv2D)            (None, 28, 28, 1)           65
activation_4 (Activation)     (None, 28, 28, 1)           0
=====
Total params: 5,637,601.0
Trainable params: 5,537,137.0
Non-trainable params: 100,464.0

Layer (type)                 Output Shape                 Param #
=====
input_x (InputLayer)         (None, 28, 28, 1)           0
conv2d_4 (Conv2D)            (None, 14, 14, 256)         6656
leaky_re_lu_1 (LeakyReLU)    (None, 14, 14, 256)         0
dropout_1 (Dropout)          (None, 14, 14, 256)         0
conv2d_5 (Conv2D)            (None, 7, 7, 512)           3277312
leaky_re_lu_2 (LeakyReLU)    (None, 7, 7, 512)           0
dropout_2 (Dropout)          (None, 7, 7, 512)           0
flatten_1 (Flatten)          (None, 25088)                0
dense_2 (Dense)              (None, 256)                  6422784
leaky_re_lu_3 (LeakyReLU)    (None, 256)                  0
dropout_3 (Dropout)          (None, 256)                  0
dense_3 (Dense)              (None, 1)                    257
=====
Total params: 9,707,009.0
Trainable params: 9,707,009.0
Non-trainable params: 0.0
    
```

А на этом снимке мы видим количество примеров, использованных для обучения и для контроля:

```

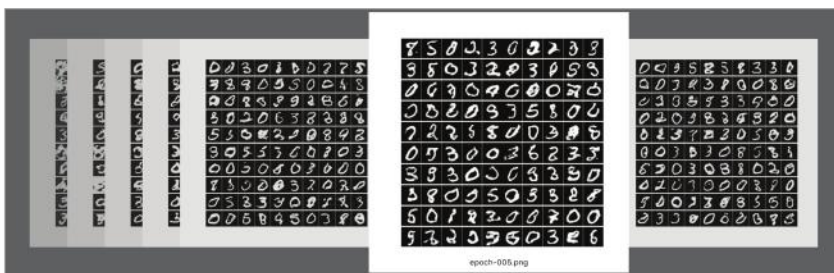
examples --- -bash --- 123x35
~/Keras/codeBook/keras_adversarial/examples --- -bash
~/Keras/codeBook/new/keras_adversarial/examples --- -bash ... +

Trainable params: 15,244,418
Non-trainable params: 100,736

Train on 60000 samples, validate on 10000 samples
Epoch 1/100
60000/60000 [=====] - 7579s - loss: 18.6313 - generator_loss: 18.4795 - generator_yfake_loss: 10.7765 - generator_yreal_loss: 7.7030 - discriminator_loss: 0.1519 - discriminator_yfake_loss: 0.0793 - discriminator_yreal_loss: 0.0726 - val_loss: 16.2338 - val_generator_loss: 15.9988 - val_generator_yfake_loss: 7.5463 - val_generator_yreal_loss: 8.4445 - val_discriminator_loss: 0.2431 - val_discriminator_yfake_loss: 0.2044 - val_discriminator_yreal_loss: 0.0386
Epoch 2/100
60000/60000 [=====] - 7737s - loss: 14.5333 - generator_loss: 14.2141 - generator_yfake_loss: 7.2208 - generator_yreal_loss: 6.9933 - discriminator_loss: 0.3192 - discriminator_yfake_loss: 0.1523 - discriminator_yreal_loss: 0.1668 - val_loss: 13.6769 - val_generator_loss: 13.4410 - val_generator_yfake_loss: 6.0405 - val_generator_yreal_loss: 7.4006 - val_discriminator_loss: 0.2359 - val_discriminator_yfake_loss: 0.1093 - val_discriminator_yreal_loss: 0.1265
Epoch 3/100
60000/60000 [=====] - 7842s - loss: 11.6130 - generator_loss: 11.2356 - generator_yfake_loss: 4.0883 - generator_yreal_loss: 7.1473 - discriminator_loss: 0.3775 - discriminator_yfake_loss: 0.1708 - discriminator_yreal_loss: 0.2067 - val_loss: 10.7820 - val_generator_loss: 10.3857 - val_generator_yfake_loss: 2.3510 - val_generator_yreal_loss: 8.0348 - val_discriminator_loss: 0.3963 - val_discriminator_yfake_loss: 0.3050 - val_discriminator_yreal_loss: 0.0913
Epoch 4/100
60000/60000 [=====] - 7567s - loss: 9.6041 - generator_loss: 9.1363 - generator_yfake_loss: 3.1345 - generator_yreal_loss: 6.0018 - discriminator_loss: 0.4678 - discriminator_yfake_loss: 0.2147 - discriminator_yreal_loss: 0.2531 - val_loss: 7.8728 - val_generator_loss: 7.4582 - val_generator_yfake_loss: 2.1574 - val_generator_yreal_loss: 5.3007 - val_discriminator_loss: 0.4146 - val_discriminator_yfake_loss: 0.2797 - val_discriminator_yreal_loss: 0.1350
Epoch 5/100
60000/60000 [=====] - 7971s - loss: 9.0191 - generator_loss: 8.5203 - generator_yfake_loss: 2.9953 - generator_yreal_loss: 5.5250 - discriminator_loss: 0.4988 - discriminator_yfake_loss: 0.2289 - discriminator_yreal_loss: 0.2699 - val_loss: 7.3837 - val_generator_loss: 6.9287 - val_generator_yfake_loss: 1.9533 - val_generator_yreal_loss: 4.9756 - val_discriminator_loss: 0.4558 - val_discriminator_yfake_loss: 0.3175 - val_discriminator_yreal_loss: 0.1375
Epoch 6/100
60000/60000 [=====] - 7333s - loss: 8.7654 - generator_loss: 8.2540 - generator_yfake_loss: 2.9357 - generator_yreal_loss: 5.3184 - discriminator_loss: 0.5114 - discriminator_yfake_loss: 0.2354 - discriminator_yreal_loss: 0.2759 - val_loss: 7.4316 - val_generator_loss: 6.9901 - val_generator_yfake_loss: 2.2178 - val_generator_yreal_loss: 4.7722 - val_discriminator_loss: 0.4416 - val_discriminator_yfake_loss: 0.2494 - val_discriminator_yreal_loss: 0.1922
Epoch 7/100

```

После 5–6 итераций мы уже имеем искусственные изображения приемлемого качества, т. е. компьютер обучился порождать рукописные цифры:



Применение Keras adversarial для создания ПСС, подделывающей CIFAR

Теперь применим ПСС для подделывания набора данных CIFAR-10 и получения синтетических изображений, похожих на настоящие. Исходный код находится по адресу https://github.com/bstriner/keras-adversarial/blob/master/examples/example_gan_cifar10.py. Как и раньше, используется синтаксис Keras 1.x, но благодаря вспомогательным

функциям из файла `legasy.py` код работает и для Keras 2.x. Сначала импортируется ряд пакетов:

```
import matplotlib as mpl
# Эта строка позволяет использовать mpl без определения DISPLAY
mpl.use('Agg')
import pandas as pd
import numpy as np
import os
from keras.layers import Dense, Reshape, Flatten, Dropout, LeakyReLU,
    Activation, BatchNormalization, SpatialDropout2D
from keras.layers.convolutional import Convolution2D, UpSampling2D,
    MaxPooling2D, AveragePooling2D
from keras.models import Sequential, Model
from keras.optimizers import Adam
from keras.callbacks import TensorBoard
from keras.regularizers import l1l2
from keras_adversarial import AdversarialModel, ImageGridCallback,
    simple_gan, gan_targets
from keras_adversarial import AdversarialOptimizerSimultaneous,
    normal_latent_sampling, fix_names
import keras.backend as K
from cifar10_utils import cifar10_data
from image_utils import dim_ordering_fix, dim_ordering_unfix,
    dim_ordering_shape
```

Затем определяется генератор, в котором используется комбинация сверток с регуляризацией по норме L1 и L2, пакетной нормировкой и повышающей передискретизацией. Параметр `axis=1` говорит, по какому измерению тензора производить нормировку сначала, а `mode=0` означает попризнаковую нормировку. Эта конкретная сеть стала результатом многочисленных экспериментов, но по сути дела она по-прежнему является последовательностью операций двумерной свертки и повышающей передискретизации, в начале которой используется модуль `Dense`, а в конце – функция активации `sigmoid`. Кроме того, во всех свертках применяется функция активации `LeakyReLU` и пакетная нормировка `BatchNormalization`:

```
def model_generator():
    model = Sequential()
    nch = 256
    reg = lambda: l1l2(l1=1e-7, l2=1e-7)
    h = 5
    model.add(Dense(input_dim=100, output_dim=nch*4*4, W_regularizer=reg()))
    model.add(BatchNormalization(mode=0))
    model.add(Reshape(dim_ordering_shape((nch, 4, 4))))
    model.add(Convolution2D(nch/2, h, h, border_mode='same', W_regularizer=reg()))
```



```

model.add(BatchNormalization(mode=0, axis=1))
model.add(LeakyReLU(0.2))
model.add(UpSampling2D(size=(2, 2)))
model.add(Convolution2D(nch / 2, h, h, border_mode='same', W_regularizer=reg()))
model.add(BatchNormalization(mode=0, axis=1))
model.add(LeakyReLU(0.2))
model.add(UpSampling2D(size=(2, 2)))
model.add(Convolution2D(nch / 4, h, h, border_mode='same', W_regularizer=reg()))
model.add(BatchNormalization(mode=0, axis=1))
model.add(LeakyReLU(0.2))
model.add(UpSampling2D(size=(2, 2)))
model.add(Convolution2D(3, h, h, border_mode='same', W_regularizer=reg()))
model.add(Activation('sigmoid'))
return model

```

Затем определяется дискриминатор. И снова мы имеем последовательность операций двумерной свертки, но в этом случае используется модуль `SpatialDropout2D`, который отбрасывает целые двумерные карты признаков, а не отдельные элементы. По тем же причинам используются модули `MaxPooling2D` и `AveragePooling2D`:

```

def model_discriminator():
    nch = 256
    h = 5
    reg = lambda: l1l2(l1=1e-7, l2=1e-7)
    c1 = Convolution2D(nch / 4, h, h, border_mode='same', W_regularizer=reg(),
        input_shape=dim_ordering_shape((3, 32, 32)))
    c2 = Convolution2D(nch / 2, h, h, border_mode='same', W_regularizer=reg())
    c3 = Convolution2D(nch, h, h, border_mode='same', W_regularizer=reg())
    c4 = Convolution2D(1, h, h, border_mode='same', W_regularizer=reg())

    def m(dropout):
        model = Sequential()
        model.add(c1)
        model.add(SpatialDropout2D(dropout))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(LeakyReLU(0.2))
        model.add(c2)
        model.add(SpatialDropout2D(dropout))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(LeakyReLU(0.2))
        model.add(c3)
        model.add(SpatialDropout2D(dropout))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(LeakyReLU(0.2))
        model.add(c4)
        model.add(AveragePooling2D(pool_size=(4, 4), border_mode='valid'))
        model.add(Flatten())
        model.add(Activation('sigmoid'))

```

```

    return model

return m

```

Теперь можно построить и сами ПСС. Следующая функция принимает несколько аргументов, включая генератор, дискриминатор, число латентных измерений и цели ПСС:

```

def example_gan(adversarial_optimizer, path, opt_g, opt_d, nb_epoch,
               generator, discriminator, latent_dim, targets=gan_targets,
               loss='binary_crossentropy'):
    csvpath = os.path.join(path, "history.csv")
    if os.path.exists(csvpath):
        print("Already exists: {}".format(csvpath))
    return

```

Далее создаются две ПСС, в одной из них дискриминатор с про-реживанием, в другой – без:

```

print("Training: {}".format(csvpath))
# ПСС (x -> yfake, yreal), z генерируются на GPU
# можно также поэкспериментировать с uniform_latent_sampling
d_g = discriminator(0)
d_d = discriminator(0.5)
generator.summary()
d_d.summary()
gan_g = simple_gan(generator, d_g, None)
gan_d = simple_gan(generator, d_d, None)
x = gan_g.inputs[1]
z = normal_latent_sampling((latent_dim,))(x)
# исключить z из входных данных
gan_g = Model([x], fix_names(gan_g([z, x]), gan_g.output_names))
gan_d = Model([x], fix_names(gan_d([z, x]), gan_d.output_names))

```

Теперь обе ПСС комбинируются в состязательную модель с раз-дельными весами, и модель компилируется:

```

# построить состязательную модель
model = AdversarialModel(player_models=[gan_g, gan_d],
                        player_params=[generator.trainable_weights, d_d.trainable_weights],
                        player_names=["generator", "discriminator"])
model.adversarial_compile(adversarial_optimizer=adversarial_optimizer,
                        player_optimizers=[opt_g, opt_d], loss=loss)

```

А это простая функция обратного вызова для выборки изображений и записи их в файл.

```

# создать обратный вызов для генерации изображений
zsamples = np.random.normal(size=(10 * 10, latent_dim))

```

```
def generator_sampler():
    xpred = dim_ordering_unfix(generator.predict(zsamples)).transpose((0, 2, 3, 1))
    return xpred.reshape((10, 10) + xpred.shape[1:])

generator_cb = ImageGridCallback(os.path.join(path, "epoch-{:03d}.png"),
    generator_sampler, cmap=None)
```

Теперь настало время загрузить набор данных CIFAR-10 и обучить модель. Если в качестве базовой библиотеки используется TensorFlow, то информация о потере сохраняется в TensorBoard, чтобы можно было следить за тем, как потеря со временем уменьшается. История хранится в формате CSV, а веса модели – в формате h5:

```
# обучить модель
xtrain, xtest = cifar10_data()
y = targets(xtrain.shape[0])
ytest = targets(xtest.shape[0])
callbacks = [generator_cb]
if K.backend() == "tensorflow":
    callbacks.append(TensorBoard(log_dir=os.path.join(path, 'logs'),
        histogram_freq=0, write_graph=True, write_images=True))
history = model.fit(x=dim_ordering_fix(xtrain), y=y,
    validation_data=(dim_ordering_fix(xtest), ytest),
    callbacks=callbacks, nb_epoch=nb_epoch, batch_size=32)
# сохранить историю в формате CSV
df = pd.DataFrame(history.history)
df.to_csv(csvpath)
# сохранить модели
generator.save(os.path.join(path, "generator.h5"))
d_d.save(os.path.join(path, "discriminator.h5"))
```

И наконец, можно запустить всю сеть. Генератор производит выборку из пространства со 100 латентными измерениями, а в качестве оптимизатора для обеих ПСС выбран Adam:

```
def main():
    # z принадлежит  $R^{100}$ 
    latent_dim = 100
    # x принадлежит  $R^{28 \times 28}$ 
    # генератор (z -> x)
    generator = model_generator()
    # дискриминатор (x -> y)
    discriminator = model_discriminator()
    example_gan(AdversarialOptimizerSimultaneous(), "output/gan-cifar10",
        opt_g=Adam(1e-4, decay=1e-5),
        opt_d=Adam(1e-3, decay=1e-5),
        nb_epoch=100, generator=generator, discriminator=discriminator,
```

```
latent_dim=latent_dim)

if __name__ == "__main__":
    main()
```

Чтобы составить полное представление об исходном коде, нужно еще рассмотреть несколько вспомогательных функций для сохранения сетки изображений:

```
from matplotlib import pyplot as plt, gridspec
import os

def write_image_grid(filepath, imgs, figsize=None, cmap='gray'):
    directory = os.path.dirname(filepath)
    if not os.path.exists(directory):
        os.makedirs(directory)
    fig = create_image_grid(imgs, figsize, cmap=cmap)
    fig.savefig(filepath)
    plt.close(fig)

def create_image_grid(imgs, figsize=None, cmap='gray'):
    n = imgs.shape[0]
    m = imgs.shape[1]
    if figsize is None:
        figsize=(n,m)
    fig = plt.figure(figsize=figsize)
    gsl = gridspec.GridSpec(n, m)
    gsl.update(wspace=0.025, hspace=0.025) # set the spacing between axes.
    for i in range(n):
        for j in range(m):
            ax = plt.subplot(gsl[i, j])
            img = imgs[i, j, :]
            ax.imshow(img, cmap=cmap)
            ax.axis('off')
    return fig
```

Кроме того, необходимы методы для работы с различными форматами изображений (применяемыми в Theano и TensorFlow):

```
import keras.backend as K
import numpy as np
from keras.layers import Input, Reshape

def dim_ordering_fix(x):
    if K.image_dim_ordering() == 'th':
        return x
    else:
        return np.transpose(x, (0, 2, 3, 1))

def dim_ordering_unfix(x):
```

```
if K.image_dim_ordering() == 'th':
    return x
else:
    return np.transpose(x, (0, 3, 1, 2))

def dim_ordering_shape(input_shape):
    if K.image_dim_ordering() == 'th':
        return input_shape
    else:
        return (input_shape[1], input_shape[2], input_shape[0])

def dim_ordering_input(input_shape, name):
    if K.image_dim_ordering() == 'th':
        return Input(input_shape, name=name)
    else:
        return Input((input_shape[1], input_shape[2], input_shape[0]), name=name)

def dim_ordering_reshape(k, w, **kwargs):
    if K.image_dim_ordering() == 'th':
        return Reshape((k, w, w), **kwargs)
    else:
        return Reshape((w, w, k), **kwargs)

# И еще одна функция для исправления имен
def fix_names(outputs, names):
    if not isinstance(outputs, list):
        outputs = [outputs]
    if not isinstance(names, list):
        names = [names]
    return [Activation('linear', name=name)(output)
            for output, name in zip(outputs, names)]
```

На следующем снимке экрана показаны характеристики определенных сетей:

examples — python example_gan_cifar10.py — 140x78

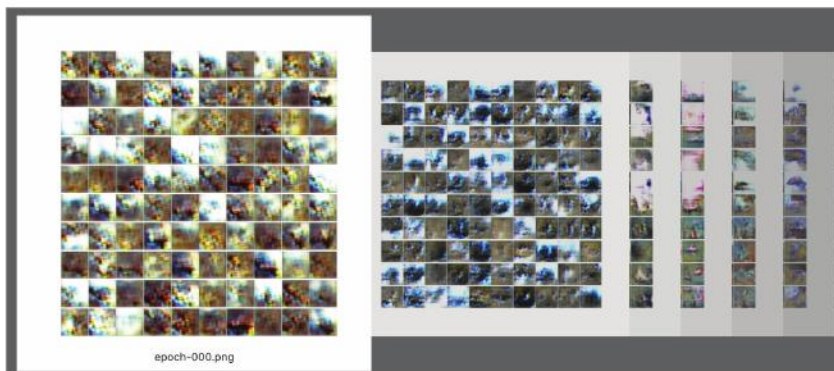
gull-macbookpro:examples gullis python example_gan_cifar10.py
Using TensorFlow backend.
Training: output/gen-cifar10/history.csv

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 4096)	413696
batch_normalization_1 (Batch Normalization)	(None, 4096)	16384
reshape_1 (Reshape)	(None, 256, 4, 4)	0
conv2d_1 (Conv2D)	(None, 128, 4, 4)	819328
batch_normalization_2 (Batch Normalization)	(None, 128, 4, 4)	512
leaky_re_lu_1 (LeakyReLU)	(None, 128, 4, 4)	0
up_sampling2d_1 (UpSampling2D)	(None, 128, 8, 8)	0
conv2d_2 (Conv2D)	(None, 128, 8, 8)	489728
batch_normalization_3 (Batch Normalization)	(None, 128, 8, 8)	512
leaky_re_lu_2 (LeakyReLU)	(None, 128, 8, 8)	0
up_sampling2d_2 (UpSampling2D)	(None, 128, 16, 16)	0
conv2d_3 (Conv2D)	(None, 64, 16, 16)	264864
batch_normalization_4 (Batch Normalization)	(None, 64, 16, 16)	256
leaky_re_lu_3 (LeakyReLU)	(None, 64, 16, 16)	0
up_sampling2d_3 (UpSampling2D)	(None, 64, 32, 32)	0
conv2d_4 (Conv2D)	(None, 3, 32, 32)	4803
activation_1 (Activation)	(None, 3, 32, 32)	0
Total params: 1,878,883.0		
Trainable params: 1,861,251.0		
Non-trainable params: 8,832.0		

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 64, 32, 32)	4864
max_pooling2d_1 (MaxPooling2D)	(None, 64, 16, 16)	0
leaky_re_lu_4 (LeakyReLU)	(None, 64, 16, 16)	0
conv2d_6 (Conv2D)	(None, 128, 16, 16)	264928
max_pooling2d_2 (MaxPooling2D)	(None, 128, 8, 8)	0
leaky_re_lu_5 (LeakyReLU)	(None, 128, 8, 8)	0
conv2d_7 (Conv2D)	(None, 256, 8, 8)	819456
max_pooling2d_3 (MaxPooling2D)	(None, 256, 4, 4)	0
leaky_re_lu_6 (LeakyReLU)	(None, 256, 4, 4)	0
conv2d_8 (Conv2D)	(None, 1, 4, 4)	6401
average_pooling2d_1 (Average Pooling2D)	(None, 1, 1, 1)	0
flatten_1 (Flatten)	(None, 1)	0
activation_2 (Activation)	(None, 1)	0
Total params: 1,835,649.0		
Trainable params: 1,835,649.0		
Non-trainable params: 0.0		

Train on 50000 samples, validate on 10000 samples

Если запустить эту программу, то на первой итерации будут генерироваться нереалистичные изображения. Но после 99 итераций сеть обучается порождать изображения, которые выглядят, как настоящие изображения из набора CIFAR-10:



Ниже мы видим настоящие изображения из набора CIFAR-10 справа и подделки слева:



WaveNet – порождающая модель для обучения генерации звука

WaveNet – глубокая порождающая модель для генерации звуковых сигналов. Эта прорывная технология была изобретена компанией Google DeepMind (<https://deepmind.com/blog/wavenet-generative-model-raw-audio/>) для обучения пользователей тому, как говорить с компьютерами. Результаты поистине впечатляют, вы можете найти в сети примеры того, как компьютер говорит голосами знаменитостей, например, Мэтта Деймона. Спрашивается, почему синтез

аудиосигналов – такая трудная задача? Слышимый нами цифровой звук имеет частоту 16 000 отсчетов в секунду (иногда 48 000 и даже больше), и построить прогностическую модель, которая обучается воспроизводить отсчет на основе всех предыдущих – очень непростое дело. Тем не менее эксперименты показывают, что WaveNet улучшила качество систем **речевого воспроизведения текста** (text-to-speech, TTS), уменьшив различие с человеческим голосом на 50 % для американского варианта английского языка и мандаринского диалекта китайского. Более того, DeepMind доказала, что WaveNet можно также использовать для генерации звучания музыкальных инструментов, в частности фортепьяно. Теперь пора дать несколько определений. TTS-системы обычно делят на два класса:

- **Компиляционный синтез.** Каждый фрагмент произнесенной речи сначала запоминается, а затем восстанавливается, когда нужно воспроизвести голос. Однако этот подход не масштабируется, потому что воспроизвести возможно только ранее запомненные фрагменты и нельзя воспроизвести новые голоса или другие типы звучания, для которых нет запомненных фрагментов.
- **Параметрический синтез.** В этом случае создается модель для хранения всех характерных признаков синтезируемого аудио. До появления WaveNet звучание, генерируемое параметрическими TTS-системами, было менее естественным, чем у компиляционных. WaveNet удалось улучшить качество за счет прямого моделирования порождения звука вместо применения промежуточных алгоритмов обработки сигналов.

В принципе WaveNet можно рассматривать как стек одномерных сверточных слоев (с двумерной сверткой мы познакомились в главе 3) с постоянным шагом 1 и без слоев пулинга. Отметим, что вход и выход по построению имеют одинаковый размер, поэтому сверточная сеть прекрасно подходит для моделирования таких последовательных данных, как звуковые сигналы. Однако было показано, что для достижения большого размера рецептивного поля выходного нейрона (напомним, что рецептивным полем нейрона называется множество нейронов предыдущего слоя, от которых данный нейрон получает входные сигналы) необходимо либо иметь много больших фильтров, либо чрезмерно

увеличить глубину сети. По этой причине чистые сверточные сети не слишком эффективны для обучения синтезу звука. Основная идея WaveNet – каузальная свертка с пропусками (dilated causal convolution) (см. статью Fisher Yu, Vladlen Koltun «Multi-Scale Context Aggregation by Dilated Convolutions», 2016, доступную по адресу <https://www.semanticscholar.org/paper/Multi-Scale-Context-Aggregation-by-Dilated-Yu-Koltun/420c46d7cafc841309f02ad04cf51cblf190a48>), иногда называемая дырчатой (atrous) сверткой (atrous – вульгаризация французского выражения *à trous*, означающего «с дырами»). Это просто означает, что при применении фильтра в сверточном слое некоторые входные значения пропускаются. Например, в одномерном случае фильтр w ширины 3 с пропуском 1 вычисляет следующую сумму:

$$w[0]x[0] + w[1]x[2] + w[3]x[4]$$

Благодаря этой простой идее дырок становится возможным собрать несколько дырчатых сверточных слоев с экспоненциально увеличивающимися фильтрами и распространить область влияния входных нейронов, не увеличивая глубину сети до запредельных значений. Таким образом, WaveNet – это сверточная сеть, слои которой имеют разные коэффициенты пропуска, в результате чего рецептивное поле может экспоненциально расти с увеличением глубины и, следовательно, покрывать тысячи временных шагов звукового сигнала. Во время обучения входными данными являются звуки записанной человеческой речи. Сигналы подвергаются дискретизации с фиксированным целочисленным шагом. WaveNet определяет начальный сверточный слой, имеющий доступ только к текущему и предыдущему входу. В конце получается последовательность плотных слоев, комбинирующих предыдущие результаты, за которыми следует слой с функцией активации softmax для порождения категориальных выходов. На каждом шаге значение, предсказанное сетью, подается обратно на вход. В то же время вычисляется новое предсказание для следующего шага. В роли функции потерь выступает перекрестная энтропия между выходом текущего шага и входом следующего. По адресу <https://github.com/basveeling/wavenet> размещена реализация на Keras, разработанная Басом Веелингом (Bas Veeling). Ее легко установить с помощью git:

```
pip install virtualenv
mkdir ~/virtualenvs && cd ~/virtualenvs
virtualenv wavenet
source wavenet/bin/activate
```

```
cd ~
git clone https://github.com/basveeling/wavenet.git
cd wavenet
pip install -r requirements.txt
```

Этот код написан для Keras 1.x, информация о ходе его переноса на Keras 2.x публикуется по адресу <https://github.com/basveeling/wavenet/issues/29>. Процедура обучения очень проста, но требует больших вычислительных ресурсов (убедитесь, что имеется адекватная поддержка GPU).

```
$ python wavenet.py with 'data_dir=your_data_dir_name'
```

Генерация звуковых сигналов после обучения ничуть не сложнее:

```
python wavenet.py predict with 'models/[run_folder]/config.json predict_seconds=1'
```

В Интернете можно найти сведения о многочисленных гиперпараметрах для настройки процесса обучения. Сеть получается весьма глубокой, как следует из приведенной ниже распечатки внутренних слоев. Заметим, что входной звуковой сигнал дискретизируется с параметрами `fragment_length = 1152` и `nb_output_bins = 256`, это и есть тензор, подаваемый WaveNet. Сеть WaveNet устроена в виде повторяющихся блоков, называемых остатками (residual). Каждый блок является объединением в режиме умножения двух модулей свертки с пропусками (в одном функция активация – сигмоида, в другом – tanh) с последующим объединением в режиме суммы с модулем одномерной свертки. Отметим, что размер дырок в свертках с пропусками экспоненциально возрастает (равен $2 \cdot i$) от 1 до 512, как видно из следующего фрагмента кода:

```
def residual_block(x):
    original_x = x
    tanh_out = CausalAtrousConvolution1D(nb_filters, 2, atrous_rate=2 ** i,
        border_mode='valid', causal=True, bias=use_bias,
        name='dilated_conv_%d_tanh_s%d' % (2 ** i, s), activation='tanh',
        W_regularizer=l2(res_l2))(x)
    sigm_out = CausalAtrousConvolution1D(nb_filters, 2, atrous_rate=2 ** i,
        border_mode='valid', causal=True, bias=use_bias,
        name='dilated_conv_%d_sigm_s%d' % (2 ** i, s), activation='sigmoid',
        W_regularizer=l2(res_l2))(x)
    x = layers.Merge(mode='mul',
        name='gated_activation_%d_s%d' % (i, s))([tanh_out, sigm_out])
    res_x = layers.Convolution1D(nb_filters, 1, border_mode='same', bias=use_bias,
        W_regularizer=l2(res_l2))(x)
    skip_x = layers.Convolution1D(nb_filters, 1, border_mode='same', bias=use_bias,
        W_regularizer=l2(res_l2))(x)
```



```
res_x = layers.Merge(mode='sum')([original_x, res_x])
return res_x, skip_x
```

После остаточного блока с пропусками идет последовательность объединенных сверточных модулей, за ними два сверточных модуля и за ними выходной слой с функцией активации softmax, порождающий `nb_output_bins` категорий. Вот полная структура сети:

```
Layer (type) Output Shape Param # Connected to
```

```
=====
```

```
input_part (InputLayer) (None, 1152, 256) 0
```

```
=====
```

```
initial_causal_conv (CausalAtrou) (None, 1152, 256) 131328 input_part[0][0]
```

```
=====
```

```
dilated_conv_1_tanh_s0 (CausalAt (None, 1152, 256) 131072
initial_causal_conv[0][0]
```

```
=====
```

```
dilated_conv_1_sigm_s0 (CausalAt (None, 1152, 256) 131072
initial_causal_conv[0][0]
```

```
=====
```

```
gated_activation_0_s0 (Merge) (None, 1152, 256) 0
dilated_conv_1_tanh_s0[0][0]
dilated_conv_1_sigm_s0[0][0]
```

```
=====
```

```
convolution1d_1 (Convolution1D) (None, 1152, 256) 65536
gated_activation_0_s0[0][0]
```

```
=====
```

```
merge_1 (Merge) (None, 1152, 256) 0 initial_causal_conv[0][0]
convolution1d_1[0][0]
```

```
=====
```

```
dilated_conv_2_tanh_s0 (CausalAt (None, 1152, 256) 131072 merge_1[0][0]
```

```
=====
```

```
dilated_conv_2_sigm_s0 (CausalAt (None, 1152, 256) 131072 merge_1[0][0]
```

```
=====
```

```
gated_activation_1_s0 (Merge) (None, 1152, 256) 0
dilated_conv_2_tanh_s0[0][0]
dilated_conv_2_sigm_s0[0][0]
```

convolution1d_3 (Convolution1D) (None, 1152, 256) 65536
gated_activation_1_s0[0][0]

merge_2 (Merge) (None, 1152, 256) 0 merge_1[0][0]
convolution1d_3[0][0]

dilated_conv_4_tanh_s0 (CausalAt (None, 1152, 256) 131072 merge_2[0][0]

dilated_conv_4_sigm_s0 (CausalAt (None, 1152, 256) 131072 merge_2[0][0]

gated_activation_2_s0 (Merge) (None, 1152, 256) 0
dilated_conv_4_tanh_s0[0][0]
dilated_conv_4_sigm_s0[0][0]

convolution1d_5 (Convolution1D) (None, 1152, 256) 65536
gated_activation_2_s0[0][0]

merge_3 (Merge) (None, 1152, 256) 0 merge_2[0][0]
convolution1d_5[0][0]

dilated_conv_8_tanh_s0 (CausalAt (None, 1152, 256) 131072 merge_3[0][0]

dilated_conv_8_sigm_s0 (CausalAt (None, 1152, 256) 131072 merge_3[0][0]

gated_activation_3_s0 (Merge) (None, 1152, 256) 0
dilated_conv_8_tanh_s0[0][0]
dilated_conv_8_sigm_s0[0][0]

convolution1d_7 (Convolution1D) (None, 1152, 256) 65536
gated_activation_3_s0[0][0]

merge_4 (Merge) (None, 1152, 256) 0 merge_3[0][0]
convolution1d_7[0][0]

dilated_conv_16_tanh_s0 (CausalAt (None, 1152, 256) 131072 merge_4[0][0]

dilated_conv_16_sigm_s0 (CausalA (None, 1152, 256) 131072 merge_4[0][0])

gated_activation_4_s0 (Merge) (None, 1152, 256) 0

dilated_conv_16_tanh_s0[0][0]

dilated_conv_16_sigm_s0[0][0]

convolution1d_9 (Convolution1D) (None, 1152, 256) 65536

gated_activation_4_s0[0][0]

merge_5 (Merge) (None, 1152, 256) 0 merge_4[0][0]

convolution1d_9[0][0]

dilated_conv_32_tanh_s0 (CausalA (None, 1152, 256) 131072 merge_5[0][0])

dilated_conv_32_sigm_s0 (CausalA (None, 1152, 256) 131072 merge_5[0][0])

gated_activation_5_s0 (Merge) (None, 1152, 256) 0

dilated_conv_32_tanh_s0[0][0]

dilated_conv_32_sigm_s0[0][0]

convolution1d_11 (Convolution1D) (None, 1152, 256) 65536

gated_activation_5_s0[0][0]

merge_6 (Merge) (None, 1152, 256) 0 merge_5[0][0]

convolution1d_11[0][0]

dilated_conv_64_tanh_s0 (CausalA (None, 1152, 256) 131072 merge_6[0][0])

dilated_conv_64_sigm_s0 (CausalA (None, 1152, 256) 131072 merge_6[0][0])

gated_activation_6_s0 (Merge) (None, 1152, 256) 0

dilated_conv_64_tanh_s0[0][0]

dilated_conv_64_sigm_s0[0][0]

convolution1d_13 (Convolution1D) (None, 1152, 256) 65536

gated_activation_6_s0[0][0]

```
merge_7 (Merge) (None, 1152, 256) 0 merge_6[0][0]
convolution1d_13[0][0]
```

```
dilated_conv_128_tanh_s0 (Causal (None, 1152, 256) 131072 merge_7[0][0])
```

```
dilated_conv_128_sigm_s0 (Causal (None, 1152, 256) 131072 merge_7[0][0])
```

```
gated_activation_7_s0 (Merge) (None, 1152, 256) 0
dilated_conv_128_tanh_s0[0][0]
dilated_conv_128_sigm_s0[0][0]
```

```
convolution1d_15 (Convolution1D) (None, 1152, 256) 65536
gated_activation_7_s0[0][0]
```

```
merge_8 (Merge) (None, 1152, 256) 0 merge_7[0][0]
convolution1d_15[0][0]
```

```
dilated_conv_256_tanh_s0 (Causal (None, 1152, 256) 131072 merge_8[0][0])
```

```
dilated_conv_256_sigm_s0 (Causal (None, 1152, 256) 131072 merge_8[0][0])
```

```
gated_activation_8_s0 (Merge) (None, 1152, 256) 0
dilated_conv_256_tanh_s0[0][0]
dilated_conv_256_sigm_s0[0][0]
```

```
convolution1d_17 (Convolution1D) (None, 1152, 256) 65536
gated_activation_8_s0[0][0]
```

```
merge_9 (Merge) (None, 1152, 256) 0 merge_8[0][0]
convolution1d_17[0][0]
```

```
dilated_conv_512_tanh_s0 (Causal (None, 1152, 256) 131072 merge_9[0][0])
```

```
dilated_conv_512_sigm_s0 (Causal (None, 1152, 256) 131072 merge_9[0][0])
```

```
gated_activation_9_s0 (Merge) (None, 1152, 256) 0
dilated_conv_512_tanh_s0[0][0]
```

dilated_conv_512_sigm_s0[0][0]

convolution1d_2 (Convolution1D) (None, 1152, 256) 65536
gated_activation_0_s0[0][0]

convolution1d_4 (Convolution1D) (None, 1152, 256) 65536
gated_activation_1_s0[0][0]

convolution1d_6 (Convolution1D) (None, 1152, 256) 65536
gated_activation_2_s0[0][0]

convolution1d_8 (Convolution1D) (None, 1152, 256) 65536
gated_activation_3_s0[0][0]

convolution1d_10 (Convolution1D) (None, 1152, 256) 65536
gated_activation_4_s0[0][0]

convolution1d_12 (Convolution1D) (None, 1152, 256) 65536
gated_activation_5_s0[0][0]

convolution1d_14 (Convolution1D) (None, 1152, 256) 65536
gated_activation_6_s0[0][0]

convolution1d_16 (Convolution1D) (None, 1152, 256) 65536
gated_activation_7_s0[0][0]

convolution1d_18 (Convolution1D) (None, 1152, 256) 65536
gated_activation_8_s0[0][0]

convolution1d_20 (Convolution1D) (None, 1152, 256) 65536
gated_activation_9_s0[0][0]

merge_11 (Merge) (None, 1152, 256) 0 convolution1d_2[0][0]
convolution1d_4[0][0]
convolution1d_6[0][0]
convolution1d_8[0][0]
convolution1d_10[0][0]
convolution1d_12[0][0]
convolution1d_14[0][0]


```
convolution1d_16[0][0]
convolution1d_18[0][0]
convolution1d_20[0][0]
```

```
activation_1 (Activation) (None, 1152, 256) 0 merge_11[0][0]
```

```
convolution1d_21 (Convolution1D) (None, 1152, 256) 65792 activation_1[0][0]
```

```
activation_2 (Activation) (None, 1152, 256) 0 convolution1d_21[0][0]
```

```
convolution1d_22 (Convolution1D) (None, 1152, 256) 65792 activation_2[0][0]
```

