Обработка естественного языка применяется в различных приложениях машинного обучения, а TensorFlow – важнейшая библиотека для реализации систем глубокого обучения на практике.

Эта книга знакомит читателя с методами обработки естественного языка и содержит практическое руководство по работе с TensorFlow, предоставляя мощный инструмент для работы с огромными объемами неструктурированных данных и решения уникальных задач по обработке естественного языка.

Книга начинается с изучения общих понятий NLP и принципа работы TensorFlow и через ряд усложняющихся тем подводит читателя к самостоятельному созданию системы нейронного машинного перевода.

В книге описываются:

- основные понятия и подходы в области обработки естественного языка;
- методы решения задач NLP с помощью функций TensorFlow для создания нейронных сетей;
- стратегии обработки больших объемов данных и способы представления слов для использования в приложениях глубокого обучения;
- технологии улучшенной классификации предложений и генерации текста при помощи сверточных и рекуррентных нейросетей;
- применение передовых рекуррентных сетей, таких как LSTM, для решения комплексной задачи генерации текста;
- принципы машинного перевода и реализация уникальной системы нейронного перевода;
- тенденции и инновации, от которых зависит будущее обработки естественного языка.

Научите компьютер разговаривать, используя библиотеки глубокого обучения на языке Python

Packt›

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-756-5
9 785970 607565 >

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
e-mail: books@alians-kniga.ru

Обработка естественного языка с TensorFlow

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Тушан Ганегедара

Обработка естественного языка с TensorFlow

Тушан Ганегедара

Обработка естественного языка с TensorFlow

Natural Language Processing with TensorFlow

Teach language to machines using Python's deep learning library

Thushan Ganegedara

Packt

BIRMINGHAM – MUMBAI

Обработка естественного языка с TensorFlow

Научите компьютер разговаривать, используя библиотеки глубокого обучения на языке Python

Тушан Ганегедара

ДМК ПРЕСС
издательство
Москва, 2020

УДК 004.032.2
ББК 32.972.1
Г19

Ганегедара Т.
Г19 Обработка естественного языка с TensorFlow / пер. с анг. В. С. Яценкова. – М.: ДМК Пресс, 2020. – 382 с.: ил.

ISBN 978-5-97060-756-5

TensorFlow – библиотека на языке Python для реализации систем глубокого обучения, позволяющих решать в том числе уникальные задачи по обработке естественного языка.

Автор книги излагает общие принципы работы NLP и построения нейронных сетей, описывает стратегии обработки больших объемов данных, а затем переходит к практическим темам. Вы узнаете, как использовать технологию World2vec и ее расширения для создания представлений, превращающих последовательности слов в числовые векторы, рассмотрите примеры решения задач по классификации предложений и генерации текста, научитесь применять продвинутые рекуррентные модели и сможете самостоятельно создать систему нейронного машинного перевода.

Издание предназначено для разработчиков, которые, используя лингвистические данные, применяют и совершенствуют методы машинной обработки естественного языка.

УДК 004.032.2
ББК 32.972.1

ISBN 978-1-78847-831-1 (анг.)
ISBN 978-5-97060-756-5 (рус.)

Содержание

Об авторе

Тушан Ганегедара написал эту книгу на третьем году обучения в аспирантуре университета Сиднея, Австралия, а перед этим получил степень бакалавра с отличием в университете Моратува, Шри-Ланка. Он специализируется на машинном обучении и особенно увлечен глубокими нейросетями. Тушан любит рисковать и часто запускает алгоритмы на непроверенных данных. Он также работает в качестве главного аналитика данных в австралийском стартапе AssessThreat и регулярно пишет технические статьи и учебные пособия по машинному обучению. Кроме того, он стремится к здоровому образу жизни и ежедневно занимается плаванием.

Я хочу поблагодарить моих родителей, моих братьев и сестер и мою жену за веру в меня и за поддержку, которую они оказали, а также всех моих учителей и моего научного руководителя за наставления, которые они дали мне.

О рецензентах

Мотаз Саад окончил аспирантуру по информатике в Университете Лотарингии. Он любит данные и все, что с ними связано. Он более 10 лет занимается обработкой естественных языков, компьютерной лингвистикой, наукой о данных и машинным обучением. В настоящее время работает доцентом на факультете информационных технологий IUG (Islamic University of Gaza).

Доктор Джозеф О'Коннор – специалист по данным, искренне увлеченный глубоким обучением. Его компания Deep Learn Analytics, британская консалтинговая компания, специализирующаяся на данных, оказывает предприятиям услуги по разработке приложений и инфраструктуры машинного обучения от концепции до развертывания. Ему была присуждена степень доктора философии в университете Лондона за работу по анализу данных эксперимента по физике высоких энергий MINOS. С того времени он разработал продукты машинного обучения для ряда компаний частного сектора, специализирующихся на NLP и прогнозировании временных рядов. Вы можете найти его на **http://deeplearnanalytics.com/**.

Предисловие

В наш век цифровой информации, в котором мы живем, объем данных растет в геометрической прогрессии. Пока вы читаете эти слова, мировые запасы данных продолжают расти с ошеломляющей скоростью. Большая часть этих данных относится к языковым данным – текстовым или устным, – таким как электронные письма, сообщения в социальных сетях, телефонные звонки и статьи в интернете. Машинная *обработка естественного языка* (natural language processing, NLP) эффективно использует эти данные, чтобы помочь людям в их бизнесе или в повседневных задачах. Технология NLP уже произвела революцию в повседневном использовании данных, помогает бизнесу, облегчает жизнь людей и продолжит делать это в будущем.

Одним из наиболее распространенных примеров применения NLP являются *виртуальные помощники* (virtual assistants, VA), такие как Siri от Apple, Google Assistant и Amazon Alexa. Всякий раз, когда вы просите своего виртуального помощника найти «отели в Швейцарии по низким ценам», запускается серия сложных задач обработки естественного языка. Во-первых, ваш помощник должен понять смысл запроса (например, определить, что надо искать низкие цены на отели, а не ближайшие площадки для выгула собак). Еще одно решение, которое должен принять помощник, – это определить критерий «низкой цены». Затем помощник должен ранжировать города в Швейцарии, возможно, исходя из вашей прошлой истории путешествий. Помощник может просканировать сайты агрегаторов отелей, чтобы извлечь оттуда цены на отели в Швейцарии и «прочитать» отзывы посетителей для каждого отеля. Как видите, ответ на запрос, который вы получаете через несколько секунд, является результатом разносторонней работы системы NLP.

Итак, что делает системы NLP настолько универсальными и точными в решении наших повседневных задач? В основе успеха лежат алгоритмы глубокого обучения. Это, по сути, сложные нейронные сети, которые могут проецировать необработанные данные в желаемый результат, не требуя какой-либо сложной ручной настройки алгоритма. Это означает, что турист напишет отзыв об отеле на естественном человеческом языке, а компьютер безошибочно ответит на вопрос «Насколько положителен отзыв клиента об этом отеле?». Кроме того, глубокое обучение уже достигло и даже превысило уровень человеческих возможностей в различных задачах NLP, таких как распознавание речи и машинный перевод.

Прочитав эту книгу, вы узнаете, как решать многие интересные задачи NLP, используя глубокое обучение. Итак, если вы хотите быть влиятельным человеком, который меняет мир, изучение NLP имеет решающее значение. Прикладные задачи варьируются от изучения семантики слов до генерации новых историй и выполнения машинного перевода с обучением «на ходу». Все главы содержат упражнения, практические примеры и пошаговые инструкции по внедрению рассматриваемых решений. Для всех упражнений в этой книге мы будем использовать Python с TensorFlow – популярной библиотекой распределенных вычислений, которая делает реализацию глубоких нейронных сетей очень простой и удобной.

Для кого эта книга

Эта книга предназначена для начинающих исследователей и разработчиков, которые стремятся сделать мир лучше, используя лингвистические данные. Книга предоставит вам прочную практическую основу для решения задач NLP. В этой книге мы рассмотрим различные аспекты NLP, уделяя больше внимания практической реализации, чем теоретическим основам. Обладание практическими навыками решения различных задач NLP поможет вам совершить более плавный переход к изучению более сложных теоретических аспектов этих методов. Кроме того, хорошее понимание практической части NLP поможет при выполнении более точной настройки ваших алгоритмов, чтобы получить максимальную отдачу от конкретного проекта.

Какие темы охватывает эта книга

Глава 1 знакомит вас с NLP. В этой главе вы узнаете причины, по которым востребована обработка естественного языка. Далее перечислены некоторые общие проблемы и определены две основные эпохи NLP – период использования традиционных методов и современные приемы глубокого обучения. Сначала мы в общих чертах рассмотрим, как задача моделирования языка решается с помощью традиционных алгоритмов. Затем обсудим современную эпоху, когда задачи машинной обработки языка решаются с помощью моделей глубокого обучения, и рассмотрим основные семейства алгоритмов глубокого обучения. Потом вы познакомитесь с принципом работы одного из основных современных алгоритмов – полносвязной нейронной сети. Главу завершает «дорожная карта», в которой дается краткое введение в последующие главы.

Глава 2 знакомит вас с библиотекой Python TensorFlow – основной платформой, на которой реализованы все решения, рассмотренные в этой книге. Вы начнете с написания кода для реализации простых вычислений в TensorFlow, а затем узнаете, как выполняется этот код, начиная с запуска программы и заканчивая получением результатов. Таким образом, вы на простом примере познакомитесь с основными компонентами TensorFlow. Вы еще больше укрепите понимание TensorFlow благодаря красочной аналогии с процессом выполнения заказов в ресторане. Далее мы обсудим технические детали TensorFlow, такие как структуры данных и операции с нейронными сетями, встроенные в TensorFlow. Наконец, вы создадите полносвязную нейронную сеть для распознавания рукописных цифр. Это пример реализации комплексного решения с помощью TensorFlow.

Глава 3 начинается с обсуждения того, как решать задачи NLP с помощью TensorFlow. В этой главе вы узнаете, как нейронные сети применяются для изучения векторов слов, известных также как представления слов. Векторы слов являют собой числовые представления слов с учетом смыслового окружения. Сначала мы обсудим несколько традиционных подходов к достижению этой цели, которые включают использование большой базы знаний, созданной человеком, известной как WordNet. Затем мы рассмотрим современный подход на основе нейронных сетей, известный как Word2vec и основанный на изучении векторов слов без вмешательства человека. Сначала вы исследуете механику Word2vec, проработав практиче-

ский пример, а затем познакомитесь с двумя усовершенствованными методами – *словосочетаниями с пропуском* (Skip-Gram) и *непрерывным мультимножеством слов* (continuous bag-of-words, CBOW). Главу завершает обсуждение концептуальных особенностей алгоритмов, а также способы их реализации в TensorFlow.

Глава 4 раскрывает более сложные темы, связанные с векторами слов. Сначала мы сравним подходы Skip-Gram и CBOW, чтобы узнать, существует ли победитель. Далее мы обсудим несколько улучшений, которые можно использовать для повышения качества работы алгоритмов Word2vec. Затем вы познакомитесь с более современным и мощным подходом к изучению векторизации смыслов – алгоритмом *глобальных векторов* GloVe. Наконец, вы познакомитесь с векторами слов в действии в задаче классификации документов. В этом упражнении вы увидите, что векторный подход достаточно точен и может правильно классифицировать тему документа.

Глава 5 посвящена обсуждению *сверточных нейронных сетей* (convolutional neural network, CNN) – семейства нейронных сетей, которые превосходны при обработке пространственных данных, таких как изображения или предложения. Мы начнем с изучения общих принципов работы CNN, обсудив, как они обрабатывают данные и какие операции выполняют. Далее мы детально разберем каждую операцию, связанную с вычислениями, чтобы понять внутреннюю математику CNN. Наконец, вы выполните два упражнения. Во-первых, вы будете классифицировать изображения рукописных цифр с помощью CNN. Вы убедитесь, что CNN способны быстро достичь очень высокой точности при решении задач такого типа. Далее вы узнаете, как можно использовать CNN для классификации предложений. В завершающем главу примере нейросеть предсказывает, относится ли предложение к объекту, человеку, местоположению и т. д.

Глава 6 рассказывает о *рекуррентных нейронных сетях* (recurrent neural network, RNN) – мощном семействе нейронных сетей, которые могут обрабатывать последовательности данных. Сначала вы изучите внутреннюю математику и правила функционирования RNN в процессе обучения. Затем вы познакомитесь с различными вариантами RNN и их применением (например, структуры *один-к-одному* и *один-ко-многим*). Наконец, вы выполните упражнение, в котором RNN решает задачу генерации текста. В этом примере RNN обучается на наборе сказок и пытается сочинить новую сказку. Вы увидите, что RNN плохо работают с долговременной памятью. Наконец, мы обсудим более продвинутый вариант RNN под названием RNN-CF, обладающий более надежной долгосрочной памятью.

Глава 7 рассказывает про более мощные рекуррентные нейросети, которые способны запоминать данные в течение более длительного периода времени, поскольку стандартные RNN плохо поддерживают долговременную память. В этой главе рассказано про сети с *долгой краткосрочной памятью* (long short-term memory, LSTM). Известно, что LSTM превосходят другие последовательные модели во многих задачах временных рядов. Сначала вы исследуете базовую математику и новые правила LSTM на ярком примере, который иллюстрирует, почему каждое вычисление имеет значение. Затем вы узнаете, как LSTM могут сохранять память еще дольше. Далее мы обсудим, как можно улучшить прогнозные возможности LSTM. Наконец, мы рассмотрим несколько вариантов LSTM, имеющих более сложную структуру, включая *управляемые рекуррентные блоки* (Gated Recurrent Unit, GRU).

Глава 8 подробно рассказывает о том, как LSTM-сети работают в задаче генерации текста. Вы качественно и количественно оцените, насколько хорош текст, сгенерированный LSTM, а также проведете сравнение различных модификаций LSTM. Наконец, вы узнаете, как добавить в модель механизм векторного представления слов, чтобы улучшить качество сгенерированного текста.

Глава 9 от работы с текстами переносит вас к *мультимодальным данным*, то есть комбинации изображения и текста. В этой главе вы узнаете, как автоматически генерировать текстовое описание для заданного изображения. Решение включает в себя объединение сверточной модели прямого распространения (CNN) со слоем представления слов и последовательной модели (LSTM) таким образом, чтобы сформировать полный цикл машинного обучения.

Глава 10 рассказывает о реализации модели *нейронного машинного перевода* (neural machine translation, NMT). В машинном переводе мы переводим предложение/фразу с исходного языка на целевой язык. Сначала вы познакомитесь с краткой историей и основами машинного перевода. Затем детально изучите архитектуру современных моделей нейронного машинного перевода, включая процедуры обучения и вывода. Далее вы узнаете, как реализовать систему NMT с нуля. Наконец, вы познакомитесь со способами улучшения стандартных систем NMT.

Глава 11 завершает книгу и посвящена текущим тенденциям и будущему NLP. Мы обсудим последние открытия, связанные с системами и задачами, о которых рассказано в предыдущих главах. В этой главе будет рассказано о самых интересных нововведениях, а также будет дано углубленное представление о внедрении некоторых технологий.

Приложение познакомит читателя с различными математическими структурами данных и операциями. Мы также обсудим несколько важных понятий в теории вероятности. Затем вы познакомитесь с Keras – библиотекой высокого уровня, которая использует TensorFlow. Keras упрощает реализацию нейронных сетей, скрывая некоторые детали в TensorFlow, что может сначала показаться запутанным. Вы познакомитесь с примером реализации сверточной сети с помощью Keras. Далее мы рассмотрим использование библиотеки seq2seq в TensorFlow для реализации системы машинного перевода с гораздо меньшим количеством кода, чем в упражнении из главы 11. Наконец, вы познакомитесь с руководством по использованию TensorBoard для визуализации смысловых связей слов. TensorBoard – это удобный инструмент визуализации, который поставляется с TensorFlow. Его можно использовать для визуализации и мониторинга различных переменных в вашем клиенте TensorFlow.

Как получить максимальную отдачу от этой книги

Чтобы извлечь максимальную пользу из этой книги, читатель должен:

- иметь твердую волю и желание изучить современные методы NLP;
- ознакомиться с базовым синтаксисом языка Python и структурами данных, такими как списки и словари;
- знать основы математики, например умножение матриц/векторов.

Не обязательно, но желательно:

- иметь достаточно обширные знания в области математики, чтобы лучше понять разделы, в которых идет речь о конкретных проблемах и методах их решения;
- прочитать научные статьи по тематике глубокого обучения, чтобы иметь представление о достижениях науки и методах, помимо тех, что описаны в книге.

Условные обозначения и соглашения, принятые в книге

В книге используются следующие типографские соглашения.

Курсив – используется для смыслового выделения важных положений, новых терминов, имен команд и утилит, а также слов и предложений на естественном языке.

`Моноширинный шрифт` – применяется для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов, ключевых слов и других программных конструкций и элементов исходного кода.

`Моноширинный полужирный шрифт` – используется для обозначения команд или фрагментов текста, которые пользователь должен ввести дословно без изменений, а также в листингах программ, если необходимо обратить особое внимание на фрагмент кода.

`Моноширинный курсив` – применяется для обозначения в исходном коде или в командах шаблонных меток-заполнителей, которые должны быть заменены соответствующими контексту реальными значениями.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Введение в обработку естественного языка

Обработка естественного языка (natural language processing, NLP) является важным инструментом для понимания и обработки огромного объема неструктурированных данных в современном мире. В последнее время глубокое обучение широко применяется в NLP, потому что алгоритмы глубокого обучения чрезвычайно эффективно решают задачи классификации изображений, распознавания речи и генерации реалистичных текстов. TensorFlow, в свою очередь, является одной из наиболее простых для освоения и эффективных систем глубокого обучения, существующих в настоящее время. Эта книга адресована начинающим разработчикам систем глубокого обучения и рассказывает про обработку больших объемов данных с использованием NLP и TensorFlow.

В этой главе мы познакомимся с понятием NLP и ответим на вопрос «Что такое обработка естественного языка?». Также мы рассмотрим некоторые из наиболее важных применений NLP. Мы исследуем как традиционные подходы к NLP, так и более поздние подходы на основе глубокого обучения, включая *полносвязные нейронные сети* (fully-connected neural network, FCNN). Глава завершается обзором остальной части книги и программных инструментов, которые мы будем использовать.

Что такое обработка естественного языка?

По данным IBM, в 2017 г. каждый день вырабатывалось 2,5 эксабайта (1 эксабайт = 1 млрд гигабайт) данных, и это число постоянно растет, пока вы читаете данную книгу. Если бы все люди в мире взялись за обработку такого объема данных, то на каждого жителя планеты приходилось бы по 300 МБ в день. В основном эти данные состоят из неструктурированного текста и речи, ведь люди каждый день создают миллионы электронных писем и записей в социальных сетях, а также разговаривают по телефону.

Эта статистика красноречиво говорит нам о том, что такое NLP. Проще говоря, цель NLP – научить компьютер понимать нашу разговорную и письменную речь. Более того, мы уже регулярно используем NLP в повседневной жизни. *Виртуальные помощники* (virtual assistant, VA), такие как Google Assistant, Cortana и Apple Siri, в основном представляют собой системы NLP. Когда кто-то просит виртуаль-

ного помощника: «Покажи мне хороший итальянский ресторан поблизости», – за кулисами сервиса решается множество сложных задач. Во-первых, помощник должен преобразовать речь в текст. Далее он должен понять *семантику запроса* (пользователь ищет хороший ресторан с итальянской кухней) и сформулировать *структурированный запрос* (например, кухня = итальянская, рейтинг = 3–5, расстояние < 10 км). Затем помощник должен отфильтровать известные рестораны по местоположению и кухне. И наконец, нужно отсортировать оставшиеся рестораны по общему рейтингу. Чтобы рассчитать общий рейтинг ресторана, хорошая система NLP может посмотреть как оценки, так и текстовые отзывы, оставленные предыдущими посетителями. Наконец, когда пользователь оказался в ресторане, помощник может перевести различные пункты меню с итальянского на язык пользователя. Этот пример показывает, что обработка естественного языка стала неотъемлемой частью человеческой жизни.

Следует понимать, что NLP – это невероятно сложная область исследований, поскольку слова и смысловое наполнение имеют неоднозначную и нелинейную взаимосвязь, и подобная информация очень плохо поддается представлению в цифровом виде. Что еще хуже, каждый язык имеет свою собственную грамматику, синтаксис и словарный запас. Поэтому обработка текстовых данных включает в себя различные сложные задачи, такие как синтаксический анализ текста (например, токенизация и выделение основы), морфологический анализ, устранение неоднозначности смысла слов и понимание грамматической структуры языка. Например, в предложениях «*Я ничего не должен банку*» и «*Я разбил стеклянную банку*» слово «*банку*» имеет два совершенно разных значения. Чтобы распознать актуальный смысл слова, нам необходимо понять контекст, в котором оно используется. Машинное обучение стало ключевым инструментом NLP, помогая решать упомянутые задачи с помощью машин.

Задачи обработки естественного языка

NLP имеет множество прикладных применений. Хорошая система NLP – это система, которая решает комплекс задач. Когда вы просите Google озвучить прогноз погоды или используете Google Translate, чтобы узнать написание фразы на французском языке, вы запускаете решение цепочки задач NLP. Мы перечислим некоторые из наиболее распространенных задач и расскажем в этой книге об их решении.

- **Токенизация** (tokenization) – это задача разделения *текстового корпуса* (text corpora, большой набор текстовых документов) на неделимые единицы, например слова. Несмотря на обманчивую простоту, токенизация – это важная задача. Например, в японском языке слова не разделяются ни пробелами, ни знаками препинания.
- **Устранение неоднозначности слов** (word-sense disambiguation, WSD) – это задача определения правильного значения слова. Например, в предложениях «*Кредитная карта заблокирована*» и «*Политическая карта Африки*» слово «*карта*» имеет два разных значения. WSD имеет решающее значение для таких задач, как ответы на вопросы.
- **Выделение именованных сущностей** (named entity recognition, NER) – пытается извлечь сущности (например, человека, местоположение и органи-

зацию) из заданного текста или текстового корпуса. Например, предложение «*Джон дал Мэри два яблока в школе в понедельник*» будет преобразовано в [*Джон*]$_{имя}$ дал [*Мэри*]$_{имя}$ [*два*]$_{число}$ яблока в [*школе*]$_{организация}$ в [*понедельник*]$_{время}$. Без NER невозможно обойтись в таких областях, как поиск информации и представление знаний.

- **Морфологическая разметка** (part of speech tagging, PoS) – это задача определения частей речи в предложении и их аннотирование. Это могут быть *основные теги*, например существительное, глагол, прилагательное, наречие и предлог, или же *гранулированные теги*, такие как собственное существительное, имя нарицательное, фразовый глагол и т. д.
- **Классификация предложений/синопсисов** (sentence/synopsis classification). Классификация *предложений* или *синопсисов* (например, обзоров фильмов) имеет множество вариантов использования, таких как обнаружение спама, классификация новостных статей (например, политические, технологические и спортивные) и распознавание отзывов о продукте (например, положительные или отрицательные). Это достигается обучением *модели классификации* на помеченных данных (то есть на обзорах, аннотированных людьми).
- **Генерация естественного языка**. Компьютерная модель, например нейронная сеть, с помощью текстового корпуса обучается генерации новых текстов. Например, можно сгенерировать совершенно новый научно-фантастический рассказ, используя для обучения модели существующие рассказы.
- **Вопросно-ответные системы** (question answering, QA). Технологии вопросно-ответных систем имеют высокую коммерческую ценность и лежат в основе чат-ботов и виртуальных помощников (например, Google Assistant и Apple Siri). Чат-боты широко используются для ответов на вопросы и решения простых проблем клиентов (например, изменения тарифного плана мобильной связи), которые могут быть выполнены без вмешательства человека. Реализация QA-систем охватывает обширные аспекты NLP, такие как поиск информации и представление знаний. Разработка полноценных QA-систем – это сложный и дорогостоящий процесс.
- **Машинный перевод** (machine translation, MT) – это задача преобразования предложения/фразы из исходного языка (например, немецкого) в целевой язык (например, английский). Это очень сложная задача, поскольку разные языки имеют очень разные морфологические структуры, следовательно, это не взаимно однозначное преобразование. Кроме того, межсловные отношения между языками могут строиться по схеме один-ко-многим, один-к-одному, многие-к-одному или многие-ко-многим. В публикациях про MT это принято называть *задачей выравнивания слов* (word alignment problem).

Наконец, в прикладной системе, помогающей человеку в повседневных делах (например, речевой помощник или чат-бот), многие из этих задач должны выполняться совместно. Как мы видели в предыдущем примере, если пользователь просит: «*Покажи мне хороший итальянский ресторан поблизости*», необходимо выполнить несколько различных задач NLP, таких как преобразование устной речи в текст, семантический анализ и анализ настроений, ответы на вопросы и машин-

ный перевод. На рис. 1.1 представлена таксономия задач NLP. Набор задач делится на две широкие категории: *анализ* существующего текста и *генерация* нового текста. В свою очередь, анализ разделяется на три подкатегории: *синтаксический* (задачи, основанные на структуре языка), *семантический* (задачи, основанные на значении) и *прагматический* (открытые проблемы, которые трудно решить):

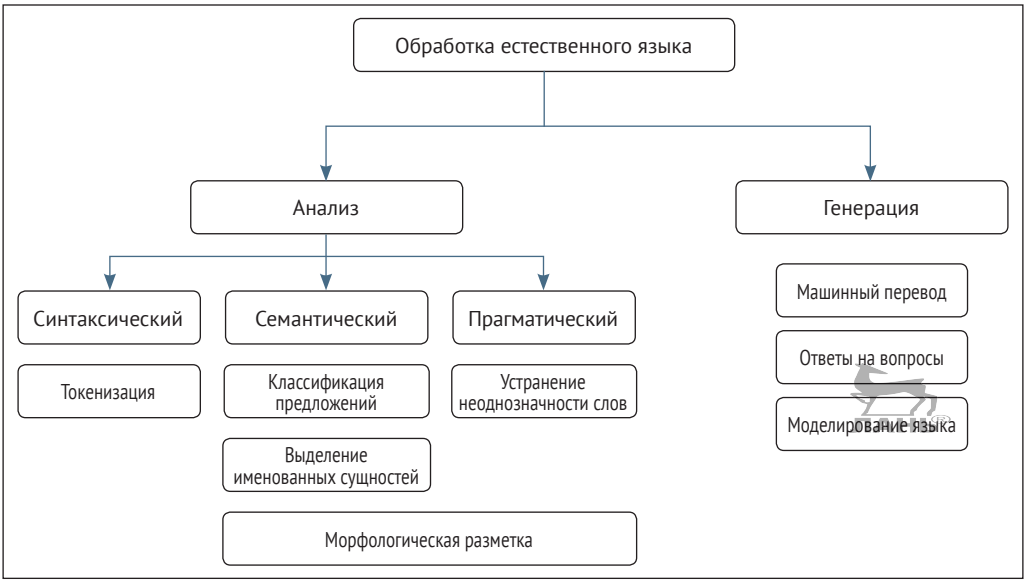


Рис. 1.1 ❖ Таксономия популярных задач NLP на основе наиболее обширных категорий

Разобравшись с типами задач в NLP, давайте теперь перейдем к обсуждению того, как мы можем решать эти задачи с помощью машин.

Традиционный подход к обработке естественного языка

Традиционный подход к NLP основан на статистике и представляет собой последовательность из нескольких ключевых этапов. Фактически это самостоятельные задачи – предварительная обработка, конструирование признаков, обучение модели с помощью обучающих данных и прогнозирование с незнакомыми данными. Наиболее трудоемким и решающим шагом, от которого зависит эффективность и качество NLP, является конструирование признаков.

Подробности традиционного подхода

Традиционный подход к решению задач NLP включает в себя набор отдельных подзадач. Во-первых, текстовые корпусы должны быть предварительно обработаны с целью сокращения *словарного запаса* (vocabulary) и удаления *помех* (distractions). Под помехами я подразумеваю вещи (например, знаки пунктуации

и *стоп-слова*[1]), которые отвлекают алгоритм от сбора важной лингвистической информации, необходимой для выполнения задачи.