```
output_softmax (Activation) (None, 1152, 256) 0 convolution1d_22[0][0]
```

```
=====
Total params: 4,129,536
```

```
Trainable params: 4,129,536
```

```
Non-trainable params: 0
```

DeepMind обучала сеть на наборах данных, содержащих речь нескольких людей, это заметно улучшило способность обучаться разделяемому представлению языков и интонаций, а, значит, приблизило результаты к естественному звучанию речи. В Интернете (<https://deepmind.com/blog/wavenet-generative-model-raw-audio/>) имеется потрясающая подборка примеров синтезированной речи, и интересно отметить, что качество звучания повышается, когда WaveNet видит не только звуковые сигналы, но дополнительный текст, преобразованный в последовательность лингвистических и фонетических признаков. Мне больше всего нравятся примеры, в которых одно и то же предложение произносится с различной интонацией. И конечно, завораживает музыка, созданная и исполняемая сетью WaveNet. Послушайте, не пожалеете!

Резюме

В этой главе мы обсудили порождающие сверточные сети (ПСС). Типичная ПСС состоит из двух сетей: одна обучается порождать синтетические данные, похожие на подлинные, а другая – отличать подлинные данные от поддельных. Обе сети соревнуются друг с другом и тем самым способствуют взаимному совершен-

ствованию. Мы рассмотрели исходный код сетей, обучающихся подделывать изображения из наборов MNIST и CIFAR-10. Кроме того, мы обсудили глубокую порождающую сеть WaveNet, разработанную компанией Google DeepMind для обучения компьютеров высококачественному воспроизведению человеческого голоса и звучания музыкальных инструментов. WaveNet непосредственно порождает звуковые сигналы, применяя параметрический синтез речи на основе сверточных сетей с пропусками. Сверточная сеть с пропусками – это специальный вид сверточных сетей, в которых сверточные фильтры имеют дырки, что позволяет рецептивному полю экспоненциально расти с увеличением глубины и тем самым охватывать тысячи временных шагов аудио. DeepMind показала, что сеть WaveNet можно использовать для синтеза человеческого голоса и звучания музыкальных инструментов с качеством, значительно превышающим предыдущие достижения. В следующей главе мы обсудим погружения слов – набор методов глубокого обучения для обнаружения связей между словами и группировки похожих слов.

Глава 5

Погружения слов

В википедии погружение, или векторное представление слов (word embedding) определяется как общее название различных методов языкового моделирования и обучения признаков, применяемых в **обработке естественных языков** (ОЕЯ, англ. NLP), когда слова или фразы из словаря отображаются на векторы вещественных чисел.

Погружение слов – это способ преобразовать текстовое представление слов в числовые векторы, допускающие анализ стандартными алгоритмами машинного обучения, принимающими на входе числа.

В главе 1 мы уже встречались с одним видом погружения слов – унитарным кодированием. Это самый простой подход к погружению. Напомним, что унитарным кодом слова будет вектор, число элементов которого равно размеру словаря, такой, что элемент, соответствующий данному слову, равен 1, а все остальные 0.

Основная проблема унитарного кодирования в том, что нет никакого способа представить сходство слов. В любом заданном корпусе текстов мы ожидаем, что между словами «кошка» и «собака» или «нож» и «вилка» есть какое-то сходство. Сходство векторов вычисляется с помощью скалярного произведения, т. е. суммы произведений соответственных элементов. В случае унитарного кодирования скалярное произведение любых двух слов равно нулю.

Для преодоления ограничений унитарного кодирования сообщество ОЕЯ заимствовало из **информационного поиска** (ИП) идею векторизации текста с использованием документа в качестве контекста. Здесь стоит отметить такие подходы, как TF-IDF (<https://en.wikipedia.org/wiki/Tf%E2%80%93idf>), **латентно-семантический анализ** (ЛСА) (https://en.wikipedia.org/wiki/Latent_semantic_analysis) и тематическое моделирование (https://en.wikipedia.org/wiki/Topic_model). Но эти представления улавливают несколько иную, документо-центрическую, идею семантического сходства.

Серьезная разработка методов погружения слов началась в 2000 году. Погружение слов отличается от предшествующих методов, применяемых в ИП, тем, что слова используются как собственный контекст, что приводит к более естественной форме семантического сходства с точки зрения понимания человеком. В настоящее время погружение слов – общепринятая техника векторизации текста во всех задачах ОЕЯ: классификация текстов, кластеризация документов, частеречная разметка, распознавание именованных сущностей, анализ эмоциональной окраски и т. п.

В этой главе мы рассмотрим две формы погружения слов, GloVe и word2vec, известные под общим названием «распределенное представление слов». Эти представления оказались более эффективными и потому широко распространены в среде специалистов по глубокому обучению и ОЕЯ.

Мы также узнаем о способах порождения собственных погружений в программе на Keras, а равно о том, как использовать и настраивать предобученные модели на основе word2vec и GloVe.

Будут рассмотрены следующие темы:

- построение различных распределенных представлений слов в контексте;
- построение моделей на основе погружений для решения таких задач ОЕЯ, как грамматический разбор предложения и анализ эмоциональной окраски.

Распределенные представления

Распределенное представление – это попытка уловить смысл слова путем рассмотрения его связей с другими словами в контексте. Эта идея сформулирована в следующем высказывании Дж. Р. Фирта (J. R. Firth) (см. статью Andrew M. Dai, Christopher Olah, Quoc V. Le «Document Embedding with Paragraph Vectors», arXiv:1507.07998, 2015), лингвиста, который первым выдвинул ее:

Мы узнаем слово по компании, с которой оно дружит.

Рассмотрим такие два предложения:

Париж – столица Франции.

Берлин – столица Германии.

Даже если вы совсем не знаете географию (или русский язык), все равно нетрудно сообразить, что пары слов (Париж, Берлин) и

(Франция, Германия) как-то связаны и что между соответственными словами связи одинаковы, т. е.

Париж : Франция :: Берлин : Германия

Следовательно, задача распределенного представления – найти такую общую функцию φ преобразования слова в соответствующий ему вектор, что справедливы соотношения следующего вида:

$$\varphi(\text{«Париж»}) - \varphi(\text{«Франция»}) \approx \varphi(\text{«Берлин»}) - \varphi(\text{«Германия»})$$

Иными словами, цель распределенного представления – преобразовать слова в векторы, так чтобы сходство векторов коррелировало с семантическим сходством слов.

В следующих разделах мы рассмотрим два наиболее известных погружения слов: word2vec и GloVe.

word2vec

Группа моделей word2vec была разработана в 2013 году группой исследователей Google под руководством Томаша Миколова (Tomas Mikolov). Модели обучаются без учителя на большом корпусе текстов и порождают векторное пространство слов. Размерность пространства погружения word2vec обычно меньше размерности пространства погружения для унитарного кодирования, которая равна размеру словаря. Кроме того, это пространство погружения плотнее разреженного погружения при унитарном кодировании.

Существуют две архитектуры word2vec:

- непрерывный мешок слов (Continuous Bag Of Words, CBOW);
- skip-граммы.

В архитектуре CBOW модель предсказывает текущее слово, если известно окно окружающих его слов. Кроме того, порядок контекстных слов не влияет на предсказание (это допущение модели мешка слов). В архитектуре skip-грамм модель предсказывает окружающие слова по известному центральному слову. Согласно заявлению авторов, CBOW быстрее, но skip-граммы лучше предсказывают редкие слова.

Интересно отметить, что хотя word2vec создает погружения, используемые в моделях глубокого обучения ОЕЯ, оба варианта word2vec, которые мы будем обсуждать и которые снискали наибольший успех и признание, являются мелкими нейронными сетями.

Модель skip-грамм

Модель skip-грамм обучается предсказывать окружающие слова по известному центральному. Чтобы понять, как она работает, рассмотрим такое предложение:

I love green eggs and ham. (Я люблю зеленые яйца и ветчину)¹

В предположении, что размер окна равен 3, предложение можно разложить на такие пары (контекст, слово):

(*[I, green], love*)
 (*[love, eggs], green*)
 (*[green, and], eggs*)
 ...

Поскольку модель skip-грамм предсказывает контекстное слово по центральному, мы можем преобразовать этот набор данных в набор пар (вход, выход). То есть, зная входное слово, мы ожидаем, что модель skip-грамм предскажет соответствующее выходное:

(*love, I*), (*love, green*), (*green, love*), (*green, eggs*), (*eggs, green*), (*eggs, and*), ...

Мы можем также сгенерировать дополнительные отрицательные примеры, объединяя в пару каждое входное слово со случайным словом из словаря, например:

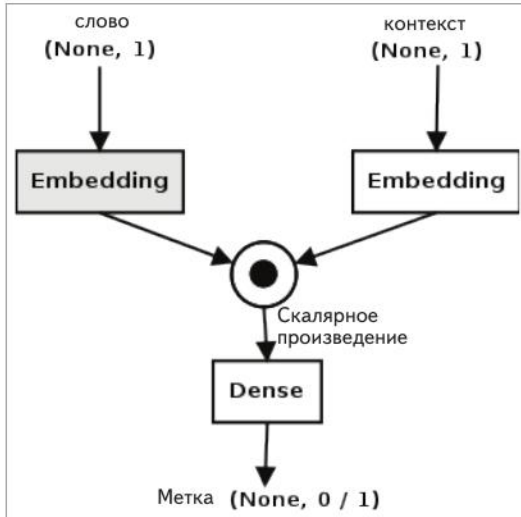
(*love, Sam*), (*love, zebra*), (*green, thing*), ...

Наконец, мы генерируем положительные и отрицательные примеры для классификатора:

((*love, I*), 1), ((*love, green*), 1), ..., ((*love, Sam*), 0), ((*love, zebra*), 0), ...

Теперь можно обучить классификатор, который принимает вектор слов и контекстный вектор и предсказывает 0 или 1 в зависимости от того, какой пример видит: положительный или отрицательный. Результатом обучения сети являются веса слоя погружения слов (серый блок на рисунке ниже):

¹ «Green eggs and ham» – детская сказка, написанная доктором Сьюзом. – Прим. перев.



Опишем, как строится модель skip-грамм в Keras. Предположим, что размер словаря равен 5000, размер выходного пространства погружения 300, размер окна 1. Последнее означает, что контекст состоит из предыдущего и следующего слова. Сначала импортируем нужные модули и инициализируем переменные:

```

from keras.layers import Merge
from keras.layers.core import Dense, Reshape
from keras.layers.embeddings import Embedding
from keras.models import Sequential

vocab_size = 5000
embed_size = 300

```

Затем создадим последовательную модель слова. Входом модели являются идентификаторы слов в словаре. Весам погружений первоначально присваиваются небольшие случайно выбранные значения. В процессе обучения модель будет обновлять веса, применяя алгоритм обратного распространения. Следующий слой адаптирует форму входа под размер погружения:

```

word_model = Sequential()
word_model.add(Embedding(vocab_size, embed_size,
    embeddings_initializer="glorot_uniform",
    input_length=1))
word_model.add(Reshape((embed_size, )))

```

Нам также нужна еще одна модель для контекстных слов. Для каждой пары skip-грамм мы имеем одно контекстное слово, соответствующее целевому слову, так что эта модель идентична модели слов:

```
context_model = Sequential()
context_model.add(Embedding(vocab_size, embed_size,
    embeddings_initializer="glorot_uniform",
    input_length=1))
context_model.add(Reshape((embed_size,)))
```

На выходе обеих моделей получаются векторы размера `embed_size`. Их скалярное произведение подается на вход плотному слою с сигмоидной функцией активации, который порождает один выход. Напомним, что сигмоида преобразует свой аргумент в число из диапазона $[0,1]$ и асимптотически быстро приближается к единице на положительной полуоси и к 0 – на отрицательной.

```
model = Sequential()
model.add(Merge([word_model, context_model], mode="dot"))
model.add(Dense(1, init="glorot_uniform", activation="sigmoid"))
model.compile(loss="mean_squared_error", optimizer="adam")
```

В качестве функции потерь используется `mean_squared_error`; идея в том, чтобы минимизировать скалярное произведение для положительных примеров и максимизировать для отрицательных. Напомним, что скалярное произведение равно сумме произведений соответственных элементов, поэтому скалярное произведение похожих векторов будет больше, чем непохожих, т. к. у похожих векторов больше одинаковых элементов в соответственных позициях.

Keras предоставляет функцию для извлечения skip-грамм из текста, преобразованного в список индексов слов. Ниже приведен пример ее использования для извлечения первых 10 из 56 сгенерированных skip-грамм (положительных и отрицательных).

Сначала импортируем пакеты и инициализируем подлежащий анализу текст:

```
from keras.preprocessing.text import *
from keras.preprocessing.sequence import skipgrams

text = "I love green eggs and ham ."
```

Затем объявляем объект для выделения лексем и пропускаем через него текст. Получается список лексем:

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts([text])
```

Объект `tokenizer` создает словарь, сопоставляющий каждому уникальному слову целочисленный идентификатор, и делает его доступным через атрибут `word_index`. Мы читаем этот атрибут и создаем две таблицы соответствия:

```
word2id = tokenizer.word_index
id2word = {v:k for k, v in word2id.items()}
```

Наконец, входной список слов преобразуется в список идентификаторов и передается функции `skipgrams`. Затем мы печатаем первые 10 из 56 сгенерированных `skip-грамм` (пара, метка):

```
wids = [word2id[w] for w in text_to_word_sequence(text)]
pairs, labels = skipgrams(wids, len(word2id))
print(len(pairs), len(labels))
for i in range(10):
    print("({:s} ({:d}), {:s} ({:d})) -> {:d}".format(
        id2word[pairs[i][0]], pairs[i][0],
        id2word[pairs[i][1]], pairs[i][1],
        labels[i]))
```

Ниже показаны результаты работы этой программы. На вашей машине результаты могут отличаться, потому что функция `skipgrams` производит случайную выборку из множества возможных положительных примеров. Кроме того, генерация отрицательных примеров производится путем выборки случайных пар лексем из текста. С увеличением размера входного текста вероятность выбрать пары не связанных между собой слов возрастает. Но в нашем примере текст очень короткий, поэтому высоки шансы, что будут сгенерированы и положительные примеры тоже:

```
(and (1), ham (3)) -> 0
(green (6), i (4)) -> 0
(love (2), i (4)) -> 1
(and (1), love (2)) -> 0
(love (2), eggs (5)) -> 0
(ham (3), ham (3)) -> 0
(green (6), and (1)) -> 1
(eggs (5), love (2)) -> 1
(i (4), ham (3)) -> 0
(and (1), green (6)) -> 1
```

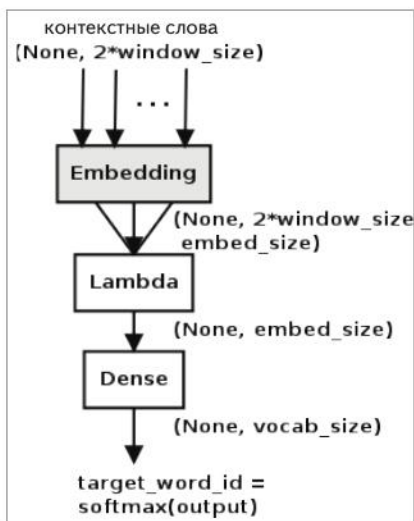
Код этого примера находится в файле `skipgram_example.py` в исходном коде к этой главе.

Модель CBOW

Теперь рассмотрим модель CBOW из семейства word2vec. Напомним, что она предсказывает слово по известным контекстным словам. Таким образом, для первого кортежа из примера ниже CBOW должна предсказать слово *love*, зная контекстные слова *I* и *green*:

$([I, green], love)$ $([love, eggs], green)$ $([green, and], eggs) \dots$

Как и модель skip-грамм, модель CBOW представляет собой классификатор, который получает на входе контекстные слова и предсказывает целевое слово. Но архитектура проще, чем в модели skip-грамм. Входными данными модели являются идентификаторы контекстных слов. Они поступают на вход слоя погружения, веса которого инициализируются небольшими случайными значениями. Этот слой преобразует каждый идентификатор в вектор размера `embed_size`. Следовательно, каждая строка входного контекста преобразуется в матрицу размера $(2 * window_size, embed_size)$. Затем матрица подается на вход слоя `lambda`, который вычисляет среднее по всем погружениям. Полученная величина передается плотному слою, который создает плотный вектор размера `vocab_size` для каждой строки. В качестве функции активации в плотном слое используется `softmax`, которая возвращает вероятность, равную максимальному элементу выходного вектора. Идентификатор с максимальной вероятностью соответствует целевому слову.



Ниже приведен код модели на Keras. Мы снова предполагаем, что размер словаря равен 5000, размер выходного пространства погружения 300, размер контекстного окна 1. Сначала импортируем пакеты и инициализируем переменные:

```
from keras.models import Sequential
from keras.layers.core import Dense, Lambda
from keras.layers.embeddings import Embedding
import keras.backend as K

vocab_size = 5000
embed_size = 300
window_size = 1
```

Затем строим последовательную модель, в которую включаем слой погружения с весами, инициализированными малыми случайными значениями. Отметим, что длина входа `input_length` этого слоя равна числу контекстных слов. Каждое контекстное слово подается на вход слоя, и веса обновляются в процессе обратного распространения. На выходе слоя получается матрица погружений контекстных слов, которая усредняется в один вектор (на каждую строку входа) слоем `lambda`. Наконец, плотный слой преобразует строки в плотный вектор размера `vocab_size`. Целевым словом будет то, для которого вероятность идентификатора в плотном выходном векторе максимальна.

```
model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=embed_size,
                    embeddings_initializer='glorot_uniform',
                    input_length=window_size*2))
model.add(Lambda(lambda x: K.mean(x, axis=1), output_shape=(embed_size,)))
model.add(Dense(vocab_size, kernel_initializer='glorot_uniform',
                activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer="adam")
```

В качестве функции потерь здесь используется `categorical_crossentropy` – типичный выбор для случая, когда категорий две или больше (в нашем примере `vocab_size`).

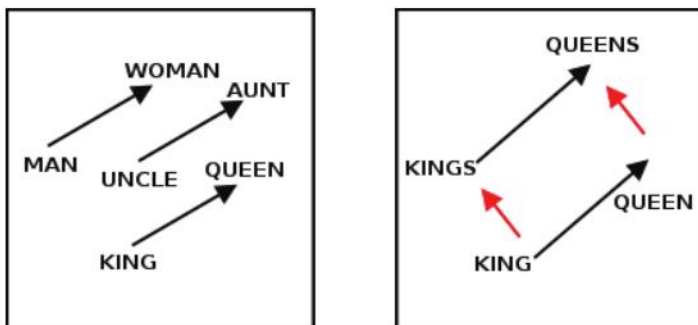
Код этого примера находится в файле `keras_cbow.py` в исходном коде к этой главе.

Извлечение погружений word2вес из модели

Выше уже отмечалось, что хотя обе модели семейства word2vec можно свести к задаче классификации, сама эта задача нас не интересует. А интересен нам побочный эффект классификации – ма-

трица весов, которая преобразует слово из словаря в плотное распределенное представление низкой размерности.

Есть немало примеров того, как распределенные представления выявляют удивительную синтаксическую и семантическую информацию. Так, на следующем рисунке, взятом из презентации Томаша Миколова на конференции NIPS 2013 (см. T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Q. Le, T. Strohmann «Learning Representations of Text using Neural Networks», NIPS 2013), показано, что векторы, соединяющие слова, имеющие одинаковый смысл, но относящиеся к разным полам, приблизительно параллельны в редуцированном двумерном пространстве и что зачастую можно получить согласующиеся с интуицией результаты, производя арифметические действия над векторами слов. В презентации много таких примеров.



Интуитивно кажется, что процесс обучения привносит достаточно информации во внутреннюю кодировку, чтобы предсказать выходное слово, встречающееся в контексте входного. Поэтому точки, представляющие слова в этом пространстве, располагаются ближе к точкам слов, с которыми они встречаются совместно. В результате похожие слова образуют кластеры. И слова, встречающиеся вместе с похожими словами, тоже образуют кластеры. Следовательно, векторы, соединяющие точки, представляющие семантически связанные слова в распределенном представлении, демонстрируют регулярное поведение.

Keras предоставляет средства для извлечения весов из обученных моделей. В случае skip-грамм веса слоя погружения можно получить следующим образом:

```
merge_layer = model.layers[0]
word_model = merge_layer.layers[0]
```

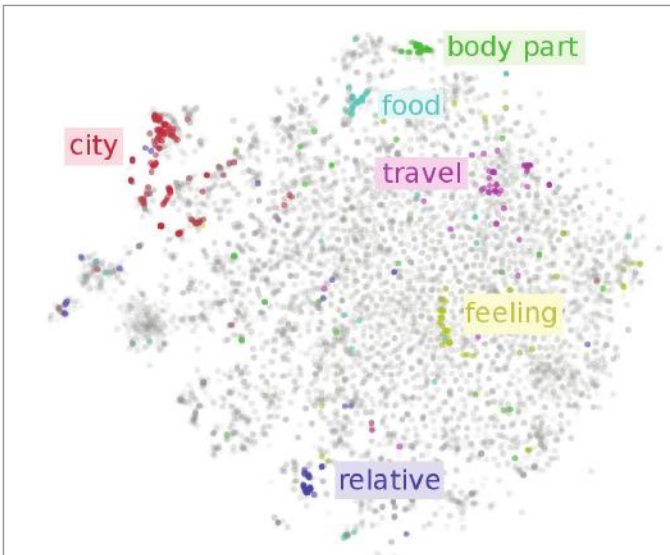
```
word_embed_layer = word_model.layers[0]
weights = word_embed_layer.get_weights()[0]
```

А в случае CBOW для получения весов достаточно одной строки:

```
weights = model.layers[0].get_weights()[0]
```

В обоих случаях матрица весов имеет размер `vocab_size × embed_size`. Для вычисления распределенного представления слова из словаря нужно построить унитарный вектор, записав 1 в элемент вектора размера `vocab_size` с индексом, равным идентификатору слова, и умножить его на матрицу весов, получив в результате вектор погружения размера `embed_size`.

Ниже показана визуализация погружений слов, выполненная Кристофером Ола (см. Andrew M. Dai, Christopher Olah, and Quoc V. Le «Document Embedding with Paragraph Vectors», arXiv:1507.07998, 2015). Для этого было произведено понижение размерности до двух измерений и использована библиотека T-SNE. Слова, образующие типы сущностей, выбраны из синонимических рядов (синсетов) семантической сети WordNet. Как видно, точки, соответствующие похожим типам сущностей, образуют кластеры.



Код этого примера находится в файле `keras_skipgram.py` в исходном коде к этой главе.

Сторонние реализации word2vec

В предыдущих разделах было подробно рассмотрено семейство моделей word2vec. Вы понимаете, как работают модели skip-грамм и CBOW и как самостоятельно построить их реализацию с помощью Keras. Однако существуют готовые реализации word2vec и в предположении, что ваша задача не слишком сложна и не сильно отличается от типичной, имеет смысл воспользоваться одной из них и не изобретать велосипед.

Одна такая реализация word2vec – библиотека gensim. И хотя эта книга посвящена Keras, а не gensim, мы решили включить ее обсуждение, поскольку Keras не поддерживает word2vec, а интеграция gensim и Keras – распространенная практика.



Установка gensim не вызывает сложностей и подробно описана на странице <https://radimrehurek.com/gensim/install.html>.

Ниже показано, как построить модель word2vec с помощью gensim и обучить ее на тексте из корпуса text8, доступном по адресу <http://mattmahoney.net/dc/text8.zip>. Этот файл содержит около 17 миллионов слов, взятых из статей википедии. Текст был подвергнут очистке – удалению разметки, знаков препинания и символов, не принадлежащих кодировке ASCII. Первые 100 миллионов знаков очищенного текста и составили корпус text8. Он часто используется в качестве примера для модели word2vec, потому что обучение на нем происходит быстро и дает хорошие результаты. Сначала импортируем необходимые пакеты:

```
from gensim.models import KeyedVectors
import logging
import os
```

Затем читаем поток слов из корпуса text8 и разбиваем его на предложения по 50 слов в каждом. Библиотека gensim содержит встроенный обработчик text8, который делает нечто подобное. Поскольку мы хотим показать, как построить модель для любого (предпочтительно большого) корпуса, который может и не помещаться целиком в память, то продемонстрируем порождение этих предложений с помощью генератора Python.

Класс `Text8Sentences` порождает предложения по `maxlen` слов в каждом из файла `text8`. В данном случае мы таки читаем весь файл в память, но при обходе файлов, находящихся в нескольких каталогах, генератор позволяет загрузить в память часть данных, обработать ее и отдать вызывающей стороне:

```
class Text8Sentences(object):
    def __init__(self, fname, maxlen):
        self.fname = fname
        self.maxlen = maxlen

    def __iter__(self):
        with open(os.path.join(DATA_DIR, "text8"), "rb") as ftext:
            text = ftext.read().split(" ")
            sentences, words = [], []
            for word in text:
                if len(words) >= self.maxlen:
                    yield words
                    words = []
                words.append(word)
            yield words
```

Теперь займемся вызывающей программой. В библиотеке `gensim` используется имеющийся в Python механизм протоколирования для уведомления о ходе обработке, поэтому для начала активируем его. В следующей строке создается экземпляр класса `Text8Sentences`, а затем модель обучается на предложениях из набора данных. Мы задали размер векторов погружения 300 и рассматриваем только слова, которые встречаются в корпусе не менее 30 раз. Размер окна по умолчанию равен 5, поэтому контекстом для слова w_i будут слова $w_{i-5}, w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}, w_{i+3}, w_{i+4}, w_{i+5}$. По умолчанию создается модель CBOW, но это можно изменить, задав параметр `sg=1`:

```
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
                    level=logging.INFO)

DATA_DIR = "../data/"
sentences = Text8Sentences(os.path.join(DATA_DIR, "text8"), 50)
model = word2vec.Word2Vec(sentences, size=300, min_count=30)
```

Реализация `word2vec` производит два прохода по данным: сначала строится словарь, а затем – фактическая модель. За ходом работы можно следить по распечатке на консоли:


```

2017-01-30 16:16:27,786 : INFO : PROGRESS: at 76.44% examples, 691859 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:28,801 : INFO : PROGRESS: at 77.74% examples, 693040 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:29,807 : INFO : PROGRESS: at 79.00% examples, 693746 words/s, in_qsize 2, out_qsize 0
2017-01-30 16:16:30,815 : INFO : PROGRESS: at 79.99% examples, 692107 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:31,819 : INFO : PROGRESS: at 80.03% examples, 682583 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:32,842 : INFO : PROGRESS: at 81.15% examples, 682090 words/s, in_qsize 1, out_qsize 0
2017-01-30 16:16:33,869 : INFO : PROGRESS: at 82.46% examples, 683117 words/s, in_qsize 0, out_qsize 1
2017-01-30 16:16:34,873 : INFO : PROGRESS: at 83.77% examples, 684403 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:35,882 : INFO : PROGRESS: at 85.02% examples, 685224 words/s, in_qsize 5, out_qsize 0
2017-01-30 16:16:36,884 : INFO : PROGRESS: at 86.36% examples, 686831 words/s, in_qsize 0, out_qsize 1
2017-01-30 16:16:37,925 : INFO : PROGRESS: at 87.51% examples, 686556 words/s, in_qsize 2, out_qsize 0
2017-01-30 16:16:38,925 : INFO : PROGRESS: at 88.57% examples, 685873 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:39,933 : INFO : PROGRESS: at 89.84% examples, 686756 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:40,936 : INFO : PROGRESS: at 91.17% examples, 688126 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:41,939 : INFO : PROGRESS: at 92.43% examples, 688894 words/s, in_qsize 0, out_qsize 1
2017-01-30 16:16:42,946 : INFO : PROGRESS: at 93.69% examples, 689612 words/s, in_qsize 1, out_qsize 0
2017-01-30 16:16:43,960 : INFO : PROGRESS: at 94.97% examples, 690484 words/s, in_qsize 1, out_qsize 0
2017-01-30 16:16:44,978 : INFO : PROGRESS: at 96.30% examples, 691348 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:45,982 : INFO : PROGRESS: at 97.58% examples, 692158 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:46,980 : INFO : PROGRESS: at 98.83% examples, 692731 words/s, in_qsize 2, out_qsize 0
2017-01-30 16:16:48,092 : INFO : PROGRESS: at 99.92% examples, 691317 words/s, in_qsize 4, out_qsize 1
2017-01-30 16:16:48,124 : INFO : worker thread finished; awaiting finish of 2 more threads
2017-01-30 16:16:48,125 : INFO : worker thread finished; awaiting finish of 1 more threads
2017-01-30 16:16:48,126 : INFO : worker thread finished; awaiting finish of 0 more threads
2017-01-30 16:16:48,129 : INFO : training on 85026040 raw words (59645573 effective words) took 86.2s, 691572
effective words/s
2017-01-30 16:16:48,129 : INFO : precomputing L2-norms of word weight vectors

```

После создания модели нужно нормировать получившиеся векторы. В документации сказано, что это сэкономит много памяти. Обученную модель можно сохранить на диске:

```
model.init_sims(replace=True)
model.save("word2vec_gensim.bin")
```

Для загрузки сохраненной модели в память служит такой метод:

```
model = Word2Vec.load("word2vec_gensim.bin")
```

Далее можно запросить у модели все известные ей слова:

```
>>> model.vocab.keys()[0:10]
['homomorphism',
'woods',
'spiders',
'hanging',
'woody',
'localized',
'sprague',
'originality',
'alphabetic',
'hermann']
```

Можно получить векторное представление заданного слова:

```
>>> model["woman"]  
array([-3.13099056e-01, -1.85702944e+00, 1.18816841e+00,  
       -1.86561719e-01, -2.23673001e-01, 1.06527400e+00,  
       ...,      ], dtype=float64)
```

```
4.31755871e-01, -2.90115297e-01, 1.00955181e-01,
-5.17173052e-01, 7.22485244e-01, -1.30940580e+00], dtype="float32")
```

Можно также найти слова, похожие на заданное:

```
>>> model.most_similar("woman")
[('child', 0.7057571411132812),
 ('girl', 0.702182412147522),
 ('man', 0.6846336126327515),
 ('herself', 0.6292711496353149),
 ('lady', 0.6229539513587952),
 ('person', 0.6190367937088013),
 ('lover', 0.6062309741973877),
 ('baby', 0.5993420481681824),
 ('mother', 0.5954475402832031),
 ('daughter', 0.5871444940567017)]
```

Можно давать указания о том, какие слова считать похожими. Так, следующая команда возвращает первые 10 слов, похожие на woman (женщина) и king (король), но не похожие на man (мужчина):

```
>>> model.most_similar(positive=['woman', 'king'], negative=['man'], topn=10)
[('queen', 0.6237582564353943),
 ('prince', 0.5638638734817505),
 ('elizabeth', 0.5557916164398193),
 ('princess', 0.5456407070159912),
 ('throne', 0.5439794063568115),
 ('daughter', 0.5364126563072205),
 ('empress', 0.5354889631271362),
 ('isabella', 0.5233952403068542),
 ('regent', 0.520746111869812),
 ('matilda', 0.5167444944381714)]
```

Можно также запросить меру сходства между заданными словами. Чтобы получить представление о том, как положение слов в пространстве погружения коррелирует с их семантикой, рассмотрим следующие пары слов:

```
>>> model.similarity("girl", "woman")
0.702182479574
>>> model.similarity("girl", "man")
0.574259909834
>>> model.similarity("girl", "car")
0.289332921793
>>> model.similarity("bus", "car")
0.483853497748
```

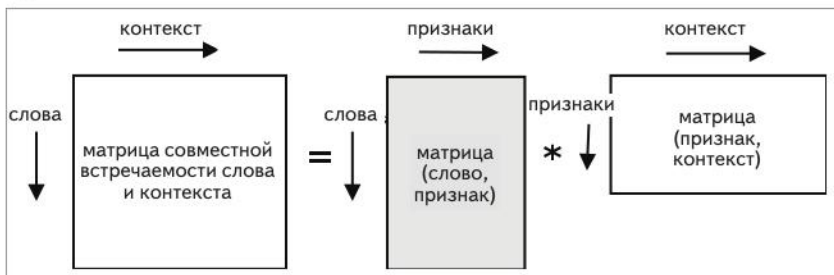
Как видим, слова girl и woman больше похожи, чем girl и man, а car (автомобиль) и bus (автобус) больше, чем girl и car. Это хорошо согласуется с тем, как ранжировал бы схожесть человек.

Код этого примера находится в файле `word2vec_gensim.py` в исходном коде к этой главе.

Введение в GloVe

Погружение слов GloVe (Global Vector – глобальный вектор) было предложено в работе J. Pennington, R. Socher, and C. Manning «GloVe: Global Vectors for Word Representation», Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1532–1543, 2013. Авторы описывают GloVe как алгоритм обучения без учителя, цель которого – получение векторных представлений слов. Обучение производится на агрегированной глобальной статистике совместной встречаемости слов из корпуса, а получающиеся представления вскрывают интересные линейные структуры в векторном пространстве слов.

GloVe отличается от word2vec тем, что word2vec – прогностическая модель, тогда как GloVe основана на счетчиках. На первом шаге строится большая матрица совместной встречаемости пар (слово, контекст) в обучающем корпусе. Каждый элемент матрицы описывает, как часто слово, представленное данной строкой, встречается в контексте (обычно это последовательность слов), представленном данным столбцом.



Алгоритм GloVe преобразует матрицу совместной встречаемости в пару матриц: (слово, признак) и (признак, контекст). Этот процесс называется **факторизацией матрицы** и выполняется итеративно с помощью метода **стохастического градиентного спуска (СГС)**. В форме уравнения он записывается так:

$$R = P * Q \approx R'$$

Здесь R – исходная матрица совместной встречаемости. Сначала мы инициализируем P и Q случайными значениями и пытаем-

ся воссоздать R' путем их перемножения. Разница между реконструированной матрицей R' и исходной R показывает, как надо изменить значения P и Q , чтобы R' стала ближе к R , т. е. ошибка реконструкции уменьшилась. Эта операция повторяется несколько раз, пока алгоритм СГС не сойдется и ошибка реконструкции не станет ниже заданного порогового значения. Получившаяся в этот момент матрица (слово, признак) и является погружением в смысле GloVe. Для ускорения процесса СГС часто выполняется параллельно, как описано в статье «Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent» (<https://people.eecs.berkeley.edu/~brecht/papers/hogwildTR.pdf>).

Отметим, что прогностические модели на основе нейронных сетей типа word2vec и основанные на счетчиках модели типа GloVe преследуют одну и ту же цель. Те и другие строят векторное пространство, так что положение слова в нем зависит от соседних слов. Нейронная сеть начинает работу с отдельных примеров совместной встречаемости слов, а основанные на счетчиках модели – со статистики совместной встречаемости всех слов в корпусе. Недавно было опубликовано несколько работ, в которых демонстрируется корреляция между моделями обоих типов.

В этой книге мы не будем подробно рассматривать генерацию векторов GloVe. Хотя в общем случае GloVe отличается большей верностью, чем word2vec, и быстрее обучается при использовании распараллеливания, поддержка ее на Python пока не столь развитая, как word2vec. На данный момент единственным доступным инструментом был проект GloVe-Python (<https://github.com/maciejkula/glove-python>), предлагающий модельную реализацию GloVe на Python.

Использование предобученных погружений

Вообще говоря, обучать модель word2vec или GloVe с нуля следует только тогда, когда имеется очень большой объем узкоспециализированных текстов. Чаще всего тем или иным способом используются предобученные погружения. Есть три основных способа включения погружений в собственную сеть:

- обучение погружений с нуля;

- настройка погружений на основе предобученных моделей GloVe/word2vec;
- поиск погружений в предобученных моделях GloVe/word2vec.

В первом случае веса погружений инициализируются небольшими случайными значениями и обучаются методом обратного распространения. Такой способ мы видели на примере моделей skip-грамм и CBOW в Keras. Это режим по умолчанию при использовании слоя Embedding в собственной сети.

Во втором случае из предобученной модели берется матрица весов и используется для инициализации весов слоя погружения. Сеть также обновляет веса методом обратного распространения, но модель сходится быстрее вследствие хорошего выбора начальных весов.

В третьем случае погружения слов ищутся в предобученной модели, и входные данные преобразуются в векторные погружения. На преобразованных данных можно затем обучить любую модель (необязательно сеть глубокого обучения). Если предобученная модель обучалась на текстах из той же предметной области, что и целевая, то этот способ обычно дает прекрасные результаты и является самым дешевым.

Для англоязычных текстов общего характера можно использовать модель word2vec от Google, обученную на 10 миллиардах слов из набора данных Google news. Размер словаря составляет примерно 3 миллиона слов, а размерность пространства погружения равна 300. Модель Google news (размером около 1.5 ГБ) можно скачать по адресу <https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing>.

С сайта GloVe можно скачать модель, обученную на 6 миллиардах лексем из англоязычной википедии и корпусе текстов, содержащем порядка миллиарда слов. Размер словаря составляет примерно 400 000 слов, для скачивания доступны модели с размерностью пространства погружения 50, 100, 200 и 300. Размер модели составляет приблизительно 822 МБ. Скачать модель можно по адресу <http://nlp.stanford.edu/data/glove.6B.zip>. Там же имеются более крупные модели, обученные на данных из репозитория проекта Common Crawl и из Twitter.

В следующих разделах мы рассмотрим, как использовать эти модели тремя вышеупомянутыми способами.

Обучение погружений с нуля

В этом примере мы обучим одномерную **сверточную нейронную сеть (СНС)** классифицировать предложения как окрашенные положительно или отрицательно. В главе 3 мы уже рассматривали обучение двумерной СНС для классификации изображений. Напомним, что в этой СНС используется пространственная структура изображения путем обеспечения локальной связности между нейронами соседних слоев.

Для слов предложения характерна линейная структура, аналогичная пространственной структуре в изображении. Традиционные (не на базе глубокого обучения) подходы к языковому моделированию подразумевают создание словесных n -грамм (<https://en.wikipedia.org/wiki/N-gram>), улавливающих эту линейную структуру. Одномерные СНС делают нечто похожее, обучая сверточные фильтры, которые затрагивают сразу несколько слов предложения, и применяя к результатам max-пулинг, чтобы создать вектор, представляющий важнейшие смысловые аспекты предложения.

Существует еще один класс нейронных сетей, **рекуррентные нейронные сети (РНС)**, специально предназначенные для обработки последовательных данных, в т. ч. текста, который есть не что иное, как последовательность слов. Порядок обработки в РНС не такой, как в СНС. Подробнее о РНС мы будем говорить в следующей главе.

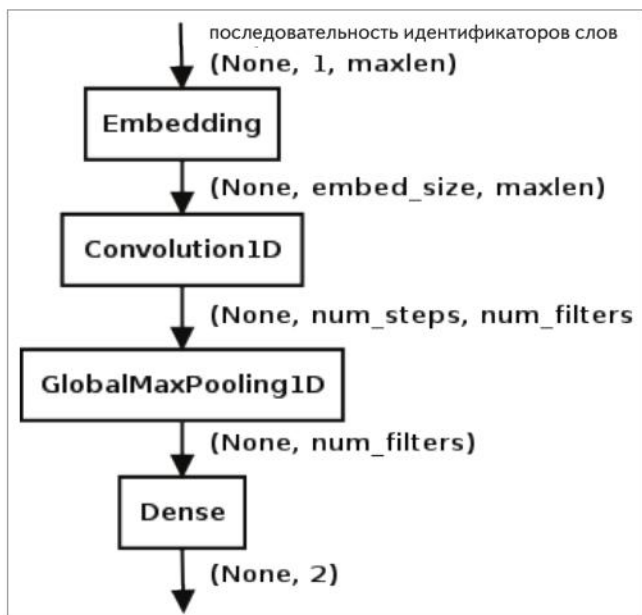
В нашей сети входной текст преобразуется в последовательность индексов слов. Для грамматического разбора текста мы воспользовались библиотекой для обработки естественных языков NLTK (natural language toolkit). Можно было бы применить для этой цели регулярные выражения, но статистические модели, предлагаемые NLTK, более пригодны для разбора текста. Если вас интересуют погружения слов, то, скорее всего, вы занимаетесь ОЕЯ, так что NLTK уже установлена.



По адресу <http://www.nltk.org/install.html> приведена информация об установке NLTK. Понадобятся также включенные в состав NLTK данные – обученные модели, корпуса текстов и прочее. Инструкции по их установке имеются на [странице http://www.nltk.org/data.html](http://www.nltk.org/data.html).

Последовательность индексов слов загружается в слой погружения заданного размера (в нашем случае число слов в самом

длинном предложении). По умолчанию слой погружения инициализируется случайными значениями. Выход слоя погружения соединяется с одномерным сверточным слоем, который сворачивает (в нашем примере) словесные триграммы 256 различными способами (по сути дела применяет различные обученные линейные комбинации весов к погружениям слов). Эти признаки затем сводятся к единственному слову слоем глобального max-пулинга. Вектор длины 256 подается на вход плотного слоя, который выводит вектор длины 2. Функция активации softmax возвращает две вероятности: положительной и отрицательной эмоциональной окраски. Сеть показана на рисунке ниже.



Посмотрим, как это реализуется с помощью Keras. Сначала – импорт пакетов. После этого задается начальное значение генератора случайных чисел – 42. Это сделано для того, чтобы результаты прогона программы были воспроизводимыми (напомним, что матрицы весов инициализируются случайными значениями).

```

from keras.layers.core import Dense, Dropout, SpatialDropout1D
from keras.layers.convolutional import Conv1D
from keras.layers.embeddings import Embedding
from keras.layers.pooling import GlobalMaxPooling1D

```

```

from keras.models import Sequential
from keras.preprocessing.sequence import pad_sequences
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np

np.random.seed(42)

```

Далее объявляются константы. Во всех оставшихся примерах в этой главе мы будем классифицировать предложения из конкурса UMICH SI650, проводившегося на сайте Kaggle. В этом наборе данных около 7000 предложений, причем положительно окрашенные снабжены меткой 1, а отрицательно – меткой 0. Константа `INPUT_FILE` определяет путь к файлу помеченных предложений. Первый символ в каждой строке файла – метка (0 или 1), затем знак табуляции и за ним предложение.

Константа `VOCAB_SIZE` говорит, что мы будем рассматривать только первые 5000 лексем текста. `EMBED_SIZE` – размер погружения, генерируемого слоем погружения нашей сети. `NUM_FILTERS` – число сверточных фильтров, обучаемых в сверточном слое, а `NUM_WORDS` – размер каждого фильтра, т. е. количество сворачиваемых за один раз слов. Константы `BATCH_SIZE` и `NUM_EPOCHS` – соответственно число загружаемых в одном пакете записей и количество проходов по всему набору данных в процессе обучения.

```

INPUT_FILE = "../data/umich-sentiment-train.txt"
VOCAB_SIZE = 5000
EMBED_SIZE = 100
NUM_FILTERS = 256
NUM_WORDS = 3
BATCH_SIZE = 64
NUM_EPOCHS = 20

```

Далее мы читаем входные предложения и строим словарь, содержащий наиболее часто встречающиеся в корпусе слова. Затем этот словарь используется для преобразования входных предложений в список индексов слов.

```

counter = collections.Counter()
fin = open(INPUT_FILE, "rb")
maxlen = 0
for line in fin:
    _, sent = line.strip().split("t")
    words = [x.lower() for x in nltk.word_tokenize(sent)]

```

```

if len(words) > maxlen:
    maxlen = len(words)
for word in words:
    counter[word] += 1
fin.close()

word2index = collections.defaultdict(int)
for wid, word in enumerate(counter.most_common(VOCAB_SIZE)):
    word2index[word[0]] = wid + 1
vocab_size = len(word2index) + 1
index2word = {v:k for k, v in word2index.items()}

```

Каждое предложение дополняется до длины `maxlen` (в нашем примере это число слов в самом длинном предложении из обучающего набора). Кроме того, метки преобразуются в категориальный формат с помощью служебной функции Keras. Последние два шага – стандартные операции обработки текста, с которыми мы еще не раз встретимся.

```

xs, ys = [], []
fin = open(INPUT_FILE, "rb")
for line in fin:
    label, sent = line.strip().split("\t")
    ys.append(int(label))
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    wids = [word2index[word] for word in words]
    xs.append(wids)
fin.close()
X = pad_sequences(xs, maxlen=maxlen)
Y = np_utils.to_categorical(ys)

```

Наконец, мы разбиваем данные на обучающий и тестовый набор в пропорции 70:30. Теперь данные приведены к виду, пригодному для загрузки в сеть:

```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.3, random_state=42)
```

Определим описанную ранее сеть:

```

model = Sequential()
model.add(Embedding(vocab_size, EMBED_SIZE, input_length=maxlen))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(Conv1D(filters=NUM_FILTERS, kernel_size=NUM_WORDS, activation="relu"))
model.add(GlobalMaxPooling1D())
model.add(Dense(2, activation="softmax"))

```

Теперь откомпилируем модель. Поскольку мы строим бинарный классификатор (положительная или отрицательная эмоциональная окраска), то в качестве функции потерь выбираем `categorical_crossentropy`. В качестве оптимизатора возьмем `adam`. Обучим модель

на нашем обучающем наборе, указав размер пакета 64 и число периодов 20:

```
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
history = model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE,
                    epochs=NUM_EPOCHS, validation_data=(Xtest, Ytest))
```

Протокол обучения показан ниже:

```
Epoch 9/20
4960/4960 [=====] - 3s - loss: 0.0337 - acc: 0.9855 - val_loss: 0.0263 - val_acc: 0.9882
Epoch 10/20
4960/4960 [=====] - 3s - loss: 0.0369 - acc: 0.9843 - val_loss: 0.0277 - val_acc: 0.9878
Epoch 11/20
4960/4960 [=====] - 3s - loss: 0.0331 - acc: 0.9881 - val_loss: 0.0303 - val_acc: 0.9878
Epoch 12/20
4960/4960 [=====] - 3s - loss: 0.0289 - acc: 0.9879 - val_loss: 0.0291 - val_acc: 0.9882
Epoch 13/20
4960/4960 [=====] - 3s - loss: 0.0261 - acc: 0.9901 - val_loss: 0.0305 - val_acc: 0.9878
Epoch 14/20
4960/4960 [=====] - 3s - loss: 0.0261 - acc: 0.9895 - val_loss: 0.0310 - val_acc: 0.9859
Epoch 15/20
4960/4960 [=====] - 3s - loss: 0.0355 - acc: 0.9857 - val_loss: 0.0307 - val_acc: 0.9873
Epoch 16/20
4960/4960 [=====] - 3s - loss: 0.0247 - acc: 0.9893 - val_loss: 0.0283 - val_acc: 0.9868
Epoch 17/20
4960/4960 [=====] - 3s - loss: 0.0249 - acc: 0.9891 - val_loss: 0.0329 - val_acc: 0.9854
Epoch 18/20
4960/4960 [=====] - 3s - loss: 0.0299 - acc: 0.9895 - val_loss: 0.0285 - val_acc: 0.9882
Epoch 19/20
4960/4960 [=====] - 3s - loss: 0.0282 - acc: 0.9887 - val_loss: 0.0287 - val_acc: 0.9882
Epoch 20/20
4960/4960 [=====] - 3s - loss: 0.0401 - acc: 0.9839 - val_loss: 0.0311 - val_acc: 0.9878

2126/2126 [=====] - 0s
Test score: 0.031, accuracy: 0.986
```

Как видим, на тестовом наборе верность сети составила 98.6%.

Код этого примера находится в файле `learn_embedding_from_scratch.py` в исходном коде к этой главе.

Настройка погружений на основе предобученной модели word2vec

В этом примере мы будем использовать ту же сеть, что в предыдущем. В программе единственное существенное отличие – дополнительный код для загрузки модели word2vec и построения матрицы весов для слоя погружения.

Как всегда, начинаем с импорта и задания начального значения случайного генератора для воспроизводимости. Помимо того, что было импортировано в предыдущем примере, мы еще импортируем модель word2vec из пакета gensim:

```
from gensim.models import KeyedVectors
from keras.layers.core import Dense, Dropout, SpatialDropout1D
from keras.layers.convolutional import Conv1D
```



```

from keras.layers.embeddings import Embedding
from keras.layers.pooling import GlobalMaxPooling1D
from keras.models import Sequential
from keras.preprocessing.sequence import pad_sequences
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np

np.random.seed(42)

```

Далее задаются константы. По сравнению с предыдущим случаем мы уменьшили `NUM_EPOCHS` с 20 до 10. Напомним, что инициализация весов значениями, взятыми из предобученной модели, обычно ускоряет сходимость:

```

INPUT_FILE = "../data/umich-sentiment-train.txt"
WORD2VEC_MODEL = "../data/GoogleNews-vectors-negative300.bin.gz"
VOCAB_SIZE = 5000
EMBED_SIZE = 300
NUM_FILTERS = 256
NUM_WORDS = 3
BATCH_SIZE = 64
NUM_EPOCHS = 10

```

В следующей части из набора данных извлекаются слова и создается словарь самых частых термов, после чего набор данных разбирается с целью создания списка списков дополненных слов. Кроме того, метки преобразуются в категориальный формат. И наконец, мы разбиваем данные на обучающий и тестовый набор. Этот блок ничем не отличается от предыдущего примера, где были приведены подробные пояснения.

```

counter = collections.Counter()
fin = open(INPUT_FILE, "rb")
maxlen = 0
for line in fin:
    _, sent = line.strip().split("t")
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    if len(words) > maxlen:
        maxlen = len(words)
    for word in words:
        counter[word] += 1
fin.close()

word2index = collections.defaultdict(int)
for wid, word in enumerate(counter.most_common(VOCAB_SIZE)):

```

```

word2index[word[0]] = wid + 1
vocab_sz = len(word2index) + 1
index2word = {v:k for k, v in word2index.items()}

xs, ys = [], []
fin = open(INPUT_FILE, "rb")
for line in fin:
    label, sent = line.strip().split("t")
    ys.append(int(label))
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    wids = [word2index[word] for word in words]
    xs.append(wids)
fin.close()
X = pad_sequences(xs, maxlen=maxlen)
Y = np_utils.to_categorical(ys)

Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.3,
random_state=42)

```

Далее мы загружаем модель `word2vec`, предобученную на 10 миллиардах слов из новостей Google News со словарем на 3 миллиона слов. После загрузки мы ищем в модели векторы погружений для слов из нашего словаря и записываем вектор погружений в нашу матрицу весов `embedding_weights`. Строки этой матрицы весов соответствуют словам из словаря, а столбцы – вектору погружений слова.

Матрица `embedding_weights` имеет размер `vocab_sz × EMBED_SIZE`. Величина `vocab_sz` на единицу больше числа уникальных термов в словаре, дополнительная фиктивная лексема `_UNK_` представляет отсутствующие в словаре слова.

Вполне возможно, что в нашем словаре есть слова, отсутствующие в модели `word2vec` на базе GoogleNews. Для таких слов вектор погружений принимает значение по умолчанию – все нули.

```

# загрузить модель word2vec
word2vec = Word2Vec.load_word2vec_format(WORD2VEC_MODEL, binary=True)
embedding_weights = np.zeros((vocab_sz, EMBED_SIZE))
for word, index in word2index.items():
    try:
        embedding_weights[index, :] = word2vec[word]
    except KeyError:
        pass

```

Теперь определим нашу сеть. Отличие от предыдущего примера заключается в том, что веса слоя погружения, хранящиеся в матрице `embedding_weights`, инициализированы в предшествующей части программы:

```

model = Sequential()
model.add(Embedding(vocab_sz, EMBED_SIZE, input_length=maxlen,
    weights=[embedding_weights]))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(Conv1D(filters=NUM_FILTERS, kernel_size=NUM_WORDS, activation="relu"))
model.add(GlobalMaxPooling1D())
model.add(Dense(2, activation="softmax"))

```

Затем мы компилируем модель, применяя категориальную перекрестную энтропию в качестве функции потерь и оптимизатор Adam, и обучаем сеть при размере пакета 64 на протяжении 10 периодов. После этого оцениваем обученную модель.

```

model.compile(optimizer="adam", loss="categorical_crossentropy",
    metrics=["accuracy"])
history = model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE,
    epochs=NUM_EPOCHS, validation_data=(Xtest, Ytest))
score = model.evaluate(Xtest, Ytest, verbose=1)
print("Test score: {:.3f}, accuracy: {:.3f}".format(score[0], score[1]))

```

Ниже показаны результаты выполнения этой программы:

```

((4960, 42), (2126, 42), (4960, 2), (2126, 2))
Train on 4960 samples, validate on 2126 samples
Epoch 1/10
4960/4960 [=====] - 7s - loss: 0.1766 - acc:
0.9369 - val_loss: 0.0397 - val_acc: 0.9854
Epoch 2/10
4960/4960 [=====] - 7s - loss: 0.0725 - acc:
0.9706 - val_loss: 0.0346 - val_acc: 0.9887
Epoch 3/10
4960/4960 [=====] - 7s - loss: 0.0553 - acc:
0.9784 - val_loss: 0.0210 - val_acc: 0.9915
Epoch 4/10
4960/4960 [=====] - 7s - loss: 0.0519 - acc:
0.9790 - val_loss: 0.0241 - val_acc: 0.9934
Epoch 5/10
4960/4960 [=====] - 7s - loss: 0.0576 - acc:
0.9746 - val_loss: 0.0219 - val_acc: 0.9929
Epoch 6/10
4960/4960 [=====] - 7s - loss: 0.0515 - acc:
0.9764 - val_loss: 0.0185 - val_acc: 0.9929
Epoch 7/10
4960/4960 [=====] - 7s - loss: 0.0528 - acc:
0.9790 - val_loss: 0.0204 - val_acc: 0.9920
Epoch 8/10
4960/4960 [=====] - 7s - loss: 0.0373 - acc:
0.9849 - val_loss: 0.0221 - val_acc: 0.9934
Epoch 9/10
4960/4960 [=====] - 7s - loss: 0.0360 - acc:

```

```
0.9845 - val_loss: 0.0194 - val_acc: 0.9929
Epoch 10/10
4960/4960 [=====] - 7s - loss: 0.0389 - acc:
0.9853 - val_loss: 0.0254 - val_acc: 0.9915
2126/2126 [=====] - 1s
Test score: 0.025, accuracy: 0.993
```

После 10 периодов обучения модель показывает верность 99.3% на тестовом наборе. Это лучше, чем предыдущий пример, где была достигнута верность 98.6% после 20 периодов.

Код этого примера находится в файле `finetune_word2vec_embeddings.py` в исходном коде к этой главе.

Настройка погружений на основе предобученной модели GloVe

Погружения на основе предобученной модели GloVe настраиваются примерно так же, как в случае модели word2vec. На самом деле отличается только код построения матрицы весов для слоя погружения. Только его мы и рассмотрим.

Есть несколько видов предобученных моделей GloVe. Мы будем работать с той, что обучена на 6 миллиардах лексем и на корпусе текстов объемом порядка миллиарда слов из англоязычной википедии. Размер словаря модели составляет примерно 400 000 слов, имеются загружаемые файлы для размерности погружения 50, 100, 200 и 300. Мы возьмем файл для размерности 300.

Единственное, что нужно изменить в коде предыдущего примера, — часть, где создается модель word2vec и инициализируется ее матрица весов. А если бы мы взяли модель с размерностью, отличной от 300, то нужно было бы еще изменить константу `EMBED_SIZE`.

Векторы записаны в файле в текстовом формате через пробел, поэтому наша первая задача — прочитать их в словарь `word2emb`. Это делается аналогично разбору строки файла данных для модели word2vec.

```
GLOVE_MODEL = "../data/glove.6B.300d.txt"
word2emb = {}
fglove = open(GLOVE_MODEL, "rb")
for line in fglove:
    cols = line.strip().split()
    word = cols[0]
    embedding = np.array(cols[1:], dtype="float32")
    word2emb[word] = embedding
fglove.close()
```


Затем создаем матрицу весов погружения размера `vocab_sz × EMBED_SIZE` и заполняем ее векторами из словаря `word2emb`. Векторы, которые соответствуют словам, имеющимся в словаре, но отсутствующим в модели GloVe, остаются нулевыми:

```
embedding_weights = np.zeros((vocab_sz, EMBED_SIZE))
for word, index in word2index.items():
    try:
        embedding_weights[index, :] = word2emb[word]
    except KeyError:
        pass
```

Код этого примера находится в файле `finetune_glove_embeddings.py` в исходном коде к этой главе. Ниже показаны результаты выполнения программы:

```
((4960, 42), (2126, 42), (4960, 2), (2126, 2))
Train on 4960 samples, validate on 2126 samples
Epoch 1/10
4960/4960 [=====] - 7s - loss: 0.1748 - acc: 0.9240 - val_loss: 0.0390 - val_acc: 0.9840
Epoch 2/10
4960/4960 [=====] - 7s - loss: 0.0859 - acc: 0.9649 - val_loss: 0.0431 - val_acc: 0.9845
Epoch 3/10
4960/4960 [=====] - 7s - loss: 0.0586 - acc: 0.9754 - val_loss: 0.0528 - val_acc: 0.9779
Epoch 4/10
4960/4960 [=====] - 8s - loss: 0.0565 - acc: 0.9798 - val_loss: 0.0386 - val_acc: 0.9873
Epoch 5/10
4960/4960 [=====] - 8s - loss: 0.0792 - acc: 0.9683 - val_loss: 0.0233 - val_acc: 0.9892
Epoch 6/10
4960/4960 [=====] - 8s - loss: 0.0618 - acc: 0.9746 - val_loss: 0.0247 - val_acc: 0.9911
Epoch 7/10
4960/4960 [=====] - 7s - loss: 0.0569 - acc: 0.9752 - val_loss: 0.0266 - val_acc: 0.9906
Epoch 8/10
4960/4960 [=====] - 8s - loss: 0.0419 - acc: 0.9829 - val_loss: 0.0211 - val_acc: 0.9920
Epoch 9/10
4960/4960 [=====] - 7s - loss: 0.0371 - acc: 0.9849 - val_loss: 0.0206 - val_acc: 0.9920
Epoch 10/10
4960/4960 [=====] - 9s - loss: 0.0422 - acc: 0.9815 - val_loss: 0.0266 - val_acc: 0.9906
2126/2126 [=====] - 1s
Test score: 0.027, accuracy: 0.991
```

При обучении на протяжении 10 периодов достигается верность 99.1%, что почти не уступает результатам, полученным после настройки весов модели `word2vec`.

Поиск погружений

И последняя стратегия – поиск погружений в предобученной сети. Для этого проще всего задать в наших примерах параметр `trainable` слоя погружения равным `False`. Тогда при обратном распространении не будут обновляться веса этого слоя:

```
model.add(Embedding(vocab_sz, EMBED_SIZE, input_length=maxlen,
                    weights=[embedding_weights], trainable=False))
model.add(SpatialDropout1D(Dropout(0.2)))
```

Поступив так в примерах для моделей word2vec и GloVe, мы получим соответственно верность 98.7% и 98.9% после 10 периодов обучения.

Однако в общем случае предобученные погружения используются не так. Обычно производится предварительная обработка набора данных, цель которой – построить векторные представления слов путем поиска в какой-то предобученной модели, а затем воспользоваться этими данными для обучения другой модели. Вторая модель не будет содержать слоя погружения и вообще может не быть сетью глубокого обучения.

В примере ниже описана плотная сеть, которая принимает на входе вектор размера 100, представляющий предложение, и выводит 1, если предложение имеет положительную эмоциональную окраску, и 0 – если отрицательную. Мы по-прежнему используем набор данных из конкурса UMICH S1650, содержащий примерно 7000 предложений.

Как и раньше, большие куски кода повторяются, поэтому мы заострим внимание только на новых частях, нуждающихся в пояснении. В начале импортируются пакеты, инициализируется генератор случайных чисел и задаются значения констант. Для создания 100-мерных векторов для каждого предложения нам понадобится модель GloVe размерности 100, которая хранится в файле glove.6B.100d.txt:

```
from keras.layers.core import Dense, Dropout, SpatialDropout1D
from keras.models import Sequential
from keras.preprocessing.sequence import pad_sequences
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np

np.random.seed(42)

INPUT_FILE = "../data/umich-sentiment-train.txt"
GLOVE_MODEL = "../data/glove.6B.100d.txt"
VOCAB_SIZE = 5000
EMBED_SIZE = 100
BATCH_SIZE = 64
NUM_EPOCHS = 10
```

Далее мы читаем предложения и создаем таблицу частот слов. Из этой таблицы мы отбираем 5000 самых частых лексем и строим

таблицы соответствия (отображающие слова на индексы и наоборот). Для лексем, отсутствующих в словаре, в таблице создается фиктивная лексема `_UNK_`. Пользуясь этими таблицами, мы преобразуем каждое предложение в последовательность идентификаторов слов, дополняя все предложения до одинаковой длины (равной числу слов в самом длинном предложении). Кроме того, метки преобразуются в категориальный формат.

```
counter = collections.Counter()
fin = open(INPUT_FILE, "rb")
maxlen = 0
for line in fin:
    _, sent = line.strip().split("t")
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    if len(words) > maxlen:
        maxlen = len(words)
    for word in words:
        counter[word] += 1
fin.close()

word2index = collections.defaultdict(int)
for wid, word in enumerate(counter.most_common(VOCAB_SIZE)):
    word2index[word[0]] = wid + 1
vocab_sz = len(word2index) + 1
index2word = {v:k for k, v in word2index.items()}
index2word[0] = "_UNK_"

ws, ys = [], []
fin = open(INPUT_FILE, "rb")
for line in fin:
    label, sent = line.strip().split("t")
    ys.append(int(label))
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    wids = [word2index[word] for word in words]
    ws.append(wids)
fin.close()
W = pad_sequences(ws, maxlen=maxlen)
Y = np_utils.to_categorical(ys)
```

Векторы GloVe загружаются в словарь. Если бы мы захотели использовать модель `word2vec`, то нужно было бы лишь заменить этот блок вызовом функции `Word2Vec.load_word2vec_format()` из библиотеки `gensim`, а следующий – поиском в модели `word2vec`, а не в словаре `word2emb`:

```
word2emb = collections.defaultdict(int)
fglove = open(GLOVE_MODEL, "rb")
for line in fglove:
```

```
cols = line.strip().split()
word = cols[0]
embedding = np.array(cols[1:], dtype="float32")
word2emb[word] = embedding
fglove.close()
```

В следующем фрагменте мы ищем слова каждого предложения в матрице идентификаторов слов `W` и записываем в матрицу `E` соответствующий вектор погружения. Сумма этих векторов образует вектор предложения, который записывается в матрицу `X`. На выходе получается матрица `X` размера `num_records × EMBED_SIZE`:

```
X = np.zeros((W.shape[0], EMBED_SIZE))
for i in range(W.shape[0]):
    E = np.zeros((EMBED_SIZE, maxlen))
    words = [index2word[wid] for wid in W[i].tolist()]
    for j in range(maxlen):
        E[:, j] = word2emb[words[j]]
    X[i, :] = np.sum(E, axis=1)
```

Итак, мы завершили предварительную обработку данных с использованием предобученной модели и готовы применить их для обучения и оценки окончательной модели. Как обычно, разобьем данные на обучающий и тестовый набор в пропорции 70:30:

```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.3,
random_state=42)
```

Для анализа эмоциональной окраски мы обучим простую плотную сеть. При компиляции задаем категориальную перекрестную энтропию в качестве функции потерь и оптимизатор Adam и обучаем сеть на векторах предложений, построенных на основе предобученных погружений. И наконец, оцениваем модели на тестовом наборе.

```
model = Sequential()
model.add(Dense(32, input_dim=100, activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(2, activation="softmax"))
model.compile(optimizer="adam", loss="categorical_crossentropy",
metrics=["accuracy"])
history = model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE,
epochs=NUM_EPOCHS, validation_data=(Xtest, Ytest))
score = model.evaluate(Xtest, Ytest, verbose=1)
print("Test score: {:.3f}, accuracy: {:.3f}".format(score[0], score[1]))
```

Ниже показаны результаты выполнения программы:


```

((4960, 100), (2126, 100), (4960, 2), (2126, 2))
Train on 4960 samples, validate on 2126 samples
Epoch 1/10
4960/4960 [=====] - 0s - loss: 1.9577 - acc: 0.5667 - val_loss: 0.4448 - val_acc: 0.8556
Epoch 2/10
4960/4960 [=====] - 0s - loss: 0.5245 - acc: 0.7942 - val_loss: 0.3167 - val_acc: 0.9078
Epoch 3/10
4960/4960 [=====] - 0s - loss: 0.3026 - acc: 0.9002 - val_loss: 0.2456 - val_acc: 0.9473
Epoch 4/10
4960/4960 [=====] - 0s - loss: 0.2338 - acc: 0.9270 - val_loss: 0.2068 - val_acc: 0.9398
Epoch 5/10
4960/4960 [=====] - 0s - loss: 0.1802 - acc: 0.9520 - val_loss: 0.1720 - val_acc: 0.9581
Epoch 6/10
4960/4960 [=====] - 0s - loss: 0.1561 - acc: 0.9552 - val_loss: 0.1561 - val_acc: 0.9610
Epoch 7/10
4960/4960 [=====] - 0s - loss: 0.1396 - acc: 0.9631 - val_loss: 0.1535 - val_acc: 0.9577
Epoch 8/10
4960/4960 [=====] - 0s - loss: 0.1216 - acc: 0.9645 - val_loss: 0.1338 - val_acc: 0.9628
Epoch 9/10
4960/4960 [=====] - 0s - loss: 0.1152 - acc: 0.9641 - val_loss: 0.1273 - val_acc: 0.9643
Epoch 10/10
4960/4960 [=====] - 0s - loss: 0.1044 - acc: 0.9706 - val_loss: 0.1257 - val_acc: 0.9647

1888/2126 [=====>....] - ETA: 0s
Test score: 0.126, accuracy: 0.965

```

Плотная сеть с предварительной обработкой на 100-мерной модели GloVe дает верность 96.5% на тестовом наборе после обучения на протяжении 10 периодов. Сеть с предварительной обработкой на 300-мерной модели word2vec дает верность 98.5%.

Код этого примера находится в файле `transfer_glove_embeddings.py` (для примера с моделью GloVe) и в файле `transfer_word2vec_embeddings.py` (для примера с моделью word2vec) в исходном коде к этой главе.

Резюме

В этой главе мы изучили, как преобразовать слова из текста в векторные представления с сохранением дистрибутивной семантики слов. Мы также на интуитивном уровне поняли, почему погружения слов в векторное пространство демонстрируют такое поведение и почему они полезны для применения в глубоких моделях текстовых данных.

Затем мы рассмотрели две популярные модели погружения слов, word2vec и GloVe, и уяснили, как они работают. Мы также познакомились с применением библиотеки `gensim` для обучения модели word2vec на данных.

Наконец, мы узнали о различных способах использования погружений в собственной сети. Первый – обучить веса погружений с нуля в процессе обучения сети. Второй – импортировать веса погружений из предобученной модели word2vec или GloVe в свою

сеть и настроить их в процессе обучения сети. Третий – использовать предобученные веса непосредственно в своем приложении.

В следующей главе мы узнаем о рекуррентных нейронных сетях, оптимизированных для обработки последовательности, в т. ч. текста.

Глава 6

Рекуррентная нейронная сеть – РНС

В главе 3 мы познакомились со **сверточными нейронными сетями** (СНС) узнали, как в них используется пространственная геометрия входных данных. Так, операции свертки и пулинга применяются вдоль временной оси для звуковых данных, в двух пространственных измерениях для изображений и в трех измерениях (высота, ширина, время) для видео.

В этой главе мы будем говорить о **рекуррентных нейронных сетях** (РНС, англ. RNN) – классе нейронных сетей, в которых учитывается последовательный характер входных данных. На вход такой сети может подаваться текст, речь, временной ряд или еще какие-то данные, в которых появление элемента в последовательности зависит от предшествующих элементов. Например, следующим словом в предложении «собака...» скорее будет «лает», чем «машина», и именно его РНС предскажет с большей вероятностью.

РНС можно рассматривать как граф, состоящий из элементарных ячеек, каждая из которых выполняет одну и ту же операцию для каждого элемента последовательности. РНС обладают большой гибкостью и применяются для решения таких задач, как распознавание речи, языковое моделирование, машинный перевод, анализ эмоциональной окраски, подписывание изображений и многих других. РНС можно адаптировать к различным типам задач, изменяя конфигурацию ячеек в графе. Мы рассмотрим несколько таких конфигураций и их применение к конкретным задачам.

Мы также узнаем об основном ограничении простой ячейки РНС и о двух ее вариантах: **долгой краткосрочной памяти** (long short term memory, LSTM) и **вентильном рекуррентном блоке** (gated recurrent unit, GRU), позволяющих преодолеть это ограни-

чение. LSTM и GRU можно подставить вместо простой ячейки, и зачастую это заметно улучшает качество сети. Хотя LSTM и GRU – не единственные варианты, эмпирически показано (см. статьи R. Jozefowicz, W. Zaremba, I. Sutskever «An Empirical Exploration of Recurrent Network Architectures», JMLR, 2015 и K. Greff «LSTM: A Search Space Odyssey», arXiv:1503.04069, 2015), что для большинства задач обработки последовательностей они оказываются наилучшими.

Наконец, мы дадим несколько рекомендаций о том, как повысить качество РНС, и о том, когда и как их следует применять.

В этой главе рассматриваются следующие вопросы:

- простая ячейка РНС;
- реализация РНС для порождения текста с помощью Keras;
- топологии РНС;
- LSTM, GRU и другие варианты РНС.

Простые ячейки РНС

Традиционно в нейронных сетях на основе многослойных перцептронов предполагается, что все входы независимы. Для последовательных данных это предположение нарушается. В предыдущем разделе мы видели пример, когда первое слово предложения влияет на второе. То же относится и к речи – разговаривая в шумной комнате, я могу выдвинуть разумную гипотезу о слове, которое не расслышал, исходя из слов, произнесенных собеседником ранее. Для временных рядов, например цен на акции или прогнозов погоды, также характерна зависимость от прошлых данных, это явление называется долговременным трендом.

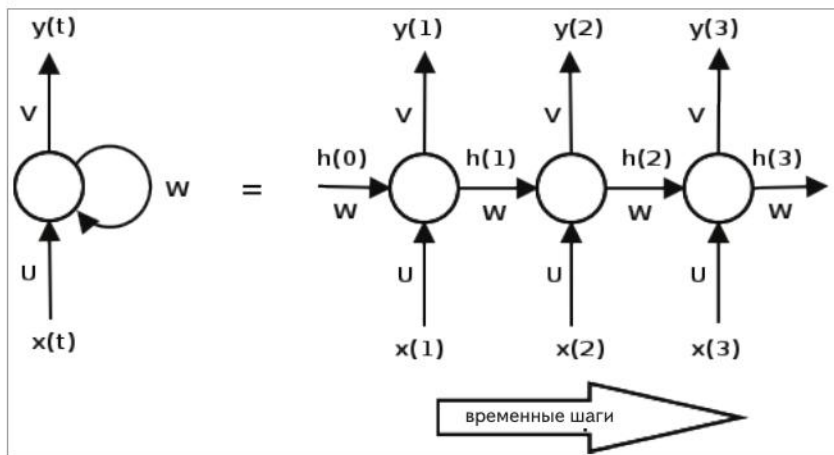
В ячейках РНС эта зависимость представляется с помощью скрытого состояния, или памяти, в которой хранится сводка прошлой информации. Значение скрытого состояния в любой момент времени – функция его значения на предыдущем шаге и значения данных на текущем шаге:

$$h_t = \varphi(h_{t-1}, x_t),$$

где h_t и h_{t-1} – значения скрытого состояния на шаге t и $t-1$ соответственно, и x_t – входное значение в момент t . Отметим, что это уравнение рекуррентное, т. е. h_{t-1} можно выразить через h_{t-2} и x_{t-1} и т. д., пока мы не дойдем до начала последовательности. Именно

так в РНС кодируется и запоминается информация о сколь угодно длинной последовательности.

Мы также можем представить ячейку РНС графически, как показано в левой части следующего рисунка. В момент t ячейка получает на входе значение x_t и выводит значение y_t . Часть y_t (скрытое состояние h_t) подается обратно на вход ячейки для использования на следующем шаге $t+1$. Если параметры традиционной нейронной сети хранятся в матрице весов, то параметры РНС задаются тремя матрицами весов, U , V и W , соответствующими входу, выводу и скрытому состоянию.



Еще один взгляд на РНС – *развертка*, показанная на том же рисунке справа. Это означает, что мы рисуем сеть на протяжении всей последовательности. На рисунке изображена РНС с тремя слоями, пригодная для обработки последовательностей с тремя элементами. Заметим, что матрицы весов U , V и W разделяются между всеми шагами, поскольку на каждом шаге к разным данным применяются одни и те же операции. Благодаря использованию одних и тех же весов на всех временных шагах удастся существенно снизить количество обучаемых параметров РНС.

Вычисления, выполняемые РНС, можно также описать в виде уравнений. Внутреннее состояние РНС в момент t определяется значением вектора h_t , равного результату применения нелинейности $\tanh$ к сумме произведения матрицы весов W на скрытое состояние h_{t-1} в момент $t-1$ и произведения матрицы весов U на входное значение x_t в момент t . Выбор нелинейности $\tanh$, а не какой-то

другой, связан с тем, что ее вторая производная очень медленно убывает, приближаясь к нулю. Поэтому градиенты остаются в линейной части функции активации, что помогает справиться с проблемой исчезающего градиента. Подробнее об этой проблеме мы поговорим ниже.

Выходной вектор y_t в момент t равен результату применения функции *softmax* к произведению матрицы весов V на скрытое состояние h_t и представляет набор вероятностей выхода:

$$\begin{aligned}h_t &= \tanh(W h_{t-1} + U x_t) \\ y_t &= \text{softmax}(V h_t)\end{aligned}$$

Keras предоставляет слой рекуррентной сети SimpleRNN (см. <https://keras.io/layers/recurrent/>), включающий всю описанную выше логику, а также более эффективные варианты LSTM и GRU, которые будут рассмотрены ниже. Чтобы использовать их, необязательно точно понимать, как они работают. Однако знать структуру и уравнения полезно, если вы захотите построить свою РНС для решения поставленной задачи.

Простая РНС с применением Keras – порождение текста

РНС активно используются в **обработке естественных языков** (ОЕЯ, англ. NLP) для решения различных задач. Одна из них – построение языковых моделей. Такая модель позволяет предсказать вероятность появления слова в тексте при условии известных предыдущих слов. Языковые модели важны для таких высокоуровневых приложений, как машинный перевод, исправление правописания и т. д.

Побочным эффектом умения предсказывать следующее слово по известным предыдущим является порождающая модель, которая генерирует текст путем выборки слов из выходного распределения. В случае языкового моделирования входом обычно является последовательность слов, а выходом – последовательность предсказанных слов. В роли обучающих данных выступает имеющийся непомеченный текст, и метка y_t в момент t становится входом x_{t+1} в момент $t+1$.

Первым нашим примером использования Keras для построения РНС будет языковая модель, обученная предсказывать следующий символ по 10 предыдущим на тексте «Алисы в Стране чудес». Мы

остановились на модели для предсказания символа, потому что у нее меньше словарь и обучение проходит быстрее. Но та же идея применима и к предсказанию слов, нужно только символы заменить словами. Обученная модель будет использована для порождения нового текста в том же стиле.

Сначала импортируем модули:

```
from __future__ import print_function
from keras.layers import Dense, Activation
from keras.layers.recurrent import SimpleRNN
from keras.models import Sequential
from keras.utils.visualize_util import plot
import numpy as np
```

Входной текст «Алисы в Стране чудес» (на английском языке) берем с сайта проекта Гутенберг по адресу <http://www.gutenberg.org/files/11/11-0.txt>. Файл содержит символы конца строки и символы не в кодировке ASCII, поэтому произведем предварительную обработку и запишем результат в переменную `text`:

```
fin = open("../data/alice_in_wonderland.txt", 'rb')
lines = []
for line in fin:
    line = line.strip().lower()
    line = line.decode("ascii", "ignore")
    if len(line) == 0:
        continue
    lines.append(line)
fin.close()
text = " ".join(lines)
```

Поскольку наша РНС будет предсказывать символы, то и словарь состоит и множества символов, встречающихся в тексте. Таковых в нашем случае 42. Мы будем иметь дело не с самими символами, а с их индексами, поэтому в следующем фрагменте создаются необходимые таблицы соответствия:

```
chars = set([c for c in text])
nb_chars = len(chars)
char2index = dict((c, i) for i, c in enumerate(chars))
index2char = dict((i, c) for i, c in enumerate(chars))
```

Следующий шаг – создание входных строк и меток. Для этого проходим по тексту с шагом `STEP` символов (в нашем случае 1) и выделяем отрезки длиной `SEQLEN` (в нашем случае 10). Следующий после отрезка символ будем меткой:

```

SEQLEN = 10
STEP = 1

input_chars = []
label_chars = []
for i in range(0, len(text) - SEQLEN, STEP):
    input_chars.append(text[i:i + SEQLEN])
    label_chars.append(text[i + SEQLEN])

```

Этот код строит из текста *it turned into a pig* такую последовательность входных строк и меток:

```

it turned -> i
t turned i -> n
turned in -> t
turned int -> o
urned into ->
rned into -> a
ned into a ->
ed into a -> p
d into a p -> i
into a pi -> g

```

Следующий шаг – векторизация входных строк и меток. На вход РНС подаются построенные выше входные строки. В каждой из них `SEQLEN` символов, а поскольку размер нашего словаря составляет `nb_chars` символов, то каждый входной символ представляется унитарным вектором длины `nb_chars`. Следовательно, каждый входной элемент представляет собой тензор формы $SEQLEN \times nb_chars$. Выходная метка – это единственный символ, поэтому по аналогии с представлением входных символов она представляется унитарным вектором длины `nb_chars`.

```

X = np.zeros((len(input_chars), SEQLEN, nb_chars), dtype=np.bool)
y = np.zeros((len(input_chars), nb_chars), dtype=np.bool)
for i, input_char in enumerate(input_chars):
    for j, ch in enumerate(input_char):
        X[i, j, char2index[ch]] = 1
    y[i, char2index[label_chars[i]]] = 1

```

И наконец-то мы готовы построить модель. Размерность выхода РНС пусть будет равна 128. Это гиперпараметр, определяемый в ходе экспериментов. В общем случае, если выбрать слишком маленькое значение, то емкость модели будет недостаточна для порождения хороших текстов, и мы увидим длинные серии повторяющихся символов или повторяющиеся группы слов. Если же значение велико, то у модели будет слишком много параметров,

так что для ее обучения потребуется гораздо больше данных. Мы хотим получать на выходе один символ, а не последовательность, поэтому задаем параметр `return_sequences=False`. Входные данные РНС имеют, как мы видели, форму матрицы $SEQLEN \times nb_chars$. Кроме того, мы задаем `unroll=True`, потому что при этом повышается качество работы базовой библиотеки TensorFlow.

РНС соединяется с плотным (полносвязным) слоем. В плотном слое `nb_char` нейронов, которые выдают оценки появления каждого символа из словаря. Функцией активации в этом слое является `softmax`, которая нормирует оценки, преобразуя их в вероятности. Символ с наибольшей вероятностью возвращается в качестве предсказания. При компиляции модели задается категориальная перекрестная энтропия в качестве функции потерь (она хорошо подходит для категориального выхода) и оптимизатор `RMSprop`:

```
HIDDEN_SIZE = 128
BATCH_SIZE = 128
NUM_ITERATIONS = 25
NUM_EPOCHS_PER_ITERATION = 1
NUM_PREDS_PER_EPOCH = 100

model = Sequential()
model.add(SimpleRNN(HIDDEN_SIZE, return_sequences=False,
    input_shape=(SEQLEN, nb_chars), unroll=True))
model.add(Dense(nb_chars))
model.add(Activation("softmax"))
model.compile(loss="categorical_crossentropy", optimizer="rmsprop")
```

Подход к обучению немного отличается от того, что мы видели раньше. До сих пор мы обучали модель в течение фиксированного числа периодов, а затем оценивали ее на зарезервированных для этой цели тестовых данных. Поскольку в данном случае у нас нет помеченных данных, то мы выполняем один период обучения (`NUM_EPOCHS_PER_ITERATION=1`), а затем тестируем модель. Так происходит на протяжении 25 итераций (`NUM_ITERATIONS=25`). Следовательно, по существу мы выполняем `NUM_ITERATIONS` периодов обучения и тестируем модель после каждого периода.

Тестирование производится так: модель порождает символ по заданным входным данным, затем первый символ входной строки отбрасывается, в конец дописывается предсказанный на предыдущем прогоне символ и у модели запрашивается следующее предсказание. Так повторяется 100 раз (`NUM_PREDS_PER_EPOCH=100`), по-

сле чего получившаяся строка печатается. Эта строка и является индикатором качества модели:

```
for iteration in range(NUM_ITERATIONS):
    print("=" * 50)
    print("Iteration #: %d" % (iteration))
    model.fit(X, y, batch_size=BATCH_SIZE, epochs=NUM_EPOCHS_PER_ITERATION)

    test_idx = np.random.randint(len(input_chars))
    test_chars = input_chars[test_idx]
    print("Generating from seed: %s" % (test_chars))
    print(test_chars, end="")
    for i in range(NUM_PREDS_PER_EPOCH):
        Xtest = np.zeros((1, SEQLen, nb_chars))
        for i, ch in enumerate(test_chars):
            Xtest[0, i, char2index[ch]] = 1
        pred = model.predict(Xtest, verbose=0)[0]
        ypred = index2char[np.argmax(pred)]
        print(ypred, end="")
        # сдвинуться вперед на test_chars + ypred
        test_chars = test_chars[1:] + ypred
    print()
```

Ниже показан результат работы программы. Вначале модель предсказывает вздор, но к концу 25-го периода она уже пишет почти без ошибок, хотя со связностью мыслей дело обстоит неважно. Удивительно, что модель обучалась выводить символы и не имеет ни малейшего представления о словах, и тем не менее она научилась порождать слова, выглядящие так, будто взяты из оригинального текста.

```
=====
Iteration #: 21
Epoch 1/1
142544/142544 [=====] - 10s - loss: 1.3916
Generating from seed: e with the
e with the white rabbit had no the that the mouse the mouse the mouse the mouse the mouse the mouse
=====
Iteration #: 22
Epoch 1/1
142544/142544 [=====] - 10s - loss: 1.3831
Generating from seed: and an ol
and an ollar the caterpillar the seapped did not a moment the cook of the courter the caterpillar the seapped
=====
Iteration #: 23
Epoch 1/1
142544/142544 [=====] - 10s - loss: 1.3757
Generating from seed: ' the mock
' the mock turtle said the dormouse some of the conce in the dormouse some of the conce in the dormouse some o
=====
Iteration #: 24
Epoch 1/1
142544/142544 [=====] - 10s - loss: 1.3685
Generating from seed: raving mad
raving made to goon of the sord alice could got to the dormouse so they looked at the sord alice could got to
```

Порождение следующего символа или слова – не единственное, на что способна такая модель. Подобные модели успешно применялись для предсказания цен акций (см. A. Bernal, S. Fok, R. Pidaparathi «Financial Market Time Series Prediction with Recurrent Neural Networks», 2012) и для генерации классической музыки (см. G. Hadjeres, F. Pachet «DeepBach: A Steerable Model for Bach Chorales Generation», arXiv:1612.01010, 2016). Андрей Карпатый приводит еще несколько любопытных примеров и исходный код для Linux в статье «The Unreasonable Effectiveness of Recurrent Neural Networks» в своем блоге по адресу <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>.

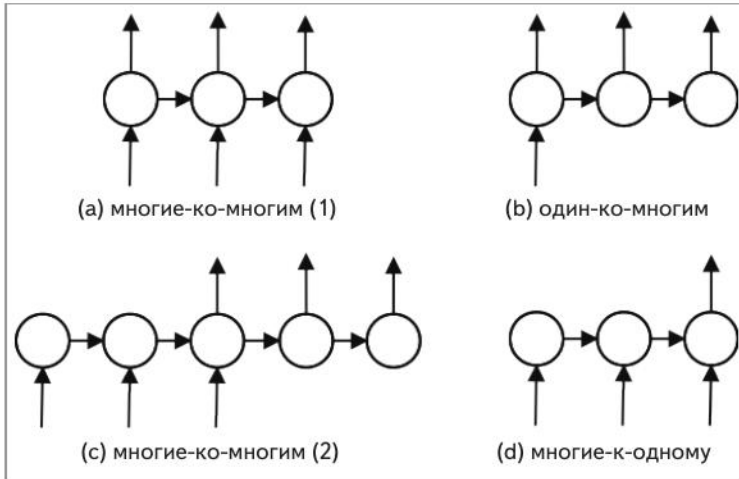
Код этого примера находится в файле `alice_chargen_rnn.py` в исходном коде к этой главе. Сами данные можно найти на сайте проекта Гутенберг.

Топологии РНС

API многослойного перцептрона и СНС ограничены. Обе архитектуры принимают на входе и порождают на выходе тензоры фиксированного размера, а для преобразования входа в выход выполняют фиксированное число шагов, определяемое числом слоев сети. У РНС такого ограничения нет – входом, выходом или тем и другим могут быть последовательности. Это означает, что для решения конкретных задач РНС можно конфигурировать разными способами.

Как мы уже знаем, РНС комбинирует входной вектор с предыдущим вектором состояния для получения нового вектора состояния. Это можно рассматривать как аналог выполнения программы с некоторыми входными данными и внутренними переменными. Следовательно, РНС можно считать способом описания компьютерных программ. На самом деле, доказано, что РНС являются полными по Тьюрингу исполнителями (см. Н. Т. Siegelmann, E. D. Sontag «On the Computational Power of Neural Nets», Proceedings of the Fifth Annual Workshop on Computational Learning Theory, ACM, 1992) в том смысле, что при задании надлежащих весов они могут моделировать произвольные программы.

Умение работать с последовательностями открывает возможность для различных топологий, некоторые из которых мы обсудим ниже.



Все эти топологии вытекают из общей базовой структуры, показанной на предыдущем рисунке. В базовой структуре все входные последовательности имеют одинаковую длину, а выход порождается на каждом временном шаге. Пример мы уже видели в сети порождения символов, обученной на тексте «Алисы в Стране чудес».

Другой пример РНС типа многие-ко-многим – сеть машинного перевода на рисунке (b), являющаяся представителем общего семейства сетей последовательность-в-последовательность (см. O. Vinyals «Grammar as a Foreign Language», *Advances in Neural Information Processing Systems*, 2015). Они принимают на входе последовательность и порождают другую последовательность. В случае машинного перевода входом может быть, например, последовательность английских слов, а выходом – переведенное предложение на испанском языке. Такой же тип модели используется для **частеречной разметки**, когда входом являются слова предложения, а выходом – соответствующие метки частей речи. От предыдущей топологии эта отличается тем, что в некоторые моменты времени может отсутствовать вход, а в некоторые – выход. С примером такой сети мы встретимся ниже.

Еще один вариант топологии – сеть типа один-ко-многим на рисунке (c), примером которой может служить сеть для подписывания изображений (см. A. Karpathy, F. Li «Deep Visual-Semantic Alignments for Generating Image Descriptions», *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015),

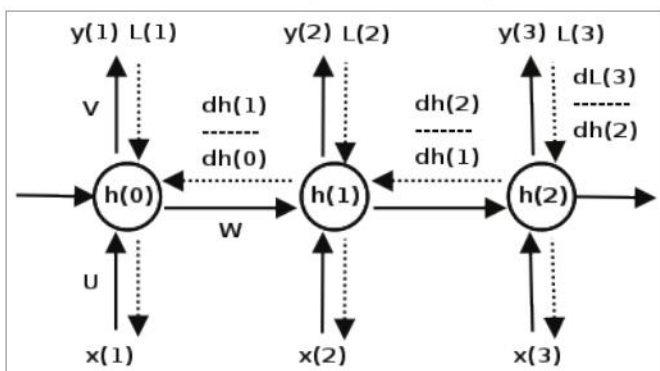
где входом является изображение, а выходом – последовательность слов.

Пример сети типа многие-к-одному на рисунке (d) – сеть анализа эмоциональной окраски предложений, когда входом является последовательность слов, а выходом – индикатор положительной или отрицательной окраски (см. R. Socher «Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank», Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP). Vol. 1631, 2013). Ниже в этой главе мы рассмотрим пример такой топологии (правда, существенно упрощенный по сравнению с цитированной выше моделью).

Проблема исчезающего и взрывного градиента

Как и в традиционных нейронных сетях, обучение РНС включает обратное распространение. Различие в том, что поскольку на всех шагах используются одинаковые параметры, то градиент в каждом выходе зависит не только от текущего временного шага, но и от предыдущих.