Далее следует несколько этапов *конструирования признаков* (feature engineering). Основная задача конструирования признаков – облегчить обучение алгоритмов. Часто эти признаки создаются вручную и смещены в сторону человеческого понимания языка. Конструирование признаков имеет огромное значение для классических алгоритмов NLP, и, следовательно, наиболее эффективные системы обычно имеют более проработанные наборы признаков. Например, для задачи классификации настроений можно представить предложение с помощью *дерева синтаксического разбора* (parse tree) и назначить положительные, отрицательные или нейтральные метки каждому узлу/поддереву в дереве, чтобы классифицировать это предложение как положительное или отрицательное. Кроме того, для конструирования более совершенных признаков можно использовать внешние ресурсы, такие как WordNet (лексическая база данных). Вскоре мы рассмотрим простую технику конструирования признаков, известную как *мультимножество слов* (bag-of-words, BOW).

Затем алгоритм учится хорошо выполнять поставленную задачу, используя полученные признаки и, при необходимости, внешние ресурсы. Например, для задачи *обобщения текста* (text summarization) полезным внешним ресурсом может служить тезаурус, содержащий синонимы слов. И наконец, выполняется прогнозирование: вы берете входные данные, подаете их на вход обученной модели и получаете выходные данные – прогноз. Традиционный подход в общем виде изображен на рис. 1.2.

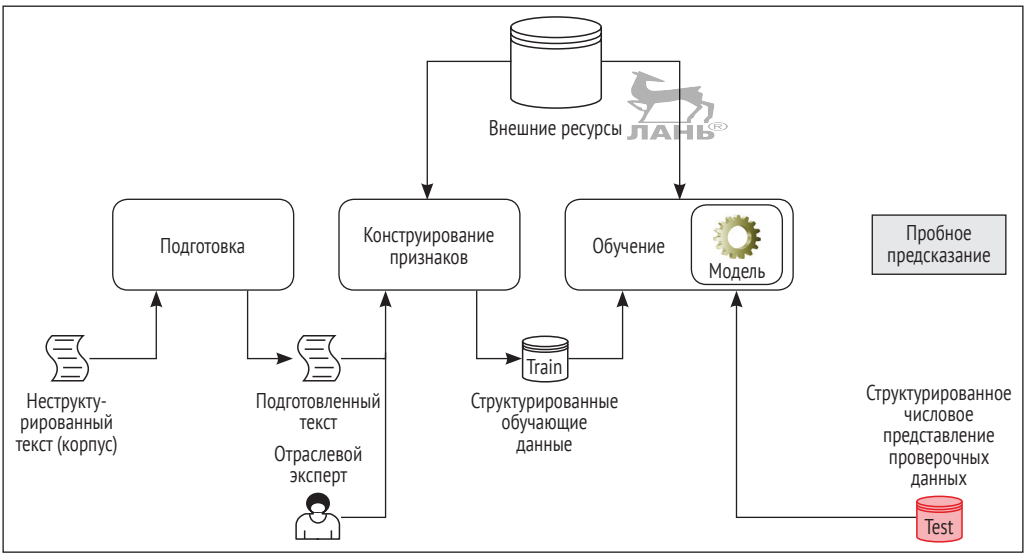


Рис. 1.2 ❖ Традиционный подход к задаче NLP

[1] Стоп-слова – это лексические единицы текста, лишенные смысловой нагрузки (вводные слова, предлоги, междометия и т. д.). – *Прим. перев.*

Пример: создание обзора футбольной игры

Чтобы глубже понять традиционный подход к NLP, рассмотрим задачу автоматической генерации текста на основе статистики футбольного матча. У нас есть несколько наборов показателей игры (например, счет, пенальти и желтые карточки) и соответствующие статьи, написанные для этой игры журналистом, в качестве обучающих данных. Давайте также предположим, что для данной игры у нас есть сопоставление каждого показателя с наиболее релевантной фразой статьи. Наша задача заключается в том, чтобы на основе показателей новой игры сгенерировать обзорную статью об этой игре на естественном языке. Конечно, при наличии достаточно обширного обучающего набора задачу можно решить в лоб: найти для новой игры наиболее похожие показатели в обучающих данных и взять оттуда соответствующую готовую статью. Но есть более сложные и элегантные способы генерации текста.

Если для генерации статьи на естественном языке мы используем машинное обучение, то наверняка будем последовательно выполнять предварительную обработку текста, токенизацию, конструирование признаков, обучение и прогнозирование.

Предварительная обработка текста включает в себя такие операции, как *стемминг*[1] (stemming) и удаление знаков препинания, чтобы уменьшить словарный запас (то есть количество признаков), тем самым уменьшив требования к памяти. Может показаться, что стемминг – это примитивная операция, основанная на простом наборе правил, таких как удаление приставок, суффиксов и окончаний (например, стемминг слова «*прослушивание*» дает стемму «*слуш*»); тем не менее для хорошего алгоритма стемминга требуется нечто большее, чем простая база правил, поскольку иногда правило не очевидно. Допустим, попробуйте из «*наличия весомой аргументации*» алгоритмически получить стемму «*аргумент*». Кроме того, усилия, необходимые для правильного нахождения стеммы, могут различаться по сложности в зависимости от языка.

Токенизация – это процесс разделения корпуса на мелкие объекты, например слова. Токенизация выглядит тривиальной для такого языка, как английский, поскольку слова изолированы; однако это не относится к таким языкам, как тайский, японский и китайский, поскольку в них нет разделения на слова.

Конструирование признаков применяется для преобразования необработанных текстовых данных в оговоренный числовой формат, чтобы можно было обучить модель на этих данных, например преобразовать текст в мультимножество слов или использовать представление в виде n-граммы, которое мы обсудим позже. Но учтите, что современные классические модели основаны на гораздо более сложных приемах конструирования признаков. Давайте рассмотрим несколько примеров.

Мультимножество слов – это методика конструирования признаков, которая создает представления элементов на основе частоты появления слов. Допустим, у нас есть два предложения:

- *Боб пошел на рынок, чтобы купить цветы.*
- *Боб купил цветы, чтобы подарить Мэри.*

---

[1] Стемминг – это нахождение основы слова (стеммы), передающей его лексическое значение. – *Прим. перев.*

На основе этих предложений мы можем составить следующий словарь:

[«Боб», «пошел», «на», «рынок», «чтобы», «купить», «цветы», «купил», «подарить», «Мэри»]

Далее мы создадим вектор признаков с размерностью *V* (размер словаря) для каждого предложения, показывающий, сколько раз каждое словарное слово появляется в предложении. В этом примере векторы признаков для предложений будут следующими:

[1, 1, 1, 1, 1, 1, 1, 0, 0, 0]
[1, 0, 0, 0, 1, 0, 0, 1, 1, 1]

К сожалению, в данном методе теряется контекстная информация о порядке слов.

n-грамма – это еще одна техника конструирования признаков, которая разбивает текст на более мелкие компоненты, состоящие из *n* букв или слов. Например, 2-грамма разбивает текст на компоненты, состоящие из двух букв или двух слов. Давайте разложим на 2-граммы знакомое предложение

Боб пошел на рынок, чтобы купить цветы.

Разложение на 2-граммы уровня букв для этого предложения выглядит следующим образом:

[«Бо», «об», «б «, « п», «по», «ош», …, «ит», «ть», «ь «, « ц», «цв», «ве», «ет», «ты»]

Основанное на словах разложение на 2-граммы выглядит так:

[«Боб пошел», «пошел на», «на рынок», «рынок чтобы», «чтобы купить», «купить цветы»]

Преимущество *n*-грамм низкого уровня заключается в том, что словарный запас для больших корпусов будет значительно меньше, чем при использовании слов в качестве признаков.

Затем нам нужно структурировать наши данные, чтобы иметь возможность использовать их для обучения модели. Например, у нас будут кортежи данных вида (`статистика`, `поясняющая фраза`):

(Общее количество голов = 4, Каждая команда забила по два гола в конце первого тайма)

(Команда 1 = Манчестер Юнайтед, Игра между командами Манчестер Юнайтед и Барселона)

(Голы команды 1 = 5, Манчестер Юнайтед сумели забить 5 голов)

Процесс обучения может состоять из трех блоков: *скрытая марковская модель* (hidden Markov model, HMM), *планировщик фразы* (sentence planner) и *планировщик дискурса* (discourse planner). В нашем примере HMM изучает морфологическую структуру и грамматические свойства языка, анализируя совокупность связанных

фраз. Сначала мы изучаем каждую фразу в нашем наборе данных и формируем пары, где первым элементом идет статистика, а за ней следует поясняющая фраза. Затем мы обучаем НММ, попросив его предсказать следующее слово с учетом текущей последовательности. Сначала мы вводим статистику в НММ, а затем получаем прогноз. Далее объединяем последний прогноз с текущей последовательностью и просим НММ дать следующий прогноз и т. д. Таким образом, НММ выводит длинные значимые фразы с учетом статистики.

Далее, у нас может следовать планировщик фраз, исправляющий любые лингвистические ошибки, которые могут встретиться во фразах. Например, планировщик заменяет фразу «*я идти в дом*» фразой «*Я иду домой*». Он может использовать базу данных правил, описывающих правильные способы передачи значений (например, отсутствие предлога между глаголом «*иду*» и существительным «*домой*»).

Теперь мы можем сгенерировать набор фраз для данного набора статистики, используя НММ. Затем нам нужно объединить эти фразы таким образом, чтобы обзор, составленный из набора фраз, был удобочитаемым и правильно передавал ход событий. Допустим, на выходе планировщика фраз мы получили три предложения:

1. Игрок номер 10 команды «Барселона» забил гол во втором тайме.
2. «Барселона» сыграла с «Манчестер Юнайтед».
3. Игрок номер 3 из «Манчестер Юнайтед» получил желтую карточку в первом тайме.

Это лингвистически правильные предложения, но их порядок явно не соответствует логике текста. Намного лучше, когда предложения расположены в таком порядке:

«Барселона» сыграла с «Манчестер Юнайтед». В первом тайме игрок номер 3 из «Манчестер Юнайтед» получил желтую карточку, а во втором тайме игрок номер 10 из «Барселоны» забил гол.

Для упорядочивания и структурирования набора предложений мы используем планировщик дискурса.

Теперь мы можем взять набор произвольных статистик и получить текст обзора футбольного матча, пошагово выполняя рабочий процесс (рис. 1.3).

Имейте в виду, что это объяснение очень высокого уровня, охватывающее только основные компоненты общего назначения, которые, скорее всего, будут включены в традиционный подход к решению задачи NLP. Детали могут существенно различаться в зависимости от конкретной задачи. Для некоторых задач могут потребоваться дополнительные критически важные компоненты, например база правил и модель выравнивания в машинном переводе. Мы не будем углубляться в такие подробности, потому что цель этой книги – рассказать о более современных способах обработки естественного языка.

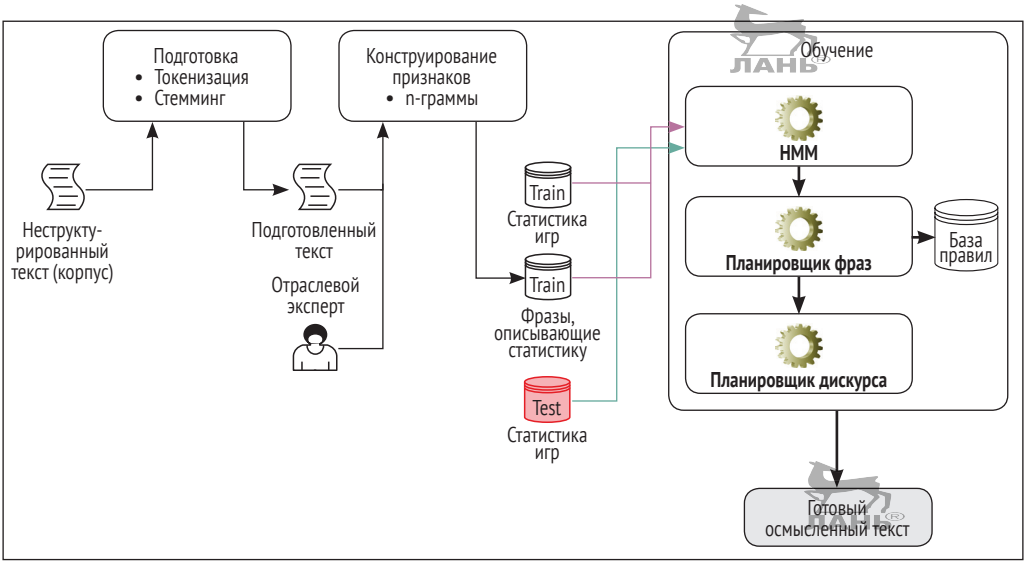


Рис. 1.3 ❖ Этап из примера классического подхода к решению задачи моделирования языка

Недостатки традиционного подхода

Давайте перечислим несколько ключевых недостатков традиционного подхода, поскольку они послужат хорошей основой для понимания причин перехода к глубокому обучению:

- необходимость предварительной обработки текста в традиционном подходе NLP вынуждает искать компромисс между размером словарного запаса и потенциально полезной информацией, встроенной в текст (например, пунктуацией и эмоциональной окраской). Хотя предварительная обработка все еще используется в современных решениях, основанных на глубоком обучении, она не столь важна благодаря большой репрезентативной способности глубоких сетей;
- конструирование признаков должно выполняться вручную. Чтобы получить качественную систему, необходимо сконструировать хорошие признаки. Этот процесс может быть очень трудоемким, так как необходимо тщательно исследовать различные пространства признаков. Кроме того, для эффективного изучения надежных признаков требуется экспертиза предметной области, которая доступна не для всех задач NLP;
- для нормальной работы метода требуются различные внешние ресурсы, но свободно доступных ресурсов не так уж много. Такие внешние ресурсы зачастую состоят из подготовленной вручную информации, хранящейся в больших базах данных. Создание базы данных (например, базы правил машинного перевода) может занять несколько лет, в зависимости от сложности задачи.