Этот процесс называется **обратным распространением во времени** (backpropagation through time, BPTT) (см. статью G. E. Hinton, D. E. Rumelhart, R. J. Williams «Learning Internal Representations by Backpropagating errors», Parallel Distributed Processing: Explorations in the Microstructure of Cognition 1, 1985).



Рассмотрим небольшую трехслойную РНС, показанную на рисунке выше. В процессе прямого распространения (сплошные

линии) сеть порождает предсказания, которые сравниваются с метками для вычисления потери L_t на каждом временном шаге. В процессе обратного распространения (пунктирные линии) на каждом временном шаге вычисляются градиенты функции потерь по параметрам U , V и W , и сумма градиентов применяется для обновления параметров.

В следующем уравнении показан градиент функции потерь по W – матрице, в которой закодированы веса для долгосрочных зависимостей. Мы акцентируем внимание на этой части обновления, потому что именно она – причина проблемы исчезающего и взрывного градиента. Два других градиента функции потерь по матрицам U и V также суммируются по всем временным шагам:

$$\frac{\partial L}{\partial W} = \sum_t \frac{\partial L_t}{\partial W}$$

Теперь посмотрим, что происходит с градиентом функции потерь на последнем временном шаге ($t = 3$). Как видим, этот градиент можно разложить в произведение трех подградиентов, применив правило дифференцирования сложной функции. Градиент скрытого состояния h_2 по W можно затем представить в виде суммы градиентов каждого скрытого состояния по предыдущему. Наконец, градиент скрытого состояния по предыдущему можно разложить в произведение градиентов текущего скрытого состояния по предыдущему:

$$\begin{aligned} \frac{\partial L_3}{\partial W} &= \frac{\partial L_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial W} \\ &= \sum_{i=0}^2 \frac{\partial L_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_i} \cdot \frac{\partial h_i}{\partial W} \\ &= \sum_{i=0}^2 \frac{\partial L_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial h_2} \cdot \left(\prod_{j=i+1}^2 \frac{\partial h_j}{\partial h_{j-1}} \right) \cdot \frac{\partial h_i}{\partial W} \end{aligned}$$

Аналогично вычисляются градиенты функции потерь L_1 и L_2 (на шагах 1 и 2) по W , после чего их сумма используется для обновления градиента по W . Мы не станем в этой книге вдаваться в математические детали. Если вам это интересно, почитайте в

блоге WILDML (<https://goo.gl/1061bX>) статью, содержащую очень хорошее объяснение БРТТ с подробными математическими выкладками.

Ну а нам последнего выражения градиента в формуле выше достаточно, чтобы понять, откуда возникает проблема исчезающего и взрывного градиента в РНС. Рассмотрим случай, когда отдельные градиенты скрытого состояния по предыдущему меньше 1. При обратном распространении через несколько временных шагов произведение градиентов становится все меньше и меньше, что и ведет к проблеме исчезающего градиента. С другой стороны, если градиенты больше 1, то произведения растут – и вот вам проблема взрывного градиента.

Из-за эффекта исчезающего градиента получается, что градиенты на отдаленных шагах не дают никакого вклада в процесс обучения, так что РНС не может учесть долговременные зависимости. Эта проблема может возникнуть и в традиционной нейронной сети, но в случае РНС она проявляется более рельефно, потому что в РНС больше слоев (временных шагов), через которые происходит обратное распространение.

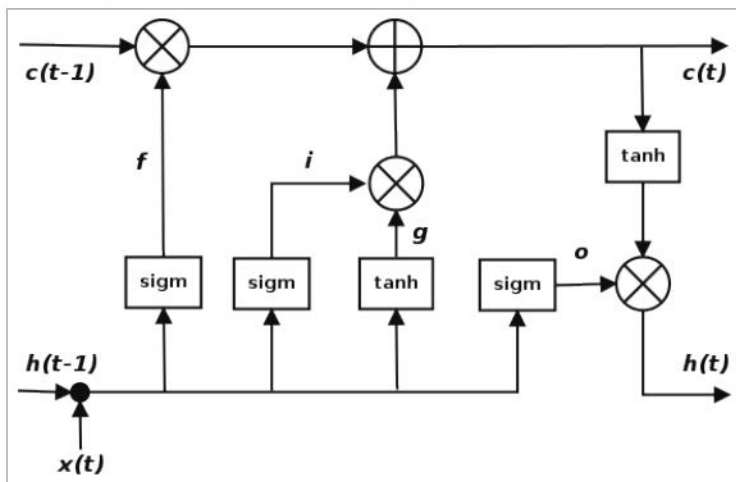
Взрывные градиенты обнаруживаются проще, поскольку когда градиент становится слишком большим и превращается в **нечисло (NaN)**, процесс обучения аварийно завершается. Рост градиентов можно контролировать, обрезая их при достижении заданного порога (см. R. Pascanu, T. Mikolov, Y. Bengio «On the Difficulty of Training Recurrent Neural Networks», ICML, Pp 1310–1318, 2013).

Существует несколько подходов к смягчению проблемы исчезающих градиентов, в частности, хорошая инициализация матрицы W , использование функции активации ReLU вместо tanh и предобучение слоев без учителя, но наиболее популярны архитектуры LSTM и GRU. Они специально проектировались для борьбы с исчезающим градиентом и более эффективно обучаются долговременным зависимостям.

Долгая краткосрочная память – LSTM

LSTM – это вариант РНС, способный обучаться долгосрочным зависимостям. LSTM-сети впервые были предложены Хохрайтером и Шмидхубером, а затем улучшены многими другими исследователями. Они хорошо работают для широкого круга задач и являются самым популярным типом РНС.

Мы видели, как в простой РНС для реализации рекуррентности используется комбинация скрытого состояния на предыдущем шаге и текущих входных данных в слое с функцией активации. В LSTM-сети рекуррентность реализуется аналогично, но $\tanh$ -слоев не один, а четыре, и взаимодействуют они весьма специфичным образом. На рисунке ниже показаны преобразования, применяемые к скрытому состоянию на временном шаге t :



Выглядит сложно, но мы рассмотрим эту схему по шагам. На горизонтальной линии сверху показано состояние ячейки c , оно представляет внутреннюю память блока. На линии снизу показано скрытое состояние, а вентили i , f , o и g – это механизмы, благодаря которым LSTM-сеть обходит проблему исчезающего градиента. В процессе обучения LSTM находит параметры этих вентилей.

Чтобы лучше понять, как эти вентили модулируют скрытое состояние LSTM-сети, рассмотрим формулы вычисления скрытого состояния h_t в момент t по состоянию h_{t-1} на предыдущем шаге:

$$\begin{aligned}
 i &= \sigma(W_i h_{t-1} + U_i x_t) \\
 f &= \sigma(W_f h_{t-1} + U_f x_t) \\
 o &= \sigma(W_o h_{t-1} + U_o x_t) \\
 g &= \tanh(W_g h_{t-1} + U_g x_t) \\
 c_t &= (c_{t-1} \otimes f) \oplus (g \otimes i) \\
 h_t &= \tanh(c_t) \otimes o
 \end{aligned}$$

Здесь i , f и o – входной вентиль, вентиль забывания и выходной вентиль. Все они вычисляются по одним и тем же формулам, но с разными матрицами параметров. Сигмоидная функция модулирует выход вентиля, приводя его к диапазону от 0 до 1, так что порождаемый выходной вектор можно умножить поэлементно на другой вектор, чтобы определить, какая часть второго вектора может пройти через первый.

Вентиль забывания определяет, какую часть предыдущего состояния h_{t-1} желательно пропустить дальше. Входной вентиль определяет, какую часть вновь вычисленного состояния для текущего входа x_t пропустить, а выходной вентиль – какую часть внутреннего состояния передать следующему слою. Внутреннее скрытое состояние g вычисляется на основе текущего входа x_t и предыдущего скрытого состояния h_{t-1} . Отметим, что выражение для g совпадает с аналогичным выражением для ячейки простой РНС, но в данном случае мы модулируем выход, смешивая его с выходом входного вентиля i .

Зная i , f , o и g , мы можем вычислить состояние ячейки c_t в момент t в терминах произведения c_{t-1} на вентиль забывания и произведения g на входной вентиль i . Это и есть способ комбинирования предыдущего содержимого памяти с новым входом. Если вентиль забывания установлен в 0, то старое запомненное состояние полностью игнорируется, а если установить в 0 входной вентиль, то игнорируется новое вычисленное состояние.

Наконец, скрытое состояние h_t в момент t вычисляется путем умножения памяти c_t на значение выходного вентиля.

Важно понимать, что LSTM всегда можно подставить вместо ячейки типа SimpleRNN, и единственная разница состоит в том, что LSTM устойчива к проблеме исчезающего градиента. Заменяв ячейку РНС на LSTM, мы можем не волноваться ни о каких побочных эффектах.

Как правило результат получается тем лучше, чем больше время обучения. Для интересующихся в блоге WILDML имеется очень подробное объяснение принципов работы вентилях LSTM. Более наглядное пояснение есть в статье Кристофера Ола «Understanding LSTMs» (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>), где все вычисления разбираются шаг за шагом, и каждый шаг сопровождается иллюстрациями.

Пример LSTM – анализ эмоциональной окраски

Keras предоставляет слой LSTM, которым мы воспользуемся, чтобы построить и обучить РНС типа многие-к-одному. Сеть будет принимать предложение (последовательность слов) и выдавать индикатор эмоциональной окраски (положительной или отрицательной). Обучающий набор состоит примерно из 7000 коротких предложений, предлагавшихся на конкурсе Kaggle UMich SI650 по классификации эмоциональной окраски (<https://inclass.kaggle.com/c/si650winter11>). Каждое предложение снабжено меткой 1 (положительная окраска) или 0 (отрицательная окраска).

Как обычно, начинаем с импорта:

```
from keras.layers.core import Activation, Dense, Dropout, SpatialDropout1D
from keras.layers.embeddings import Embedding
from keras.layers.recurrent import LSTM
from keras.models import Sequential
from keras.preprocessing import sequence
from sklearn.model_selection import train_test_split
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np
import os
```

Предварительно займемся исследовательским анализом данных. Нам нужно знать, сколько уникальных слов встречается в корпусе текстов и сколько слов в каждом предложении:

```
maxlen = 0
word_freqs = collections.Counter()
num_recs = 0
ftrain = open(os.path.join(DATA_DIR, "umich-sentiment-train.txt"), 'rb')
for line in ftrain:
    label, sentence = line.strip().split("\t")
    words = nltk.word_tokenize(sentence.decode("ascii", "ignore").lower())
    if len(words) > maxlen:
        maxlen = len(words)
    for word in words:
        word_freqs[word] += 1
    num_recs += 1
ftrain.close()
```

Для нашего корпуса получаем такие числа:

```
maxlen : 42
len(word_freqs) : 2313
```

Зная количество уникальных слов `len(word_freqs)`, мы задаем фиксированный размер словаря, а все остальные слова считаем несловарными и заменяем их фиктивным словом UNK (unknown). На этапе предсказания это позволит нам обрабатывать ранее не встречавшиеся слова как несловарные.

Зная число слов в предложении (`maxlen`), мы можем задать фиксированную длину предложения и более короткие предложения дополнять нулями, а более длинные обрезать. Хотя РНС способна обрабатывать последовательности переменной длины, достигается это обычно дополнением и обрезанием, как описано выше, или группировкой входных данных в пакеты, содержащие последовательности одинаковой длины. Мы будем использовать первый подход. Что касается второго, Keras рекомендует пакеты длины 1 (см. <https://github.com/fchollet/keras/issues/40>).

Исходя из вычисленных показателей, мы задаем `VOCABULARY_SIZE` равным 2002. Это 2000 слов в словаре плюс фиктивное слово UNK плюс фиктивное слово PAD (используется для дополнения предложений до фиксированного числа слов, в нашем случае `MAX_SENTENCE_LENGTH = 40`).

```
DATA_DIR = "../data"
```

```
MAX_FEATURES = 2000
```

```
MAX_SENTENCE_LENGTH = 40
```

Далее нам понадобится пара таблиц соответствия. Входными данными для РНС является строка индексов слов, причем слова упорядочены по убыванию частоты встречаемости в обучающем наборе. Таблицы соответствия позволяют находить индекс по слову и слово по индексу (включая фиктивные слова PAD и UNK):

```
vocab_size = min(MAX_FEATURES, len(word_freqs)) + 2
word2index = {x[0]: i+2 for i, x in enumerate(word_freqs.most_common(MAX_FEATURES))}
word2index["PAD"] = 0
word2index["UNK"] = 1
index2word = {v:k for k, v in word2index.items()}
```

Затем мы преобразуем входные предложения в последовательности индексов слов, дополняя их до `MAX_SENTENCE_LENGTH` слов. Поскольку в нашем случае результатом является бинарная величина (положительная или отрицательная эмоциональная окраска), обрабатывать метки не нужно:

```
X = np.empty((num_recs, ), dtype=list)
y = np.zeros((num_recs, ))
```

```

i = 0
ftrain = open(os.path.join(DATA_DIR, "umich-sentiment-train.txt"), 'rb')
for line in ftrain:
    label, sentence = line.strip().split("t")
    words = nltk.word_tokenize(sentence.decode("ascii", "ignore").lower())
    seqs = []
    for word in words:
        if word2index.has_key(word):
            seqs.append(word2index[word])
        else:
            seqs.append(word2index["UNK"])
    X[i] = seqs
    y[i] = int(label)
    i += 1
ftrain.close()
X = sequence.pad_sequences(X, maxlen=MAX_SENTENCE_LENGTH)

```

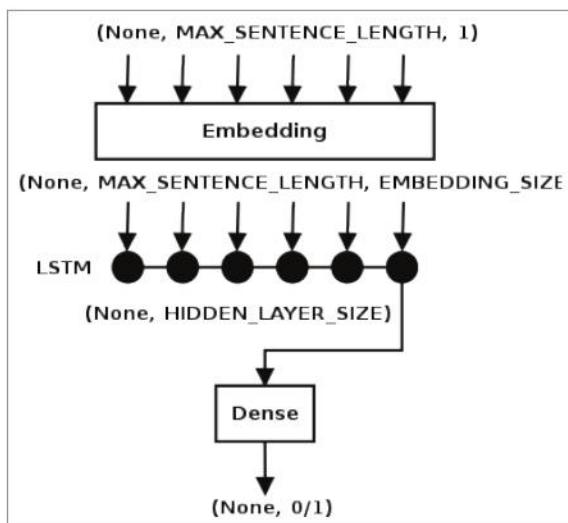
Наконец, разбиваем весь набор данных на обучающий и тестовый в пропорции 80:20:

```

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2,
        random_state=42)

```

На рисунке ниже показана структура нашей PHC



Входными данными является последовательность индексов слов. Длина последовательности равна `MAX_SENTENCE_LENGTH`. Первому измерению тензора присваивается значение `None`, показывающее, что размер пакета (число записей, загружаемых в сеть за один раз)

в момент определения сети неизвестен; он будет задан на этапе выполнения с помощью параметра `batch_size`. Таким образом, в предположении, что размер пакета пока неизвестен, входной тензор имеет форму `(None, MAX_SENTENCE_LENGTH, 1)`. Такие тензоры подаются на вход слоя погружения размера `EMBEDDING_SIZE`, веса которого инициализированы небольшими случайными значениями и подлежат обучению. Этот слой преобразует входной тензор к форме `(None, MAX_SENTENCE_LENGTH, EMBEDDING_SIZE)`. Выход слоя погружения загружается в LSTM с длиной последовательности `MAX_SENTENCE_LENGTH` и размером выходного слоя `HIDDEN_LAYER_SIZE`. На выходе LSTM получается тензор формы `(None, HIDDEN_LAYER_SIZE, MAX_SENTENCE_LENGTH)`. По умолчанию LSTM выводит единственный тензор формы `(None, HIDDEN_LAYER_SIZE)` в качестве результирующей последовательности (`return_sequences=False`). Он подается на вход плотного слоя с размером выхода 1 и сигмоидной функцией активации, который выводит 0 (отрицательная окраска) или 1 (положительная окраска).

При компиляции модели указывается бинарная перекрестная энтропия в качестве функции потерь, поскольку модель предсказывает бинарное значение, и Adam – хороший универсальный оптимизатор. Гиперпараметры `EMBEDDING_SIZE`, `HIDDEN_LAYER_SIZE`, `BATCH_SIZE` и `NUM_EPOCHS` (заданные ниже в виде констант) выбраны по результатам нескольких экспериментов:

```
EMBEDDING_SIZE = 128
HIDDEN_LAYER_SIZE = 64
BATCH_SIZE = 32
NUM_EPOCHS = 10

model = Sequential()
model.add(Embedding(vocab_size, EMBEDDING_SIZE,
                    input_length=MAX_SENTENCE_LENGTH))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(LSTM(HIDDEN_LAYER_SIZE, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1))
model.add(Activation("sigmoid"))

model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
```

Затем мы обучаем сеть на протяжении 10 (`NUM_EPOCHS`) периодов с размером пакета (`BATCH_SIZE`) 32. После каждого периода модель проверяется на тестовых данных:

```
history = model.fit(Xtrain, ytrain, batch_size=BATCH_SIZE,
                    epochs=NUM_EPOCHS, validation_data=(Xtest, ytest))
```

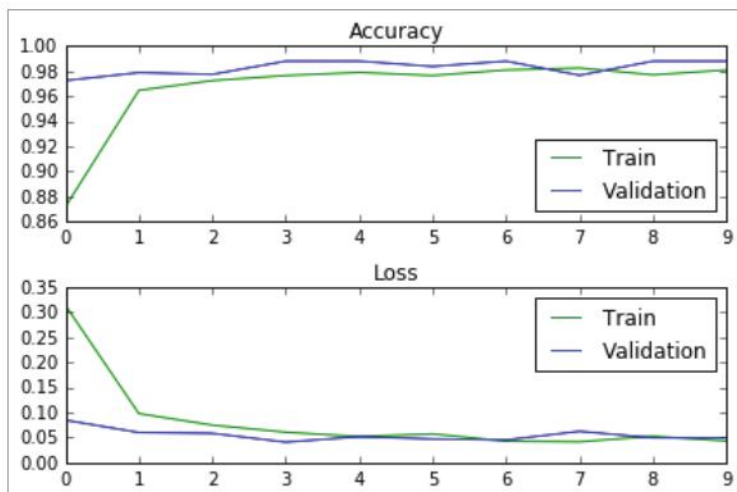
На результатах выполнения программы видно, как убывает потеря и возрастает верность:

```
Train on 5668 samples, validate on 1418 samples
Epoch 1/10
5668/5668 [=====] - 20s - loss: 0.3316 - acc: 0.8626 - val_loss: 0.0799 - val_acc: 0.9746
Epoch 2/10
5668/5668 [=====] - 19s - loss: 0.0911 - acc: 0.9626 - val_loss: 0.0512 - val_acc: 0.9810
Epoch 3/10
5668/5668 [=====] - 18s - loss: 0.0649 - acc: 0.9730 - val_loss: 0.0553 - val_acc: 0.9859
Epoch 4/10
5668/5668 [=====] - 19s - loss: 0.0642 - acc: 0.9746 - val_loss: 0.0596 - val_acc: 0.9845
Epoch 5/10
5668/5668 [=====] - 20s - loss: 0.0531 - acc: 0.9787 - val_loss: 0.0434 - val_acc: 0.9845
Epoch 6/10
5668/5668 [=====] - 19s - loss: 0.0575 - acc: 0.9762 - val_loss: 0.0396 - val_acc: 0.9852
Epoch 7/10
5668/5668 [=====] - 19s - loss: 0.0494 - acc: 0.9797 - val_loss: 0.0374 - val_acc: 0.9873
Epoch 8/10
5668/5668 [=====] - 19s - loss: 0.0467 - acc: 0.9809 - val_loss: 0.0374 - val_acc: 0.9859
Epoch 9/10
5668/5668 [=====] - 18s - loss: 0.0440 - acc: 0.9811 - val_loss: 0.0425 - val_acc: 0.9852
Epoch 10/10
5668/5668 [=====] - 18s - loss: 0.0464 - acc: 0.9795 - val_loss: 0.0378 - val_acc: 0.9873
1418/1418 [=====] - 0s
```

В следующем фрагменте мы строим графики зависимости потери и верности от времени:

```
plt.subplot(211)
plt.title("Accuracy")
plt.plot(history.history["acc"], color="g", label="Train")
plt.plot(history.history["val_acc"], color="b", label="Validation")
plt.legend(loc="best")
plt.subplot(212)
plt.title("Loss")
plt.plot(history.history["loss"], color="g", label="Train")
plt.plot(history.history["val_loss"], color="b", label="Validation")
plt.legend(loc="best")
plt.tight_layout()
plt.show()
```

Вот что получается:



Наконец, модель оценивается на полном тестовом наборе и печатается оценка и верность. Мы также выбираем несколько случайных предложений из тестового набора и печатаем предсказание РНС, метку и само предложение:

```
score, acc = model.evaluate(Xtest, ytest, batch_size=BATCH_SIZE)
print("Test score: %.3f, accuracy: %.3f" % (score, acc))

for i in range(5):
    idx = np.random.randint(len(Xtest))
    xtest = Xtest[idx].reshape(1,40)
    ylabel = ytest[idx]
    ypred = model.predict(xtest)[0][0]
    sent = " ".join([index2word[x] for x in xtest[0].tolist() if x != 0])
    print("%.0ft%dt%s" % (ypred, ylabel, sent))
```

Как видим, верность близка к 99 %. На этом конкретном наборе предсказания модели в точности совпадают с метками, но это верно не для всех предсказаний.

Test score: 0.038, accuracy: 0.987

#pred label sentence

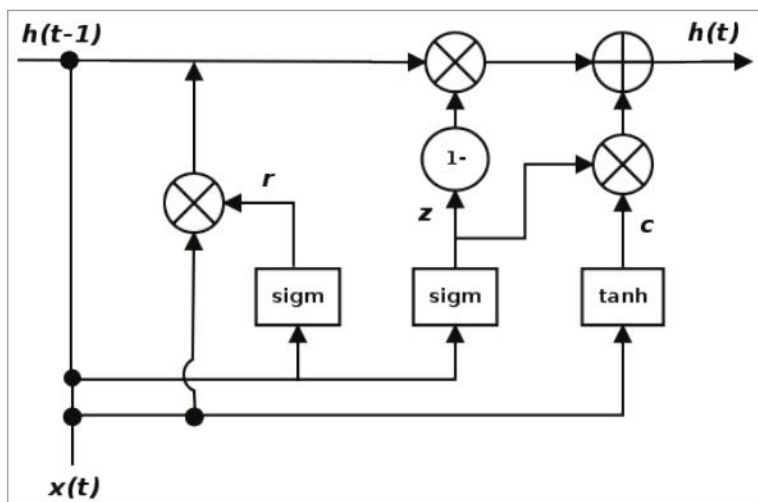
```
1 1 i like th mission impossible one ...
1 1 we're gon na like watch mission impossible or hoot . (
1 1 the people who are worth it know how much i love the da vinci code .
0 0 ok brokeback mountain is such a horrible movie .
1 1 brokeback mountain is the most amazing / beautiful / romantic /
Heartbraking movie i have ever or will ever see in my life
```

Если хотите выполнить эту программу на своей машине, то нужно будет скачать данные с сайта Kaggle.

Код этого примера находится в файле `umich_sentiment_lstm.py` в составе исходного кода к этой главе.

Вентильный рекуррентный блок – GRU

Вентильный рекуррентный блок (gated recurrent unit, GRU) – это вариант LSTM, впервые предложенный К. Чо (см. «Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation», arXiv:1406.1078, 2014). Он также обладает устойчивостью к проблеме исчезающего градиента, но его внутренняя структура проще, а потому и обучается он быстрее, т. е. для обновления скрытого состояния нужно меньше вычислений. На следующем рисунке показаны вентили в ячейке GRU:



Вместо трех вентилей в ячейке LSTM – входного, забывания и выходного, в ячейке GRU всего два вентиль: обновления z и сброса r . Вентиль обновления определяет, какую часть предыдущего запомненного значения сохранять, а вентиль сброса – как смешивать новый вход с предыдущей памятью. Не существует никакого постоянного состояния ячейки, отличного от скрытого состояния, как в LSTM. Механизм работы GRU описывается следующими формулами:

$$\begin{aligned}
 z &= \sigma(W_z h_{t-1} + U_z x_t) \\
 r &= \sigma(W_r h_{t-1} + U_r x_t) \\
 c &= \tanh(W_c (h_{t-1} \otimes r) + U_c x_t) \\
 h_t &= (z \otimes c) \oplus ((1 - z) \otimes h_{t-1})
 \end{aligned}$$

Согласно эмпирическим оценкам (см. статьи R. Jozefowicz, W. Zaremba, and I. Sutskever «An Empirical Exploration of Recurrent Network Architectures», JMLR, 2015 и J. Chung «Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling», arXiv:1412.3555. 2014), качество GRU и LSTM сравнимо, и дать априорную рекомендацию, какую модель выбрать для конкретной задачи, невозможно. GRU быстрее обучаются и требуют меньше данных для достижения обобщаемости, но в ситуациях, когда обучающих данных достаточно, большая выразительная способность LSTM может приводить к лучшим результатам. Как и LSTM, GRU можно подставить вместо ячейки типа SimpleRNN.

Keras предоставляет встроенные реализации LSTM и GRU наряду с рассмотренным выше классом SimpleRNN.

Пример GRU – частеречная разметка

В Keras есть реализация GRU, которой мы воспользуемся для построения сети частеречной разметки, которая распознает грамматические категории слов: существительные, глаголы, прилагательные и т. д. Раньше частеречную разметку приходилось выполнять вручную, но теперь это делается автоматически с помощью статистических моделей. В последние годы к этой задаче было также применено глубокое обучение (см. R. Collobert «Natural Language Processing (almost) from Scratch», Journal of Machine Learning Research, pp. 2493–2537, 2011).

В качестве обучающих данных нам нужны предложения, в которых проставлены метки частей речи. Один из таких наборов, Penn Treebank (<https://catalog.ldc.upenn.edu/ldc99t42>), содержит размеченный людьми корпус текстов, содержащий примерно 4.5 миллиона слова на американском диалекте английского языка. Но этот ресурс платный. 10%-ая выборка набора Penn Treebank свободно доступна в составе пакета NLTK (<http://www.nltk.org/>), и мы воспользуемся ей для обучения нашей сети.

Наша модель принимает последовательность слов предложения и выводит метки частей речи для каждого слова. Так, для входной

последовательности, состоящей из слов [*The, cat, sat, on, the, mat, .*], выходная последовательность будет содержать метки [*DT, NN, VB, IN, DT, NN*].

Начинаем с импорта:

```
from keras.layers.core import Activation, Dense, Dropout, RepeatVector, SpatialDropout1D
from keras.layers.embeddings import Embedding
from keras.layers.recurrent import GRU
from keras.layers.wrappers import TimeDistributed
from keras.models import Sequential
from keras.preprocessing import sequence
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
import collections
import nltk
import numpy as np
import os
```

Далее скачиваем корпус NLTK Treebank в формате, удобном для последующей обработки – в уже разобранном виде. Показанный ниже код загружает данные в два параллельных файла: в одном слова, составляющие предложения, а в другом – метки частей речи.

```
DATA_DIR = "../data"

fedata = open(os.path.join(DATA_DIR, "treebank_sents.txt"), "wb")
ffdata = open(os.path.join(DATA_DIR, "treebank_pos.txt"), "wb")

sents = nltk.corpus.treebank.tagged_sents()
for sent in sents:
    words, poss = [], []
    for word, pos in sent:
        if pos == "-NONE-":
            continue
        words.append(word)
        poss.append(pos)
    fedata.write(":{s}\n".format(" ".join(words)))
    ffdata.write(":{s}\n".format(" ".join(poss)))

fedata.close()
ffdata.close()
```

И на этот раз мы выполним предварительную обработку, чтобы определить размер словаря. Но теперь у нас будет два словаря: исходных слов и целевых меток. Требуется вычислить количество уникальных элементов в каждом словаре. Кроме того, нам нужно знать максимальное число слов в предложении и количество записей. В силу взаимной однозначности частеречной разметки последние два значения одинаковы для обоих словарей.

```

def parse_sentences(filename):
    word_freqs = collections.Counter()
    num_recs, maxlen = 0, 0
    fin = open(filename, "rb")
    for line in fin:
        words = line.strip().lower().split()
        for word in words:
            word_freqs[word] += 1
        if len(words) > maxlen:
            maxlen = len(words)
        num_recs += 1
    fin.close()
    return word_freqs, maxlen, num_recs

s_wordfreqs, s_maxlen, s_numrecs = parse_sentences(
    os.path.join(DATA_DIR, "treebank_sents.txt"))
t_wordfreqs, t_maxlen, t_numrecs = parse_sentences(
    os.path.join(DATA_DIR, "treebank_poss.txt"))
print(len(s_wordfreqs), s_maxlen, s_numrecs, len(t_wordfreqs), t_maxlen, t_numrecs)

```

Выясняется, что корпус содержит 10 947 уникальных слов и 45 уникальных меток. Размер самого длинного предложения равен 249, а число предложений – 3914. Зная все это, мы принимаем решение включать в исходный словарь только первые 5000 слов. А в целевом словаре будет 45 уникальных меток частей речи, поскольку мы хотим предсказывать все метки. И в качестве максимальной длины последовательности зададим 250.

```

MAX_SEQLEN = 250
S_MAX_FEATURES = 5000
T_MAX_FEATURES = 45

```

Как и в примере анализа эмоциональной окраски, входные данные будут представлены последовательностью индексов слов. А на выходе будет последовательность индексов меток. Поэтому нужно построить таблицы соответствия между словами (метками) и их индексами. Показанный ниже код именно это и делает. При рассмотрении словаря слов мы включаем в индекс два дополнительных элемента для фиктивных слов PAD и UNK. А в словаре меток фиктивное слово UNK не нужно, поскольку никакие метки не отбрасываются.

```

s_vocabsize = min(len(s_wordfreqs), S_MAX_FEATURES) + 2
s_word2index = {x[0]:i+2 for i, x in
    enumerate(s_wordfreqs.most_common(S_MAX_FEATURES))}
s_word2index["PAD"] = 0
s_word2index["UNK"] = 1

```

```
s_index2word = {v:k for k, v in s_word2index.items()}

t_vocabsize = len(t_wordfreqs) + 1
t_word2index = {x[0]:i for i, x in
enumerate(t_wordfreqs.most_common(T_MAX_FEATURES))}
t_word2index["PAD"] = 0
t_index2word = {v:k for k, v in t_word2index.items()}
```

Следующий шаг – построение наборов данных для подачи на вход сети. Мы воспользуемся этими таблицами соответствия для преобразования входных предложений в последовательность идентификаторов слов длиной `MAX_SEQLEN` (250). Метки должны быть представлены в виде последовательности унитарных векторов размера `T_MAX_FEATURES + 1` (46), также длиной `MAX_SEQLEN` (250). Функция `build_tensor` читает данные из обоих файлов и преобразует их во входной и выходной тензор. Для построения выходного тензора передаются дополнительные параметры, подразумеваемые по умолчанию. Наша функция обращается к функции `np_utils.to_categorical()`, которая преобразует выходную последовательность идентификаторов меток частей речи в унитарное представление:

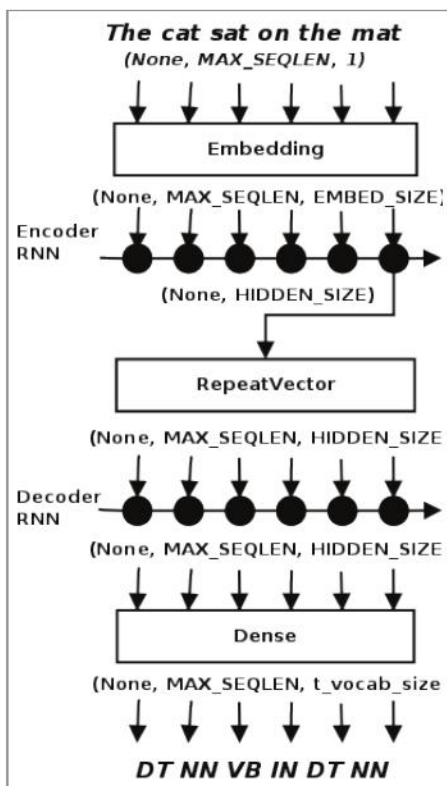
```
def build_tensor(filename, numrecs, word2index, maxlen,
    make_categorical=False, num_classes=0):
    data = np.empty((numrecs, ), dtype=list)
    fin = open(filename, "rb")
    i = 0
    for line in fin:
        wids = []
        for word in line.strip().lower().split():
            if word2index.has_key(word):
                wids.append(word2index[word])
            else:
                wids.append(word2index["UNK"])
        if make_categorical:
            data[i] = np_utils.to_categorical(wids, num_classes=num_classes)
        else:
            data[i] = wids
        i += 1
    fin.close()
    pdata = sequence.pad_sequences(data, maxlen=maxlen)
    return pdata

X = build_tensor(os.path.join(DATA_DIR, "treebank_sents.txt"),
    s_numrecs, s_word2index, MAX_SEQLEN)
Y = build_tensor(os.path.join(DATA_DIR, "treebank_poss.txt"),
    t_numrecs, t_word2index, MAX_SEQLEN, True, t_vocabsize)
```


Затем разбиваем набор данных на обучающий и тестовый в пропорции 80:20:

```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.2,
                                              random_state=42)
```

На рисунке ниже показана схема нашей сети. Выглядит сложно-вато, поэтому разберем ее по шагам.



Как и раньше, в предположении, что размер пакета еще не определен, входом в сеть является тензор идентификаторов слов формы $(None, MAX_SEQLEN, 1)$. Он проходит через слой погружения, который преобразует каждое слово в плотный вектор размера $EMBED_SIZE$, так что тензор на выходе этого слоя имеет форму $(None, MAX_SEQLEN, EMBED_SIZE)$. Этот тензор подается на вход кодирующего GRU-слоя с размером выхода $HIDDEN_SIZE$. GRU настроен на возврат единственного контекстного вектора (`return_sequences=False`) после

обработки последовательности длиной `MAX_SEQLEN`, поэтому тензор на выходе GRU-слоя имеет форму `(None, HIDDEN_SIZE)`.

Этот контекстный вектор далее реплицируется слоем `RepeatVector` в тензор формы `(None, MAX_SEQLEN, HIDDEN_SIZE)` и подается на вход декодирующего GRU-слоя. Результат поступает плотному слою, который порождает выходной тензор формы `(None, MAX_SEQLEN, t_vocab_size)`. В качестве функции активации в плотном слое используется `softmax`. Значением `argmax` для каждого столбца этого тензора является индекс предсказанной метки части речи для слова в соответствующей позиции.

Ниже приведено определение этой модели; `EMBED_SIZE`, `HIDDEN_SIZE`, `BATCH_SIZE` и `NUM_EPOCHS` – гиперпараметры, значения которых выбраны по результатам экспериментов. При компиляции модели была указана функция потерь `categorical_crossentropy`, поскольку метки принадлежат нескольким категориям, и популярный оптимизатор `Adam`:

```
EMBED_SIZE = 128
HIDDEN_SIZE = 64
BATCH_SIZE = 32
NUM_EPOCHS = 1

model = Sequential()
model.add(Embedding(s_vocabsize, EMBED_SIZE, input_length=MAX_SEQLEN))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(GRU(HIDDEN_SIZE, dropout=0.2, recurrent_dropout=0.2))
model.add(RepeatVector(MAX_SEQLEN))
model.add(GRU(HIDDEN_SIZE, return_sequences=True))
model.add(TimeDistributed(Dense(t_vocabsize)))
model.add(Activation("softmax"))

model.compile(loss="categorical_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

Модель обучается на протяжении одного периода. Поскольку количество параметров модели велико, то после первого периода начинает проявляться тенденция к переобучению – если одни и те же данные подавать в последующих периодах, то модель подгоняется к ним и хуже ведет себя на контрольных данных.

```
model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE, epochs=NUM_EPOCHS,
          validation_data=[Xtest, Ytest])

score, acc = model.evaluate(Xtest, Ytest, batch_size=BATCH_SIZE)
print("Test score: %.3f, accuracy: %.3f" % (score, acc))
```

Ниже показан результат обучения и оценивания модели. Как видим, уже после первого периода модель ведет себя весьма плохо.

```
Train on 3131 samples, validate on 783 samples
Epoch 1/1
3131/3131 [=====] - 81s - loss: 0.3013 - acc: 0.8263 - val_loss: 0.2934 - val_acc: 0.9159

783/783 [=====] - 3s
Test score: 0.293, accuracy: 0.916
```

Как и положено, три класса рекуррентных сетей в Keras (`SimpleRNN`, `LSTM` и `GRU`) взаимозаменяемы. Чтобы убедиться в этом, заменим все три вхождения слова `GRU` в предыдущей программе на `LSTM` и снова запустим ее. Изменить придется только директиву импорта и определение модели:

```
from keras.layers.recurrent import LSTM

model = Sequential()
model.add(Embedding(s_vocabsize, EMBED_SIZE, input_length=MAX_SEQLen))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(LSTM(HIDDEN_SIZE, dropout=0.2, recurrent_dropout=0.2))
model.add(RepeatVector(MAX_SEQLen))
model.add(LSTM(HIDDEN_SIZE, return_sequences=True))
model.add(TimeDistributed(Dense(t_vocabsize)))
model.add(Activation("softmax"))
```

Результаты сетей на основе `GRU` и `LSTM` сопоставимы.

Класс моделей вида последовательность-в-последовательность обладает мощными выразительными возможностями. Его каноническое приложение – машинный перевод, но есть и много других, в т. ч. рассмотренное выше. В качестве примеров можно назвать различные задачи обработки естественного языка: распознавание именованных сущностей (см. J. Hammerton «Named Entity Recognition with Long Short Term Memory», Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL, Association for Computational Linguistics, 2003), грамматический разбор предложений (см. O. Vinyals «Grammar as a Foreign Language», Advances in Neural Information Processing Systems, 2015), а также более сложные сети, например, для подписывания изображения (см. A. Karpathy, F. Li «Deep Visual-Semantic Alignments for Generating Image Descriptions», Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015).

Полный код примера находится в файле `pos_tagging_gru.py` в составе исходного кода к этой главе.

Двунаправленные РНС

Выход РНС в момент времени t зависит от выходов на всех предшествующих временных шагах. Но вполне может случиться, что выход зависит также от будущих выходов. Это справедливо, в частности, для приложений ОЕЯ, когда атрибуты слова или фразы, которые мы пытаемся предсказать, могут зависеть от контекста, определяемого всем объемлющим предложением, а не только предшествующими словами. Двунаправленные РНС помогают построить архитектуру сети, которая придает одинаковую важность началу и концу предложения, и позволяют увеличить объем данных, доступных для обучения.

Двунаправленная РНС – это две РНС, собранные вместе, которые читают входные данные в разных направлениях. В нашем примере одна РНС будет читать слова от начала предложения к концу, а другая – от конца к началу. Выход на каждом временном шаге будет зависеть от скрытого состояния обеих РНС.

Keras поддерживает двунаправленные РНС с помощью оберты-вающего слоя *Bidirectional*. Так, в случае частеречной разметки мы могли бы сделать LSTM-сети двунаправленными, обернув их слоем *Bidirectional*, как показано ниже:

```
from keras.layers.wrappers import Bidirectional

model = Sequential()
model.add(Embedding(s_vocabsize, EMBED_SIZE, input_length=MAX_SEQLEN))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(Bidirectional(LSTM(HIDDEN_SIZE, dropout=0.2, recurrent_dropout=0.2)))
model.add(RepeatVector(MAX_SEQLEN))
model.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True)))
model.add(TimeDistributed(Dense(t_vocabsize)))
model.add(Activation("softmax"))
```

Получаемое качество сравнимо с качеством в случае однонаправленной LSTM-сети:

```
Train on 3131 samples, validate on 783 samples
Epoch 1/1
3131/3131 [=====] - 268s - loss: 0.2889 - acc: 0.8226 - val_loss: 0.2788 - val_acc: 0.9036

783/783 [=====] - 12s
Test score: 0.279, accuracy: 0.904
```


РНС с запоминанием состояния

РНС может сохранять состояние при переходе от одного пакета к другому во время обучения. Иначе говоря, скрытое состояние, вычисленное для одного пакета обучающих данных, используется в качестве начального скрытого состояния для следующего пакета. Но этот режим необходимо задавать явно, потому что в Keras РНС по умолчанию не запоминает состояние и сбрасывает его после обработки каждого пакета. Запоминание состояния позволяет РНС строить свое внутреннее состояние на протяжении обработки всей последовательности обучающих данных и даже использовать его на этапе предсказания.

К достоинствам РНС с запоминанием состояния следует отнести меньший размер сети и (или) сокращение времени обучения, а к недостаткам – то, что мы теперь несем ответственность за выбор такого размера пакета, который отражает периодичность данных, и за сброс состояния после каждого периода. Кроме того, данные не следует перетасовывать в процессе обучения, потому что для сетей с запоминанием состояния порядок предъявления данных существенен.

Пример LSTM с запоминанием состояния – предсказание потребления электричества

В этом примере мы предскажем потребление электричества с помощью LSTM-сети с запоминанием и без запоминания состояния и сравним результаты. Напомним, что в Keras РНС по умолчанию не запоминает состояние. В моделях с запоминанием внутреннее состояние, вычисленное для элемента i в предыдущем пакете, будет использоваться в качестве начального состояния элемента i в следующем пакете.

Для обучения мы будем использовать набор данных из репозитория машинного обучения UCI (<https://archive.ics.uci.edu/ml/datasets/ElectricityLoadDiagrams20112014>), который содержит информацию о потреблении электричества 370 потребителями с интервалом 15 минут за период с 2011 по 2014 год. Для примера мы случайно выбрали потребителя с номером 250.

Большинство задач можно решить с помощью РНС без запоминания состояния, поэтому, прибегая к РНС с запоминанием состояния, нужно понимать, зачем вы это делаете. Как правило,

такая необходимость возникает, когда в данных имеется периодическая составляющая. И для потребления электричества действительно характерна периодичность – потребление выше днем и ниже ночью. Выделим данные для потребителя 250, нарисуем график за первые десять дней и сохраним данные в двоичном формате NumPy для следующего шага.