Революция глубокого обучения в обработке естественного языка

Думаю, никто не станет спорить, что глубокое обучение произвело революцию в машинном обучении, особенно в таких областях, как компьютерное зрение, распознавание речи и, конечно, NLP. Глубокое обучение вызвало волну смены парадигм во многих областях машинного обучения, потому что глубокие модели умеют самостоятельно извлекать мощные признаки из необработанных данных вместо использования ограниченных искусственных признаков. Это привело к тому, что утомительное и затратное ручное конструирование признаков устарело. Глубокие модели повысили эффективность рабочего процесса, поскольку они одновременно извлекают признаки и учатся решать задачу. Более того, благодаря огромному количеству параметров (т. е. весов) в глубокой модели она может учитывать значительно больше признаков, чем мог бы придумать человек. Однако глубокие модели считаются «черным ящиком» из-за плохой интерпретируемости результатов. Например, понимание того, *как* и *что* использует глубокая модель для решения определенной задачи, все еще остается открытой проблемой.

Глубокая модель – это, по сути, искусственная нейронная сеть, имеющая входной слой, множество взаимосвязанных скрытых слоев в середине и, наконец, выходной слой (например, классификатор или регрессор). Как видите, получается *сквозная модель* (end-to-end model), охватывающая весь процесс, от необработанных данных до прогнозов. Скрытые слои в середине – это сердце глубокой модели, поскольку они отвечают за извлечение полезных признаков из необработанных данных, что в конечном итоге приводит к выполнению поставленной задачи.

История глубокого обучения

Давайте кратко обсудим происхождение глубокого обучения и то, как эта область превратилась в очень многообещающую технологию. В 1960 году Хьюбел (Hubel) и Визель (Wiesel) провели интересный эксперимент и обнаружили, что зрительная кора кошки состоит из простых и сложных клеток и что эти клетки организованы в иерархической форме. Также эти клетки по-разному реагируют на разные раздражители. Например, простые клетки срабатывают в ответ на простые визуальные сигналы, такие как ориентация границ, в то время как сложные клетки не чувствительны к пространственной ориентации объекта. Это открытие разожгло интерес ученых к имитации подобного поведения в машинах, породив концепцию глубокого обучения.

В последующие годы нейронные сети привлекли внимание многих исследователей. В 1965 году Ивахненко и его коллеги представили нейронную сеть, обученную по *методу группового учета аргументов* (group method of data handling, GMDH), основанному на знаменитом *персептроне Розенблатта*. Позже, в 1979 году, Фукусима (Fukushima) представил *неокогнитрон* (neocognitron), который лег в основу одного из самых известных вариантов глубоких моделей – сверточной нейронной сети. В отличие от персептронов, которые всегда обладают одномерным входом, неокогнитрон мог обрабатывать двумерные входы, используя операции свертки.

Искусственные нейронные сети используют алгоритм *обратного распространения ошибки* для оптимизации весовых коэффициентов сети путем вычисления якобиана предшествующего слоя. Кроме того, проблема *исчезающих градиентов* (vanishing gradient) жестко ограничивает потенциальную *глубину*, т. е. количество слоев нейронной сети. Чем ближе слой расположен ко входу, тем меньше становится его градиент, т. е. изменение весов в процессе распространения ошибки, что известно как явление исчезающего градиента. Это происходит по причине цепного вычисления градиентов весов нижних слоев путем последовательного перемножения малых величин.

Затем в 2006 году было обнаружено, что предварительное обучение глубокой нейронной сети на сжатых и нормализованных данных для каждого слоя сети по отдельности обеспечивает хорошие начальные значения весовых коэффициентов, особенно для начальных слоев, что позволяет, по сути, устранить эффект исчезающего градиента. Кроме того, эти более глубокие модели смогли превзойти традиционные модели машинного обучения во многих задачах, в основном в компьютерном зрении. Например, точность распознавания стандартного набора рукописных цифр MNIST близка к 100 %. После этого прорыва глубокое обучение стало модным словом в сообществе машинного обучения.

Глубокие нейросети начали резко набирать обороты, когда в 2012 году Алекс Крижевский (**http://www.cs.toronto.edu/~kriz/**), Илья Суцкевер (**http://www.cs.toronto.edu/~ilya/**) и Джефф Хинтон (Geoff Hinton) представили глубокую сверточную нейросеть AlexNet и выиграли соревнование по визуальному распознаванию изображений 2012 года, уменьшив ошибку на 10 % по сравнению с предыдущим победителем. Примерно в это же время глубокие нейронные сети достигли современного уровня точности распознавания речи. Кроме того, разработчики нейросетей обнаружили, что *графические процессоры* (graphical processor unit, GPU) за счет своей архитектуры обеспечивают отличную параллельность вычислений по сравнению с *центральными процессорами* (central, processor unit, CPU), что позволяет быстрее обучать большие и более глубокие сети.

Глубокие модели были дополнительно улучшены с помощью более совершенных методов инициализации, например *инициализации Завьера* (Xavier), что делает ненужным длительное предварительное обучение. Кроме того, были предложены более подходящие нелинейные функции активации, такие как *выпрямленная линейная функция* (rectified linear unit, ReLU), которые ослабили вредный эффект исчезающего градиента в более глубоких моделях. Более эффективные методы обучения, такие как Adam, автоматически подстраивают индивидуальные скорости обучения каждого параметра среди миллионов параметров, которые мы имеем в модели нейронной сети, что радикально увеличивает производительность нейросетей в таких ресурсоемких областях, как классификация объектов и распознавание речи. Благодаря этим улучшениям удалось увеличить количество скрытых слоев, т. е. сделать нейросети более глубокими. Глубина нейросети является одним из основных факторов, определяющих значительно более высокую точность глубоких моделей по сравнению с другими моделями машинного обучения. Кроме того, улучшенные промежуточные нормализаторы, такие как *слои пакетной нормализации* (batch normalization layer), увеличили производительность глубоких сетей для многих задач.

Позже были представлены феноменально глубокие модели, такие как ResNet, Highway Net и Ladder Net, которые имели сотни слоев и миллиарды параметров.

Это стало возможным благодаря оригинальному сочетанию результатов теоретических и экспериментальных исследований. Например, ResNet использует механизм ссылок для соединения слоев, расположенных далеко друг от друга, что сводит к минимуму послойное уменьшение градиентов, о котором говорилось ранее.

Современное состояние глубокого обучения и NLP

Много разных глубоких моделей увидело свет с момента их появления в начале 2000 года. Несмотря на то что все они используют нелинейное преобразование входных данных и параметров, детали реализации могут сильно различаться. Например, *сверточные нейронные сети* (convolution neural network, CNN) могут извлекать признаки непосредственно из двумерных данных (например, изображений RGB), в то время как многослойный персептрон требует, чтобы входные данные были развернуты в одномерный вектор, что приводит к потере важной пространственной информации.

Понимание текста заключается в том, чтобы уметь интерпретировать его как последовательность символов. Поэтому глубокая модель должна уметь выполнять моделирование временных рядов, что требует памяти прошлых состояний. Чтобы понять это, подумайте о задаче моделирования языка: слову «*кошка*» и слову «*самолет*» обычно предшествуют разные слова. Одна из наиболее популярных моделей, обладающих способностью памяти, известна как *рекуррентная нейронная сеть* (recurrent neural network, RNN). В главе 6 вы увидите, каким образом RNN добиваются этого, выполняя операции с обратной связью.

Следует подчеркнуть, что память – это весьма нетривиальная опция глубокой модели. Способы реализации памяти должны быть тщательно продуманы. Кроме того, термин «память» не следует путать с сохранением весов *одношаговой* глубокой сети, которая смотрит только на текущий вход, тогда как *последовательная* глубокая модель (например, RNN) будет смотреть как на сохраненные веса, так и на предыдущий элемент входной последовательности для прогнозирования следующего вывода.

Одним из существенных недостатков RNN является то, что они могут запомнить лишь несколько – около семи – временных шагов, поэтому им не хватает долговременной памяти. Сети с *долгой краткосрочной памятью* (long short-term memory, LSTM) являются расширением сетей RNN, которые инкапсулируют долговременную память. Поэтому в настоящее время сети с LSTM зачастую предпочтительнее стандартных RNN. Мы обсудим подробности в главе 7.

Таким образом, мы можем разделить глубокие сети на две основные категории: одношаговые сети, которые имеют дело только с одним входом в один момент времени для обучения и прогнозирования (например, классификация изображений), и последовательные сети, которые работают с последовательностями входов произвольной длины (например, генерация текста, где одно слово является одним входом). Затем мы можем разделить одношаговые сети, также называемые сетями *прямого распространения*, на глубокие – приблизительно около 20 слоев, и очень глубокие сети – более чем сотни слоев. Последовательные сети подразделяются на модели кратковременной памяти (например, RNN), которые могут запоминать только краткосрочные паттерны, и модели долговременной памяти, которые могут запоминать более длинные паттерны. На рис. 1.4 схематически изображена обсуждаемая таксономия. На данном этапе от вас не ждут полного

понимания всех этих моделей глубокого обучения. Пока это лишь иллюстрация разнообразия моделей.

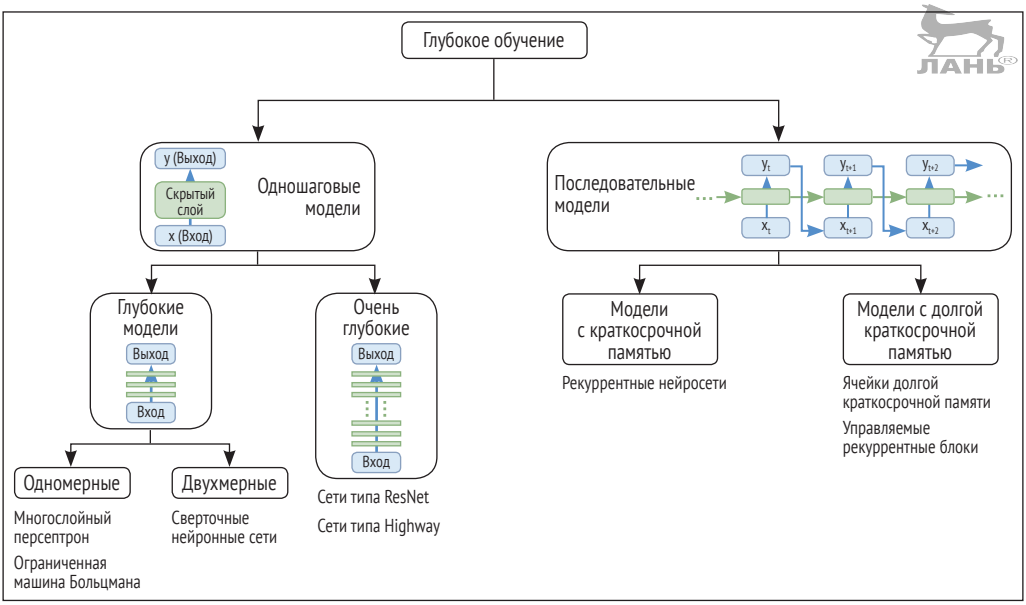


Рис. 1.4 ❖ Общая таксономия наиболее часто используемых методов глубокого обучения

Устройство простой глубокой модели – полносвязной нейронной сети

Теперь давайте подробнее рассмотрим устройство глубокой нейронной сети. Хотя существует множество различных вариантов глубоких сетей, рассмотрим одну из самых ранних моделей 1950–1960 гг., известную как *полносвязная нейронная сеть* (fully connected neural network, FCNN), или *многослойный персептрон*. На рис. 1.5 изображена стандартная трехслойная нейросеть.

Цель FCNN – выполнить классификацию, то есть сопоставить входные данные (например, изображение или предложение) с определенной меткой или аннотацией, например категорией объекта на картинке. Сначала вычисляют h – скрытое представление ввода x с помощью преобразования $h = sigma(W*x + b)$, где W и b – веса и смещение FCNN соответственно, а *sigma* – сигмоидная функция активации. Далее поверх слоев помещается классификатор, изучающий функции скрытых слоев и выполняющий классификацию входных данных. Классификатор, по сути, является частью FCNN и еще одним скрытым слоем с некоторыми весами W_s и смещением b_s. Кроме того, мы можем рассчитать конечный выход FCNN как $output = softmax(W_s*h + b_s)$. Например, классификатор *softmax* обеспечивает нормализованное представление результатов на уровне классификатора. Меткой считается выходной узел с наибольшим значением *softmax*. Получив результат, мы можем определить *ошибку классификации*, которая рассчитывается как разница между прогнозируемой меткой и фактической меткой. Примером такой функ-

ции ошибки является среднеквадратичная ошибка. Вам не нужно беспокоиться, если вы незнакомы с функциями ошибок. Мы обсудим их в следующих главах. Затем параметры нейронной сети W, b, W_s и b_s оптимизируют с использованием стандартного стохастического оптимизатора, например стохастического градиентного спуска, чтобы уменьшить потери классификации всех входных данных. На рис. 1.5 изображен рабочий процесс для трехслойной полносвязной сети. Мы подробно разберем применение такой модели для задач NLP в главе 3.

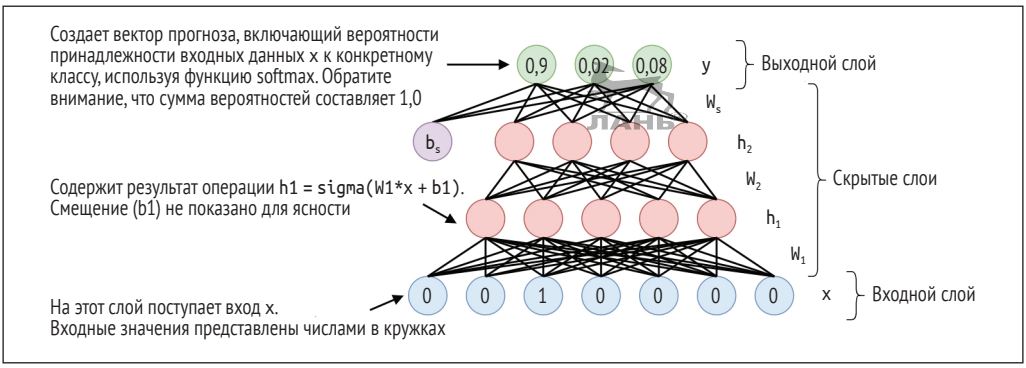


Рис. 1.5 ❖ Пример рабочего процесса полносвязной нейронной сети (FCNN)

Давайте рассмотрим пример использования нейронной сети для задачи *анализа настроений* (sentiment analysis). Предположим, что у нас есть обучающий набор данных в виде предложений на естественном языке, выражающих положительное или отрицательное мнение о фильме. Предложения снабжены метками, означающими, является отзыв положительным (1) или отрицательным (0). Затем нам дают проверочный набор данных, в котором есть отзывы на фильмы в одном предложении, и наша задача – классифицировать эти новые предложения как положительные или отрицательные.