```
import numpy as np
import matplotlib.pyplot as plt
import os
import re

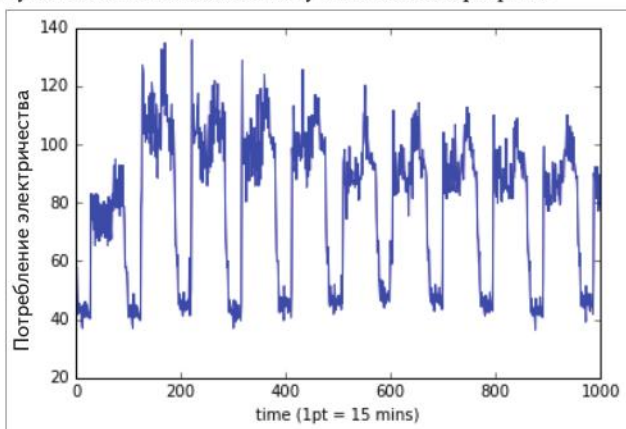
DATA_DIR = "../data"

fld = open(os.path.join(DATA_DIR, "LD2011_2014.txt"), "rb")
data = []
cid = 250
for line in fld:
    if line.startswith(";;;"):
        continue
    cols = [float(re.sub(",", ".", x)) for x in line.strip().split(";")[1:]]
    data.append(cols[cid])
fld.close()

NUM_ENTRIES = 1000
plt.plot(range(NUM_ENTRIES), data[0:NUM_ENTRIES])
plt.ylabel("electricity consumption")
plt.xlabel("time (1pt = 15 mins)")
plt.show()

np.save(os.path.join(DATA_DIR, "LD_250.npy"), np.array(data))
```

На рисунке ниже показан получившийся график:



Как видите, имеется отчетливая суточная периодичность, так что для этой задачи модель с запоминанием состояния отлично подойдет. Кроме того, из результатов наблюдения следует, что имеет смысл взять `BATCH_SIZE` равным 96 (количество 15-минутных отсчетов за 24 часа).

Мы будем показывать одновременно код обеих версий: с запоминанием и без запоминания состояния, поскольку почти весь код совпадает. На отличия будем специально обращать внимание.

Как обычно, сначала импортируем необходимые библиотеки и классы:

```
from keras.layers.core import Dense
from keras.layers.recurrent import LSTM
from keras.models import Sequential
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import math
import os
```

Затем загрузим данные для потребителя 250 в большой массив (размера 140256) из сохраненного ранее двоичного файла NumPy и приведем его к диапазону (0, 1). Наконец, изменим форму входных данных на трехмерную, как того требует наша сеть:

```
DATA_DIR = "../data"
data = np.load(os.path.join(DATA_DIR, "LD_250.npy"))
data = data.reshape(-1, 1)
scaler = MinMaxScaler(feature_range=(0, 1), copy=False)
data = scaler.fit_transform(data)
```

При обработке каждого пакета модель принимает последовательность 15-минутных отсчетов и предсказывает следующий. Длина входной последовательности определяется переменной `NUM_TIMESTEPS`. По результатам экспериментов мы выбрали значение `NUM_TIMESTEPS=20`, т. е. длина входной последовательности равна 20, а выходной – 1. На следующем шаге входной массив преобразовывается в тензоры `x` и `y` формы `(None, 4)` и `(None, 1)`. И наконец, входной тензор `x` преобразуется в трехмерный в соответствии с требованиями сети:

```
X = np.zeros((data.shape[0], NUM_TIMESTEPS))
Y = np.zeros((data.shape[0], 1))
for i in range(len(data) - NUM_TIMESTEPS - 1):
    X[i] = data[i:i + NUM_TIMESTEPS].T
    Y[i] = data[i + NUM_TIMESTEPS + 1]
```

```
# изменить форму X, приведя его к трем измерениям (отсчеты, временные шаги, признаки)
X = np.expand_dims(X, axis=2)
```

Далее мы разбиваем тензоры x и y на обучающий и тестовый набор в пропорции 70:30. Поскольку мы работаем с временными рядами, то просто выбираем точку разделения и разрезаем данные на две части, не пользуясь функцией `train_test_split`, которая дополнительно перетасовывает данные:

```
sp = int(0.7 * len(data))
Xtrain, Xtest, Ytrain, Ytest = X[0:sp], X[sp:], Y[0:sp], Y[sp:]
print(Xtrain.shape, Xtest.shape, Ytrain.shape, Ytest.shape)
```

Сначала определяется модель без сохранения соединения. Кроме того, задаются значения переменных `BATCH_SIZE` и `NUM_TIMESTEPS`. Размер выхода LSTM-сети определяется переменной `HIDDEN_SIZE`, это еще один гиперпараметр, который обычно выставляется по результатам экспериментов. Мы задали значение 10, потому что наша цель – просто сравнить две сети:

```
NUM_TIMESTEPS = 20
HIDDEN_SIZE = 10
BATCH_SIZE = 96 # 24 часа (количество 15-минутных интервалов)
```

```
# без сохранения состояния
model = Sequential()
model.add(LSTM(HIDDEN_SIZE, input_shape=(NUM_TIMESTEPS, 1), return_sequences=False))
model.add(Dense(1))
```

Определение соответствующей модели с сохранением состояния очень похоже (см. ниже). В конструкторе LSTM нужно задать параметр `stateful=True`, а вместо параметра `input_shape`, в котором подразумевается, что размер пакета определяется на этапе выполнения, мы задаем параметр `batch_input_shape`, где этот размер указывается явно. Кроме того, размеры обучающего и тестового набора данных должны быть кратны размеру пакета. Как этого добиться, мы увидим ниже при рассмотрении кода обучения.

```
# с сохранением состояния
model = Sequential()
model.add(LSTM(HIDDEN_SIZE, stateful=True,
batch_input_shape=(BATCH_SIZE, NUM_TIMESTEPS, 1), return_sequences=False))
model.add(Dense(1))
```

Код компиляции модели одинаков для обеих РНС. Отметим, что в роли показателя качества выступает не верность, как обычно, а среднеквадратическая ошибка, поскольку мы имеем задачу регрессии: нас интересует не совпадение предсказания с меткой, а расхождение между предсказаниями и метками. Полный перечень встроенных в Keras показателей качества имеется в документации.


```
model.compile(loss="mean_squared_error", optimizer="adam",
              metrics=["mean_squared_error"])
```

Для обучения модели без состояния достаточно одной строки, которая вам, наверное, уже стала привычной:

```
BATCH_SIZE = 96 # 24 часа (количество 15-минутных интервалов)

# без сохранения состояния
model.fit(Xtrain, Ytrain, epochs=NUM_EPOCHS, batch_size=BATCH_SIZE,
        validation_data=(Xtest, Ytest), shuffle=False)
```

Соответствующий код для модели с сохранением состояния показан ниже. В нем есть три важных момента.

Во-первых, размер пакета должен отражать периодичность данных, поскольку РНС с сохранением состояния сопоставляет состояния соответственных элементов соседних пакетов, а это значит, что при правильном размере пакета сеть будет обучаться быстрее. Размеры обучающего и тестового набора должны быть кратны размеру пакета. Мы обеспечили выполнение этого условия, отбросив последние несколько записей в обоих наборах.

Во-вторых, мы в цикле обучаем модель на протяжении заданного числа периодов, при этом состояние сохраняется при переходе от пакета к пакету. Но после каждого периода состояние модели необходимо сбросить вручную.

В-третьих, данные необходимо подавать строго последовательно. По умолчанию Keras перетасовывает данные в каждом пакете, а это нарушает соответствие элементов, необходимое для эффективного обучения РНС с сохранением состояния. Чтобы отменить тасование, задается параметр `shuffle=False` при обращении к `model.fit()`:

```
BATCH_SIZE = 96 # 24 часа (количество 15-минутных интервалов)

# с сохранением состояния
# размеры обучающего и тестового набора должны быть кратны BATCH_SIZE
train_size = (Xtrain.shape[0] // BATCH_SIZE) * BATCH_SIZE
test_size = (Xtest.shape[0] // BATCH_SIZE) * BATCH_SIZE
Xtrain, Ytrain = Xtrain[0:train_size], Ytrain[0:train_size]
Xtest, Ytest = Xtest[0:test_size], Ytest[0:test_size]
print(Xtrain.shape, Xtest.shape, Ytrain.shape, Ytest.shape)
for i in range(NUM_EPOCHS):
    print("Epoch {:d}/{:d}".format(i+1, NUM_EPOCHS))
    model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE, epochs=1,
            validation_data=(Xtest, Ytest), shuffle=False)
    model.reset_states()
```

Наконец, мы оцениваем модель на тестовых данных и печатаем результаты:

```
score, _ = model.evaluate(Xtest, Ytest, batch_size=BATCH_SIZE)
rmse = math.sqrt(score)
print("MSE: {:.3f}, RMSE: {:.3f}".format(score, rmse))
```

После обучения модели без сохранения состояния на протяжении пяти периодов получают такие результаты:

```
(98179, 20, 1) (42077, 20, 1) (98179, 1) (42077, 1)
Train on 98179 samples, validate on 42077 samples
Epoch 1/5
98179/98179 [=====] - 41s - loss: 0.0086 - mean_squared_error: 0.0086 - val_loss: 0.0040 -
val_mean_squared_error: 0.0040
Epoch 2/5
98179/98179 [=====] - 41s - loss: 0.0045 - mean_squared_error: 0.0045 - val_loss: 0.0039 -
val_mean_squared_error: 0.0039
Epoch 3/5
98179/98179 [=====] - 43s - loss: 0.0041 - mean_squared_error: 0.0041 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
Epoch 4/5
98179/98179 [=====] - 44s - loss: 0.0039 - mean_squared_error: 0.0039 - val_loss: 0.0040 -
val_mean_squared_error: 0.0040
Epoch 5/5
98179/98179 [=====] - 44s - loss: 0.0038 - mean_squared_error: 0.0038 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038

42077/42077 [=====] - 2s

MSE: 0.004, RMSE: 0.062
```

Для модели с сохранением состояния, которая пять раз обучалась в цикле с одним периодом, получились такие результаты (обратите внимание, что число примеров в обучающем и тестовом наборе уменьшено, чтобы сделать его кратным размеру пакета):

```
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 37s - loss: 0.0056 - mean_squared_error: 0.0056 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
Epoch 2/5
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 36s - loss: 0.0044 - mean_squared_error: 0.0044 - val_loss: 0.0037 -
val_mean_squared_error: 0.0037
Epoch 3/5
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 38s - loss: 0.0043 - mean_squared_error: 0.0043 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
Epoch 4/5
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 37s - loss: 0.0042 - mean_squared_error: 0.0042 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
Epoch 5/5
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 37s - loss: 0.0040 - mean_squared_error: 0.0040 - val_loss: 0.0035 -
val_mean_squared_error: 0.0035
41952/42048 [=====>] - ETA: 0s

MSE: 0.003, RMSE: 0.059
```

Как видим, результаты модели с сохранением состояния чуть лучше. Учитывая, что мы привели данные к диапазону (0,1), в абсолютных цифрах модель без сохранения состояния дает частоту ошибок 6.2%, а с сохранением – 5.9%. То же самое можно выразить, сказав, что верность составляет соответственно 93.8% и 94.1%.

Код этого примера находится в двух файлах: `econs_data.py` содержит код разбора исходного набора данных, а `econs_stateful.py` – код определения и обучения обеих моделей. Оба файла входят в состав исходного кода к этой главе.

Другие варианты РНС

В заключение этой главы кратко рассмотрим другие варианты ячейки РНС. В этой области ведутся активные исследования и были предложены многочисленные варианты для решения конкретных задач.

Один из популярных вариантов LSTM характеризуется добавлением «смотровых глазков» (peerhole connections), благодаря которым вентильные слои могут видеть состояние ячейки. Этот вариант предложен в 2002 году Герсом и Шмидхубером (см. F. A. Gers, N. N. Schraudolph, J. Schmidhuber «Learning Precise Timing with LSTM Recurrent Networks», *Journal of Machine Learning Research*, pp. 115–43).

Еще один вариант LSTM, который в конечном итоге привел к GRU, – спарить вентиль забывания с выходным вентилем. Решения о том, что забыть, а что оставить, принимаются совместно обоими вентилями, а новая информация замещает забытую.

Keras предлагает только три основных варианта в виде слоев SimpleRNN, LSTM и GRU. Но это не обязательно плохо. Греф экспериментально исследовал много вариантов LSTM (см. K. Greff «LSTM: A Search Space Odyssey», arXiv:1503.04069, 2015) и пришел к выводу, что ни один из них не дает значительного выигрыша по сравнению со стандартной архитектурой LSTM. Таким образом, компонентов, имеющихся в Keras, обычно достаточно для решения большинства задач.

В случае, если вам действительно необходимо сконструировать собственный слой, можно прибегнуть к предоставляемому Keras механизму пользовательских слоев. В следующей главе мы посмотрим, как это делается. Существует также каркас Recurrent Shop с открытым исходным кодом (<https://github.com/datalogai/>

`recurrentshop`), позволяющий создавать сложные рекуррентные нейронные сети с помощью Keras.

Резюме

В этой главе мы рассмотрели базовую архитектуру рекуррентных нейронных сетей и объяснили, почему они превосходят традиционные нейронные сети в применении к последовательным данным. Мы также видели, как можно использовать РНС для обучения авторскому стилю письма и порождения новых текстов с помощью обученной модели. Мы показали, как этот пример можно распространить на предсказание цен на акции или других временных рядов, на выделение речи из зашумленного звукового сигнала и т. д.

Мы обсудили различные способы соединения блоков РНС и применение этих топологий к решению таких задач, как анализ эмоциональной окраски, машинный перевод, подписывание изображений, классификация и т. д.

Затем мы остановились на главном недостатке архитектуры простой РНС – проблеме исчезающих и взрывных градиентов. Мы видели, что проблему исчезающего градиента можно решить с помощью архитектуры LSTM (и GRU). Мы подробно рассмотрели обе эти архитектуры и привели два примера: предсказание эмоциональной окраски текста с помощью LSTM-модели и предсказание меток частей речи с помощью основанной на GRU архитектуры типа последовательность-в-последовательность.

Затем мы поговорили о РНС с сохранением состояния и ее поддержке в Keras. Мы привели пример использования такой РНС для предсказания потребления электричества.

Наконец, мы кратко упомянули о других вариантах РНС, отсутствующих в Keras, о способах построения соответствующих моделей.

В следующей главе мы будем рассматривать модели, не укладывающиеся ни в один из описанных выше шаблонов. Мы также покажем, как из простых моделей можно создавать более сложные, пользуясь функциональным API Keras, и приведем несколько примеров адаптации Keras под свои потребности.

Глава 7

Дополнительные модели машинного обучения

До сих пор мы обсуждали в основном модели, предназначенные для классификации. Они обучаются на основе признаков и меток объектов с целью предсказывать метки ранее не предъявлявшихся объектов. У таких моделей довольно простая архитектура – все рассмотренные выше модели основаны на линейном конвейере, который строится с помощью последовательного API Keras.

Темой этой главы будут более сложные архитектуры, в которых конвейер необязательно линейный. Мы узнаем, как определить сеть с помощью функционального API. Отметим, впрочем, что функциональный API пригоден и для построения линейной архитектуры.

Простейшее обобщение сетей классификации – регрессионные сети. Вообще, классификация и регрессия – две самых широких категории машинного обучения без учителя. Вместо категории регрессионная сеть предсказывает значение непрерывной величины. С примером такого рода мы встречались при обсуждении РНС с состоянием и без. Многие задачи регрессии можно почти без усилий решить с помощью моделей классификации. В этой главе мы рассмотрим пример подобной сети для предсказания содержания бензола в атмосфере.

Еще один класс моделей предназначен для выявления структуры непомеченных данных. Это так называемое **обучение без учителя**. Отличие от моделей классификации в том, что метки присутствуют лишь неявно. Примеры нам уже встречались: модели CBOW и skip-граммы из семейства word2vec. Другой пример – автокодировщики. Мы рассмотрим автокодировщики подробнее и в качестве примера приведем построение компактного векторного представления предложения.

Затем мы поговорим о том, как составлять из рассмотренных моделей более крупные графы вычислений. Цель такого графа – достичь некоторой цели, для которой одной последовательной модели недостаточно. В частности, граф может иметь несколько входов и выходов и соединяться с различными внешними компонентами. Мы рассмотрим пример сети для ответов на вопросы.

Затем мы немного отвлечемся на рассмотрение базового API Keras и расскажем, как его можно использовать для создания пользовательских компонент, расширяющих функциональность Keras.

После этого мы вернемся к неразмеченным данным. Еще один класс моделей, не требующих меток, – порождающие сети. Они обучаются на наборе существующих объектов и пытаются выявить их истинное распределение вероятности. Найдя распределение, мы сможем делать из него выборку, получая примеры, похожие на обучающие данные. Подобный пример мы видели в предыдущей главе, когда обучали РНС для генерации текстов, похожих на «Алису в Стране чудес». Возвращаться к этому примеру мы не станем, а посмотрим, как применить идею обучения распределению данных для создания интересных визуальных эффектов с помощью сети VGG-16, предобученной на наборе данных ImageNet.

Еще раз перечислим темы, рассматриваемые в этой главе:

- функциональный API Keras;
- регрессионные сети;
- автокодировщики как пример обучения без учителя;
- композиция сложных сетей с помощью функционального API;
- пользовательские расширения Keras;
- порождающие сети.

Функциональный API Keras

В функциональном API Keras каждый слой определяется как функция и предоставляются операторы для объединения этих функций в большой граф вычислений. Функция представляет собой преобразование с одним входом и одним выходом. Так, выражение $y = f(x)$ определяет функцию f с входом x и выходом y . Рассмотрим простую последовательную модель Keras (см. <https://keras.io/getting-started/sequential-model-guide/>):

```

from keras.models import Sequential
from keras.layers.core import dense, Activation

model = Sequential([
    dense(32, input_dim=784),
    Activation("sigmoid"),
    dense(10),
    Activation("softmax"),
])

model.compile(loss="categorical_crossentropy", optimizer="adam")

```

Как видим, последовательная модель представляет сеть в виде линейного конвейера, т. е. списка слоев. Но можно также представить сеть в виде показанной ниже композиции функций. Здесь x – входной тензор формы $(None, 784)$, а y – выходной тензор формы $(None, 10)$, где *None* означает, что размер пакета пока не определен.

$$y = \sigma_K(f(\sigma_2(g(x)))) ,$$

где

$$g(x) = W_g x + b_g$$

$$\sigma_2(x) = \frac{1}{1 + e^{-x}}$$

$$f(x) = W_f x + b_f$$

$$\sigma_K(x) = \frac{e^x}{\sum_{k=1}^K e^{x_k}}$$

Ниже показано, как определить эту сеть с помощью функционального API Keras. Обратите внимание, что переменная `predictions` является композицией тех самых функций, которые присутствуют в уравнении выше:

```

from keras.layers import Input
from keras.layers.core import dense
from keras.models import Model
from keras.layers.core import Activation

inputs = Input(shape=(784,))

x = dense(32)(inputs)
x = Activation("sigmoid")(x)
x = dense(10)(x)

```

```

predictions = Activation("softmax")(x)

model = Model(inputs=inputs, outputs=predictions)

model.compile(loss="categorical_crossentropy", optimizer="adam")

```

Поскольку модель есть композиция слоев, которые являются также функциями, то и модель является функцией. Следовательно, обученную модель можно рассматривать просто как еще один слой и вызывать ее, передав тензор подходящей формы. Таким образом, если мы уже построили модель, которая делает нечто полезное, например, классифицирует изображения, то легко можем распространить ее на последовательность изображений, обернув слоем Keras `TimeDistributed`:

```
sequence_predictions = TimeDistributed(model)(input_sequences)
```

С помощью функционального API можно определить любую сеть, допускающую определение посредством последовательного API. Но обратное неверно – следующие типы сетей можно определить только с помощью функционального API:

- модели с несколькими входами и выходами;
- модели, являющиеся композицией нескольких подмоделей;
- модели, в которых используются разделяемые слои.

Модели с несколькими входами и выходами определяются путем раздельной композиции входов и выходов, как показано в предыдущем примере, с последующей передачей массивов входных и выходных функций в качестве параметров `inputs` и `outputs` конструктору класса `Model`:

```
model = Model(inputs=[input1, input2], outputs=[output1, output2])
```

Обычно модели с несколькими входами и выходами состоят из нескольких подсетей, результаты вычисления которых объединяются в конечный результат. Функция `merge` предлагает несколько способов объединения промежуточных результатов, например: векторное сложение, скалярное произведение и конкатенация. С примерами объединения мы встретимся при рассмотрении вопросно-ответной системы ниже.

Не менее полезным применением функционального API являются модели с разделяемыми слоями. Разделяемый слой определяется один раз, и используется во всех конвейерах, где нужны его веса.

В этой главе мы будем пользоваться в основном функциональным API, так что в примерах недостатка не будет. А на сайте Keras имеется еще много примеров на эту тему.

Регрессионные сети

Два основных вида обучения с учителем – классификация и регрессия. В обоих случаях модель обучается на данных с известными метками. В случае классификации метки – дискретные значения, например, жанр текста или категория изображения. В случае регрессии метками являются значения непрерывной величины, например, цены акций или коэффициент интеллектуального развития человека (IQ).

В большинстве рассмотренных выше примеров модели глубокого обучения использовались для классификации. В этом разделе мы рассмотрим, как такая модель позволяет выполнить регрессию.

Напомним, что в моделях классификации в конце имеется плотный слой с нелинейной функцией активации, число выходов которого равно числу классов в модели. Так, в модели классификации изображений из набора ImageNet размерность плотного выходного слоя равна 1000, т. е. он может предсказывать 1000 классов. Аналогично в модели анализа эмоциональной окраски плотный слой имеет два выхода, соответствующие положительной и отрицательной окраске.

В регрессионных моделях в конце тоже имеется плотный слой, но только с одним выходом и без нелинейной функции активации. Это значит, что плотный слой возвращает просто сумму откликов нейронов предыдущего слоя. Кроме того, в качестве функции потерь обычно используется **среднеквадратическая ошибка (СКО, англ. MSE)**, но допустимо и несколько других функций (см. <https://keras.io/losses/>).

Пример регрессии – предсказание содержания бензола в воздухе

В этом примере мы займемся предсказанием концентрации бензола в атмосфере, зная такие величины, как концентрации окиси углерода, окиси азота и т. д., а также температуру и относительную влажность. В качестве обучающих данных используется набор из репозитория машинного обучения UCI (<https://archive.ics.uci>.

edu/ml/datasets/Air+Quality). Этот набор содержит 9358 показаний, полученных с часовым интервалом от матрицы из пяти химических датчиков окислов металлов. Матрица датчиков находилась в одном итальянском городе, а показания регистрировались с марта 2004 по февраль 2005.

Как обычно, сначала импортируем нужные библиотеки:

```
from keras.layers import Input
from keras.layers.core import Dense
from keras.models import Model
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
```

Набор данных представлен в виде CSV-файла. Данные загружаются во фрейм данных Pandas (см. <http://pandas.pydata.org/>). Pandas – популярная библиотека для анализа данных, в основе которой лежит понятие фрейма данных (объект `DataFrame`), заимствованное из языка R. Мы используем здесь Pandas по двум причинам. Во-первых, набор данных содержит пустые поля, соответствующие отсутствию показаний датчиков. Во-вторых, вместо десятичных точек используются запятые, как принято в некоторых европейских странах. В Pandas встроена поддержка того и другого, а также ряд дополнительных удобств, о которых мы скажем в свое время.

```
DATA_DIR = "../data"
AIRQUALITY_FILE = os.path.join(DATA_DIR, "AirQualityUCI.csv")

aqdf = pd.read_csv(AIRQUALITY_FILE, sep=";", decimal=".", header=0)

# удалить 2 первых и 2 последних столбца
del aqdf["Date"]
del aqdf["Time"]
del aqdf["Unnamed: 15"]
del aqdf["Unnamed: 16"]

# Вместо NaN подставить средние значения по столбцу
aqdf = aqdf.fillna(aqdf.mean())

Xorig = aqdf.as_matrix()
```

Здесь мы удаляем первые два столбца, содержащие дату и время наблюдения, и два последних столбца, которые, похоже, не содержат ничего полезного. Затем заменяем пустые поля средними

значениями по столбцу. И наконец, экспортируем, фрейм данных в виде матрицы для последующего использования.

Отметим, что масштаб в каждом столбце разный, потому что измерялись различные величины. Например, концентрация окиси олова измерялась в масштабе 1:1000, а концентрации неметановых углеводородов – в масштабе 1:100. Во многих ситуациях признаки однородны, так что различие масштабов не составляет проблемы, но в таких, как эта, лучше привести данные к единому масштабу. В данном случае масштабирование сводится к вычитанию из каждого значения среднего по столбцу и делению на стандартное отклонение:

$$z = \frac{x - \mu}{\sigma}$$

Для этого воспользуемся классом `StandardScaler` из библиотеки `scikit-learn`, как показано ниже. Мы сохраняем среднее и стандартное отклонение, потому что они еще понадобятся для вывода результатов и предсказания на основе новых данных. Меткой является четвертый столбец набора данных, поэтому мы разделяем масштабированные данные на входные переменные x и целевую переменную y :

```
scaler = StandardScaler()
Xscaled = scaler.fit_transform(Xorig)
# сохраняем для предсказаний на основе новых данных
Xmeans = scaler.mean_
Xstds = scaler.scale_

y = Xscaled[:, 3]
X = np.delete(Xscaled, 3, axis=1)
```

Затем мы разбиваем данные на обучающий и тестовый набор в пропорции 70:30, т. е. оставляем 6549 записей для обучения и 2808 для тестирования:

```
train_size = int(0.7 * X.shape[0])
Xtrain, Xtest, ytrain, ytest = X[0:train_size], X[train_size:],
    y[0:train_size], y[train_size:]
```

Далее определяется сеть. Это простая плотная сеть с двумя слоями, которая принимает на входе вектор из 12 признаков и выводит масштабированное предсказание. Плотный скрытый слой содержит восемь нейронов. Матрица весов в обоих плотных сло-

ях инициализируется по схеме, которая называется *glorot uniform* (равномерная инициализация Глорота). Полный перечень схем инициализации приведен по адресу <https://keras.io/initializers/>. В качестве функции потерь мы задаем среднеквадратическую ошибку (*mse*), а в качестве оптимизатора *adam*:

```
readings = Input(shape=(12,))
x = dense(8, activation="relu", kernel_initializer="glorot_uniform")(readings)
benzene = dense(1, kernel_initializer="glorot_uniform")(x)

model = Model(inputs=[readings], outputs=[benzene])
model.compile(loss="mse", optimizer="adam")
```

Мы обучаем эту модель на протяжении 20 периодов с размером пакета 10:

```
NUM_EPOCHS = 20
BATCH_SIZE = 10

history = model.fit(Xtrain, ytrain, batch_size=BATCH_SIZE,
                    epochs=NUM_EPOCHS, validation_split=0.2)
```

В результате получается модель со среднеквадратической ошибкой 0.0003 (квадратный корень из нее приблизительно равен 2%) на обучающем наборе и 0.0016 (квадратный корень примерно 4%) на контрольном наборе. Это видно из показанного ниже протокола обучения:

```
Epoch 8/20
5239/5239 [=====] - 0s - loss: 0.0015 - val_loss: 0.0024
Epoch 9/20
5239/5239 [=====] - 0s - loss: 0.0012 - val_loss: 0.0020
Epoch 10/20
5239/5239 [=====] - 0s - loss: 9.5742e-04 - val_loss: 0.0018
Epoch 11/20
5239/5239 [=====] - 0s - loss: 8.2761e-04 - val_loss: 0.0019
Epoch 12/20
5239/5239 [=====] - 0s - loss: 7.1237e-04 - val_loss: 0.0021
Epoch 13/20
5239/5239 [=====] - 0s - loss: 6.4492e-04 - val_loss: 0.0018
Epoch 14/20
5239/5239 [=====] - 0s - loss: 6.0119e-04 - val_loss: 0.0019
Epoch 15/20
5239/5239 [=====] - 0s - loss: 5.1915e-04 - val_loss: 0.0017
Epoch 16/20
5239/5239 [=====] - 0s - loss: 4.4686e-04 - val_loss: 0.0014
Epoch 17/20
5239/5239 [=====] - 0s - loss: 5.6912e-04 - val_loss: 0.0019
Epoch 18/20
5239/5239 [=====] - 0s - loss: 3.6897e-04 - val_loss: 0.0013
Epoch 19/20
5239/5239 [=====] - 0s - loss: 3.6652e-04 - val_loss: 0.0012
Epoch 20/20
5239/5239 [=====] - 0s - loss: 3.2395e-04 - val_loss: 0.0016
```


Мы также сравним зарегистрированные концентрации бензола с предсказанными моделью. Те и другие преобразуются к исходному масштабу.

```
ytest_ = model.predict(Xtest).flatten()
for i in range(10):
    label = (ytest[i] * Xstds[3]) + Xmeans[3]
    prediction = (ytest_[i] * Xstds[3]) + Xmeans[3]
    print("Benzene Conc. expected: {:.3f}, predicted: {:.3f}".format(label,
prediction))
```

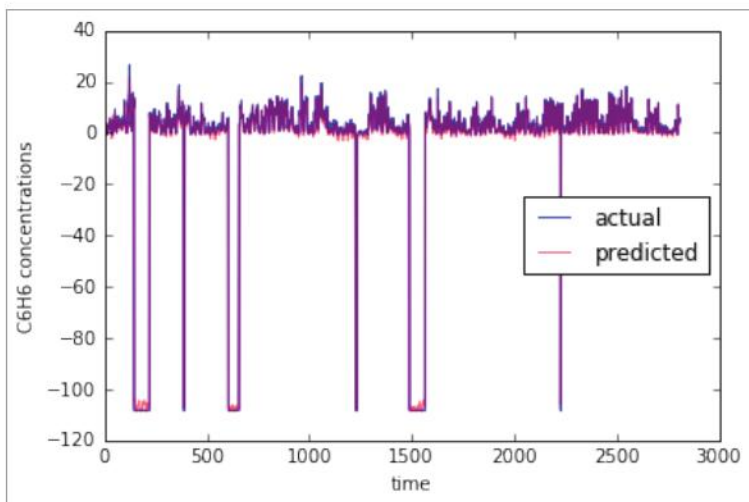
Сравнение показывает, что предсказания довольно близки к истинным значениям:

```
Benzene Conc. expected: 4.600, predicted: 5.254
Benzene Conc. expected: 5.500, predicted: 4.932
Benzene Conc. expected: 6.500, predicted: 5.664
Benzene Conc. expected: 10.300, predicted: 8.482
Benzene Conc. expected: 8.900, predicted: 6.705
Benzene Conc. expected: 14.000, predicted: 12.928
Benzene Conc. expected: 9.200, predicted: 7.128
Benzene Conc. expected: 8.200, predicted: 5.983
Benzene Conc. expected: 7.200, predicted: 6.256
Benzene Conc. expected: 5.500, predicted: 5.184
```

Наконец, нанесем на общий график фактические и предсказанные значения для всего тестового набора. И снова мы видим, что предсказания сети очень близки к истинным наблюдениям:

```
plt.plot(np.arange(ytest.shape[0]), (ytest * Xstds[3]) / Xmeans[3],
         color="b", label="actual")
plt.plot(np.arange(ytest_.shape[0]), (ytest_ * Xstds[3]) / Xmeans[3],
         color="r", alpha=0.5, label="predicted")
plt.xlabel("time")
plt.ylabel("C6H6 concentrations")
plt.legend(loc="best")
plt.show()
```

Построенный график показан ниже.



Обучение без учителя – автокодировщики

Автокодировщики – это класс нейронных сетей, которые пытаются реконструировать входные данные с применением обратного распространения. Автокодировщик состоит из двух частей: кодировщик и декодер. Кодировщик читает входные данные и сжимает их, порождая более компактное представление, а декодер читает это представление и пытается восстановить по нему вход. Иными словами, автокодировщик пытается обучить тождественную функцию, минимизируя ошибку реконструкции.

На первый взгляд, тождественная функция не представляет ничего интересного, но важно, как именно производится обучение. Число скрытых слоев автокодировщика обычно меньше числа входных (и выходных) блоков. Это вынуждает кодировщик обучаться сжато́му представлению входа, которое декодер реконструирует. Если входные данные обладают структурой в виде корреляций между входными признаками, то автокодировщик выявит некоторые корреляции и в итоге обучится представлению данных меньшей размерности аналогично тому, как это делается в **методе главных компонент** (principal component analysis, PCA).

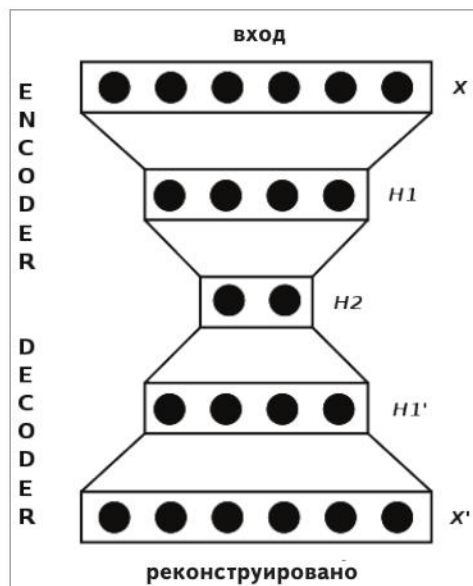
Обучив автокодировщик, декодер обычно отбрасывают и используют только кодировщик для порождения компактных представлений входных данных. Можно вместо этого использовать кодировщик как детектор признаков, порождающий компактное, семантически полноценное представление входа, и построить классификатор, присоединив к скрытому слою слой с функцией активации softmax.

Кодировщик и декодер можно реализовать с помощью плотной, сверточной или рекуррентной сети в зависимости от характера моделируемых данных. Так, плотные сети хороши для автокодировщиков, применяемых для построения моделей **коллаборативной фильтрации** (см. S. Sedhain «AutoRec: Autoencoders Meet Collaborative Filtering», Proceedings of the 24th International Conference on World Wide Web, ACM, 2015 и H. Cheng «Wide & Deep Learning for Recommender Systems», Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, ACM, 2016), где обучается сжатая модель пользовательских предпочтений на основе имеющихся разреженных рейтингов. Сверточные нейронные сети подходят для ситуации, рассмотренной в статье M. Runfeldt «Using Deep Learning to Remove Eyeglasses from Faces», а рекуррентные – для автокодировщиков на основе текстовых данных, в частности, для предсказания будущего пациента по электронным историям болезни (см. R. Miotto «Deep Patient: An Unsupervised Representation to Predict the Future of Patients from the Electronic Health Records», Scientific Reports 6, 2016) и предсказания предложений, окружающих данное, методом «векторов пропущенных мыслей» (skip-thought vectors) (см. R. Kiros «Skip-Thought Vectors», Advances in Neural Information Processing Systems, 2015).

Автокодировщики можно также объединять, последовательно соединяя кодировщики, которые все плотнее и плотнее сжимают входные данные, и декодеры, которые выполняют противоположные операции. Составные автокодировщики обладают большей выразительной способностью, а последовательные слои представлений улавливают иерархическую структуру входных данных по аналогии со сверточными и пулинговыми слоями в сверточных нейронных сетях.

Раньше составные автокодировщики обучались послойно. Например, в показанной ниже сети мы сначала обучили бы слой X реконструировать слой X' с помощью скрытого слоя $H1$ (игнорируя $H2$). Затем мы обучили бы слой $H1$ реконструировать $H1'$ с помо-

стью скрытого слоя $H2$. Наконец, мы соединили бы все слои вместе в изображенную на рисунке конфигурацию и произвели бы ее окончательную настройку для реконструкции X' по X . Но теперь, когда появились более совершенные функции активации и методы регуляризации, такие сети стали обучать как единое целое.



В статье «Building Autoencoders in Keras» (<https://blog.keras.io/building-autoencoders-in-keras.html>) есть интересные примеры построения автокодировщиков, которые реконструируют изображения рукописных цифр из набора MNIST с помощью полносвязных и сверточных нейронных сетей. Там же имеется обсуждение шумоподавляющих и вариационных автокодировщиков, о которых здесь мы говорить не будем.

Пример автокодировщика – векторы предложений

В этом примере мы построим и обучим основанный на LSTM автокодировщик, который будет порождать векторы предложений для документов из корпуса Reuters-21578 (<https://archive.ics.uci.edu/ml/datasets/Reuters-21578+Text+Categorization+Collection>). В главе 5 мы уже встречались с погружениями слов, в результате чего

получается вектор, представляющий смысл слова в контексте других слов, совместно с которыми оно встречается. А сейчас мы посмотрим, как построить аналогичные векторы для предложений. Предложение – это последовательность слов, а вектор предложения представляет его смысл.

Самый простой способ построить вектор предложения – сложить все векторы слов и поделить сумму на число слов. Но в этом случае предложение трактуется как мешок слов, и порядок слов не принимается во внимание. При таком подходе предложения *The dog bit the man* (Собака укусила человека) и *The man bit the dog* (Человек укусил собаку) считались бы идентичными. LSTM предназначена для работы с входными последовательностями и учитывает порядок слов, поэтому является более естественным представлением предложения.

Сначала импортируем библиотеки:

```
from sklearn.model_selection import train_test_split
from keras.callbacks import ModelCheckpoint
from keras.layers import Input
from keras.layers.core import RepeatVector
from keras.layers.recurrent import LSTM
from keras.layers.wrappers import Bidirectional
from keras.models import Model
from keras.preprocessing import sequence
from scipy.stats import describe
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np
import os
```

Данные представлены в виде набора SGML-файлов. Мы уже разбирали и консолидировали эти данные в один текстовый файл в главе 6, когда рассматривали частеречную разметку на базе GRU. Воспользуемся ими повторно, чтобы сначала преобразовать каждый блок текста в список предложений, по одному предложению в строке.

```
sents = []
fsent = open(sent_filename, "rb")
for line in fsent:
    docid, sent_id, sent = line.strip().split("t")
    sents.append(sent)
fsent.close()
```

Для построения словаря мы снова прочитаем этот список предложений пословно. При добавлении в словарь каждое слово норма-

лизуется. Нормализация заключается в том, чтобы заменить каждую лексему, похожую на число, цифрой 9 и перевести все лексемы в нижний регистр. В результате формируется таблица частот слов `word_freqs`. Кроме того, мы вычисляем длину каждого предложения и создаем список разобранных предложений, соединяя лексемы через пробел, чтобы их было проще разобрать на следующем шаге.

```
def is_number(n):
    temp = re.sub("[.,-/>]", "", n)
    return temp.isdigit()

word_freqs = collections.Counter()
sent_lens = []
parsed_sentences = []
for sent in sentences:
    words = nltk.word_tokenize(sent)
    parsed_words = []
    for word in words:
        if is_number(word):
            word = "9"
        word_freqs[word.lower()] += 1
        parsed_words.append(word)
    sent_lens.append(len(words))
    parsed_sentences.append(" ".join(parsed_words))
```

Это дает нам некоторую информацию о корпусе текстов, которая поможет вычислить подходящие значения констант для LSTM-сети:

```
sent_lens = np.array(sent_lens)
print("number of sentences: {:d}".format(len(sent_lens)))
print("distribution of sentence lengths (number of words)")
print("min:{:d}, max:{:d}, mean:{:.3f}, med:{:.3f}".format(
    np.min(sent_lens), np.max(sent_lens), np.mean(sent_lens),
    np.median(sent_lens)))
print("vocab size (full): {:d}".format(len(word_freqs)))
```

В результате получаем такие сведения о корпусе:

```
number of sentences: 131545
distribution of sentence lengths (number of words)
min: 1, max: 429, mean: 22.315, median: 21.000
vocab size (full): 50751
```

Исходя из этой информации, зададим константы. Положим `VOCAB_SIZE=5000`, т. е. словарь будет включать 5000 самых часто встречающихся слов, что покрывает свыше 93 % всех слов в корпусе. Остальные слова будем считать **несловарными** и заменять фиктивной лексемой `UNK`. На этапе предсказания любому слову, кото-

рое модель раньше не видела, также будет сопоставляться лексема UNK. В качестве `SEQUENCE_LEN` выберем значение, близкое к удвоенной медианной длине последовательностей в обучающем наборе; на самом деле примерно 110 миллионов из 131 миллиона предложений короче выбранного значения. Те предложения, которые короче `SEQUENCE_LENGTH`, дополняются специальным символом `PAD`, а те, что длиннее, обрезаются:

```
VOCAB_SIZE = 5000
SEQUENCE_LEN = 50
```

Поскольку на вход нашей LSTM-сети подаются числа, необходимо построить таблицы соответствия между словами и их идентификаторами. Так как мы ограничили словарь 5000 словами и добавили два фиктивных слова `PAD` и `UNK`, в этих таблицах будут отражены 4998 реальных слов плюс `PAD` и `UNK`:

```
word2id = {}
word2id["PAD"] = 0
word2id["UNK"] = 1
for v, (k, _) in enumerate(word_freqs.most_common(VOCAB_SIZE - 2)):
    word2id[k] = v + 2
id2word = {v:k for k, v in word2id.items()}
```

На вход сети подается последовательность слов, в которой каждое слово представлено вектором. Можно было бы упростить себе жизнь и использовать унитарное кодирование слов, но тогда объем входных данных был бы слишком велик. Поэтому слова кодируются 50-мерными погружениями GloVe. Погружение порождается матрицей формы `(VOCAB_SIZE, EMBED_SIZE)`, в которой каждая строка представляет GloVe-погружение слова из словаря. В строках, описывающих `PAD` и `UNK` (с номерами 0 и 1), находятся соответственно нули и случайно выбранная комбинация:

```
EMBED_SIZE = 50

def lookup_word2id(word):
    try:
        return word2id[word]
    except KeyError:
        return word2id["UNK"]

def load_glove_vectors(glove_file, word2id, embed_size):
    embedding = np.zeros((len(word2id), embed_size))
    fglove = open(glove_file, "rb")
    for line in fglove:
        cols = line.strip().split()
```

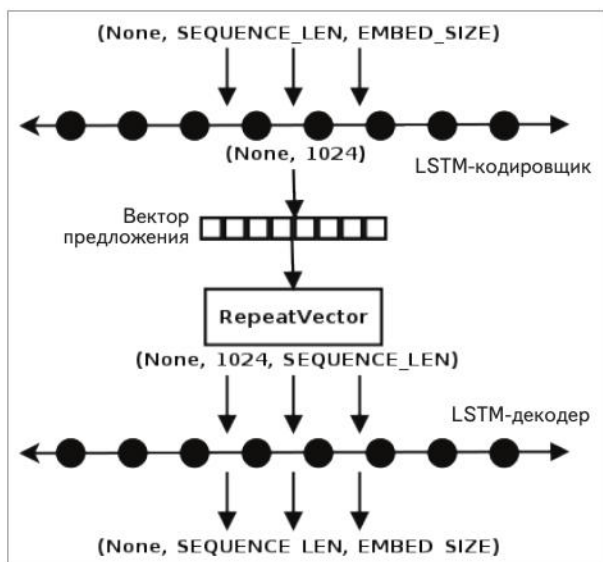
```

word = cols[0]
if embed_size == 0:
    embed_size = len(cols) - 1
if word2id.has_key(word):
    vec = np.array([float(v) for v in cols[1:]])
embedding[lookup_word2id(word)] = vec
embedding[word2id["PAD"]] = np.zeros((embed_size))
embedding[word2id["UNK"]] = np.random.uniform(-1, 1, embed_size)
return embedding

embeddings = load_glove_vectors(os.path.join(
    DATA_DIR, "glove.6B.{:d}.txt".format(EMBED_SIZE)),
    word2id, EMBED_SIZE)

```

Наша модель автокодировщика принимает последовательность GloVe-векторов слов и обучается порождать другую последовательность, похожую на входную. LSTM-кодировщик сжимает последовательность в контекстный вектор фиксированной длины, по которой LSTM-декодер реконструирует исходную последовательность. Схематически сеть представлена на следующем рисунке:



Поскольку объем входных данных очень велик, мы воспользуемся генератором, порождающим входные пакеты, а именно тензоры формы $(BATCH_SIZE, SEQUENCE_LEN, EMBED_SIZE)$. Здесь $BATCH_SIZE$ равно 64, а поскольку мы используем погружения GloVe в 50-мерное векторное пространство, то $EMBED_SIZE$ равно 50. В начале каж-

дого периода предложения перемешиваются и отдаются пакеты по 64 предложения в каждом. Каждое предложение представлено в виде вектора GloVe-векторов слов. Если у слова из словаря нет соответствующего GloVe-погружения, то оно представляется нулевым вектором. Мы создаем два генератора: один для обучающих данных, другой для тестовых, составляющих соответственно 70 % и 30 % исходного набора данных.