Для этой задачи можно использовать нейронную сеть (которая может быть глубокой или неглубокой в зависимости от сложности задачи), придерживаясь следующего рабочего процесса:

1) токенизируем предложения по словам;
2) при необходимости добавляем к предложениям специальный токен, чтобы привести все предложения к одинаковой длине;
3) переводим предложения в числовое представление, например Bag-of-Words;
4) подаем числовые входы в нейронную сеть и прогнозируем выход (положительный или отрицательный);
5) вычисляем функцию потерь и оптимизируем нейронную сеть.

Что вы найдете дальше в этой книге?

В этом разделе кратко описано содержание остальной части книги. Мы рассмотрим многочисленные интересные области NLP, от алгоритмов, которые находят сходство слов, не используя аннотированные данные, до алгоритмов, которые могут самостоятельно сочинить сказку.

Начиная со следующей главы, мы углубимся в детали нескольких популярных и интересных задач NLP. Чтобы получить глубокие знания и сделать обучение интерактивным, предусмотрены различные упражнения. Мы будем использовать Python и TensorFlow – библиотеку с открытым исходным кодом для распределенных вычислений. TensorFlow воплощает передовые технические решения, такие как оптимизация вашего кода для графических процессоров с использованием технологии Compute Unified Device Architecture (CUDA), что само по себе непросто. Кроме того, TensorFlow предоставляет встроенные функции для реализации алгоритмов глубокого обучения, например активаций, методов стохастической оптимизации и сверток, облегчающие жизнь разработчика.

Итак, мы начинаем путешествие, которое охватывает многие актуальные темы NLP и демонстрирует реализацию самых современных алгоритмов при помощи TensorFlow. Вот что вы найдете дальше в этой книге.

- *Глава 2* дает вам общее понимание того, как писать клиентские программы и запускать их в TensorFlow. Это важно, особенно если вы новичок в TensorFlow, потому что TensorFlow ведет себя иначе, чем традиционный язык программирования, такой как Python. Сначала мы подробно разберем, как в окружении TensorFlow выполняется прикладной код – так называемый *клиент*. Это поможет вам понять рабочий процесс выполнения клиента TensorFlow и чувствовать себя комфортно в новой терминологии. Далее в этой главе вы познакомитесь с различными элементами клиента TensorFlow, такими как определение переменных, определение операций/функций, ввод входных данных в алгоритм и получение результатов. Наконец, мы используем полученные знания на практике и реализуем умеренно сложную нейронную сеть для классификации рукописных изображений.
- *Глава 3* представляет Word2vec – эффективный метод числового представления слов в пространстве смыслов. Но прежде чем углубиться в изучение Word2vec, мы обсудим некоторые классические подходы, используемые для представления семантики слов. Одним из первых подходов было использование WordNet – большой лексической базы данных. WordNet можно применять для измерения семантического сходства между разными словами. Однако поддержание такой большой лексической базы данных является дорогостоящей работой. К счастью, существуют более простые методы представления семантики, не зависящие от внешних ресурсов. Мы рассмотрим эти методы. Затем мы перейдем к современному способу векторного представления слов, известному как Word2vec, где используем нейронную сеть для изучения представлений. Мы обсудим две популярные методики Word2vec: *словосочетания с пропуском* (skip-gram) и *непрерывное мультимножество слов* (continuous bag-of-words, CBOW).
- *Глава 4* начинается с нескольких сравнений, в том числе между алгоритмами skip-gram и CBOW, чтобы узнать, есть ли явный победитель. Затем мы обсудим несколько дополнений, которые были введены в оригинальную технику Word2vec в течение последних нескольких лет. Например, игнорирование распространенных в тексте слов, таких как артикли *the* и *a*, с высокой вероятностью повышает качество моделей Word2vec. С другой стороны, модель Word2vec учитывает только локальный контекст слова и игнорирует глобальную статистику всего корпуса. Отсюда мы приходим к методике

GloVe, которая учитывает как локальную, так и глобальную статистику при поиске векторов слов.

- *Глава 5* знакомит вас со сверточными нейронными сетями (CNN). Сверточные сети – это мощное семейство глубоких моделей, учитывающих пространственную структуру ввода при изучении данных. Другими словами, CNN может обрабатывать непосредственно двумерные изображения, когда многослойному персептрону необходимо развернуть изображение в одномерный вектор. Сначала мы подробно обсудим различные базовые операции CNN, такие как операции свертки и объединения. Затем разберем пример классификации рукописных цифровых изображений с помощью CNN. Далее перейдем к применению сверточных сетей в NLP. Точнее, мы рассмотрим применение CNN для задачи классификации предложений, когда необходимо определить, относится ли предложение к человеку, местоположению, объекту и т. д.
- *Глава 6* посвящена изучению рекуррентных нейронных сетей (RNN) и использованию RNN для генерации естественного языка. Отличительной особенностью рекуррентных сетей является наличие памяти. Память хранится как постоянно обновляемое состояние системы. Мы начнем с представления нейронной сети с прямым распространением и перестроим это представление для изучения последовательных данных. Таким образом, мы превратим сеть прямого распространения в RNN. За этим последует детальное описание уравнений, используемых для вычислений в RNN. Далее мы обсудим процесс обучения RNN и обзорно пройдемся по различным типам RNN, таким как одноранговые и одноконтурные сети. Затем разберем увлекательный пример и научим рекуррентную нейросеть рассказывать новые сказки, извлекая уроки из совокупности существующих сказок. Мы добьемся этого, обучая RNN предсказывать следующее слово с учетом предыдущей последовательности слов сказки. Наконец, обсудим модифицированный вариант нейросети под названием RNN-CF (RNN с контекстными функциями) и сравним его со стандартной RNN, чтобы увидеть, какой вариант работает лучше.
- *В главе 7* мы обсуждаем сети с долгой краткосрочной памятью (LSTM), даем четкое представление о том, как работают эти модели, и постепенно углубляемся в технические детали, достаточные для их самостоятельной реализации. Стандартные рекуррентные сети страдают от невозможности сохранения долговременной памяти. Однако существуют усовершенствованные модели RNN, например ячейки с долгой кратковременной памятью и управляемые рекуррентные блоки (GRU), которые могут запоминать последовательности на большое количество тактов. Мы рассмотрим, как именно LSTM облегчает задачу сохранения долговременной памяти (это называется проблемой исчезающего градиента). Затем обсудим несколько идей для дальнейшего улучшения моделей LSTM, таких как прогнозирование на несколько шагов вперед и чтение последовательностей как вперед, так и назад. Наконец, обсудим несколько вариантов моделей LSTM.
- *Глава 8* рассказывает о практической реализации сетей с долгой краткосрочной памятью. Кроме того, сравним как качественно, так и количественно производительность различных вариантов. Мы также обсудим, как реализовать некоторые расширения, рассмотренные в главе 7, такие как

прогнозирование на несколько шагов вперед (*лучевой поиск,* beam search) и использование векторов слов в качестве входных данных вместо прямого унитарного кодирования. Наконец, обсудим использование API RNN – вспомогательной библиотеки TensorFlow для облегчения реализации рекуррентных моделей.

- *Глава 9* рассказывает про другое интересное приложение, где модель учится генерировать подписи к рисункам, используя LSTM и CNN. Это приложение интересно тем, что показывает нам, как комбинировать модели двух разных типов, а также как распознавать *мультимодальные данные* (например, изображения и текст). Конкретный подход заключается в следующем: сначала получают вектор представления изображения (аналогично векторам слов) с помощью CNN и обучают LSTM, подавая ей на вход вектор изображения, а затем слова описания изображения в виде последовательности. Сначала вы узнаете, как использовать предварительно обученную сверточную сеть для получения представлений изображения. Затем мы обсудим, как получить представление слов. Далее разберемся, как применять векторы изображений вместе с векторами слов для обучения LSTM. Мы перечислим различные метрики оценки систем генерации подписей к изображениям. После этого сможем оценивать подписи, сгенерированные нашей моделью, как качественно, так и количественно. Главу завершает инструкция по внедрению той же системы с помощью RNN API TensorFlow.
- *Глава 10* рассказывает про обучение преобразованию последовательностей и машинный перевод. Машинный перевод привлекает исследователей и разработчиков массовой потребностью людей в автоматическом переводе и сложностью задачи. Мы начнем главу с краткого воспоминания о том, с чего начиналась история машинного перевода, и продолжим введением в системы *нейронного машинного перевода* (neural machine translation, NMT). Мы увидим, насколько хорошо работают современные системы NMT по сравнению со старыми системами статистического машинного перевода. После этого обсудим идею, лежащую в основе устройства систем NMT, и углубимся в технические детали. Затем выберем метрику оценки нашей системы и рассмотрим, как можно реализовать переводчик с немецкого на английский с нуля. Далее вы узнаете о способах улучшения систем NMT. Мы подробно рассмотрим один из подходов, называемый *механизмом внимания* (attention mechanism). Без механизма внимания не обойтись при обучении модели преобразованию последовательности в последовательность. Наконец, мы сравним полученные результаты и проанализируем причины повышения качества при использовании механизма внимания. Главу завершает раздел о том, как можно развить концепцию систем NMT для реализации чат-ботов. Чат-боты – это системы, которые общаются с людьми на естественном языке и помогают выполнять различные запросы пользователей.
- *Глава 11* рассказывает о современных тенденциях и будущем обработки естественного языка. Обработка естественного языка охватывает широкий спектр различных задач. Мы обсудим некоторые текущие тенденции и предположения о том, чего можно ожидать от NLP в будущем. Сначала обсудим различные расширения для обучения смысловой векторизации слов, появившиеся недавно. Мы также рассмотрим реализацию одной из таких

техник обучения векторизации, известную как TV-embedding. Далее разберем различные направления развития в области нейронного машинного перевода. Затем обсудим, как NLP сочетается с другими областями, такими как компьютерное зрение и обучение с подкреплением, для решения некоторых любопытных проблем, таких как обучение компьютерных агентов общению путем разработки собственного языка. В наши дни еще одной важной областью исследований является *общий искусственный интеллект*, связанный с разработкой систем, способных выполнять несколько задач (классифицировать изображения, переводить текст, генерировать подписи к изображениям и т. д.) в рамках одной системы. Мы исследуем несколько таких систем. Затем поговорим о внедрении NLP в майнинг социальных сетей. Мы завершим эту главу примерами некоторых новых задач, например языковым обоснованием – разработкой систем NLP на основе здравого смысла, и новыми моделями, такими как синхронизированные LSTM.

- *Приложение* познакомит читателя с различными математическими структурами данных и операциями (например, с обращением матриц). Мы также обсудим несколько важных понятий теории вероятностей. Затем мы представим Keras – высокоуровневую библиотеку, которая использует TensorFlow. Keras упрощает внедрение нейронных сетей, скрывая некоторые детали реализации TensorFlow, что может показаться достаточно запутанным. Вы познакомитесь с примером реализации сверточной сети с помощью Keras. Далее мы обсудим, как использовать библиотеку seq2seq в TensorFlow для реализации системы машинного перевода с гораздо меньшим количеством кода, чем тот, который вы использовали в главе 11. Наконец, вы познакомитесь с руководством по использованию TensorBoard для визуализации смысловых связей слов. TensorBoard – это удобный инструмент визуализации, который поставляется с TensorFlow. Его можно использовать для визуализации и мониторинга различных переменных в вашем клиенте TensorFlow.

Знакомство с рабочими инструментами

В этом разделе вы познакомитесь с техническими средствами, которые будут задействованы в упражнениях следующих глав. Сначала мы представим обзор основных инструментов. Далее дадим краткие инструкции по установке каждого инструмента вместе с гиперссылками на подробные руководства, предоставленные официальными сайтами. Кроме того, мы поделимся советами о том, как убедиться, что инструменты были установлены правильно.

Обзор основных инструментов

Для написания кода программ мы будем использовать Python. Это очень универсальный, легко настраиваемый язык программирования, который активно используется научным сообществом. Кроме того, вокруг Python существует множество научных библиотек, охватывающих различные области, от глубокого обучения до вероятностного вывода и визуализации данных. TensorFlow – одна из таких биб-

лиотек, хорошо известная в сообществе глубокого обучения и предоставляющая множество базовых и сложных операций, полезных для глубокого обучения.

Во всех наших упражнениях мы будем использовать блокноты Jupyter Notebook, поскольку они обеспечивают более интерактивную среду программирования по сравнению с использованием IDE.

Мы также будем использовать scikit-learn – еще один популярный инструментарий машинного обучения для Python – для различных прикладных целей, таких как предварительная обработка данных. Для различных операций с текстом мы будем использовать NLTK – набор инструментов Python для естественного языка. Наконец, применим пакет Matplotlib для визуализации данных.

Установка Python и scikit-learn

Python без проблем устанавливается в любой из широко используемых операционных систем, таких как Windows, macOS или Linux. Для установки и настройки Python мы будем использовать дистрибутив Anaconda, поскольку он выполняет всю кропотливую работу по настройке Python, а также устанавливает дополнительные пакеты и библиотеки.

Чтобы установить Anaconda, выполните следующие действия:

1) загрузите Anaconda с **https://www.anaconda.com/distribution/**. На момент подготовки русского перевода была доступна версия Anaconda3 2019.07;
2) выберите подходящую ОС и загрузите пакет с поддержкой Python 3.7;
3) установите Anaconda, следуя инструкциям на **https://docs.continuum.io/anaconda/install/**.

Чтобы проверить правильность установки Anaconda, выполните следующие действия:

1) в списке установленных программ найдите приложение **Anaconda Prompt** или **Anaconda Promt Shell** и запустите его;
2) теперь выполните следующую команду:

```
conda --version
```

При правильной установке в окне терминала должна отобразиться версия текущего дистрибутива Anaconda. На момент подготовки перевода это была версия 4.7.10.