```
BATCH_SIZE = 64

def sentence_generator(X, embeddings, batch_size):
    while True:
        # одна итерация цикла на период
        num_recs = X.shape[0]
        indices = np.random.permutation(np.arange(num_recs))
        num_batches = num_recs // batch_size
        for bid in range(num_batches):
            sids = indices[bid * batch_size : (bid + 1) * batch_size]
            yield Xbatch, Xbatch

train_size = 0.7
Xtrain, Xtest = train_test_split(sent_wids, train_size=train_size)
train_gen = sentence_generator(Xtrain, embeddings, BATCH_SIZE)
test_gen = sentence_generator(Xtest, embeddings, BATCH_SIZE)
```

Теперь все готово к определению автокодировщика. Как показано на рисунке, он состоит из LSTM-кодировщика и LSTM-декодера. LSTM-кодировщик читает тензор формы (BATCH_SIZE, SEQUENCE_LEN, EMBED_SIZE), представляющий пакет предложений. Каждое предложение представлено последовательностью слов, дополненной до фиксированной длины SEQUENCE_LEN, а каждое слово – 50-мерным GloVe-вектором. Размерность выхода LSTM-кодировщика, т. е. длина вектора, порождаемого обученным автокодировщиком – гиперпараметр LATENT_SIZE. Векторное пространство размерности LATENT_SIZE и есть латентное пространство, кодирующее смысл предложения. Поскольку выходом LSTM для любого предложения является вектор длины LATENT_SIZE, то выходной тензор для всего пакета имеет форму (BATCH_SIZE, LATENT_SIZE). Этот тензор подается на вход слою RepeatVector, который реплицирует его для всей последовательности, т. е. выходной тензор этого слоя имеет форму (BATCH_SIZE, SEQUENCE_LEN, LATENT_SIZE). Этот тензор подается на вход LSTM-декодеру с размерностью выхода EMBED_SIZE, так что выходной тензор имеет форму (BATCH_SIZE, SEQUENCE_LEN, EMBED_SIZE) – такую же, как у входного тензора.

При компиляции этой модели задается оптимизатор СГС и средне-квадратическая ошибка в качестве функции потерь. Такая потеря выбрана, потому что мы хотим реконструировать предложение, сохранив по возможности смысл, т. е. получить нечто близкое к исходному предложению в пространстве погружения размерности `LATENT_SIZE`:

```
inputs = Input(shape=(SEQUENCE_LEN, EMBED_SIZE), name="input")
encoded = Bidirectional(LSTM(LATENT_SIZE), merge_mode="sum",
    name="encoder_lstm")(inputs)
decoded = RepeatVector(SEQUENCE_LEN, name="repeater")(encoded)
decoded = Bidirectional(LSTM(EMBED_SIZE, return_sequences=True),
    merge_mode="sum", name="decoder_lstm")(decoded)

autoencoder = Model(inputs, decoded)

autoencoder.compile(optimizer="sgd", loss="mse")
```

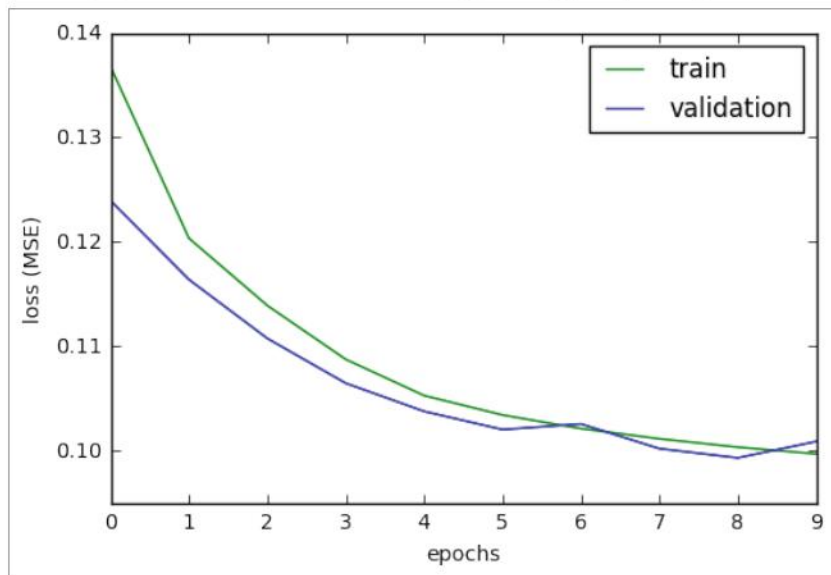
Этот автокодировщик мы обучаем на протяжении 10 периодов. Такое значение достаточно для сходимости СКО. Кроме того, мы сохраняем наилучшую с точки зрения потери СКО модель, найденную за это время.

```
num_train_steps = len(Xtrain) // BATCH_SIZE
num_test_steps = len(Xtest) // BATCH_SIZE
checkpoint = ModelCheckpoint(filepath=os.path.join(DATA_DIR,
    "sent-thoughts-autoencoder.h5"), save_best_only=True)
history = autoencoder.fit_generator(train_gen,
    steps_per_epoch=num_train_steps,
    epochs=NUM_EPOCHS,
    validation_data=test_gen,
    validation_steps=num_test_steps,
    callbacks=[checkpoint])
```

Ниже показаны результаты обучения. Как видим, СКО на обучающем наборе уменьшилось с 0.14 до 0.1, а на контрольном – с 0.12 до 0.1.

```
Epoch 1/10
92032/92032 [=====] - 542s - loss: 0.1368 - val_loss: 0.1239
Epoch 2/10
92032/92032 [=====] - 540s - loss: 0.1203 - val_loss: 0.1164
Epoch 3/10
92032/92032 [=====] - 546s - loss: 0.1139 - val_loss: 0.1107
Epoch 4/10
92032/92032 [=====] - 547s - loss: 0.1087 - val_loss: 0.1064
Epoch 5/10
92032/92032 [=====] - 542s - loss: 0.1053 - val_loss: 0.1036
Epoch 6/10
92032/92032 [=====] - 543s - loss: 0.1034 - val_loss: 0.1020
Epoch 7/10
92032/92032 [=====] - 544s - loss: 0.1021 - val_loss: 0.1025
Epoch 8/10
92032/92032 [=====] - 545s - loss: 0.1011 - val_loss: 0.1002
Epoch 9/10
92032/92032 [=====] - 545s - loss: 0.1003 - val_loss: 0.0993
Epoch 10/10
92032/92032 [=====] - 545s - loss: 0.0997 - val_loss: 0.1009
```

На графике изменение СКО выглядит следующим образом:



Поскольку мы подаем на вход матрицу погружений, то на выходе также будет матрица погружений слов. Поскольку пространство погружения непрерывное, а наш словарь дискретный, не всякое выходное погружение будет соответствовать слову. Лучшее, что можно сделать, — найти слово, ближайшее к выходному погружению, и таким образом реконструировать оригинальный текст. Это слишком громоздко, поэтому мы будем оценивать автокодировщик иначе.

Поскольку цель автокодировщика — получить хорошее латентное представление, то будем оценивать латентные векторы, порожденные кодировщиком, сопоставляя оригинальные входные данные с выходом автокодировщика:

```
encoder = Model(autoencoder.input, autoencoder.get_layer("encoder_lstm").output)
```

Затем прогоним автокодировщик на тестовом наборе и вернем предсказанные погружения. После этого пропустим входные и предсказанные погружения через кодировщик, который породит для тех и других векторы предложений, и сравним эти векторы, применяя косинусное расстояние. Если косинусное расстояние близко к 1, то векторы похожи, а если к 0, то не похожи. Показан-

ный ниже код выбирает случайное подмножество, содержащее 500 тестовых предложений, и вычисляет косинусные расстояния между векторами предложений, сгенерированными по исходному погружению и по соответствующему целевому погружению, порожденному автокодировщиком:

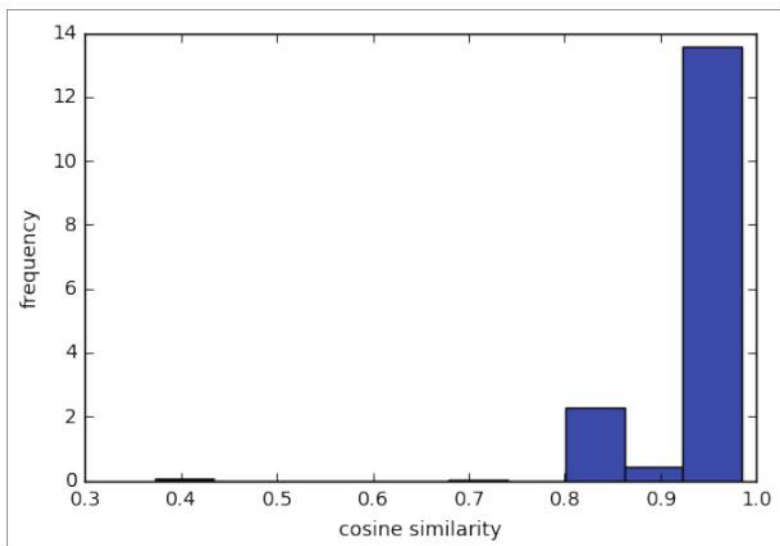
```
def compute_cosine_similarity(x, y):
    return np.dot(x, y) / (np.linalg.norm(x, 2) * np.linalg.norm(y, 2))

k = 500
cosims = np.zeros((k))
i = 0
for bid in range(num_test_steps):
    xtest, ytest = test_gen.next()
    ytest_ = autoencoder.predict(xtest)
    Xvec = encoder.predict(xtest)
    Yvec = encoder.predict(ytest_)
    for rid in range(Xvec.shape[0]):
        if i >= k:
            break
        cosims[i] = compute_cosine_similarity(Xvec[rid], Yvec[rid])
        if i <= 10:
            print(cosims[i])
            i += 1
    if i >= k:
        break
```

Ниже приведены первые 10 косинусных расстояний. Как видим, векторы очень похожи:

```
0.982818722725
0.970908224583
0.98131018877
0.974798440933
0.968060493469
0.976065933704
0.96712064743
0.949920475483
0.973583400249
0.980291545391
0.817819952965
```

На следующем рисунке показана гистограмма распределения косинусных расстояний между векторами предложений для первых 500 предложений из тестового набора. Она также подтверждает, что векторы, сгенерированные по входным данным и по выходу автокодировщика, очень похожи, а, значит, получившийся вектор предложения является хорошим представлением предложения.



Композиция глубоких сетей

Мы подробно рассмотрели три основных вида сетей глубокого обучения: **полносвязные сети (ПСС)**, сверточные сети и рекуррентные сети. Каждая из них применяется для решения определенных задач. Но из них можно составлять также более крупные и полезные модели, соединяя, как детали конструктора Lego, с помощью функционального API Keras разными интересными способами.

Такие модели обычно предназначены для решения узкоспециализированных задач, поэтому говорить об их обобщении не приходится. Как правило, они либо обучаются на данных из нескольких источников, либо генерируют несколько выходов. Примером может служить вопросно-ответная сеть, которая обучается давать ответы, получив на входе некоторую историю и вопрос. Другой пример – сиамская сеть, которая вычисляет сходство между двумя изображениями и выдает бинарный (похожи-непохожи) или категориальный (степень сходства) ответ. Еще один пример – сеть классификации и локализации объектов, которая обучается предсказывать категорию изображения, а также место изображенного объекта на предлагаемой картинке. В первых двух случаях мы имеем составные сети с несколькими входами, а в последнем – сеть с несколькими выходами.

Пример – сеть с памятью для ответов на вопросы

В этом примере мы построим сеть с памятью, которая будет отвечать на вопросы. Сети с памятью – это специализированная архитектура, в которой помимо других обучаемых нейронов (блоков), обычно РНС, имеются блоки памяти. Каждый входной элемент обновляет состояние памяти, а при вычислении окончательного выхода учитываются не только выходы обучаемых блоков, но и содержимое памяти. Эта архитектура была предложена в 2014 году в работе J. Weston, S. Chopra, A. Bordes «Memory Networks», arXiv:1410.3916, 2014. Годом позже в работе J. Weston «Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks», arXiv:1502.05698, 2015, был описан синтетический набор данных и стандартный набор из 20 вопросно-ответных задач увеличивающейся трудности, для решения которых применялись различные сети глубокого обучения. Как оказалось, сеть с памятью во всех случаях показывала наилучшие результаты. Впоследствии этот набор данных был выложен в открытый доступ в рамках проекта Facebook bAbI (<https://research.fb.com/projects/babi/>). Наша реализация сети с памятью ближе всего напоминает предложенную в статье S. Sukhbaatar, J. Weston, R. Fergus «End-To-End Memory Networks», Advances in Neural Information Processing Systems, 2015 в том смысле, что производится совместное обучение всех компонентов единой сети. Для решения первой вопросно-ответной задачи используется набор данных bAbI.

Первым делом импортируем библиотеки:

```
from keras.layers import Input
from keras.layers.core import Activation, dense, Dropout, Permute
from keras.layers.embeddings import Embedding
from keras.layers.merge import add, concatenate, dot
from keras.layers.recurrent import LSTM
from keras.models import Model
from keras.preprocessing.sequence import pad_sequences
from keras.utils import np_utils
import collections
import itertools
import nltk
import numpy as np
import matplotlib.pyplot as plt
import os
```

Данные в наборе bAbI для первой задачи состоят из 10 000 коротких предложений для обучения и тестирования. История со-

держит от двух до трех предложений. За последним предложением в каждой истории следует вопрос и ответ. Приведенный ниже код разбирает обучающие и тестовые файлы, формируя тройки (история, вопрос, ответ).

```
DATA_DIR = "../data"
TRAIN_FILE = os.path.join(DATA_DIR, "qal_single-supporting-fact_train.txt")
TEST_FILE = os.path.join(DATA_DIR, "qal_single-supporting-fact_test.txt")

def get_data(infile):
    stories, questions, answers = [], [], []
    story_text = []
    fin = open(TRAIN_FILE, "rb")
    for line in fin:
        line = line.decode("utf-8").strip()
        lno, text = line.split(" ", 1)
        if "t" in text:
            question, answer, _ = text.split("t")
            stories.append(story_text)
            questions.append(question)
            answers.append(answer)
            story_text = []
        else:
            story_text.append(text)
    fin.close()
    return stories, questions, answers

data_train = get_data(TRAIN_FILE)
data_test = get_data(TEST_FILE)
```

Следующий шаг – обработать тексты в построенных списках и создать словарь. Мы уже неоднократно делали нечто подобное. Но на этот раз словарь будет совсем маленьким, он содержит всего 22 уникальных слова, так что несловарные слова не понадобятся.

```
def build_vocab(train_data, test_data):
    counter = collections.Counter()
    for stories, questions, answers in [train_data, test_data]:
        for story in stories:
            for sent in story:
                for word in nltk.word_tokenize(sent):
                    counter[word.lower()] += 1
            for question in questions:
                for word in nltk.word_tokenize(question):
                    counter[word.lower()] += 1
            for answer in answers:
                for word in nltk.word_tokenize(answer):
                    counter[word.lower()] += 1
    word2idx = {w:(i+1) for i, (w, _) in enumerate(counter.most_common())}
```

```
word2idx["PAD"] = 0
idx2word = {v:k for k, v in word2idx.items()}
return word2idx, idx2word

word2idx, idx2word = build_vocab(data_train, data_test)

vocab_size = len(word2idx)
```

Сеть с памятью основана на РНС, в которой каждое предложение истории и вопрос рассматриваются как последовательности слов, поэтому нам нужно найти максимальную длину предложений. Это делает показанный ниже код. Как выясняется, максимальное число слов в предложениях из историй равно 14, а в вопросах – всего 4.

```
def get_maxlens(train_data, test_data):
    story_maxlen, question_maxlen = 0, 0
    for stories, questions, _ in [train_data, test_data]:
        for story in stories:
            story_len = 0
            for sent in story:
                swords = nltk.word_tokenize(sent)
                story_len += len(swords)
            if story_len > story_maxlen:
                story_maxlen = story_len
        for question in questions:
            question_len = len(nltk.word_tokenize(question))
            if question_len > question_maxlen:
                question_maxlen = question_len
    return story_maxlen, question_maxlen

story_maxlen, question_maxlen = get_maxlens(data_train, data_test)
```

Как и раньше, на вход РНС подается последовательность идентификаторов слов. Поэтому мы воспользуемся словарем, чтобы преобразовать тройку (история, вопрос, ответ) в последовательность целочисленных идентификаторов. Это делает следующий фрагмент кода, который заодно дополняет нулями результирующие последовательности до вычисленной ранее максимальной длины. После этого мы будем иметь списки дополненных последовательностей идентификаторов слов для каждой тройки из обучающего и тестового набора.

```
def vectorize(data, word2idx, story_maxlen, question_maxlen):
    Xs, Xq, Y = [], [], []
    stories, questions, answers = data
    for story, question, answer in zip(stories, questions, answers):
        xs = [[word2idx[w.lower()] for w in nltk.word_tokenize(s)] for s in story]
        xs = list(itertools.chain.from_iterable(xs))
```



```

xq = [word2idx[w.lower()] for w in nltk.word_tokenize(question)]
Xs.append(xs)
Xq.append(xq)
Y.append(word2idx[answer.lower()])
return pad_sequences(Xs, maxlen=story_maxlen),
       pad_sequences(Xq, maxlen=question_maxlen),
       np_utils.to_categorical(Y, num_classes=len(word2idx))

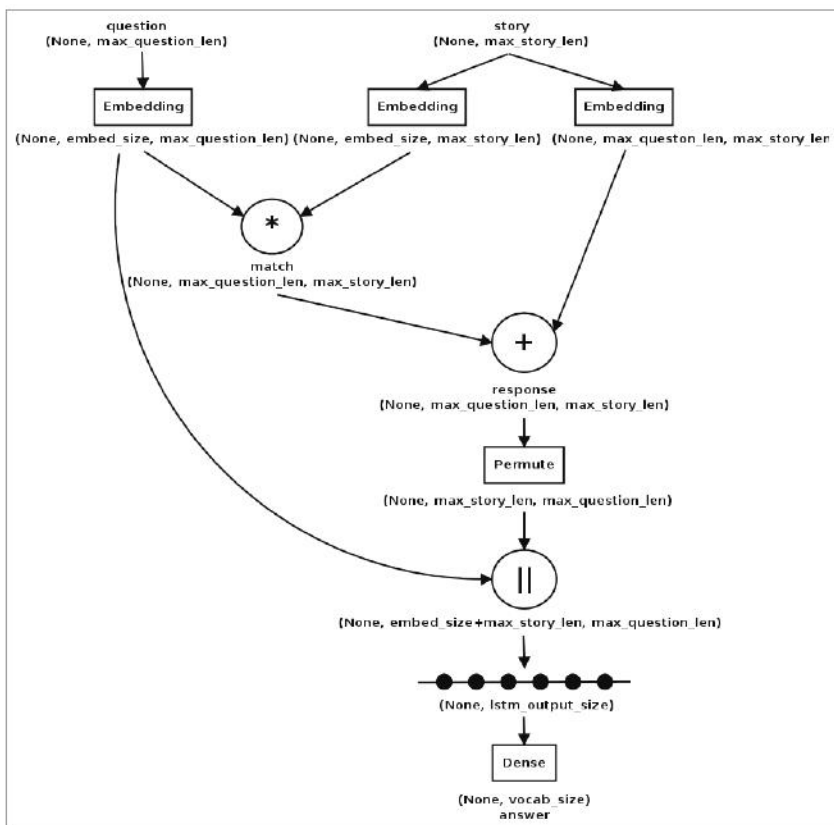
```

```

Xstrain, Xqtrain, Ytrain = vectorize(data_train, word2idx, story_maxlen,
                                     question_maxlen)
Xstest, Xqtest, Ytest = vectorize(data_test, word2idx, story_maxlen,
                                   question_maxlen)

```

Нам нужно определить модель. Определение оказывается длиннее, чем все, что мы видели раньше, поэтому, знакомясь с ним, будет удобно справляться со следующей диаграммой:



У нашей модели два входа: последовательности идентификаторов слов в вопросе и в предложениях истории. Каждая последовательность передается слою Embedding, который преобразует идентификаторы слов в векторы в 64-мерном пространстве погружения. Дополнительно последовательность, относящаяся к истории, пропускается через еще один слой Embedding, который проецирует ее в погружение размерности `max_question_len`. Все слои погружения инициализированы случайными весами и обучаются вместе с остальной сетью.

Первые два погружения (история и вопрос) объединяются с помощью операции скалярного произведения, образуя память сети. Это дает представления тех слов в истории и вопросе, которые совпадают или близки в пространстве погружения. Выход блока памяти объединяется со вторым погружением, относящимся к истории, с помощью операции сложения. Полученный отклик сети еще раз объединяется с погружением для вопроса, образуя последовательность-отклик, которая пропускается через LSTM. Контекстный вектор LSTM передается плотному слою, предсказывающему ответ – одно из слов в словаре.

Для обучения модели используется оптимизатор RMSprop и категориальная перекрестная энтропия в качестве функции потерь:

```
EMBEDDING_SIZE = 64
LATENT_SIZE = 32

# входные данные
story_input = Input(shape=(story_maxlen,))
question_input = Input(shape=(question_maxlen,))

# кодировщик историй
story_encoder = Embedding(input_dim=vocab_size, output_dim=EMBEDDING_SIZE,
                          input_length=story_maxlen)(story_input)
story_encoder = Dropout(0.3)(story_encoder)

# кодировщик вопросов
question_encoder = Embedding(input_dim=vocab_size, output_dim=EMBEDDING_SIZE,
                             input_length=question_maxlen)(question_input)
question_encoder = Dropout(0.3)(question_encoder)

# сопоставление истории с вопросом
match = dot([story_encoder, question_encoder], axes=[2, 2])

# погружение истории в векторное пространство вопроса
story_encoder_c = Embedding(input_dim=vocab_size, output_dim=question_maxlen,
                             input_length=story_maxlen)(story_input)
```

```

story_encoder_c = Dropout(0.3)(story_encoder_c)

# комбинирование векторов сопоставления и истории
response = add([match, story_encoder_c])
response = Permute((2, 1))(response)

# комбинирование векторов отклика и вопроса
answer = concatenate([response, question_encoder], axis=-1)
answer = LSTM(LATENT_SIZE)(answer)
answer = Dropout(0.3)(answer)
answer = dense(vocab_size)(answer)
output = Activation("softmax")(answer)

model = Model(inputs=[story_input, question_input], outputs=output)
model.compile(optimizer="rmsprop", loss="categorical_crossentropy",
              metrics=["accuracy"])

```

Эта сеть обучается на протяжении 50 периодов с размером пакета 32 и достигает на контрольном наборе верности 81%:

```

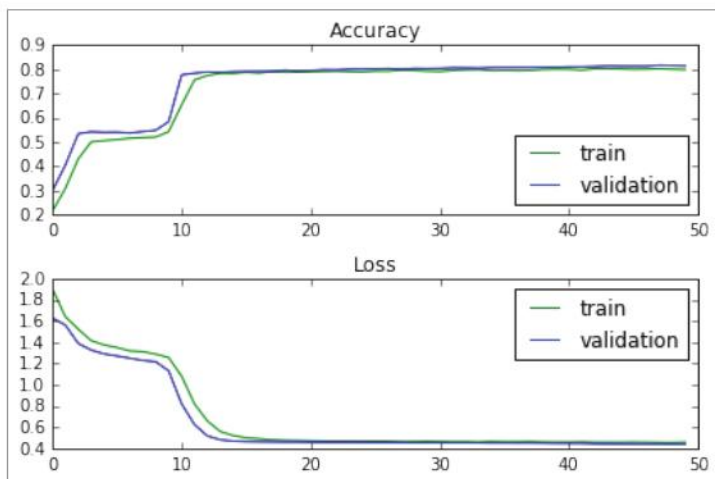
BATCH_SIZE = 32
NUM_EPOCHS = 50
history = model.fit([Xstrain, Xqtrain], [Ytrain], batch_size=BATCH_SIZE,
                    epochs=NUM_EPOCHS,
                    validation_data=([Xstest, Xqtest], [Ytest]))

```

Ниже показан протокол обучения:

Epoch 38/50	10000/10000	=====	5s	loss: 0.4636	acc: 0.7952	val_loss: 0.4499	val_acc: 0.8071
Epoch 39/50	10000/10000	=====	5s	loss: 0.4603	acc: 0.7993	val_loss: 0.4489	val_acc: 0.8083
Epoch 40/50	10000/10000	=====	5s	loss: 0.4590	acc: 0.8003	val_loss: 0.4475	val_acc: 0.8086
Epoch 41/50	10000/10000	=====	5s	loss: 0.4592	acc: 0.7997	val_loss: 0.4472	val_acc: 0.8099
Epoch 42/50	10000/10000	=====	5s	loss: 0.4611	acc: 0.7966	val_loss: 0.4466	val_acc: 0.8099
Epoch 43/50	10000/10000	=====	5s	loss: 0.4577	acc: 0.8025	val_loss: 0.4437	val_acc: 0.8114
Epoch 44/50	10000/10000	=====	5s	loss: 0.4576	acc: 0.8023	val_loss: 0.4431	val_acc: 0.8136
Epoch 45/50	10000/10000	=====	5s	loss: 0.4575	acc: 0.8013	val_loss: 0.4422	val_acc: 0.8127
Epoch 46/50	10000/10000	=====	5s	loss: 0.4587	acc: 0.7998	val_loss: 0.4420	val_acc: 0.8127
Epoch 47/50	10000/10000	=====	6s	loss: 0.4574	acc: 0.8005	val_loss: 0.4412	val_acc: 0.8126
Epoch 48/50	10000/10000	=====	5s	loss: 0.4559	acc: 0.8023	val_loss: 0.4408	val_acc: 0.8168
Epoch 49/50	10000/10000	=====	6s	loss: 0.4550	acc: 0.8003	val_loss: 0.4395	val_acc: 0.8154
Epoch 50/50	10000/10000	=====	5s	loss: 0.4577	acc: 0.7985	val_loss: 0.4407	val_acc: 0.8139

На следующих графиках показано изменение верности и потери на обучающем и контрольном наборах.



Мы прогнали модель для первых 10 историй из тестового набора для проверки качества предсказаний:

```

ytest = np.argmax(Ytest, axis=1)
Ytest_ = model.predict([Xstest, Xqtest])
ytest_ = np.argmax(Ytest_, axis=1)

for i in range(NUM_DISPLAY):
    story = " ".join([idx2word[x] for x in Xstest[i].tolist() if x != 0])
    question = " ".join([idx2word[x] for x in Xqtest[i].tolist()])
    label = idx2word[ytest[i]]
    prediction = idx2word[ytest_[i]]
    print(story, question, label, prediction)

```

Как видно из следующей таблицы, предсказания по большей части верны.

Story	Question	Answer	Predicted
mary moved to the bathroom . john went to the hallway .	where is mary ?	bathroom	bathroom
daniel went back to the hallway . sandra moved to the garden .	where is daniel ?	hallway	hallway
john moved to the office . sandra journeyed to the bathroom .	where is daniel ?	hallway	kitchen
mary moved to the hallway . daniel travelled to the office .	where is daniel ?	office	office
john went back to the garden . john moved to the bedroom .	where is sandra ?	bathroom	bedroom
sandra travelled to the office . sandra went to the bathroom .	where is sandra ?	bathroom	bathroom
mary went to the bedroom . daniel moved to the hallway .	where is sandra ?	bathroom	garden
john went to the garden . john travelled to the office .	where is sandra ?	bathroom	bathroom
daniel journeyed to the bedroom . daniel travelled to the hallway .	where is john ?	office	kitchen
john went to the bedroom . john travelled to the office .	where is daniel ?	hallway	kitchen

Расширение Keras

Составление крупных архитектур из элементарных строительных блоков позволяет строить интересные модели глубокого обучения, но иногда нужно подойти к задаче с другого конца. В Keras уже встроена богатая функциональность, поэтому вы, скорее всего, сможете создать нужные модели из готовых компонентов, и потребности в расширении вообще не возникнет. Но если все же понадобится, то в Keras есть для этого средства.

Как вы помните, Keras представляет собой высокоуровневый API, который делегирует тяжелую работу базовой библиотеке TensorFlow или Theano. Любой добавленный вами код расширения должен обращаться к одной из этих библиотек. Чтобы сохранить переносимость между ними, необходимо пользоваться переходным API Keras (<https://keras.io/backend/>), в котором имеется набор функций, образующих фасад для выбранной библиотеки. В зависимости от того, какая библиотека выбрана, фасад будет транслировать обращения в вызовы функций из TensorFlow или Theano. Полный перечень имеющихся функций с подробными описаниями имеется на вышеупомянутой странице.

Помимо переносимости, использование переходного API позволяет получить более удобный для сопровождения код, потому что код на Keras обычно более компактный и высокоуровневый, чем эквивалентный код на TensorFlow или Theano. В том маловероятном случае, когда необходимо работать с базовой библиотекой напрямую, компоненты Keras можно вызывать из кода на TensorFlow (но не Theano), как описано в блоге Keras (<https://blog.keras.io/keras-as-a-simplified-interface-to-tensorflow-tutorial.html>).

Расширение Keras обычно сводится к написанию своего слоя или своей функции расстояния. В этом разделе мы продемонстрируем создание нескольких простых слоев в Keras. Дополнительные примеры использования переходных функций для создания пользовательских компонентов Keras, например функций потерь, будут приведены в последующих разделах.

Пример – использование слоя lambda

Keras предоставляет слой lambda, позволяющий обернуть любую функцию. Например, чтобы построить слой, который поэлементно возводит в квадрат входной тензор, достаточно написать:

```
model.add(lambda (lambda x: x ** 2))
```

Слоем `lambda` можно также обертывать функции. Например, пусть нам нужен пользовательский слой, который поэлементно вычисляет евклидово расстояние между двумя входными тензорами. Тогда мы определим функцию, которая вычисляет само значение, и еще одну, которая возвращает форму ее выхода:

```
def euclidean_distance(vecs):
    x, y = vecs
    return K.sqrt(K.sum(K.square(x - y), axis=1, keepdims=True))

def euclidean_distance_output_shape(shapes):
    shape1, shape2 = shapes
    return (shape1[0], 1)
```

Далее эти функции вызываются из слоя `lambda`:

```
lhs_input = Input(shape=(VECTOR_SIZE,))
lhs = dense(1024, kernel_initializer="glorot_uniform",
            activation="relu")(lhs_input)

rhs_input = Input(shape=(VECTOR_SIZE,))
rhs = dense(1024, kernel_initializer="glorot_uniform",
            activation="relu")(rhs_input)

sim = lambda(euclidean_distance,
              output_shape=euclidean_distance_output_shape)([lhs, rhs])
```

Пример – построение пользовательского слоя нормировки

Слой `lambda` бывает очень полезен, но иногда необходим больший контроль. Рассмотрим, к примеру, код слоя нормировки, реализующий метод **локальной нормировки отклика**. Смысл метода заключается в нормировке входных данных в локальных областях, но сейчас он вышел из моды, потому что оказался не таким эффективным, как другие методы регуляризации, например, прореживание или пакетная нормировка, и к тому же появились улучшенные методы инициализации.

Для построения пользовательских слоев обычно применяются переходные функции, поэтому нужно выработать привычку рассуждать о коде в терминах тензоров. Напомним, что работа с тензорами – двухшаговый процесс. Сначала мы определяем тензоры и строим из них граф вычислений, а затем применяем этот граф к фактическим данным. Поэтому работать на этом уровне труднее, чем с остальными частями Keras. В документации имеются реко-

мендации по конструированию пользовательских слоев (<https://keras.io/layers/writing-your-own-keras-layers/>), и их нужно обязательно прочитать.

Чтобы упростить разработку кода с применением переходного API, можно подготовить простенький тестовый стенд, на котором проверяется, что код делает то, что нужно. Вот пример такого стенда, позаимствованный из исходного кода Keras, он применяет слой к входным данным и возвращает результат:

```
from keras.models import Sequential
from keras.layers.core import Dropout, Reshape

def test_layer(layer, x):
    layer_config = layer.get_config()
    layer_config["input_shape"] = x.shape
    layer = layer.__class__.from_config(layer_config)
    model = Sequential()
    model.add(layer)
    model.compile("rmsprop", "mse")
    x_ = np.expand_dims(x, axis=0)
    return model.predict(x_)[0]
```

А вот несколько тестов, которые Keras предоставляет, чтобы убедиться в правильности работы слоев на этом стенде:

```
from keras.layers.core import Dropout, Reshape
from keras.layers.convolutional import ZeroPadding2D
import numpy as np

x = np.random.randn(10, 10)
layer = Dropout(0.5)
y = test_layer(layer, x)
assert(x.shape == y.shape)

x = np.random.randn(10, 10, 3)
layer = ZeroPadding2D(padding=(1,1))
y = test_layer(layer, x)
assert(x.shape[0] + 2 == y.shape[0])
assert(x.shape[1] + 2 == y.shape[1])

x = np.random.randn(10, 10)
layer = Reshape((5, 20))
y = test_layer(layer, x)
assert(y.shape == (5, 20))
```

Прежде чем переходить к построению слоя локальной нормировки отклика, разберемся, что именно он делает. Эта техника впервые была применена в библиотеке Caffe, и в документации

(<http://caffe.berkeleyvision.org/tutorial/layers/lrn.html>) описывается как способ *латерального торможения*, в основе которого лежит нормировка в локальных входных областях. В режиме `ACROSS_CHANNEL` локальные области захватывают несколько соседних каналов, но не имеют пространственной протяженности. В режиме `WITHIN_CHANNEL` локальные области занимают место в пространстве, но только в пределах одного канала. Ниже описывается реализация режима `WITHIN_CHANNEL`. Формула нормировки локального отклика в режиме `WITHIN_CHANNEL` имеет вид:

$$LRN(x_i) = \frac{x_i}{\left(k + \frac{\alpha}{n} \sum_i x_i\right)^\beta}$$

Структура кода пользовательского слоя стандартная. В методе `__init__` задаются зависящие от приложения параметры, т. е. гиперпараметры слоя. Поскольку наш слой производит вычисления только в прямом направлении и не имеет обучаемых весов, то в методе `build` нужно лишь задать форму входа и вызвать метод `build` суперкласса, который позаботится о внутренней кухне. Если бы слой имел обучаемые веса, то здесь мы должны были бы задать их начальные значения.

В методе `call` производятся сами вычисления. Не забудьте учесть порядок измерений. Еще следует помнить, что размер пакета обычно неизвестен на этапе проектирования, поэтому операции нужно программировать так, чтобы размер пакета не упоминался. Само вычисление не вызывает трудностей и выполняется в соответствии с формулой. Сумму в знаменателе можно интерпретировать как усредненный пулинг по строке и столбцу с ядром размера (n, n) и шагом $(1, 1)$. Поскольку в процессе пулинга усреднение уже произведено, то еще раз делить сумму на n не нужно.

И последняя часть класса – метод `get_output_shape_for`. Поскольку слой нормирует каждый элемент входного тензора, размер выхода равен размеру входа:

```
from keras import backend as K
from keras.engine.topology import Layer, InputSpec

class LocalResponseNormalization(Layer):

    def __init__(self, n=5, alpha=0.0005, beta=0.75, k=2, **kwargs):
        self.n = n
```



```

self.alpha = alpha
self.beta = beta
self.k = k
super(LocalResponseNormalization, self).__init__(**kwargs)

def build(self, input_shape):
    self.shape = input_shape
    super(LocalResponseNormalization, self).build(input_shape)

def call(self, x, mask=None):
    if K.image_dim_ordering == "th":
        _, f, r, c = self.shape
    else:
        _, r, c, f = self.shape
    squared = K.square(x)
    pooled = K.pool2d(squared, (n, n), strides=(1, 1), padding="same", pool_mode="avg")
    if K.image_dim_ordering == "th":
        summed = K.sum(pooled, axis=1, keepdims=True)
        averaged = self.alpha * K.repeat_elements(summed, f, axis=1)
    else:
        summed = K.sum(pooled, axis=3, keepdims=True)
        averaged = self.alpha * K.repeat_elements(summed, f, axis=3)
    denom = K.pow(self.k + averaged, self.beta)
    return x / denom

def get_output_shape_for(self, input_shape):
    return input_shape

```

Этот слой можно тестировать в процессе разработки, пользуясь описанным выше тестовым стендом. Это проще, чем строить целую сеть, в которую можно вставить слой, или, хуже того, приступать к тестированию только после окончательного завершения разработки слоя.

```

x = np.random.randn(225, 225, 3)
layer = LocalResponseNormalization()
y = test_layer(layer, x)
assert(x.shape == y.shape)

```

Хотя создание пользовательских слоев – обычное дело для опытных разработчиков на Keras, в Интернете не так много примеров на эту тему. Возможно, это объясняется тем, что пользовательские слои пишутся для узкоспециализированных задач и не представляют интереса для широкой публики. Кроме того, из-за узкой специализации одного примера недостаточно для демонстрации всех возможностей API. Теперь, когда у вас есть представление о том, как пишутся пользовательские слои Keras, будет поучительно

взглянуть на примеры слоя Melspectogram в статье Keunwoo Choi (<https://keunwoochoi.wordpress.com/2016/11/18/for-beginners-writing-a-custom-keras-layer/>) и слоя NodeEmbeddingLayer в статье Shashank Gupta (<http://shashankg7.github.io/2016/10/12/Custom-Layer-In-Keras-Graph-Embedding-Case-Study.html>).

Порождающие модели

Порождающие модели обучаются создавать данные, подобные тем, на которых они обучались. Один такой пример мы видели в главе 6, когда обучали модель сочинять тексты, похожие на «Алису в Стране чудес». Там мы обучили модель предсказывать 11-ый символ текста по первым десяти. Еще один пример – **порождающие состязательные сети (ПСС)**, недавно появившийся весьма мощный класс моделей, – встречался нам в главе 4. Идея порождающих моделей заключается в том, что модель обучается хорошему представлению обучающих данных и потому может генерировать похожие данные на этапе предсказания.

На порождающие модели можно взглянуть также с вероятностной точки зрения. Типичная сеть для классификации или регрессии, называемая также дискриминантной моделью, обучается функции, отображающей входные данные X на некоторую метку y , т. е. эти модели обучаются условной вероятности $P(y|X)$. С другой стороны, порождающая модель обучается совместной вероятности входных данных и меток $P(X, y)$. Эти знания затем можно использовать для выборки новых примеров из распределения (X, y) . Поэтому порождающая модель способна объяснять скрытую структуру входных данных даже в отсутствии меток. На практике это очень важное преимущество, потому что помеченных данных в мире гораздо больше, чем помеченных.

Простые порождающие модели типа вышеупомянутого примера можно распространить также на звуковые данные. Так, модель можно обучить сочинению и воспроизведению музыки. Про одну такую модель можно прочитать в статье A. van den Oord «WaveNet: A Generative Model for Raw Audio», 2016, где описывается сеть WaveNet, построенная с применением дырчатых сверточных слоев, ее реализацию на Keras можно найти на сайте GitHub по адресу <https://github.com/basveeling/wavenet>.

Пример – глубокие сновидения

В этом примере мы рассмотрим немного другую порождающую сеть. Мы увидим, как воспользоваться предобученной сверточной сетью для порождения новых объектов в изображении. Сети, обученные различать изображения, накапливают столько знаний об изображениях, что способны также порождать их. Впервые это продемонстрировал Александр Мордвинцев из Google и описал в блоге Google (<https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html>). Первоначально эта идея получила название *инцепционизм*, но термин *deep dreaming*, или *Deep Dream* (глубокие сновидения) оказался более популярен.

В технологии Deep Dream активации обратно распространяющегося градиента прибавляются к изображению, и этот процесс повторяется в цикле. По ходу дела сеть оптимизирует функцию потерь, но мы посмотрим, как она это делает на примере входного изображения (трехканального), а не многомерного скрытого слоя, который трудно визуализировать.

У этой базовой стратегии много вариаций, и все они дают новые интересные эффекты. Упомянем размытие, наложение ограничений на общее число активаций, снижение градиента, бесконечное приближение изображения путем кадрирования и масштабирования, добавление дрожания путем случайного перемещения изображения в разных направлениях и т. д. В примере ниже будет продемонстрирован простейший подход – мы оптимизируем градиент средней активации выбранного слоя для каждого пулингового слоя предобученной модели VGG-16 и посмотрим, как это отражается на входном изображении.

Сначала, как обычно, импортируем библиотеки:

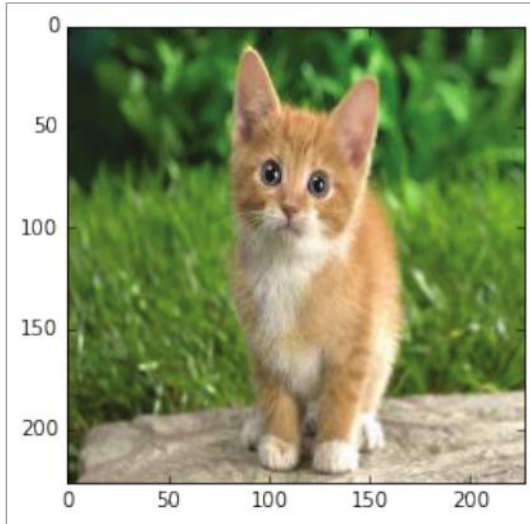
```
from keras import backend as K
from keras.applications import vgg16
from keras.layers import Input
import matplotlib.pyplot as plt
import numpy as np
import os
```

Затем загрузим входное изображение. Вы наверняка встречали его в блогах, посвященных глубокому обучению. Оригинал изображения находится по адресу <https://www.flickr.com/photos/billgarrett-newagecrap/14984990912/>:

```
DATA_DIR = "../data"
IMAGE_FILE = os.path.join(DATA_DIR, "cat.jpg")
```

```
img = plt.imread(IMAGE_FILE)
plt.imshow(img)
```

Вот результат работы этого кода:



Далее определим две функции: `preprocess` преобразует изображение в четырехмерное представление, которое можно подать на вход предобученной сети VGG-16, а `deprocess` производит обратное преобразование.

```
def preprocess(img):
    img4d = img.copy()
    img4d = img4d.astype("float64")
    if K.image_dim_ordering() == "th":
        # (H, W, C) -> (C, H, W)
        img4d = img4d.transpose((2, 0, 1))
        img4d = np.expand_dims(img4d, axis=0)
        img4d = vgg16.preprocess_input(img4d)
    return img4d

def deprocess(img4d):
    img = img4d.copy()
    if K.image_dim_ordering() == "th":
        # (B, C, H, W)
        img = img.reshape((img4d.shape[1], img4d.shape[2], img4d.shape[3]))
        # (C, H, W) -> (H, W, C)
        img = img.transpose((1, 2, 0))
    else:
        # (B, H, W, C)
```



```

img = img.reshape((img4d.shape[1], img4d.shape[2], img4d.shape[3]))
img[:, :, 0] += 103.939
img[:, :, 1] += 116.779
img[:, :, 2] += 123.68
# BGR -> RGB
img = img[:, :, ::-1]
img = np.clip(img, 0, 255).astype("uint8")
return img

```

Эти функции взаимно обратны, т. е. если сначала передать изображение `preprocess`, а затем передать результат `deprocess`, то получится исходное изображение.

Далее загружаем сеть VGG-16, предобученную на наборе данных ImageNet и входящую в состав дистрибутива Keras. В главе 3 мы уже рассказывали, как работать с предобученными моделями. Мы выбираем вариант, из которого уже удалены полносвязные слои. Мало того что это избавляет нас от необходимости удалять их самостоятельно, так мы еще можем передать изображение любого размера, потому что ширина и высота входного изображения задаются только для того, чтобы определить размер матриц весов в полносвязных слоях. Поскольку преобразования, выполняемые СНС, по природе своей локальны, размер изображения не влияет на размеры матриц весов сверточных и пулинговых слоев. Так что остается только одно ограничение на размер изображения – он должен быть одинаковым в пределах одного пакета:

```

img_copy = img.copy()
print("Original image shape:", img.shape)
p_img = preprocess(img_copy)
batch_shape = p_img.shape
dream = Input(batch_shape=batch_shape)
model = vgg16.VGG16(input_tensor=dream, weights="imagenet",
include_top=False)

```

В последующих вычислениях нам придется ссылаться на объекты слоев СНС по именам, поэтому построим словарь. Нужно также понимать соглашение об именовании слоев, поэтому распечатаем этот словарь:

```

layer_dict = {layer.name : layer for layer in model.layers}
print(layer_dict)

```

Результат выглядит так:

```

{'block1_conv1': <keras.layers.convolutional.Convolution2D at 0x1b847690>,
'block1_conv2': <keras.layers.convolutional.Convolution2D at 0x1b847f90>,

```

```
'block1_pool': <keras.layers.pooling.MaxPooling2D at 0x11c45db90>,
'block2_conv1': <keras.layers.convolutional.Convolution2D at 0x11c45ddd0>,
'block2_conv2': <keras.layers.convolutional.Convolution2D at 0x11b88f810>,
'block2_pool': <keras.layers.pooling.MaxPooling2D at 0x11c2d2690>,
'block3_conv1': <keras.layers.convolutional.Convolution2D at 0x11c47b890>,
'block3_conv2': <keras.layers.convolutional.Convolution2D at 0x11c510290>,
'block3_conv3': <keras.layers.convolutional.Convolution2D at 0x11c4afa10>,
'block3_pool': <keras.layers.pooling.MaxPooling2D at 0x11c334a10>,
'block4_conv1': <keras.layers.convolutional.Convolution2D at 0x11c345b10>,
'block4_conv2': <keras.layers.convolutional.Convolution2D at 0x11c345950>,
'block4_conv3': <keras.layers.convolutional.Convolution2D at 0x11d52c910>,
'block4_pool': <keras.layers.pooling.MaxPooling2D at 0x11d550c90>,
'block5_conv1': <keras.layers.convolutional.Convolution2D at 0x11d566c50>,
'block5_conv2': <keras.layers.convolutional.Convolution2D at 0x11d5b1910>,
'block5_conv3': <keras.layers.convolutional.Convolution2D at 0x11d5b1710>,
'block5_pool': <keras.layers.pooling.MaxPooling2D at 0x11fd68e10>,
'input_1': <keras.engine.topology.InputLayer at 0x11b847410>}
```

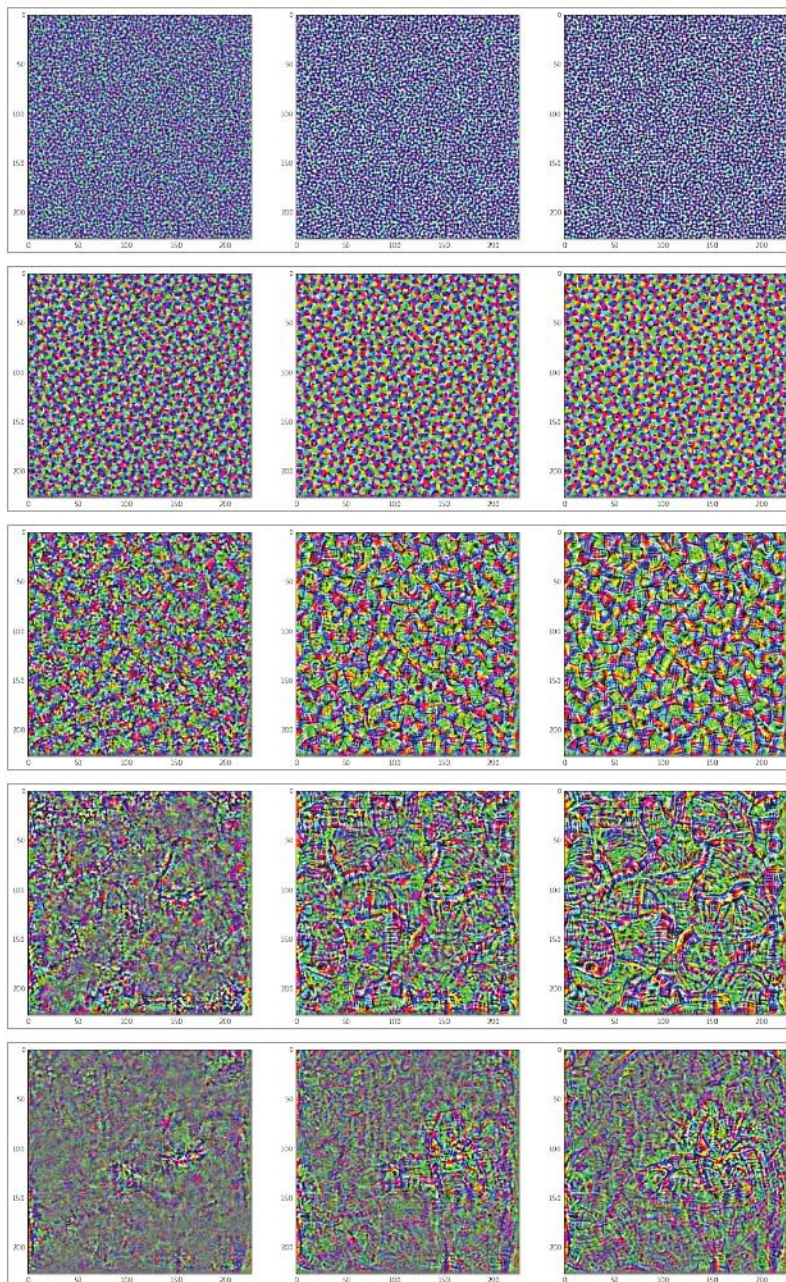
Затем вычисляется потеря в каждом из пяти пулинговых слоев и градиент средней активации для каждого из трех шагов. Градиент снова складывается с изображением и на каждом шаге выводится изображение в каждом из пулинговых слоев:

```
num_pool_layers = 5
num_iters_per_layer = 3
step = 100

for i in range(num_pool_layers):
    # идентифицировать пулинговый слой
    layer_name = "block{:d}_pool".format(i+1)
    # построить функцию потерь, максимизирующую среднюю активацию в слое
    layer_output = layer_dict[layer_name].output
    loss = K.mean(layer_output)
    # вычислить градиент изображения по потере и нормировать
    grads = K.gradients(loss, dream)[0]
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
    # определить функцию, возвращающую потерю и градиент данного входного изображения
    f = K.function([dream], [loss, grads])
    img_value = p_img.copy()
    fig, axes = plt.subplots(1, num_iters_per_layer, figsize=(20, 10))
    for it in range(num_iters_per_layer):
        loss_value, grads_value = f([img_value])
        img_value += grads_value * step
        axes[it].imshow(deprocess(img_value))
    plt.show()
```

Ниже показаны результирующие изображения:

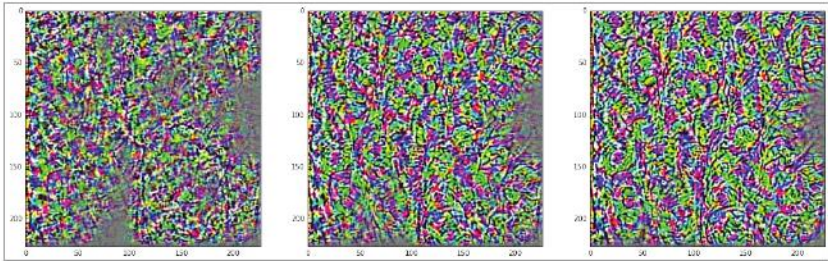




Мы можем повторить этот эксперимент с шумом на входе и вычислить потерю на одном фильтре вместо усреднения по всем фильтрам. Выбранный нами фильтр относится к изображению из набора ImageNet с меткой «африканский слон» (24). Таким образом, в предыдущем коде мы вместо вычисления среднего по всем фильтрам считаем потерей выход фильтра, представляющего класс африканского слона:

```
loss = layer_output[:, :, :, 24]
```

На выходе слоя `block4_pool` получается нечто, напоминающее повторяющиеся изображения слоньего хобота:



Пример – перенос стиля

Обобщение технологии Deep Dream описано в работе L. A. Gatys, A. S. Ecker, M. Bethge «Image Style Transfer Using Convolutional Neural Networks», Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, где показано, что нейронная сеть типа VGG-16 обучается одновременно признакам содержимого и стиля и что тем и другим можно манипулировать независимо. Это значит, что изображение объекта (содержимое) можно подстроить под стиль предъявленной картины.

Как обычно, начинаем с импорта библиотек:

```
from keras.applications import vgg16
from keras import backend as K
from scipy.misc import imresize
import matplotlib.pyplot as plt
import numpy as np
import os
```

Для примера нарисуем наше изображение котенка в стиле репродукции картины Клода Моне «Японский мостик» (<https://goo.gl/0VXC39>):

```

DATA_DIR = "../data"
CONTENT_IMAGE_FILE = os.path.join(DATA_DIR, "cat.jpg")
STYLE_IMAGE_FILE = os.path.join(DATA_DIR, "JapaneseBridgeMonetCopy.jpg")
RESIZED_WH = 400

content_img_value = imresize(plt.imread(CONTENT_IMAGE_FILE), (RESIZED_WH,
    RESIZED_WH))
style_img_value = imresize(plt.imread(STYLE_IMAGE_FILE), (RESIZED_WH,
    RESIZED_WH))

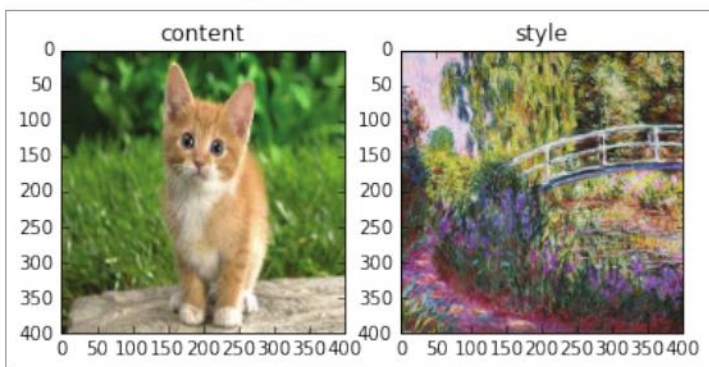
plt.subplot(121)
plt.title("content")
plt.imshow(content_img_value)

plt.subplot(122)
plt.title("style")
plt.imshow(style_img_value)

plt.show()

```

Вот что выводит эта программа:



Как и раньше, напомним функции для преобразования изображения в нужный СНС четырехмерный тензор и обратно:

```

def preprocess(img):
    img4d = img.copy()
    img4d = img4d.astype("float64")
    if K.image_dim_ordering() == "th":
        # (H, W, C) -> (C, H, W)
        img4d = img4d.transpose((2, 0, 1))
    img4d = np.expand_dims(img4d, axis=0)
    img4d = vgg16.preprocess_input(img4d)
    return img4d

def deprocess(img4d):

```

```

img = img4d.copy()
if K.image_dim_ordering() == "th":
    # (B, C, H, W)
    img = img.reshape((img4d.shape[1], img4d.shape[2], img4d.shape[3]))
    # (C, H, W) -> (H, W, C)
    img = img.transpose((1, 2, 0))
else:
    # (B, H, W, C)
    img = img.reshape((img4d.shape[1], img4d.shape[2], img4d.shape[3]))
img[:, :, 0] += 103.939
img[:, :, 1] += 116.779
img[:, :, 2] += 123.68
# BGR -> RGB
img = img[:, :, ::-1]
img = np.clip(img, 0, 255).astype("uint8")
return img

```

Объявляем тензоры для хранения обоих изображений, содержимого и стиля, а также тензор для результирующего изображения. Затем конкатенируем содержимое и стиль в единый входной тензор, который подается на вход предобученной сети VGG-16:

```

content_img = K.variable(preprocess(content_img_value))
style_img = K.variable(preprocess(style_img_value))
if K.image_dim_ordering() == "th":
    comb_img = K.placeholder((1, 3, RESIZED_WH, RESIZED_WH))
else:
    comb_img = K.placeholder((1, RESIZED_WH, RESIZED_WH, 3))
# конкатенировать изображения в единый входной тензор
input_tensor = K.concatenate([content_img, style_img, comb_img], axis=0)

```

Создаем объект сети VGG-16, предобученной на наборе данных ImageNet, с удаленными полносвязными слоями:

```

model = vgg16.VGG16(input_tensor=input_tensor, weights="imagenet",
    include_top=False)

```

Как и раньше, строим словарь, сопоставляющий имя выходу слоя обученной сети VGG-16:

```

layer_dict = {layer.name : layer.output for layer in model.layers}

```

Далее вычисляются потери `content_loss`, `style_loss` и `variational_loss`. И наконец, мы определяем общую потерю как линейную комбинацию этих трех потерь:

```

def content_loss(content, comb):
    return K.sum(K.square(comb - content))

def gram_matrix(x):
    if K.image_dim_ordering() == "th":

```

```

    features = K.batch_flatten(x)
else:
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
gram = K.dot(features, K.transpose(features))
return gram

def style_loss_per_layer(style, comb):
    S = gram_matrix(style)
    C = gram_matrix(comb)
    channels = 3
    size = RESIZED_WH * RESIZED_WH
    return K.sum(K.square(S - C)) / (4 * (channels ** 2) * (size ** 2))

def style_loss():
    stl_loss = 0.0
    for i in range(NUM_LAYERS):
        layer_name = "block{:d}_conv1".format(i+1)
        layer_features = layer_dict[layer_name]
        style_features = layer_features[1, :, :, :]
        comb_features = layer_features[2, :, :, :]
        stl_loss += style_loss_per_layer(style_features, comb_features)
    return stl_loss / NUM_LAYERS

def variation_loss(comb):
    if K.image_dim_ordering() == "th":
        dx = K.square(comb[:, :, :RESIZED_WH-1, :RESIZED_WH-1] -
                       comb[:, :, 1:, :RESIZED_WH-1])
        dy = K.square(comb[:, :, :RESIZED_WH-1, :RESIZED_WH-1] -
                       comb[:, :, :RESIZED_WH-1, 1:])
    else:
        dx = K.square(comb[:, :RESIZED_WH-1, :RESIZED_WH-1, :] -
                       comb[:, 1:, :RESIZED_WH-1, :])
        dy = K.square(comb[:, :RESIZED_WH-1, :RESIZED_WH-1, :] -
                       comb[:, :RESIZED_WH-1, 1:, :])
    return K.sum(K.pow(dx + dy, 1.25))

CONTENT_WEIGHT = 0.1
STYLE_WEIGHT = 5.0
VAR_WEIGHT = 0.01
NUM_LAYERS = 5

c_loss = content_loss(content_img, comb_img)
s_loss = style_loss()
v_loss = variation_loss(comb_img)
loss = (CONTENT_WEIGHT * c_loss) + (STYLE_WEIGHT * s_loss) + (VAR_WEIGHT * v_loss)

```


Здесь `content_loss` – квадратный корень из суммы квадратов разностей координат (**L2-расстояние**) признаков, выделенных из изображения-содержимого и результирующей комбинации изображений. Благодаря минимизации этой потери стилизованное изображение остается близким к оригинальному.

Потеря `style_loss` определяется как L2-расстояние между матрицами Грамма изображения-стиля и результирующей комбинации изображений. Матрицей Грамма матрицы M называется произведение $M^T * M$, где M^T обозначает транспонированную матрицу. Такая потеря измеряет частоту совместной встречаемости признаков в изображении-стилей и комбинированном изображении. Это, кстати, означает, что матрицы `content` и `style` должны быть квадратными.

Полная вариационная потеря `variation_loss` измеряет разность между соседними пикселями. В результате ее минимизации соседние пиксели будут похожи, так что окончательное изображение окажется плавным, а не рваным.

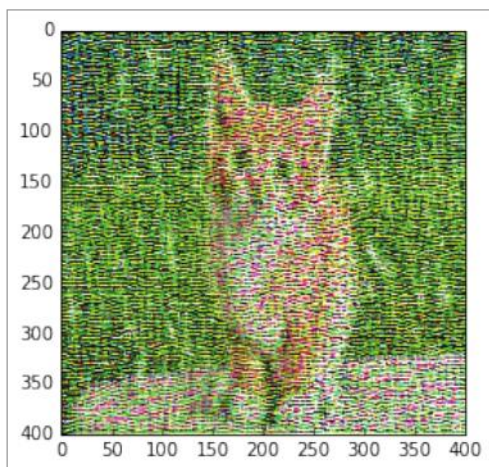
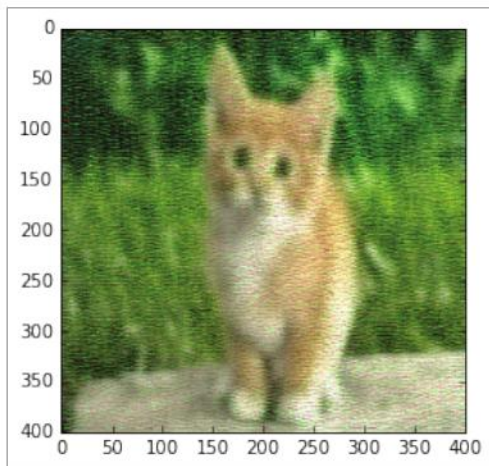
Вычисляем градиент и функцию потерь и выполняем пять итераций работы сети в обратном направлении:

```
grads = K.gradients(loss, comb_img)[0]
f = K.function([comb_img], [loss, grads])

NUM_ITERATIONS = 5
LEARNING_RATE = 0.001

content_img4d = preprocess(content_img_value)
for i in range(NUM_ITERATIONS):
    print("Epoch {:d}/{:d}".format(i+1, NUM_ITERATIONS))
    loss_value, grads_value = f([content_img4d])
    content_img4d += grads_value * LEARNING_RATE
    plt.imshow(deprocess(content_img4d))
    plt.show()
```

Ниже показан результат последних двух итераций. Как видим, изображение приобрело черты размытости, характерные для импрессионистов, а в последнем случае даже скопирована текстура холста.



Резюме

В этой главе мы рассмотрели несколько сетей глубокого обучения, не вошедших в предшествующие главы. Мы начали с краткого обзора функционального API Keras, позволяющего строить не только последовательные, но и более сложные сети. Затем мы обратились к регрессионным сетям, которые делают предсказания в непрерывном пространстве и тем самым позволяют решить широкий ряд новых задач. Однако регрессионная сеть на самом деле пред-

ставляет собой очень простую модификацию стандартной сети классификации. Далее мы занялись автокодировщиками – классом сетей, допускающих обучение без учителя, которые открывают доступ к огромному массиву непомеченных данных, находящихся в распоряжении каждого из нас. Мы также научились составлять из уже известных сетей более крупные, как в гигантском конструкторе Lego. От построения больших сетей из более мелких мы перешли к вопросу о создании пользовательских слоев с применением переходного API Keras. И напоследок рассмотрели порождающие модели, способные имитировать данные, на которых обучались, и продемонстрировали несколько новаторских применений таких моделей.

Темой следующей главы будет обучение с подкреплением. Мы исследуем эти идеи на примере построения и обучения сети, которая будет играть в простую компьютерную игру.

Глава 8

Искусственный интеллект играет в игры

В предыдущих главах мы рассматривали методы обучения с учителем – классификацию и регрессию – и без учителя – порождающие состязательные сети, автокодировщики и порождающие модели. При обучении с учителем сеть обучалась на помеченных данных и должна была предсказывать выход, когда ей предъявлялись новые данные. При обучении без учителя сети предъявлялись входные данные, а она должна была выявить в них структуру и применить полученные знания к новым данным.

В этой главе мы будем говорить о глубоком обучении с подкреплением, т. е. применении глубоких нейронных сетей к обучению с подкреплением. Обучение с подкреплением берет начало в психологии поведения. Агент обучается, получая вознаграждение за правильное поведение и наказание за неправильное. В контексте глубокого обучения с подкреплением сети предъявляются входные данные, и в зависимости от того, какой ответ она дает – правильный или неправильный – применяются положительные или отрицательные стимулы. Следовательно, в обучении с подкреплением мы имеем разреженные и отложенные во времени метки. На протяжении большого числа итераций сеть обучается давать правильный результат.

Пионером глубокого обучения с подкреплением стала небольшая британская компания DeepMind, в 2013 году опубликовавшая статью (V. Mnih «Playing Atari with Deep Reinforcement Learning», arXiv:1312.5602, 2013), в которой описывалось, как можно научить сверточную нейронную сеть играть в видеоигры с компьютером Atari 2600, показывая ей экранные пиксели и вознаграждая, когда

счет увеличивается в ее пользу. Одна и та же архитектура была использована для обучения семи разных игр, в шести из которых модель превзошла все предшествующие подходы, а в трех – результат человека.

В отличие от ранее рассмотренных стратегий обучений, когда сеть обучается только в одной предметной области, обучение с подкреплением, похоже, является универсальным алгоритмом обучения, который применим в разных ситуациях; быть может, это даже первый шаг к настоящему искусственному интеллекту. Впоследствии компанию DeepMind купила Google, и ее коллектив всегда находился на переднем крае исследований по ИИ. В следующей статье (V. Mnih «Human-Level Control through Deep Reinforcement Learning», Nature 518.7540, 2015:529–533), опубликованной в престижном журнале Nature в 2015 году, описывалось применение той же модели к 49 играм.

В этой главе мы расскажем о теоретических основаниях глубокого обучения с подкреплением. А затем применим эти идеи к построению с помощью Keras сети, которая будет обучаться игре в поимку мяча. Мы также дадим краткий обзор возможных путей улучшения этой сети и некоторых многообещающих направлений исследований в этой области.

Итак, в этой главе обсуждаются следующие концепции, касающиеся обучения с учителем:

- Q-обучение;
- исследование и использование;
- воспроизведение опыта.

Обучение с подкреплением

Наша цель – построить нейронную сеть для игры в поимку мяча. В начале игры из случайной точки сверху экрана падает мячик. Задача игрока – подставить ракетку в нижней части экрана, пользуясь клавишами со стрелками влево и вправо, и не дать мячику упасть. Как видите, все очень просто. Состояние игры в любой момент времени описывается координатами мяча и ракетки (x , y). В большинстве аркадных игр подвижных элементов гораздо больше, поэтому общее решение – считать текущим состоянием все изображение на экране. На рисунке ниже показаны четыре последовательных снимка экрана игры.



Внимательные читатели, вероятно, заметили, что эту задачу можно было бы сформулировать как задачу классификации, в которой на вход сети подаются изображения на экране игры, а выходом является одно из трех действий: двигаться влево, оставаться на месте, двигаться вправо. Но тогда мы должны были бы предъявить сети обучающие примеры, возможно, из партий, сыгранных людьми. Другой, более простой подход состоит в том, чтобы построить сеть и заставить ее играть снова и снова, организовав обратную связь, сообщающую, удалось поймать мяч или нет. Этот подход интуитивно ближе к тому, как обучаются люди и животные.

Самый распространенный способ представления таких задач – **марковский процесс принятия решений**. Игра – это окружающая среда, в которой агент пытается обучиться. Состояние среды в момент t обозначим s_t (оно включает положение мяча и ракетки). Агент может выполнять некоторые действия (двигать ракетку влево или вправо). Иногда эти действия влекут за собой вознаграждение r_t , положительное или отрицательное (например, увеличение или уменьшение счета). Действия изменяют среду и могут привести к новому состоянию s_{t+1} , в котором агент может выполнить новое действие a_{t+1} и т. д. Множество состояний, действий и вознаграждений, а также правила перехода из одного состояния в другое и составляют марковский процесс принятия решений. Одна игра является эпизодом этого процесса и представляется конечной последовательностью состояний, действий и вознаграждений:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots, s_{n-1}, a_{n-1}, r_n, s_n$$

Поскольку это марковский процесс, вероятность состояния s_{t+1} зависит только от текущего состояния s_t и действия a_t .

Максимизация будущих вознаграждений

Цель агента – максимизировать полное вознаграждение в каждой игре. Полное вознаграждение можно представить в виде:

$$R = \sum_{i=1}^n r_i$$

Для максимизации полного вознаграждения агент должен стремиться максимизировать полное вознаграждение, начиная с любого момента t в игре. Полное вознаграждение в момент t обозначается R_t и имеет вид:

$$R = \sum_{i=1}^n r_i = r_t + r_{t+1} + \dots + r_n$$

Однако чем дальше мы заглядываем в будущее, тем труднее предсказать величину вознаграждения. Чтобы учесть это, агент должен стремиться максимизировать полное дисконтированное будущее вознаграждение в момент t . Для этого мы уменьшаем вознаграждение на каждом будущем временном шаге в γ раз по сравнению с предыдущим шагом. Если γ равно 0, то сеть вообще не учитывает будущие вознаграждения, а если γ равно 1, то сеть полностью детерминирована. Хорошим является значение γ в районе 0.9. Выписав все члены предыдущего тождества, мы сможем рекурсивно выразить полное дисконтированное будущее вознаграждение в виде суммы текущего вознаграждения и полного дисконтированного будущего вознаграждения на следующем временном шаге:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{n-t} r_n \\ &= r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \dots)) \\ &= r_t + \gamma R_{t+1} \end{aligned}$$

Q-обучение

В глубоком обучении с подкреплением используется техника обучения без формирования модели окружающей среды, называемая **Q-обучением**. Q-обучение можно использовать для нахождения оптимального действия в любом заданном состоянии конечного марковского процесса принятия решений. Агент пытается максимизировать значение Q-функции, представляющей максимальное дисконтированное будущее вознаграждение при выполнении действия a в состоянии s :

$$Q(s_t, a_t) = \max(R_{t+1})$$

Если Q -функция известна, то оптимальным действием a в состоянии s будет то, для которого значение Q максимально. Следовательно, мы можем определить политику $\Pi(s)$, дающую оптимальное действие в любом состоянии:

$$\Pi(s) = \operatorname{argmax}_a Q(s, a)$$

Мы можем определить Q -функцию для переходной точки (s_t, a_t, r_t, s_{t+1}) в терминах Q -функции в следующей точке $(s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2})$ аналогично тому, как мы поступили с дисконтированным будущим вознаграждением. В результате получается **уравнение Беллмана**:

$$Q(s_t, a_t) = r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

Q -функцию можно аппроксимировать с помощью уравнения Беллмана. Q -функцию можно рассматривать как таблицу соответствия (она называется **Q -таблицей**), в которой состояния (обозначаемые буквой s) представляются строками, действия (обозначаемые буквой a) – столбцами, а элемент на пересечении s -ой строки и a -го столбца $Q(s, a)$ равен вознаграждению, получаемому в случае, когда в состоянии s выполнено действие a . Наилучшее действие в данном состоянии – то, для которого вознаграждение максимально. В начале Q -таблица инициализируется случайными значениями, затем выполняются случайные действия и Q -таблица обновляется в соответствии с наблюдаемыми вознаграждениями по следующему алгоритму:

```
инициализировать  $Q$ -таблицу  $Q$ 
наблюдать начальное состояние  $s$ 
repeat
    выбрать и выполнить действие  $a$ 
    наблюдать вознаграждение  $r$  и перейти в новое состояние  $s'$ 
     $Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$ 
     $s = s'$ 
until игра закончена
```

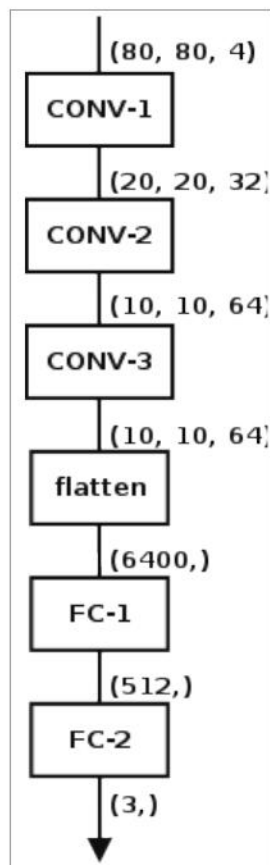
Нетрудно понять, что этот алгоритм по существу является стохастическим градиентным спуском по уравнению Беллмана с обратным распространением вознаграждения через пространство состояний (эпизод) и усреднением по многим испытаниям (периодам). В нем α – скорость обучения, определяющая, какую часть разности между предыдущим значением Q и новым дисконтированным максимальным значением Q следует включить.

Глубокая Q-сеть как Q-функция

Мы знаем, что Q-функция будет нейронной сетью, но какого вида? В случае нашей простой игры каждое состояние представляется четырьмя последовательными черно-белыми изображениями экрана размера $(80, 80)$, так что общее число возможных состояний равно $2^{80 \times 80 \times 4}$. К счастью, многие из них – невозможные или крайне маловероятные комбинации пикселей. Поскольку сверточная нейронная сеть обладает локальной связностью (т. е. каждый нейрон связан только с локальной областью своих входов), она избегает таких невозможных и невероятных комбинаций пикселей. Кроме того, в общем случае нейронные сети хорошо выделяют признаки из таких структурированных данных, как изображения. Поэтому СНС будет очень эффективным инструментом моделирования Q-функции.

В цитированной выше работе компании DeepMind описана сеть с тремя сверточными слоями, за которыми следуют два полносвязных слоя. В отличие от традиционных СНС, применяемых для классификации или распознавания изображений, пулинговых слоев здесь нет, поскольку пулинговый слой делает сеть менее чувствительной к положению конкретных объектов в изображении. Но в случае игры эта информация, скорее всего, необходима для вычисления вознаграждения, поэтому отбрасывать ее нельзя.

На следующем рисунке показана структура глубокой Q-сети в нашем примере. Она отличается от сети в оригинальной статье DeepMind только формой входного и выходного слоя. Форма каждого элемента входных данных $(80, 80, 4)$: четыре последовательных снимка черно-белого экрана игровой консоли размера 80×80 пикселей. Выходной слой имеет форму (3) , что соответствует трем возможным действиям агента (влево, на месте, вправо).



Поскольку выход состоит из трех Q -значений, это задача регрессии, и мы можем решить ее путем минимизации среднеквадратической ошибки: квадрата разности между текущим значением $Q(s, a)$ и предсказанным значением, которое вычисляется в терминах суммы вознаграждения и дисконтированного будущего Q -значения $Q(s', a')$. Текущее значение уже известно в начале итерации, а будущее вычисляется на основе вознаграждения, полученного от окружающей среды:

$$L = \frac{1}{2} [r + \gamma \max Q(s', a') - Q(s, a)]^2$$

Баланс между исследованием и использованием

Глубокое обучение с подкреплением – пример онлайн-обучения, когда шаги обучения и предсказания чередуются. В отличие от пакетного обучения, когда наилучший предиктор генерируется путем обучения на всем массиве обучающих данных, предиктор, обучаемый в процессе онлайн-обучения, постоянно улучшается по мере поступления новых данных.

Таким образом, на начальных периодах обучения глубокая Q -сеть дает случайные предсказания, что ведет к плохому качеству Q -обучения. Чтобы справиться с этой проблемой, мы можем применить простой метод исследования, например, ε -жадную стратегию. В этом случае агент выбирает действие, предложенное сетью, с вероятностью $1 - \varepsilon$ или случайное действие с равномерным распределением. Такая стратегия называется исследованием-использованием.

По мере увеличения числа периодов Q -функция сходится и начинает возвращать более осмысленные Q -значения. Чтобы учесть это, значение ε можно постепенно уменьшать: по мере того как предсказания сети становятся более согласованными, агент чаще предпочитает предлагаемое сетью действие случайному. В сети DeepMind величина ε со временем убывает от 1 до 0.1, а в нашем примере – от 0.1 до 0.001.

Следовательно, ε -жадный алгоритм гарантирует, что в начале работы система ищет баланс между ненадежными предсказаниями Q -сети и случайными перемещениями, стремясь исследовать

пространство состояний, а затем, по мере улучшения предсказаний, переходит в режим менее агрессивного исследования (отдавая предпочтение использованию).

Воспроизведение опыта

Исходя из уравнений, представляющих значение Q для пары (s_t, a_t) в терминах текущего вознаграждения r_t и дисконтированного максимального значения Q на следующем временном шаге (s_{t+1}, a_{t+1}) , было бы логично обучать сеть предсказывать наилучшее следующее состояние s' по текущим значениям (s, a, r) . Но, как выясняется, такая стратегия заводит сеть в локальный минимум. Объясняется это тем, что последовательные обучающие примеры как правило очень похожи.

Чтобы противостоять этому, мы сохраняем все предыдущие ходы (s, a, r, s') в большой очереди фиксированного размера, которая называется **накопленной памятью** (replay memory). В этой памяти представлен опыт сети. В процессе обучения сети мы генерируем случайные пакеты из накопленной памяти, а не берем последний пакет транзакций. Поскольку пакеты содержат прошлые кортежи (s, a, r, s') , взятые в случайном порядке, сеть обучается лучше и не застревает в локальном минимуме.

Опыт можно накапливать и из игр, сыгранных человеком, – вместо или в дополнение к прошлым ходам, сделанным сетью. Еще один подход к накоплению опыта – в начале дать сети поработать в режиме *наблюдения*, когда она генерирует совершенно случайные действия ($\epsilon = 1$) и запоминает полученные от игры вознаграждение и следующее состояние в накопленной памяти.

Пример – глубокая Q-сеть для поимки мяча

Цель нашей игры – поймать мячик, падающий из случайной точки сверху экрана, ракеткой, находящейся внизу экрана, которую мы можем двигать клавишами вправо или влево. Игрок выигрывает, если ему удалось поймать мячик на ракетку, и проигрывает, если мячик упал раньше. Достоинством игры является простота – как для понимания, так и для реализации. Она устроена по образцу игры, описанной в блоге Эдера Сантаны (см. Eder Santana «Keras Plays Catch, a Single File Reinforcement Learning Example, 2017»), посвященном глубокому обучению с подкреплением. Саму игру мы написали с помощью свободной библиотеки Pygame с открытым ис-

ходным кодом (<https://www.pygame.org/news>). Желающие попробовать найдут код игры в файле `game.py` в составе кода к этой главе.



Установка Pygame

Pygame написана на Python, имеются версии для Linux (различных дистрибутивов), macOS, Windows, а также для некоторых мобильных операционных систем, в т. ч. Android и Nokia. Полный список поддерживаемых платформ опубликован на странице <http://www.pygame.org/download.shtml>. Откомпилированные версии имеются для 32- и 64-разрядных версий Linux и Windows и 64-разрядной версии macOS. На этих платформах установить Pygame можно командой `pip install pygame`.

Если готовой версии для вашей платформы не существует, то ее можно собрать из исходных текстов, следуя инструкциям на странице <http://www.pygame.org/wiki/GettingStarted>.

Для пользователей Anaconda имеются откомпилированные версии Pygame в репозитории conda-forge:

```
conda install binstar
conda install anaconda-client
conda install -c https://conda.binstar.org/tlatorre
pygame # Linux
conda install -c https://conda.binstar.org/quasiben
pygame # Mac
```

Чтобы обучить нейронную сеть, понадобится внести некоторые изменения в код игры, чтобы в нее могла играть сеть вместо человека. Мы хотим, чтобы сеть могла взаимодействовать с игрой через API, а не путем нажатия на клавиши. Ниже приведен код обернутой таким образом игры:

Как обычно, начинаем с импорта:

```
from __future__ import division, print_function
import collections
import numpy as np
import pygame
import random
import os
```

Определим свой класс. В конструкторе можно задать режим *без монитора*, тогда игра не будет отображать экран Pygame. Это полезно, если мы запускаем программу на машине с GPU в облаке, где есть доступ только к текстовым терминалам. Эту строку можно

закомментировать, если обернутая игра запускается локально на графическом терминале. Затем мы вызываем метод `pygame.init()` для инициализации компонентов Pygame. И напоследок инициализируем ряд констант:

```
class MyWrappedGame(object):

    def __init__(self):
        # запустить pygame в режиме без монитора
        os.environ["SDL_VIDEODRIVER"] = "dummy"

    pygame.init()

    # инициализировать константы
    self.COLOR_WHITE = (255, 255, 255)
    self.COLOR_BLACK = (0, 0, 0)
    self.GAME_WIDTH = 400
    self.GAME_HEIGHT = 400
    self.BALL_WIDTH = 20
    self.BALL_HEIGHT = 20
    self.PADDLE_WIDTH = 50
    self.PADDLE_HEIGHT = 10
    self.GAME_FLOOR = 350
    self.GAME_CEILING = 10
    self.BALL_VELOCITY = 10
    self.PADDLE_VELOCITY = 20
    self.FONT_SIZE = 30
    self.MAX_TRIES_PER_GAME = 1
    self.CUSTOM_EVENT = pygame.USEREVENT + 1
    self.font = pygame.font.SysFont("Comic Sans MS", self.FONT_SIZE)
```

Метод `reset()` содержит операции, которые должны выполняться в начале каждой игры: очистка очереди состояний, установка положения мяча и ракетки, инициализация счета и т. д.

```
def reset(self):
    self.frames = collections.deque(maxlen=4)
    self.game_over = False
    # задать начальные положения
    self.paddle_x = self.GAME_WIDTH // 2
    self.game_score = 0
    self.reward = 0
    self.ball_x = random.randint(0, self.GAME_WIDTH)
    self.ball_y = self.GAME_CEILING
    self.num_tries = 0

    # инициализировать дисплей и часы
    self.screen = pygame.display.set_mode((self.GAME_WIDTH, self.GAME_HEIGHT))
    self.clock = pygame.time.Clock()
```

В оригинальной игре имеется очередь событий Pygame, в которую помещаются события клавиш, нажимаемых игроком, а также внутренние события, генерируемые компонентами Pygame. Стержнем кода игры является **цикл обработки событий**, где происходит чтение и обработка событий из очереди.

В обернутой версии мы переместили цикл обработки событий в код вызывающей стороны. Метод `step()` описывает, что происходит на одной итерации цикла. Он принимает значение 0, 1 или 2, представляющее операцию (влево, на месте, вправо), а затем устанавливает переменные, управляющие положением мяча и ракетки на этом временном шаге. Переменная `PADDLE_VELOCITY` задает скорость движения ракетки, т. е. на сколько пикселей она сдвигается при нажатии клавиши со стрелкой. Если мячик оказывается на одном уровне с ракеткой, метод проверяет, есть ли столкновение. Если да, то мячик пойман, и, значит, игрок (нейронная сеть) выиграл, иначе игрок проиграл. Далее метод перерисовывает экран и добавляет его в конце очереди `deque`, рассчитанной на последние четыре кадра игры. И наконец, он возвращает состояние (определяемое последними четырьмя кадрами), вознаграждение за текущее действие и флаг завершения игры.