Пакеты scikit-learn, NLTK и Matplotlib автоматически устанавливаются в составе нового дистрибутива Anaconda. Затем вы можете обновить версии пакетов. Найдите на своем компьютере приложение Anaconda Navigator и запустите его. Перейдите в окне навигатора на вкладку **Environment** (Окружение). Пакеты, для которых доступно стабильное обновление, помечены стрелкой в столбце **Version** (Версия). Щелкните правой кнопкой мыши на значке галочки и выберите пункт меню **Mark for update** (Отметить для обновления). Отметив нужные пакеты, нажмите кнопку **Apply** (Применить).

Установка Jupyter Notebook

Jupyter Notebook автоматически устанавливается вместе с новым дистрибутивом Anaconda. Вы можете установить Jupyter Notebook вручную, следуя инструкции на странице **http://jupyter.readthedocs.io/en/latest/install.html**.

Чтобы проверить правильность установки Jupyter Notebook, выполните следующие действия:

1) откройте окно терминала Anaconda Prompt;
2) выполните команду

```
jupyter notebook
```

Если новое окно браузера не открывается автоматически, то скопируйте адресную строку с секретным токеном из окна терминала и вставьте ее в адресное поле браузера. В браузере должно открыться рабочее окно, которое выглядит как на рис. 1.6:

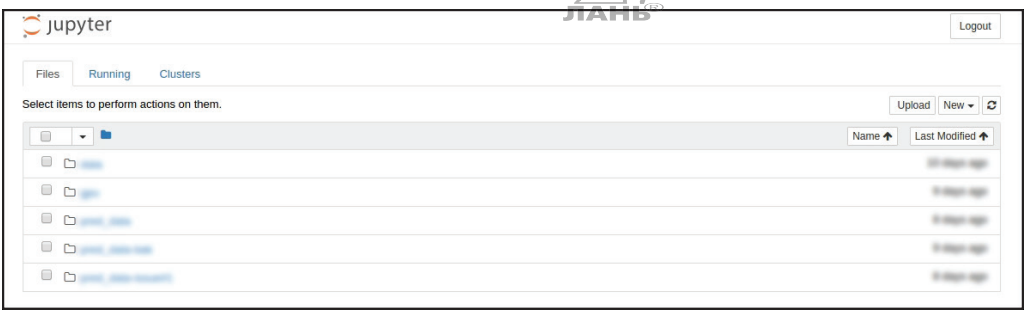


Рис. 1.6 ❖ Juputer Notebook установлен успешно

Установка TensorFlow

Для установки TensorFlow откройте окно терминала Anaconda Prompt и выполните команду

```
pip install tensorflow
```

Дождитесь окончания установки пакетов. На момент подготовки русского перевода книги была доступна версия TensorFlow 1.14.0. Для работы с упражнениями из этой книги нужна версия не ниже 1.8.0, поскольку API претерпел много изменений по сравнению с предыдущими версиями TensorFlow. Если у вас уже установлена версия TensorFlow ниже 1.8.0, обновите ее при помощи команды

```
pip update tensorflow
```

Чтобы проверить правильность установки TensorFlow, выполните следующие действия.

1. В терминале Anaconda Prompt введите команду

   ```
   python
   ```

   для запуска интерпретатора Python. В ответной строке вывода вы должны увидеть версию Python. Убедитесь, что вы используете Python 3.
2. Затем введите следующие команды в строке интерпретатора Python:

   ```
   import tensorflow as tf
   print (tf.__version__)
   ```

Если все прошло хорошо, должна отображаться версия TensorFlow 1.14.0 или выше без сообщений об ошибках. Если вы увидите предупреждение, что на вашем компьютере нет выделенного графического процессора, можете его проигнорировать.

Пользователям доступно множество облачных вычислительных платформ, где вы можете создать свой собственный виртуальный компьютер с различными настройками (операционная система, тип карты GPU, количество карт GPU и т. д.). Многие исследователи переходят на такие облачные сервисы благодаря следующим преимуществам:

- дополнительные параметры настройки;
- меньше усилий по обслуживанию;
- не требуется собственная инфраструктура.

Вот несколько популярных облачных вычислительных платформ:

- облачная платформа Google (GCP): **https://cloud.google.com/**;
- Amazon Web Services (AWS): **https://aws.amazon.com/**;
- TensorFlow Research Cloud (TFRC): **https://www.tensorflow.org/tfrc/**.

Заключение

В этой главе вы получили представление о задачах, связанных с построением хорошей системы на основе NLP. Сначала вы узнали, зачем нужна обработка естественного языка, а затем обсудили различные проблемы и осознали, насколько трудно добиться успеха в решении этих задач.

Далее мы рассмотрели классический подход к реализации NLP на примере генерации текста футбольного репортажа на естественном языке. Мы увидели, что традиционный подход обычно включает кропотливое и утомительное конструирование признаков. Например, чтобы проверить правильность сгенерированной фразы, возможно, придется сгенерировать дерево разбора для этой фразы. Затем мы обсудили смену парадигмы, которая произошла с появлением глубокого обучения, и обнаружили, что глубокое обучение сделало ненужным этап конструирования признаков. Мы начали с небольшого путешествия во времени, чтобы вернуться к истокам глубокого обучения и искусственных нейронных сетей, и постепенно добрались до огромных современных сетей с сотнями скрытых слоев. Позже разобрали простой пример, иллюстрирующий глубокую модель – многослойную модель персептрона, – чтобы прикоснуться к магии математики, скрытой в глубинах нейронных сетей.

Изучив основы традиционных и современных подходов к NLP, мы обсудили дальнейшие темы, которые будут представлены в книге, от представления слова в пространстве смыслов до мощных рекуррентных сетей с памятью, от создания подписей к изображениям до нейронных машинных переводчиков! Наконец, вы настроили на своем компьютере рабочую среду, установив Python, scikit-learn, Jupyter Notebook и TensorFlow.

В следующей главе вы изучите основы TensorFlow. К концу главы вы должны освоить реализацию простого алгоритма, который способен принимать некоторые входные данные, пропускать эти данные через определенную функцию и выводить результат.

Глава 2

Знакомство с TensorFlow

В этой главе вы познакомитесь с TensorFlow. Это открытая библиотека для реализации крупномасштабных вычислений в машинном обучении, и она будет основной платформой, на которой мы будем выполнять все наши упражнения.

Мы начнем знакомство с TensorFlow с простого вычислительного примера, а затем детально рассмотрим, как выполняется код программы. Это поможет вам понять, как фреймворк создает граф вычислений для расчета выходных данных и выполняет этот граф в сеансе. Затем вы закрепите понимание архитектуры TensorFlow на примере процесса выполнения заказа в ресторане.

Получив хорошее концептуальное и техническое понимание того, как работает TensorFlow, вы изучите некоторые важные вычислительные операции, предлагаемые платформой. Сначала вы познакомитесь с определением различных структур данных в TensorFlow, таких как переменные, заполнители и тензоры, а также узнаете, как читать входные данные. Затем освоите некоторые операции, связанные с нейронными сетями (например, свертка, вычисление потерь и оптимизация). После этого узнаете, как повторно использовать и эффективно управлять переменными TensorFlow с помощью области видимости. Наконец, вы примените эти знания в увлекательном упражнении по разработке нейронной сети, способной распознавать изображения рукописных цифр.

Что такое TensorFlow?

В главе 1 мы уже кратко упоминали TensorFlow. Теперь настало время познакомиться поближе. *TensorFlow* – это платформа для крупномасштабных вычислений с открытым исходным кодом, выпущенная Google и в основном предназначенная для облегчения реализации нейронной сети (например, вычисления производных весов нейронной сети). Более того, TensorFlow эффективно реализует прикладные вычисления с использованием Compute Unified Device Architecture (CUDA) – параллельной вычислительной платформы, представленной NVIDIA. Взгляните на интерфейс прикладного программирования (API) TensorFlow по адресу **https://www.tensorflow.org/api_docs/python/** – и вы убедитесь, что TensorFlow предоставляет тысячи операций, облегчающих жизнь разработчика.

Платформа TensorFlow не возникла в одночасье. Это результат длительного усердного труда талантливых, добросердечных людей, которые захотели изменить ситуацию и открыть доступ к глубокому обучению широкой аудитории. Если вам интересно, взгляните на код TensorFlow по адресу **https://github.com/tensorflow/tensorflow**. В настоящее время сообщество разработчиков TensorFlow насчи-

тывает около 1000 участников и включает более 25 тыс. коммитов, совершенствуясь с каждым днем.

Начало работы с TensorFlow

Теперь давайте на примере кода познакомимся с некоторыми важными компонентами структуры TensorFlow. Мы напишем пример для выполнения следующих вычислений, которые очень распространены в нейронных сетях:

```
h = sigmoid(W * x + b)
```

Здесь `W` и `x` – матрицы, а `b` – вектор. Тогда `*` обозначает скалярное произведение. Функция `sigmoid` является нелинейным преобразованием в соответствии со следующим выражением:

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

Давайте выполним это вычисление при помощи TensorFlow.

Сначала нам нужно импортировать библиотеки TensorFlow и NumPy. Их импорт необходим перед тем, как вы запустите в Python любую операцию, связанную с TensorFlow или NumPy:

```
import tensorflow as tf
import numpy as np
```

Далее мы определим объект `graph`, который позже будет заполнен операциями и переменными:

```
graph = tf.Graph () # Создает граф
session = tf.InteractiveSession (graph graph) # Создает сессию
```

Объект `graph` содержит *граф вычислений* (computational graph), соединяющий различные входы и выходы, которые мы определяем в нашей программе, чтобы получить конечный желаемый результат (т. е. он определяет, как `W`, `x` и `b` взаимодействуют, чтобы произвести `h` в терминах графа). Например, если вы думаете о выходе как о торте, тогда граф будет рецептом для приготовления этого торта с использованием ингредиентов (т. е. входных данных). Кроме того, мы определим объект `session`, который принимает определенный граф и выполняет его. Мы поговорим об этих элементах подробно в следующем разделе.

Чтобы создать новый объект `graph`, вы можете поступить как в предыдущем примере:

```
graph = tf.Graph()
```

В качестве альтернативы вы можете получить граф вычислений TensorFlow по умолчанию:

```
graph = tf.get_default_graph()
```

Мы выполним упражнения с использованием обоих этих методов.

Теперь мы определим несколько тензоров, а именно `x`, `W`, `b` и `h`. *Тензор*, по сути, является n-мерным массивом в TensorFlow. Например, одномерный вектор или

двухмерная матрица является тензором[1]. В TensorFlow есть несколько различных способов определения тензоров. Здесь мы используем три подхода:

1) во-первых, `x` является *заполнителем* (placeholder, поле для подстановки). Заполнители, как следует из названия, не инициализируются каким-либо значением. Наоборот, они получают значение «на лету» во время выполнения графа;
2) далее у нас есть переменные `W` и `b`. Переменные являются изменяемыми, это означает, что их значения могут меняться со временем;
3) наконец, у нас есть `h`, который является неизменяемым тензором, полученным путем выполнения некоторых операций с `x`, `W` и `b`:

```
x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1, maxval=0.1, dtype=tf.
float32),name='W')
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
h = tf.nn.sigmoid(tf.matmul(x,W) + b)
```

Также обратите внимание, что для `W` и `b` мы приводим некоторые важные аргументы:

```
tf.random_uniform(shape=[10,5], minval=-0.1, maxval=0.1, dtype=tf.float32)
tf.zeros(shape=[5],dtype=tf.float32)
```

Они называются *инициализаторами переменных* и являются тензорами, которые будут изначально назначены переменным `W` и `b`. Переменные не существуют без начального значения, как это могут делать заполнители, и им необходимо постоянно присваивать какое-то значение. Здесь `tf.random_uniform` означает, что мы равномерно выбираем значения между `minval(-0.1)` и `maxval(0.1)` для присвоения значений тензорам, а `tf.zeros` инициализирует тензор с нулями. Также очень важно задать *форму* тензора при его определении. Свойство `shape` определяет размер каждого измерения тензора. Например, `shape=[10,5]` означает, что тензор является двумерной структурой и будет иметь 10 элементов на оси 0 и 5 элементов на оси 1.

Далее мы запускаем операцию инициализации, которая инициализирует переменные в графе, `W` и `b`:

```
tf.global_variables_initializer().run()
```

Теперь мы выполним граф, чтобы получить конечный результат `h`. Это делается путем выполнения `session.run(...)`, где мы передаем значение заполнителю в качестве аргумента команды `session.run()`:

```
h_eval = session.run(h, feed_dict = {x: np.random.rand (1,10)})
```

Наконец, мы закрываем сессию, освобождая любые ресурсы, удерживаемые объектом сессии:

```
session.close()
```

Вот полный код этого примера TensorFlow. Все примеры кода в этой главе будут доступны в файле `tensorflow_introduction.ipynb` в папке `ch2`:

---

[1] В данном случае термин «тензор» используется в основном ради красивого названия TensorFlow и имеет мало общего с понятиями тензора и тензорного пространства в классической математике. – *Прим. перев.*

```
import tensorflow as tf
import numpy as np

# Определяем граф и сессию
graph = tf.Graph() # Создаем граф
session = tf.InteractiveSession(graph=graph) # Создаем сессию

# Строим граф
# Заполнитель принимает символьный ввод
x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,
maxval=0.1, dtype=tf.float32),name='W') # Переменная

# Переменная
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')

h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Выполняемая операция

# Выполнение операций и оценка узлов графа
tf.global_variables_initializer().run() # Инициализация переменных

# Выполнение операции с предоставлением символьного ввода x
h_eval = session.run(h,feed_dict={x: np.random.rand(1,10)})
# Закрытие сессии и освобождение всех занятых сессией ресурсов
session.close()
```

Когда вы запустите этот код, то можете столкнуться со следующим предупреждением:

```
... tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports instructions that
this TensorFlow binary was not compiled to use: ...
```

Не беспокойтесь, это предупреждение о том, что вы использовали готовую предварительно скомпилированную версию TensorFlow, не компилируя ее на своем компьютере. Это совершенно нормально. Просто иногда вы можете получить немного лучшую производительность, если скомпилируете библиотеку на своем компьютере, так как TensorFlow будет оптимизирован для вашего конкретного оборудования.