```
def step(self, action):
    pygame.event.pump()

    if action == 0: # сдвинуть ракетку влево
        self.paddle_x -= self.PADDLE_VELOCITY
        if self.paddle_x < 0:
            # отскок от стенки, движение вправо
            self.paddle_x = self.PADDLE_VELOCITY
    elif action == 2: # сдвинуть ракетку вправо
        self.paddle_x += self.PADDLE_VELOCITY
        if self.paddle_x > self.GAME_WIDTH - self.PADDLE_WIDTH:
            # отскок от стенки, движение влево
            self.paddle_x = self.GAME_WIDTH - self.PADDLE_WIDTH - self.PADDLE_VELOCITY
    else: # не двигать ракетку
        pass

    self.screen.fill(self.COLOR_BLACK)
    score_text = self.font.render("Score: {:d}/{:d}, Ball: {:d}"
                                   .format(self.game_score, self.MAX_TRIES_PER_GAME,
                                           self.num_tries), True, self.COLOR_WHITE)
    self.screen.blit(score_text,
                     ((self.GAME_WIDTH - score_text.get_width()) // 2,
                      (self.GAME_FLOOR + self.FONT_SIZE // 2)))

    # обновить положение мяча
```

```

self.ball_y += self.BALL_VELOCITY
ball = pygame.draw.rect(self.screen, self.COLOR_WHITE,
    pygame.Rect(self.ball_x, self.ball_y, self.BALL_WIDTH, self.BALL_HEIGHT))
# обновить положение ракетки
paddle = pygame.draw.rect(self.screen, self.COLOR_WHITE,
    pygame.Rect(self.paddle_x, self.GAME_FLOOR,
        self.PADDLE_WIDTH, self.PADDLE_HEIGHT))
# проверить, столкнулся ли мяч с ракеткой
self.reward = 0
if self.ball_y >= self.GAME_FLOOR - self.BALL_WIDTH // 2:
    if ball.colliderect(paddle):
        self.reward = 1
    else:
        self.reward = -1

self.game_score += self.reward
self.ball_x = random.randint(0, self.GAME_WIDTH)
self.ball_y = self.GAME_CEILING
self.num_tries += 1

pygame.display.flip()

# сохранить последние 4 кадра
self.frames.append(pygame.surfarray.array2d(self.screen))

if self.num_tries >= self.MAX_TRIES_PER_GAME:
    self.game_over = True

self.clock.tick(30)
return np.array(list(self.frames)), self.reward, self.game_over

```

Теперь рассмотрим код обучения сети.

Как обычно, сначала импортируются библиотеки. Помимо сторонних компонентов из библиотек Keras и SciPy мы импортируем описанный выше класс `wrapped_game`:

```

from __future__ import division, print_function
from keras.models import Sequential
from keras.layers.core import Activation, Dense, Flatten
from keras.layers.convolutional import Conv2D
from keras.optimizers import Adam
from scipy.misc import imresize
import collections
import numpy as np
import os

import wrapped_game

```

Определим две вспомогательные функции. Первая преобразует набор из четырех изображений в форму, ожидаемую сетью. На

вход подаются четыре изображения размера 800×800 , так что вход имеет форму (4, 800, 800). Однако сеть ожидает получить четырехмерный тензор формы (*размер пакета*, 80, 80, 4). В самом начале игры четырех кадров еще нет, поэтому мы просто повторяем первый кадр четыре раза. Выходной тензор, возвращаемый функцией, имеет форму (80, 80, 4).

Вторая функция `get_next_batch()` выбирает `batch_size` кортежей состояния из очереди накопленного опыта и получает вознаграждение и предсказанное следующее состояние от нейронной сети. Затем она вычисляет и возвращает значение Q-функции на следующем временном шаге:

```
def preprocess_images(images):
    if images.shape[0] < 4:
        # одно изображение
        x_t = images[0]
        x_t = imresize(x_t, (80, 80))
        x_t = x_t.astype("float")
        x_t /= 255.0
        s_t = np.stack((x_t, x_t, x_t, x_t), axis=2)
    else:
        # 4 изображения
        xt_list = []
        for i in range(images.shape[0]):
            x_t = imresize(images[i], (80, 80))
            x_t = x_t.astype("float")
            x_t /= 255.0
            xt_list.append(x_t)
        s_t = np.stack((xt_list[0], xt_list[1], xt_list[2], xt_list[3]), axis=2)
    s_t = np.expand_dims(s_t, axis=0)
    return s_t

def get_next_batch(experience, model, num_actions, gamma, batch_size):
    batch_indices = np.random.randint(low=0, high=len(experience), size=batch_size)
    batch = [experience[i] for i in batch_indices]
    X = np.zeros((batch_size, 80, 80, 4))
    Y = np.zeros((batch_size, num_actions))
    for i in range(len(batch)):
        s_t, a_t, r_t, s_tp1, game_over = batch[i]
        X[i] = s_t
        Y[i] = model.predict(s_t)[0]
        Q_sa = np.max(model.predict(s_tp1)[0])
        if game_over:
            Y[i, a_t] = r_t
        else:
            Y[i, a_t] = r_t + gamma * Q_sa
    return X, Y
```


Теперь определим сеть, моделирующую Q-функцию для игры. Наша сеть будет очень похожа на предложенную в статье DeepMind. Отличаются они только размерами входа и выхода. У нас вход имеет форму (80, 80, 4), а у них (84, 84, 4), у нас выход имеет форму (3), что соответствует трем действиям, для которых нужно вычислить значение Q-функции, а у них (18) – по числу действий в игре Atari.

Сеть состоит из трех сверточных и двух полносвязных (плотных) слоев. Во всех слоях, кроме последнего, используется функция активации ReLU. Поскольку мы предсказываем значения Q-функции, это регрессионная сеть, и в последнем слое нет блока активации.

```
# построить модель
model = Sequential()
model.add(Conv2D(32, kernel_size=8, strides=4,
                 kernel_initializer="normal",
                 padding="same",
                 input_shape=(80, 80, 4)))
model.add(Activation("relu"))
model.add(Conv2D(64, kernel_size=4, strides=2,
                 kernel_initializer="normal",
                 padding="same"))
model.add(Activation("relu"))
model.add(Conv2D(64, kernel_size=3, strides=1,
                 kernel_initializer="normal",
                 padding="same"))
model.add(Activation("relu"))
model.add(Flatten())
model.add(Dense(512, kernel_initializer="normal"))
model.add(Activation("relu"))
model.add(Dense(3, kernel_initializer="normal"))
```

Как уже было сказано, в качестве функции потерь используется квадрат разности между текущим значением $Q(s, a)$ и вычисленным значением – суммой вознаграждения и дисконтированного будущего значения $Q(s', a')$. Поэтому среднеквадратическая ошибка (MSE) вполне подойдет. В качестве оптимизатора выбираем Adam и задаем для него низкую скорость обучения:

```
model.compile(optimizer=Adam(lr=1e-6), loss="mse")
```

Зададим некоторые константы – гиперпараметры обучения. `NUM_ACTIONS` – количество действий, которые сеть может посылать игре. В нашем случае это действия 0, 1 и 2 (влево, на месте, вправо). `GAMMA` – коэффициент дисконтирования будущих вознаграждений. `INITIAL_EPSILON` и `FINAL_EPSILON` – начальное и конечное значение параметра ϵ -жадного исследования. `MEMORY_SIZE` – размер

очереди накопленного опыта. `NUM_EPOCHS_OBSERVE` – количество периодов, в течение которых сети разрешается исследовать игру, посылая случайные действия и наблюдая за вознаграждением. `NUM_EPOCHS_TRAIN` – количество периодов, в течение которых происходит онлайн-обучение сети. Период соответствует одной игре, или эпизоду. Общее число игр, сыгранных в процессе обучения, равно сумме `NUM_EPOCHS_OBSERVE` и `NUM_EPOCHS_TRAIN`. `BATCH_SIZE` – размер мини-пакета, используемого в ходе обучения.

```
# инициализировать параметры
DATA_DIR = "../data"
NUM_ACTIONS = 3 # число допустимых действий (влево, на месте, вправо)
GAMMA = 0.99 # коэффициент дисконтирования прошлых наблюдений
INITIAL_EPSILON = 0.1 # начальное значение эпсилон
FINAL_EPSILON = 0.0001 # конечное значение эпсилон
MEMORY_SIZE = 50000 # сколько предыдущих ходов запоминать
NUM_EPOCHS_OBSERVE = 100
NUM_EPOCHS_TRAIN = 2000

BATCH_SIZE = 32
NUM_EPOCHS = NUM_EPOCHS_OBSERVE + NUM_EPOCHS_TRAIN
```

Создаем объект игры и очередь накопленного опыта. Кроме того, открываем файл журнала и инициализируем некоторые переменные перед началом обучения:

```
game = wrapped_game.MyWrappedGame()
experience = collections.deque(maxlen=MEMORY_SIZE)

fout = open(os.path.join(DATA_DIR, "rl-network-results.tsv"), "wb")
num_games, num_wins = 0, 0
epsilon = INITIAL_EPSILON
```

Далее входим в цикл по числу периодов обучения. Как уже было сказано, период соответствует одной игре, поэтому в начале периода мы сбрасываем состояние игры. Игра начинается срыванием мяча с потолка и заканчивается либо его поимкой, либо падением на пол. Потеря равна квадрату разности между предсказанным и фактическим значением Q -функции для данной игры.

В начале игры мы посылаем фиктивное действие (в нашем случае «на месте») и получаем в ответ кортеж начального состояния игры:

```
for e in range(NUM_EPOCHS):
    game.reset()
    loss = 0.0

    # получить первое состояние
```

```

a_0 = 1 # (0 = left, 1 = stay, 2 = right)
x_t, r_0, game_over = game.step(a_0)
s_t = preprocess_images(x_t)

```

Далее начинается главный цикл игры. Это цикл обработки событий оригинальной игры, который мы перенесли в вызывающую программу. Сохраняем текущее состояние, потому что оно понадобится для запоминания в очереди накопленного опыта, а затем решаем, какое действие послать обернутой игре. Если мы работаем в режиме наблюдения, то просто генерируем случайное число, соответствующее одному из действий, в противном случае применяем ϵ -жадный алгоритм исследования, который либо выбирает случайное действие, либо пользуется нашей нейронной сетью (которую мы попутно обучаем) для предсказания действия:

```

while not game_over:
    s_tm1 = s_t

    # следующее действие
    if e <= NUM_EPOCHS_OBSERVE:
        a_t = np.random.randint(low=0, high=NUM_ACTIONS, size=1)[0]
    else:
        if np.random.rand() <= epsilon:
            a_t = np.random.randint(low=0, high=NUM_ACTIONS, size=1)[0]
        else:
            q = model.predict(s_t)[0]
            a_t = np.argmax(q)

```

Определившись с действием, посылаем его игре, вызывая метод `game.step()`, который возвращает новое состояние, вознаграждение и флаг завершения игры. Если вознаграждение положительно (т. е. мяч был пойман), то увеличиваем счетчик выигрышей и в любом случае сохраняем кортеж (*состояние, действие, вознаграждение, новое состояние, флаг завершения игры*) в очереди накопленного опыта:

```

# применить действие, получить вознаграждение
x_t, r_t, game_over = game.step(a_t)
s_t = preprocess_images(x_t)
# если вознаграждение положительно, увеличить num_wins
if r_t == 1:
    num_wins += 1
# сохранить опыт
experience.append((s_tm1, a_t, r_t, s_t, game_over))

```

Затем выбираем случайный минипакет из очереди накопленного опыта и обучаем сеть. В каждом сеансе обучения вычисляем потерю. Сумма потерь для всех сеансов в каждом периоде составляет потерю в этом периоде:

```
if e > NUM_EPOCHS_OBSERVE:
    # наблюдение закончено, начинается обучение
    # получить следующий пакет
    X, Y = get_next_batch(experience, model, NUM_ACTIONS, GAMMA, BATCH_SIZE)
    loss += model.train_on_batch(X, Y)
```

Пока сеть еще толком не обучилась, она дает плохие предсказания, поэтому имеет смысл продолжить исследование пространства состояний, чтобы уменьшить вероятность застревания в локальном минимуме. Но по мере того, как сеть обучается, мы постепенно уменьшаем величину ϵ , так что модель все чаще предсказывает действия, посылаемые сетью игре:

```
# постепенное уменьшение эпсилон
if epsilon > FINAL_EPSILON:
    epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / NUM_EPOCHS
```

Протокол обучения выводится на консоль и записывается в файл журнала для последующего анализа. После каждых 100 периодов мы сохраняем состояние модели, чтобы его можно было восстановить в случае, если по какой-то причине обучение будет прервано. Также сохраняется окончательная модель, чтобы впоследствии ей можно было воспользоваться для игры.

```
print("Epoch {:04d}/{:d} | Loss {:.5f} | Win Count {:d}"
      .format(e + 1, NUM_EPOCHS, loss, num_wins))
fout.write("{:04d}t{:.5f}t{:d}\n".format(e + 1, loss, num_wins))

if e % 100 == 0:
    model.save(os.path.join(DATA_DIR, "rl-network.h5"), overwrite=True)

fout.close()
model.save(os.path.join(DATA_DIR, "rl-network.h5"), overwrite=True)
```

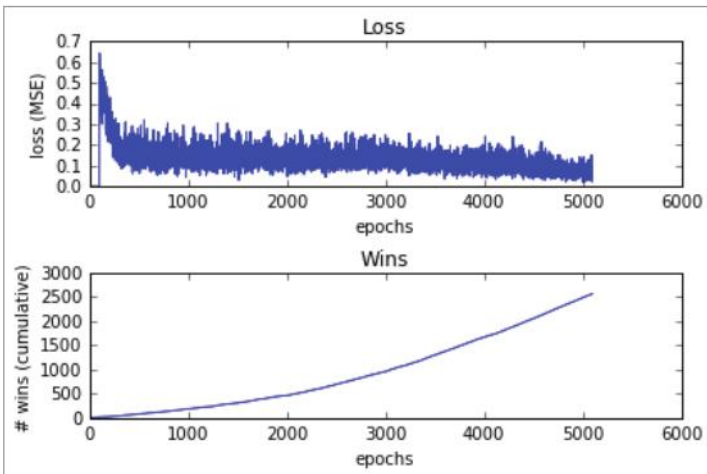
Для обучения модели мы заставили ее наблюдать за 100 играми, а затем сыграть 1000, 2000 и 5000 игр. Ниже показаны последние строки журнала для обучения на 5000 играх. Как видите, к концу обучения сеть стала играть очень неплохо.


```

Epoch 5075/5100 | Loss 0.02603 | Win Count 2548
Epoch 5076/5100 | Loss 0.06248 | Win Count 2549
Epoch 5077/5100 | Loss 0.09836 | Win Count 2550
Epoch 5078/5100 | Loss 0.05955 | Win Count 2551
Epoch 5079/5100 | Loss 0.07357 | Win Count 2552
Epoch 5080/5100 | Loss 0.05425 | Win Count 2553
Epoch 5081/5100 | Loss 0.05961 | Win Count 2553
Epoch 5082/5100 | Loss 0.05737 | Win Count 2553
Epoch 5083/5100 | Loss 0.06699 | Win Count 2554
Epoch 5084/5100 | Loss 0.04265 | Win Count 2555
Epoch 5085/5100 | Loss 0.06579 | Win Count 2556
Epoch 5086/5100 | Loss 0.06825 | Win Count 2557
Epoch 5087/5100 | Loss 0.09329 | Win Count 2557
Epoch 5088/5100 | Loss 0.06124 | Win Count 2558
Epoch 5089/5100 | Loss 0.15128 | Win Count 2559
Epoch 5090/5100 | Loss 0.03769 | Win Count 2560
Epoch 5091/5100 | Loss 0.06348 | Win Count 2560
Epoch 5092/5100 | Loss 0.03817 | Win Count 2561
Epoch 5093/5100 | Loss 0.05225 | Win Count 2562
Epoch 5094/5100 | Loss 0.04986 | Win Count 2563
Epoch 5095/5100 | Loss 0.06316 | Win Count 2564
Epoch 5096/5100 | Loss 0.07558 | Win Count 2564
Epoch 5097/5100 | Loss 0.04027 | Win Count 2565
Epoch 5098/5100 | Loss 0.03801 | Win Count 2566
Epoch 5099/5100 | Loss 0.02446 | Win Count 2567
Epoch 5100/5100 | Loss 0.04321 | Win Count 2568

```

Графики потерь и числа выигрышей говорят о том же. За 5000 периодов потеря уменьшилась с 0.6 до примерно 0.1 и, похоже, могла бы еще уменьшиться, если обучать модель подольше. И кривая числа выигрышей тоже идет вверх и тем быстрее, чем больше периодов прошло.



И наконец, оценим мастерство обученной модели, для чего заставим ее сыграть фиксированное число игр (в нашем случае 100) и посмотрим, сколько раз она выиграет. Ниже приведен соответствующий код. Как всегда, начинаем с импорта:

```
from __future__ import division, print_function
from keras.models import load_model
from keras.optimizers import Adam
from scipy.misc import imresize
import numpy as np
import os
import wrapped_game
```

Загружаем сохраненную модель и компилируем ее. Создаем объект `wrapped_game`:

```
DATA_DIR = "../data"
model = load_model(os.path.join(DATA_DIR, "rl-network.h5"))
model.compile(optimizer=Adam(lr=1e-6), loss="mse")

game = wrapped_game.MyWrappedGame()
```

Затем в цикле организуем 100 игр. В начале каждой игры вызывается ее метод `reset()`, а затем, пока игра не закончится, мы просим модель предсказать действие с наилучшим значением Q-функции. В конце печатаем, сколько игр выиграла модель.

Мы прогнали этот тест для каждой из построенных моделей. Первая, обученная на 1000 игр, выиграла 42 игры из 100, вторая, обученная на 2000 игр, – 74 из 100, а третья, обученная на 5000 игр, – 87 из 100. Это доказывает, что мастерство сети возрастает по мере обучения.

```
num_games, num_wins = 0, 0
for e in range(100):
    game.reset()

    # получить первое состояние
    a_0 = 1 # (0 = left, 1 = stay, 2 = right)
    x_t, r_0, game_over = game.step(a_0)
    s_t = preprocess_images(x_t)

    while not game_over:
        s_tml = s_t
        # следующее действие
        q = model.predict(s_t)[0]
        a_t = np.argmax(q)
        # применить действие, получить вознаграждение
        x_t, r_t, game_over = game.step(a_t)
        s_t = preprocess_images(x_t)
```

```
# если вознаграждение положительно, увеличить num_wins
if r_t == 1:
    num_wins += 1

num_games += 1
print("Game: {:03d}, Wins: {:03d}".format(num_games, num_wins), end="r")
print("")
```

Если выполнить код оценивания, закомментировав строку, где устанавливается режим без монитора, то мы будем видеть, как сеть играет. Зрелище очень увлекательное. Учитывая, что начальные предсказания значений Q-функции случайны и что обучение сети направляется разреженным механизмом вознаграждения, трудно поверить, что сеть сможет научиться хорошо играть. Тем не менее, как и в других областях глубокого обучения, это случилось.

Мы привели довольно простой пример, но он иллюстрирует, как устроен процесс глубокого обучения с подкреплением, и хочется надеяться, что он помог вам построить умозрительную модель, которую можно будет применить к более сложным приложениям. Интересна реализация игры FlappyBird с использованием Keras, описанная в статье Ben Lau «Using Keras and Deep Q-Network to Play FlappyBird», 2016 и выложенная на GitHub по адресу <https://github.com/yanpanlau/Keras-FlappyBird>. В проекте Keras-RL (<https://github.com/matthiasplappert/keras-rl>), библиотеке глубокого обучения с подкреплением на основе Keras, также есть хорошие примеры.

После выхода оригинальной статьи DeepMind появились различные усовершенствования, например, двойное Q-обучение (см. H. Van Hasselt, A. Guez, and D. Silver «Deep Reinforcement Learning with Double Q-Learning», AAAI, 2016), воспроизведение опыта с приоритетами (см. T. Schaul «Prioritized Experience Replay», arXiv:1511.05952, 2015) и архитектуры дуэльных сетей (см. Z. Wang «Dueling Network Architectures for Deep Reinforcement Learning», arXiv:1511.06581, 2015). При двойном Q-обучении используются две сети: основная выбирает действие, а целевая – значение Q-функции для этого действия. Это уменьшает возможную переоценку значений Q одной сетью и позволяет сети обучаться быстрее и лучше. Воспроизведение опыта с приоритетами увеличивает вероятность выборки из накопленного опыта кортежей, которые дают больший ожидаемый прогресс в обучении. В архитектурах дуэльных сетей Q-функция разлагается на компоненты, соответствующие состоянию и действию, а затем они вновь комбинируются по отдельности.

Весь код, приведенный в этом разделе, включая исходную игру, в которую может играть человек, имеется в составе исходного кода к данной главе.

Что дальше?

В январе 2016 компания DeepMind анонсировала выпуск AlphaGo (см. D. Silver «Mastering the Game of Go with Deep Neural Networks and Tree Search», Nature 529.7587, pp. 484–489, 2016), нейронной сети для игры в го. Считается, что игра го очень сложна для искусственного интеллекта, потому что количество возможных ходов превышает 10^{170} (см. <http://ai-depot.com/LogicGames/Go-Complexity.html>) (для сравнения в шахматах ходов приблизительно 10^{50}). Поэтому найти наилучший ход путем перебора невозможно при современных вычислительных средствах. На момент публикации AlphaGo уже выиграла со счетом 5–0 в матче из пяти партий у чемпиона Европы Фан Ху. До того компьютерным программам не удавалось побеждать человека в го. Позже, в марте 2016, AlphaGo выиграла со счетом 4–1 у Ли Седоля, второго в мире профессионального игрока в го.

В AlphaGo было воплощено несколько новых идей. Во-первых, при обучении использовалась комбинация обучения с учителем на играх опытных мастеров и обучения с подкреплением путем игры двух экземпляров AlphaGo между собой. В предыдущих главах мы видели примеры реализации обеих этих идей.

Во-вторых, AlphaGo состояла из двух сетей: оценочной и стратегической. На каждом ходе AlphaGo использует моделирование методом Монте-Карло для предсказания вероятности различных исходов в будущем при наличии случайных факторов, стремясь предугадать много вариантов развития игры, начиная с текущей позиции. Оценочная сеть служит для того, чтобы уменьшить глубину дерева поиска для оценки вероятности выигрыша и проигрыша, не вычисляя игру до конца, – это своего рода интуитивная оценка качества хода. Задача стратегической сети – уменьшить ширину поиска, направляя поиск в сторону действий, обещающих максимальное немедленное вознаграждение (значение Q). Более подробное описание см. в статье «AlphaGo: Mastering the ancient game of Go with Machine Learning», Google Research Blog, 2016.

Хотя AlphaGo стала значительным улучшением первоначальной сети DeepMind, она все же предназначена для игр, в которых

игроки видят одни и те же фишки, т. е. обладают полной информацией. В январе 2017 года исследователи из университета Карнеги-Меллон анонсировали Libratus (см. T. Revel «AI Takes on Top Poker Players», New Scientist 223.3109, pp. 8, 2017), приложение ИИ для игры в покер. Одновременно другая группа исследователей из Альбертского университета, Карлова университета в Праге и Чешского технического университета (тоже в Праге) предложила архитектуру DeepStack (см. M. Moravaak «DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker», arXiv:1701.01724, 2017) для той же цели. Покер – игра с неполной информацией, поскольку игрок не видит карт соперников. Поэтому ИИ должен не только сам обучиться играть, но и выработать интуитивное представление об игре соперников.

В Libratus нет встроенной стратегии выработки интуиции, а есть алгоритм, который вычисляет эту стратегию, стремясь найти баланс между риском и вознаграждением – равновесие Нэша. С 11 января 2017 по 31 января 2017 Libratus выставился против четырех лучших игроков в покер (см. «Upping the Ante: Top Poker Pros Face Off vs. Artificial Intelligence», Carnegie Mellon University, January 2017) и победил их с ощутимым перевесом.

Для тренировки интуиции DeepStack использовалось обучение с подкреплением на примерах из случайных партий в покер. Программа сыграла с 33 профессиональными игроками из 17 стран и набрала рейтинг, *на порядок* превышающий рейтинг хорошего игрока (см. С. Е. Perez «The Uncanny Intuition of Deep Learning to Predict Human Behavior», Medium corporation, Intuition Machine, February 13, 2017).

Как видите, нас ждут интересные времена. Прогресс, начавшийся с того, что сети глубокого обучения научились играть в аркадные игры, привел к сетям, которые читают ваши мысли или, по крайней мере, предвидят (иногда нерациональное) поведение человека и обыгрывают его на блефе. Возможности глубокого обучения, похоже, не имеют границ.

Резюме

В этой главе мы узнали об идеях, лежащих в основе обучения с подкреплением, и о том, как применить их к построению с помощью Keras сетей глубокого обучения, умеющих играть в аркадные игры с обратной связью (вознаграждением). Затем мы перешли к об-

суждению последних достижений в этой области, в т. ч. сетей, обученных играть в более трудные игры, го и покер, лучше человека. На первый взгляд, игры могут показаться несерьезным приложением, но эти идеи – первый шаг к настоящему искусственному интеллекту, когда сеть обучается на опыте, а не на больших объемах обучающих данных.

Заключение

Поздравляем читателей, добравшихся до конца книги! Давайте остановимся и посмотрим, какой путь мы прошли.

Если вы – типичный читатель, то, приступая к чтению, умели программировать на Python и обладали базовыми знаниями о машинном обучении, а хотели узнать больше о глубоком обучении и применить эти знания на практике в программах на Python.

Вы узнали, как установить библиотеку Keras на свою машину, и начали использовать ее для построения простых моделей глубокого обучения. Затем вы узнали о самой первой такой модели – многослойном перцептроне, который еще называют **полносвязной сетью**. Вы научились строить такие сети с помощью Keras.

Вы также узнали о многочисленных параметрах, которые необходимо настраивать для получения хороших результатов от сети. В Keras самая сложная работа уже сделана, поскольку в библиотеке заданы разумные значения по умолчанию, но в некоторых случаях без знания о параметрах все же не обойтись.

Далее мы перешли к **сверточным нейронным сетям**, которые изначально были предложены для учета локальности признаков в изображениях, а потом выяснилось, что их можно применить и к данным других типов, включая текст, звук и видео. Вы узнали, что Keras позволяет строить и такие сети тоже – легко и интуитивно понятно. Вы видели, как сети, предобученные на наборах изображений, позволяют делать предсказания для ваших изображений с помощью процесса переноса обучения и окончательной настройки.

Вслед за этим вы узнали о **порождающих состязательных сетях (ПСС)**, когда две сети (обычно сверточных) пытаются противостоять друг другу и в результате каждая становится сильнее. ПСС – это передний край глубокого обучения, им посвящено немало недавних работ.

Затем мы обратились к **погружению слов** – технологии, которая в последние пару лет стала активно применяться для векторного представления текста. Мы рассмотрели несколько популярных алгоритмов погружения слов и видели, как использовать предобученные погружения для представления коллекций слов. Мы так-

же познакомились с поддержкой погружения слов в библиотеках Keras и gensim.

После этого мы перешли к **рекуррентным нейронным сетям (РНС)** – классу сетей, оптимизированному для работы с последовательностями данных, в т. ч. с текстом и временными рядами. Мы узнали о недостатках базовой модели РНС и о том, как их преодолеть с помощью более эффективных вариаций: **долгой краткосрочной памяти (LSTM)** и **вентильных рекуррентных блоков (GRU)**. Мы рассмотрели несколько примеров использования этих компонентов. Мы также кратко познакомились с РНС с сохранением состояния и их возможными применениями.

Далее мы обсудили дополнительные модели, которым не нашлось места в предыдущих главах. К ним относятся **автокодировщики** – модели, обучаемая без учителя, и **регрессионные сети**, которые предсказывают не дискретную метку, а непрерывное значение. Мы познакомились с **функциональным API Keras**, который позволяет строить сложные сети с несколькими входами и выходами и разделять компоненты между несколькими конвейерами. Мы обсудили расширение Keras с целью добавления новой функциональности.

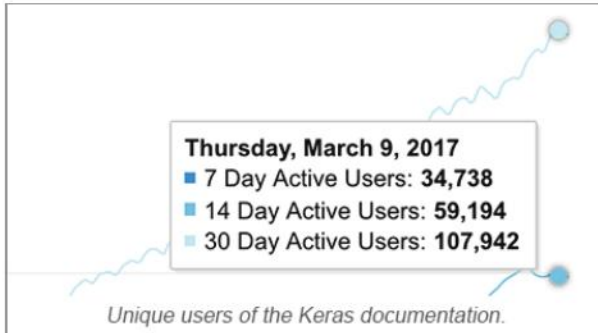
Наконец, мы рассмотрели глубокое **обучение с подкреплением** в контексте аркадных игр и заметили, что такие сети могут стать первым шагом на пути к настоящему искусственному интеллекту. Мы показали, как можно обучить сеть простой игре средствами Keras. Затем мы дали краткий обзор последних достижений в этой области на примере еще более сложных игр – покер и го, – в которых нейронные сети уже превосходили человека.

Мы полагаем, что теперь вы во всеоружии для решения новых задач машинного обучения с применением методов глубокого обучения и библиотеки Keras.

Пользуясь случаем, благодарим за то, что позволили сопроводить вас на пути к секретам мастерства в глубоком обучении.

Keras 2.0 – что нового

По словам Франсуа Шолле, библиотека Keras была выпущена в марте 2015 года. И постепенно число пользователей увеличилось с одного до сотни тысяч. На следующем рисунке, взятом из блога Keras, показан рост числа пользователей Keras со временем.



Одним из важных новшеств в Keras 2.0 является тот факт, что его API стал частью TensorFlow, начиная с версии TensorFlow 1.2. Вообще, Keras все больше обретает черты «лингва франка» (языка-посредника) в глубоком обучении, *спецификации*, используемой во все новых и новых контекстах. Например, компания Skyminд реализует спецификацию Keras на Scala для библиотеки ScalNet, а Keras.js делает то же самое для JavaScript, чтобы глубоким обучением можно было заниматься прямо в браузере. Ведутся работы по предоставлению API Keras для библиотек глубокого обучения MXNET и CNTK.

Установка Keras 2.0

Для установки Keras 2.0 достаточно выполнить команду `pip install keras --upgrade`, а вслед за ней – `pip install tensorflow --upgrade`.

Изменения API

Переход на версию Keras 2.0 вынудил пересмотреть некоторые API. Полный перечень изменений см. в замечаниях к версии (<https://github.com/fchollet/keras/wiki/Keras-2.0-release-notes>). В показанном ниже модуле `legacy.py` сведены самые существенные изменения с целью предотвратить появление предупреждений при использовании функций из версии Keras 1.x:

```
"""
```

```
Служебные функции для предотвращения предупреждений при совместном
использовании Keras 1 и 2.
```

```
"""
```

```
import keras
```

```
keras_2 = int(keras.__version__.split(".")[0]) > 1 # Keras > 1

def fit_generator(model, generator, epochs, steps_per_epoch):
    if keras_2:
        model.fit_generator(generator, epochs=epochs, steps_per_epoch=steps_per_epoch)
    else:
        model.fit_generator(generator, nb_epoch=epochs, samples_per_epoch=steps_per_epoch)

def fit(model, x, y, nb_epoch=10, *args, **kwargs):
    if keras_2:
        return model.fit(x, y, *args, epochs=nb_epoch, **kwargs)
    else:
        return model.fit(x, y, *args, nb_epoch=nb_epoch, **kwargs)

def l1l2(l1=0, l2=0):
    if keras_2:
        return keras.regularizers.L1L2(l1, l2)
    else:
        return keras.regularizers.l1l2(l1, l2)

def Dense(units, W_regularizer=None, W_initializer='glorot_uniform', **kwargs):
    if keras_2:
        return keras.layers.Dense(units, kernel_regularizer=W_regularizer,
                                    kernel_initializer=W_initializer, **kwargs)
    else:
        return keras.layers.Dense(units, W_regularizer=W_regularizer,
                                    init=W_initializer, **kwargs)

def BatchNormalization(mode=0, **kwargs):
    if keras_2:
        return keras.layers.BatchNormalization(**kwargs)
    else:
        return keras.layers.BatchNormalization(mode=mode, **kwargs)

def Convolution2D(units, w, h, W_regularizer=None,
                  W_initializer='glorot_uniform', border_mode='same', **kwargs):
    if keras_2:
        return keras.layers.Conv2D(units, (w, h), padding=border_mode,
                                     kernel_regularizer=W_regularizer,
                                     kernel_initializer=W_initializer,
                                     **kwargs)
    else:
        return keras.layers.Conv2D(units, w, h, border_mode=border_mode,
                                     W_regularizer=W_regularizer, init=W_initializer, **kwargs)

def AveragePooling2D(pool_size, border_mode='valid', **kwargs):
    if keras_2:
        return keras.layers.AveragePooling2D(pool_size=pool_size,
                                                padding=border_mode, **kwargs)
    else:
```

```
return keras.layers.AveragePooling2D(pool_size=pool_size,  
border_mode=border_mode, **kwargs)
```

Имеется также ряд несовместимых изменений, в частности:

- унаследованные слои `maxout`, `dense`, `time distributed dense` и `highway` **исключены**;
- слой пакетной нормировки больше не поддерживает аргумент `mode`, поскольку изменилась внутренняя реализация Keras;
- пользовательские слои необходимо обновить;
- любые недокументированные возможности Keras могут перестать работать.

Ко всему прочему, кодовая база Keras снабжена средствами обнаружения случаев обращения к API Keras 1.x, при этом выводятся предупреждения с рекомендацией, как модифицировать вызов, адаптировав его к API Keras 2. Если вы уже написали достаточно много кода для Keras 1.x и опасаетесь переходить на Keras 2 из-за несовместимых изменений, то эти предупреждения облегчат переход.

Предметный указатель

А

автокодировщики
 общие сведения 223, 286
 пример 225
архитектура Keras 68

Б

блок линейной ректификации 28

В

вентильный рекуррентный блок (GRU)
 176, 197, 286
 частеречная разметка 198
веса 81
 сохранение и загрузка 73
вытесняемая виртуальная машина 75

Г

гиперпараметры, настройка 52
глубокая сверточная нейронная сеть
 (ГСНС) 80
 LeNet 83
 веса 81
 пулингвые слои 82
 распознавание изображений 101
 рецептивные поля 80
 смещение 81
глубокая сверточная порождающая
 состязательная сеть (ГСПСС) 114
глубокое обучение
 общие сведения 54, 89
градиентный спуск 41
градиенты
 взрывные 188
 исчезающие 188

Д

двунаправленные РНС 205
длина шага 81

долгая краткосрочная память (LSTM)
 176, 188, 286

И

инициализаторы 25
информационный поиск 143

К

коллаборативная фильтрация 224
контрольные точки 75

Л

латентно-семантический анализ (ЛСА) 143
локальная нормировка отклика 243

М

метод главных компонент (РСА) 223
многослойный перцептрон (МСП)
 блок линейной ректификации 28
 пример нейросети 25
 проблемы обучения 26
 сигмоида 27
 функции активации 28
модели Keras
 последовательная композиция 68
 функциональная композиция 69

Н

накопленная память 269

О

обработка естественных языков (ОЕЯ) 143
обратное распространение 52
обратное распространение во времени
 (BPTT) 186
обратные вызовы для управления
 процессом обучения 74
обучение без учителя 223
обучения процесс

использование Quiver 76
 использование TensorBoard 76
 контрольные точки 75
 управление с помощью обратных вызовов 74
 оптимизаторы
 ссылка на документацию 73
 тестирование в Keras 41

П

пакетная нормировка 71
 пакетные вычисления
 увеличение размера пакета 48
 первичная зрительная кора 23
 перенос обучения
 использование сети inception-v3 105
 перенос стиля 255
 периоды
 увеличение числа 46
 показатели качества 73
 полносвязная сеть 285
 пользовательский слой, рекомендации по построению 243
 пополнение данных, на примере набора изображений CIFAR-10 97
 порождающая состязательная сеть (ПСС) 109, 247, 285
 порождающие модели 247
 Deep Dream 248
 перенос стиля 255
 последовательная модель в Keras 24
 признаки
 выделение с помощью предобученных моделей 104
 прореживание 38
 простые ячейки РНС 177
 пулинговые слои 82
 max-пулинг 82
 ссылка на документацию 83
 усредненный пулинг 82

Р

распознавание изображений
 inception-v3, применение для переноса обучения 105
 встроенный модуль VGG-16 103
 РСНС 101

использование готовых моделей для выделения признаков 104
 с помощью сети VGG-16 102
 распределенные представления 144
 расширение Keras
 использования слоя lambda 242
 общие сведения 242
 пользовательского слоя нормировки 243
 регрессионные сети 218
 регуляризация 71
 применение для предотвращения переобучения 50
 типы 51
 рекуррентная нейронная сеть (РНС) 69, 161
 другие варианты 212
 топологии 184
 рецептивное поле 80
 речевое воспроизведение текста (TTS)
 компиляционный синтез 133
 параметрический синтез 133
 РНС с запоминанием состояния 206
 предсказание потребления электричества 206
 рукописные цифры, распознавание 29
 добавление скрытых слоев 35
 настройка гиперпараметров 52
 определение нейронной сети в Keras 30
 подведение итогов экспериментов 49
 предсказание выхода 52
 регуляризация 50
 создание эталона для сравнения 34
 тестирование оптимизаторов 41
 увеличение размера пакета 48
 увеличение числа нейронов в скрытых слоях 47
 увеличение числа периодов 46
 улучшение с помощью прореживания 38
 унитарное кодирование 30
 управление скоростью обучения оптимизатора 46

С

сверточная нейронная сеть (СНС) 55, 79, 83, 161, 285
 сигмоида 27
 скорость обучения оптимизатора, управление 46
 скрытые слои 35

смещение 81
среднеквадратическая ошибка 218
стохастический градиентный спуск (СГС) 43

Т

тензор 68

У

унитарное кодирование 30, 143

Ф

факторизация матрицы 158
функции потерь 72
функциональный API Keras 215, 286
функция активации
 блок линейной ректификации (ReLU) 28
 ссылка на документацию 29
 общие сведения 28
 сигмоида 27

Ц

целевая функция
 бинарная перекрестная энтропия 32
 категориальная перекрестная энтропия 32
 среднеквадратическая ошибка 32
цикл обработки событий 272

Ч

частеречная разметка 185

А

Adam 43
AlphaGo 282
Amazon AWS
 спотовый инстанс 75
 ссылки 65
 установка Keras 64

В

bAbI проект 235

С

Caffe 101, 244

CBOW word2vec модель 150
CIFAR-10, набор изображений
 повышение качества распознавания путем пополнения 97
 повышение качества распознавания путем углубления сети 95
 предсказание 100
 ПСС для подделывания 124
 распознавание с помощью глубокого обучения 90
 ссылка 91

Д

Deep Dream (глубокие сновидения) 248
DeepMind 132, 141
Docker, установка Keras 60

Г

GloVe 158
GloVe-Python, проект 159
Google Cloud ML
 установка Keras 62

Н

HDF5, формат 75

И

ImageNet ILSVRC-2012, набор данных 101
inception-v3, использование для переноса обучения 106

Ж

Jupyter Notebooks 61

К

Keras
 код сети LeNet 83
 настройка 59
 определение простой нейронной сети 30
 проверка работоспособности 58
 расширение 242
 ссылка на дистрибутив 63
 тестирование оптимизаторов 41
 установка 56
 установка TensorFlow 57
 установка Theano 57

- установка в Amazon AWS 64
- установка в Google Cloud ML 62
- установка в Microsoft Azure 65
- установка в контейнер Docker 60
- установка зависимостей 56
- Keras 2.0 287
- Keras adversarial
 - ПСС для подделывания CIFAR-10 124
 - ПСС для подделывания MNIST 118
- Keras API
 - архитектура Keras 68
 - вспомогательные операции 73
 - готовые слои 69
 - готовые функции активации 72
 - загрузка и сохранение весов и архитектуры модели 73
 - общие сведения 67
 - оптимизаторы 73
 - показатели качества 73
 - функции потерь 72
- L**
- LeNet, сеть 83
 - код в Keras 83
- Libratus 283
- M**
- MNIST, набор данных
 - подделывание с помощью ПСС 118
 - ссылка 29, 90
- N**
- NLTK, библиотека для обработки естественных языков 161
- O**
- OpenAI 109
- P**
- Pygame 269
- Q**
- Quiver 77
- R**
- RMSprop 43
- S**
- skip-граммы 146
- T**
- TensorBoard 76
- TensorFlow
 - проверка работоспособности 59
 - ссылка 67
 - установка 57
- Theano
 - проверка работоспособности 58
 - ссылка 67
 - установка 57
- V**
- VGG-16 сеть 102
- W**
- WaveNet 132
- word2vec
 - извлечение погружений 151
 - модель CBOW 150
 - модель skip-грамм 146
 - общие сведения 145
 - сторонние реализации 154

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.aliants-kniga.ru.**

Оптовые закупки: **тел. +7 (499) 782-38-89.**

Электронный адрес: **books@aliants-kniga.ru.**

Антонио Джулли, Суджит Пал

Библиотека Keras – инструмент глубокого обучения

Реализация нейронных сетей с помощью
библиотек Theano и TensorFlow

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Слинкин А. А.*

Корректор *Синяева Г. И.*

Верстка *Паранская Н. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург».

Печать офсетная. Усл. печ. л. 18,38.

Тираж 200 экз.

Веб-сайт издательства: **www.dmk.ru**