Далее вы узнаете, как TensorFlow выполняет этот код. Учтите, что следующие два раздела будут насыщены техническими деталями. Но не волнуйтесь, если что-то останется непонятным, потому что сразу после этого мы рассмотрим наглядный и обстоятельный пример из реальной жизни, где такой же процесс объясняется с точки зрения выполнения заказа в кафе Le TensorFlow.

Подробно о клиенте TensorFlow

Предыдущий пример программы называется *клиентом* TensorFlow. В любой клиентской программе, которую вы пишете с помощью TensorFlow, будет два основных типа объектов: *операции* и *тензоры*. В предыдущем примере `tf.nn.sigmoid` – это операция, а `h` – тензор.

Еще у нас есть объект `graph`, который является графом вычислений – в нем хранится поток данных нашей программы. Когда мы добавляем последующие строки, определяющие `x`, `W`, `b` и `h`, TensorFlow автоматически добавляет эти тензоры и любые операции в граф как узлы. Граф хранит жизненно важную информацию,

такую как тензорные зависимости и какие операции и где выполнять. В нашем примере граф будет знать, что для вычисления `h` требуются тензоры `x`, `W` и `b`. Так что если вы неправильно инициализировали один из них во время выполнения кода, TensorFlow может точно указать вам ошибку инициализации, которую необходимо исправить.

Сессия занимается выполнением графа путем разделения его на подграфы, а затем на еще более мелкие фрагменты, которые потом будут назначены *исполнителям* (worker), выполняющим определенную задачу. За это отвечает функция `session.run(...)`. Мы скоро поговорим об этом. Для удобства будем дальше называть наш вычислительный пример просто *примером сигмоиды*.

Архитектура TensorFlow – что происходит при запуске клиента?

Мы знаем, что TensorFlow умеет создавать хороший граф вычислений со всеми зависимостями и операциями, чтобы точно знать, как, когда и где движутся потоки данных. Но должен быть еще один элемент, делающий TensorFlow поистине великолепным, – эффективное выполнение объявленного графа. Вот когда на сцену выходит сессия. Давайте заглянем под капот сессии, чтобы понять, как выполняется граф.

Итак, клиент TensorFlow содержит граф и сессию. Когда вы создаете сессию, она отправляет граф вычислений в виде буфера протокола `tf.GraphDef` *распределенному мастеру*[1] (distributed master). `tf.GraphDef` – это стандартизированное представление графа. Распределенный мастер видит все вычисления, входящие в граф, и поручает их выполнение разным устройствам, например разным GPU и CPU. Граф в нашем примере сигмоиды выглядит как на рис. 2.1. Одиночный элемент графа называется *узлом* (node).

Затем распределенный мастер разбивает вычислительный граф на подграфы и далее на более мелкие части. Хотя в нашем примере декомпозиция графа выглядит слишком тривиальной, граф вычислений может экспоненциально расти в реальных решениях со многими скрытыми слоями. Кроме того, с ростом сложности задачи становится принципиально важным разбить граф на несколько частей, чтобы выполнять вычисления параллельно на нескольких устройствах. Выполнение этого графа (или подграфа, если граф делится на части) называется *задачей* (task), которая выделяется одному серверу TensorFlow.

Однако в действительности каждая задача будет выполняться путем разбиения ее на две части, где каждая часть выполняется отдельным воркером:

- один исполнитель выполняет операции TensorFlow, используя текущие значения параметров (исполнитель операции, воркер);
- другой исполнитель сохраняет параметры и обновляет их новыми значениями, полученными после выполнения операций (сервер параметров).

Рабочий процесс клиента TensorFlow в общем виде изображен на рис. 2.2.

---

[1] Также известен как *менеджер распределенных вычислений*, распределяющий вычислительные задания между исполнителями. – *Прим. перев.*

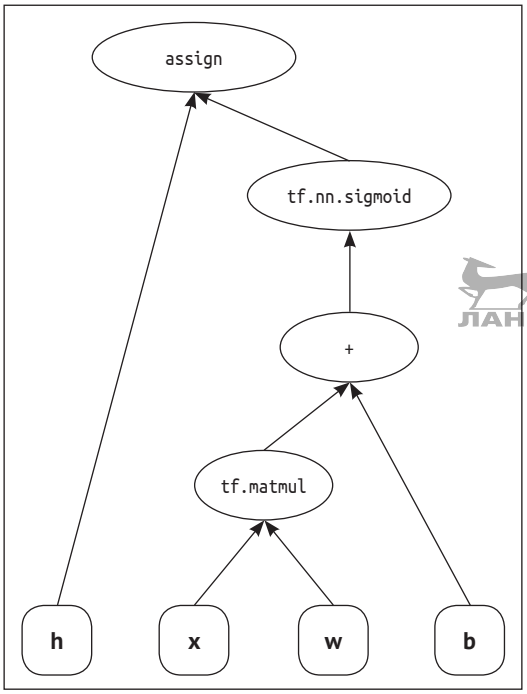


Рис. 2.1 ❖ Вычислительный граф примера сигмоиды

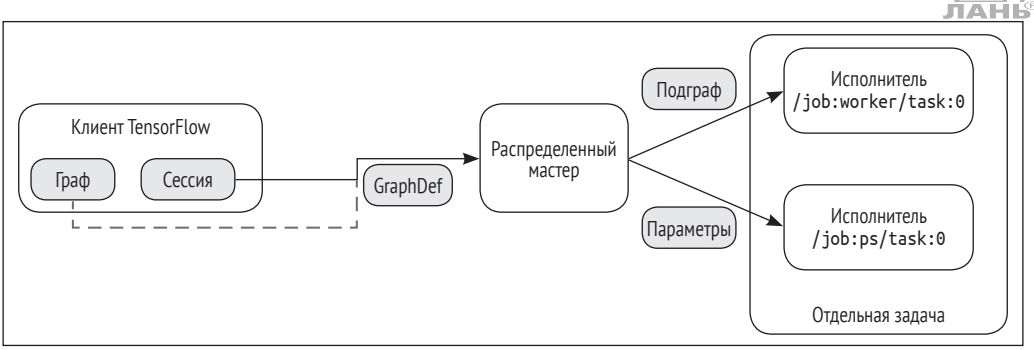


Рис. 2.2 ❖ Общая схема рабочего процесса клиента TensorFlow

Рисунок 2.3 иллюстрирует декомпозицию вычислительного графа. Кроме разбиения графа на части, TensorFlow вставляет передающие и принимающие узлы, чтобы организовать связь между сервером параметров и исполнителем операции. Передающие узлы отправляют данные всякий раз, когда данные доступны, а принимающие узлы постоянно прослушивают и получают данные, отправленные соответствующим передающим узлом.

Наконец, сессия возвращает обновленные данные клиенту с сервера параметров после завершения вычислений. Архитектура TensorFlow показана на рис. 2.4. Это объяснение основано на официальной документации TensorFlow, которая находится по адресу **https://www.tensorflow.org/guide/extend/architecture**.

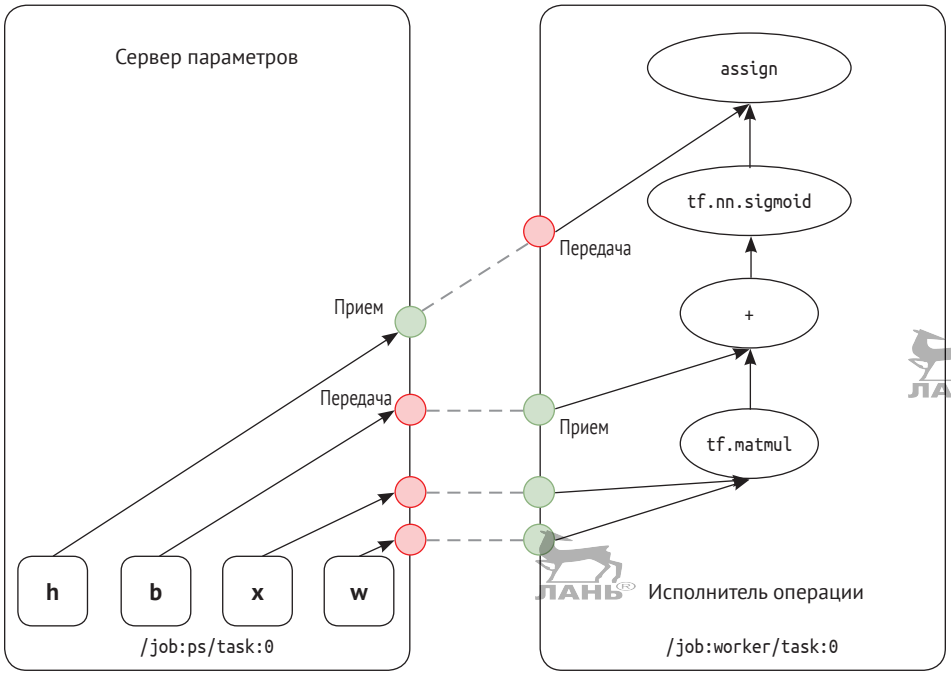


Рис. 2.3 ❖ Декомпозиция графа TensorFlow

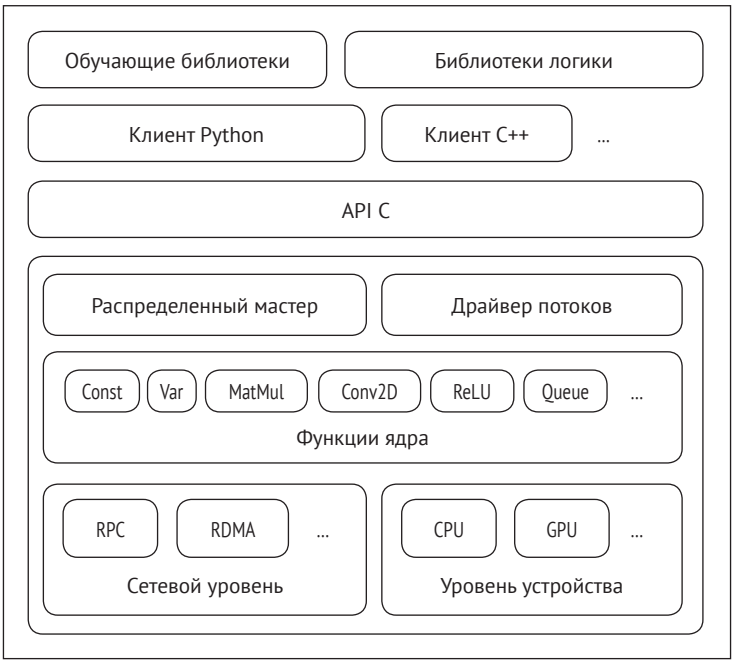


Рис. 2.4 ❖ Архитектура фреймворка TensorFlow (**https://www.tensorflow.org/extend/architecture**)

Кафе Le TensorFlow – пояснение устройства TensorFlow на примере

Если вы слегка запутались в технической информации предыдущего раздела, попробуйте понять концепцию TensorFlow с другой точки зрения. Допустим, недавно открылось новое кафе, и вам очень хочется в нем побывать. Итак, вы идете туда и занимаете место у окна.

К вашему столику подходит официант, чтобы принять ваш заказ, и вы заказываете *куриный бургер с дополнительным сыром и без помидоров*. Сейчас вы клиент TensorFlow, а ваш заказ – это определение графа. Граф задает, *что* вам нужно и *как* вам это нужно. Официант аналогичен сессии, где его обязанность – доставить заказ на кухню для выполнения. Принимая заказ, официант использует определенный формат для передачи вашего заказа, например номер столика, идентификатор пункта меню, количество и специальные требования. Эта запись в блокноте официанта является буфером протокола `GraphDef`. Затем официант относит заказ на кухню и отдает его менеджеру кухни. С этого момента менеджер берет на себя ответственность за выполнение заказа. Менеджер играет роль распределенного мастера. Он принимает решения, например сколько поваров требуется для приготовления блюд и какие повара являются лучшими исполнителями данной работы. Предположим также, что у каждого повара есть помощник, в обязанности которого входит обеспечение повара необходимыми ингредиентами, оборудованием и т. д. Поэтому менеджер передает заказ одному повару и помощнику (гамбургер не так уж и сложен в приготовлении) и поручает им приготовить блюдо. В нашем примере повар является исполнителем операций, а помощник – сервером параметров.

Повар смотрит на заказ и говорит помощнику, что нужно. Поэтому предусмотрительный помощник заранее находит вещи, которые потребуются (например, булочки, филе и лук), и держит их под рукой, чтобы как можно скорее выполнить запросы повара. Кроме того, повар может попросить временно сохранить промежуточные ингредиенты блюда (например, нарезанные овощи), пока они не потребуются снова.

Когда заказ готов, менеджер кухни получает гамбургер от повара и помощника и уведомляет официанта. В этот момент официант берет гамбургер у менеджера кухни и приносит его вам. Наконец-то вы можете насладиться вкусным гамбургером, приготовленным в соответствии с вашими требованиями. Этот процесс изображен на рис. 2.5.

Входные данные, переменные, выходные данные и операции

Получив понимание базовой архитектуры, давайте перейдем к наиболее распространенным элементам, из которых состоит клиент TensorFlow. В любом из миллионов клиентов TensorFlow, доступных в интернете, все элементы входят в одну из следующих групп:

- **входные данные** – применяются для обучения и тестирования наших алгоритмов;

- **переменные** – изменяемые тензоры, в основном определяющие параметры наших алгоритмов;
- **выходные данные** – неизменяемые тензоры, хранящие как терминальные, так и промежуточные выходы;
- **операции** – различные преобразования входов для получения желаемых выходов.

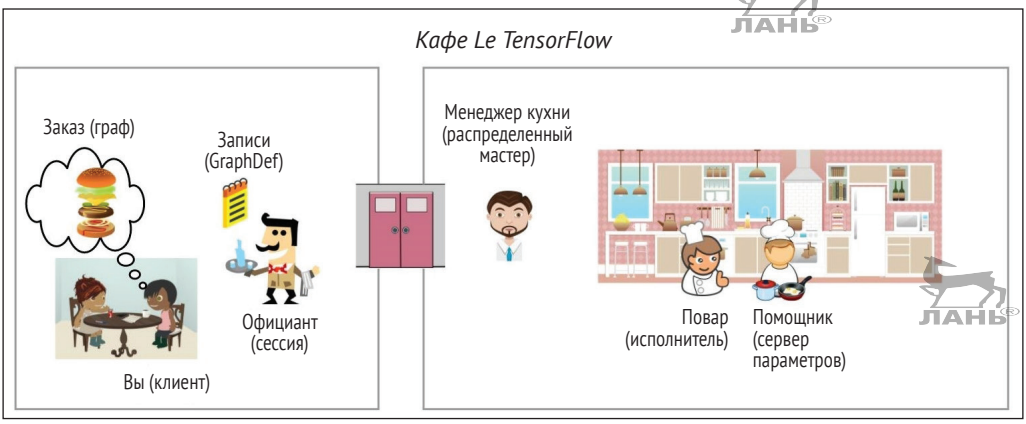


Рис. 2.5 ❖ Иллюстрация принципа работы TensorFlow на примере выполнения заказа в кафе

В нашем предыдущем примере с сигмоидом присутствуют экземпляры всех этих категорий (табл. 2.1).

Таблица 2.1. Основные элементы TensorFlow

Элемент TensorFlow	Пример
Входные данные	`x`
Переменные	`W`, `b`
Выходные данные	`h`
Операции	`tf.matmul(...)`, `tf.nn.sigmoid(...)`

Рассмотрим каждый элемент TensorFlow более подробно.

Определение входных данных в TensorFlow

Клиент TensorFlow может получать данные тремя различными способами:

- ввод данных на каждом шаге алгоритма с помощью кода Python;
- предварительная загрузка и сохранение данных в виде тензоров TensorFlow;
- построение входного конвейера.

Давайте рассмотрим каждый из этих способов.

Ввод данных с помощью кода Python

В первом методе данные могут быть переданы клиенту TensorFlow с использованием обычного кода Python. Этот метод применяется в рассмотренном выше примере с сигмоидой. Для передачи данных клиенту из внешних структур (например, `numpy.ndarray`) библиотека TensorFlow предоставляет элегантную символическую структуру, известную как *заполнитель*. Как следует из названия, заполнитель не требует фактического наличия данных на этапе построения графа. Данные передаются только при выполнении графа, вызванного с помощью `session.run(..., feed_dict={placeholder:value})` путем присвоения внешних данных аргументу `feed_dict` в форме словаря Python, где ключом является переменная `tf.placeholder`, а соответствующим значением – фактические данные (например, `numpy.ndarray`). Объявление заполнителя имеет следующий вид:

```
tf.placeholder(dtype, shape=None, name=None)
```

Аргументы заполнителя:

- `dtype` – тип данных, вводимых в заполнитель;
- `shape` – форма заполнителя, заданная как одномерный вектор;
- `name` – это имя заполнителя, и оно важно для отладки.

Предварительная загрузка и хранение данных в виде тензоров

Второй метод аналогичен первому, но его проще использовать. Нам не нужно подавать данные во время выполнения графа, поскольку они предварительно загружены. Чтобы увидеть метод в действии, давайте изменим наш пример. Ранее мы объявляли `x` как заполнитель, не имеющий определенного значения:

```
x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
```

А теперь давайте объявим его как тензор, который содержит заданные значения:

```
x = tf.constant(value=[[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]], dtype=tf.float32,name='x')
```

Итак, полный код будет выглядеть следующим образом:

```
import tensorflow as tf
# Определяем граф и сессию
graph = tf.Graph() # Создаем граф
session = tf.InteractiveSession(graph=graph) # Создаем сессию

# Строим граф

# x - предварительно заданный вход
x = tf.constant(value=[[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]],
dtype=tf.float32,name='x')

W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,
maxval=0.1, dtype=tf.float32),name='W') # Переменная
# Переменная
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')

h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Вычислительная операция
```

```
# Выполнение операций и оценка узлов графа
tf.global_variables_initializer().run() # Инициализация переменных

# Выполнение операции без feed_dict
h_eval = session.run(h)
print(h_eval)
session.close()
```

В этом коде есть два важных отличия от нашего исходного примера сигмоиды. Вместо того чтобы использовать объект-заполнитель и вводить фактическое значение при выполнении графа, мы теперь сразу назначаем конкретное значение и определяем `x` как тензор. Также мы не вводим никаких дополнительных аргументов в `session.run(...)`. С другой стороны, теперь вы не можете передавать различные значения `x` в `session.run(...)` и смотреть, как изменяется выходной результат.

Построение входного конвейера

Входные конвейеры предназначены для более загруженных клиентов, которым необходимо быстро обрабатывать много данных. По сути, *конвейер* – это очередь данных, извлекаемых по мере необходимости. TensorFlow также обладает различными инструментами для предварительной обработки данных перед подачей в алгоритм – например, настройка контрастности/яркости изображения или стандартизация. Чтобы сделать алгоритм еще эффективнее, можно запустить несколько потоков, считывающих и обрабатывающих данные параллельно.

Типичный конвейер состоит из следующих компонентов:

- список имен файлов;
- очередь имен файлов, подающая имена файлов в *ридер ввода* (input reader);
- ридер ввода, читающий входные записи;
- декодер прочитанных записей, например декодер изображения JPEG;
- этапы предварительной обработки (необязательно);
- выборка очереди (т. е. декодированные входы).

Давайте напишем простой код входного конвейера, используя TensorFlow. Допустим, у нас есть три текстовых файла `text1.txt`, `text2.txt` и `text3.txt` в формате CSV, содержащих по пять строк. Каждая строка состоит из 10 чисел, разделенных запятыми, например: `0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0`. Нам нужно читать эти данные как пакеты (несколько строк векторов данных), формируя входной конвейер на всем пути от исходных файлов до тензора, представляющего эти входные данные. Рассмотрим процесс формирования конвейера шаг за шагом.

Для получения дополнительной информации обратитесь к официальной странице TensorFlow Importing Data по адресу **https://www.tensorflow.org/guide/datasets**.

Во-первых, как и раньше, импортируем несколько важных библиотек:

```
import tensorflow as tf
import numpy as np
```

Далее определяем объекты `graph` и `session`:

```
graph = tf.Graph() # Создаем объект graph
session = tf.InteractiveSession(graph = graph) # Создаем объект session
```

Затем определяем очередь имен файлов – структуру данных, содержащую имена файлов, – и передаем ее ридеру в качестве аргумента. Очередь выдает имена файлов по запросу ридера, чтобы он мог получить очередной файл для чтения данных:

```
filenames = ['test %d.txt' %i for i in range(1,4)]
filename_queue = tf.train.string_input_producer(filenames, capacity=3, shuffle=True,
name='string_input_producer')
```

Здесь `capacity` – это объем данных, хранящихся в очереди в данный момент времени, а `shuffle` сообщает очереди, следует ли перетасовать данные перед выдачей.

TensorFlow имеет несколько различных типов ридеров. Поскольку у нас есть несколько отдельных текстовых файлов, в которых одна строка представляет одну точку данных, нам лучше всего подходит TextLineReader[1]:

```
reader = tf.TextLineReader()
```

Определив ридер, мы можем использовать функцию `read()` для чтения данных из файлов. Она выводит пары в формате (`ключ`, `значение`). Ключ идентифицирует файл и запись (т. е. строку текста), читаемую в файле. Мы не будем его использовать. Значение возвращает фактическое значение строки, прочитанной ридером:

```
key, value = reader.read(filename_queue, name = 'text_read_op')
```

Далее мы определим значение по умолчанию `record_defaults`, которое будет выводиться, если найдены какие-либо ошибочные записи:

```
record_defaults = [[-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0],
[-1.0]]
```

Теперь мы декодируем строку чтения текста в столбцы чисел, поскольку работаем с файлами CSV. Для этого используем метод `decode_csv()`. Если вы откроете файл, например `test1.txt`, в текстовом редакторе, то увидите, что строка состоит из 10 чисел, где каждое число соответствует столбцу данных.

```
col1, col2, col3, col4, col5, col6, col7, col8, col9, col10 = tf.decode_csv(value, record_
defaults = record_defaults)
```

Затем мы объединяем эти столбцы, называемые *признаками*, в единый тензор и передаем его методу `tf.train.shuffle_batch()`. Этот метод берет ранее определенный тензор, случайным образом перемешивает признаки и выводит пакет данных с заданным размером:

```
features = tf.stack([col1, col2, col3, col4, col5, col6, col7, col8, col9, col10])

x = tf.train.shuffle_batch([features], batch_size=3, capacity=5, name='data_batch', min_after_
dequeue=1, num_threads=1)
```

Аргумент `batch_size` – это размер пакета данных, который мы будем отбирать на данном шаге, `capacity` – это емкость очереди данных (чем больше очередь, тем

[1] Эта функция на момент подготовки русского перевода книги объявлена устаревшей (deprecated) и не рекомендуется к дальнейшему применению. Для работы с API TensorFlow версии 1.14 используйте `tf.data.TextLineDataset` (**https://www.tensorflow.org/api_docs/python/tf/data/TextLineDataset**). – *Прим. перев.*

больше памяти требуется), а `min_after_dequeue` представляет минимальное количество элементов, оставшихся в очереди после отбора пакета. Наконец, `num_threads` определяет, сколько потоков используется для создания пакета данных. Если в конвейере происходит сложная предварительная обработка, вы можете увеличить это число. Чтобы прочитать данные без перемешивания, следует использовать операцию `tf.train.batch`. Теперь осталось только запустить конвейер:

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(coord=coord, sess=session)
```

Класс `tf.train.Coordinator()` можно рассматривать как *менеджер потоков* (thread manager). Он реализует различные механизмы для управления потоками, такие как запуск потоков и присоединение потоков к основному потоку после завершения задачи. Этот класс необходим, потому что входной конвейер порождает много потоков для заполнения очереди, исключения из очереди и других задач. В следующей строке происходит запуск заполнителя очереди с участием созданного ранее менеджера потоков. Класс `QueueRunner()` содержит операции постановки в очередь, и они автоматически создаются во время определения входного конвейера. Итак, чтобы заполнить определенные очереди, нам нужно запустить эти обработчики очередей с помощью функции `tf.train.start_ queue_runners`.

Затем после завершения необходимых вычислений необходимо явно остановить потоки очередей и присоединить их к основному потоку, иначе программа будет выполняться бесконечно. Это достигается с помощью команд `coord.request_stop()` и `coord.join(threads)`.

Уже знакомый нам пример сигмоиды – теперь с чтением данных в конвейер непосредственно из файла – будет выглядеть следующим образом:

```
import tensorflow as tf
import numpy as np
import os

# Определяем граф и сессию.
graph = tf.Graph() # Создаем граф
session = tf.InteractiveSession(graph=graph) # Создаем сессию.

### Строим входной конвейер ###
# Очередь имен файлов.
filenames = ['test %d.txt' %i for i in range(1,4)]
filename_queue = tf.train.string_input_producer(filenames, capacity=3,
shuffle=True,name='string_input_producer')

# Проверяем наличие файлов в указанном месте.
for f in filenames:
    if not tf.gfile.Exists(f):
        raise ValueError('Failed to find file: ' + f)
    else:
        print('File  %s found.' %f)

# Ридер берет имена файлов из очереди
# и построчно читает содержимое этих файлов.
reader = tf.TextLineReader()

# Получаем данные в виде пар key, value.
```

```
# Мы игнорируем значение key.
key, value = reader.read(filename_queue, name='text_read_op')

# Если при чтении данных из файла возникла ошибка,
# возвращаем это значение по умолчанию.
record_defaults = [[-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0],
[-1.0], [-1.0], [-1.0], [-1.0]]

# Декодируем данные из строки CSV в столбцы данных.
col1, col2, col3, col4, col5, col6, col7, col8, col9, col10 =
tf.decode_csv(value, record_defaults=record_defaults)
# Теперь собираем все столбцы в один тензор
features = tf.stack([col1, col2, col3, col4, col5, col6, col7, col8,
col9, col10])

# x получает случайный пакет данных с размером batch_size,
# где данные прочитаны из текстовых файлов.
x = tf.train.shuffle_batch([features], batch_size=3,
                           capacity=5, name='data_batch',
                           min_after_dequeue=1,num_threads=1)
# QueueRunner запрашивает данные из очередей.
# Coordinator управляет работой нескольких QueueRunner.
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(coord=coord, sess=session)
# Строим граф, определяя переменные и вычисления.
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,
maxval=0.1, dtype=tf.float32),name='W') # Переменная.
# Переменная.
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')

h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Вычислительная операция.

# Выполнение операций и оценка узлов графа
tf.global_variables_initializer().run() # Инициализация переменных.

# Вычисление h по текущему x и вывод результатов в цикле из 5 проходов.
for step in range(5):
    x_eval, h_eval = session.run([x,h])
    print('========== Step  %d ==========' %step)
    print('Evaluated data (x)')
    print(x_eval)
    print('Evaluated data (h)')
    print(h_eval)
    print('')
# Необходимо явно остановить процессы,
# иначе они будут выполняться бесконечно.
coord.request_stop()
coord.join(threads)
session.close()
```

Объявление переменных в TensorFlow

Переменные играют важную роль в TensorFlow. *Переменная* – это, по существу, тензор с заданной формой, определяющей, сколько измерений будет иметь пе-

ременная и размер каждого измерения. Однако, в отличие от обычного тензора, переменные являются изменяемыми тензорами, т. е. значение переменных может измениться после того, как они объявлены. Это идеальное свойство для реализации параметров обучаемой модели, например весов нейронной сети, где веса незначительно меняются после каждого шага обучения. Например, если вы объявили переменную с помощью операции `x=tf.Variable(0,dtype=tf.int32)`, впоследствии можно изменить значение этой переменной с помощью операции TensorFlow, такой как `tf.assign(x,x+1)`. Однако если вы объявляете константу, например `x=tf.constant(0, dtype=tf.int32)`, то не сможете изменить ее значение после операции объявления.

Создать переменную довольно просто. В нашем примере сигмоиды мы уже создали две переменные, `W` и `b`. При объявлении переменной применяются следующие параметры:

- форма переменной;
- тип данных;
- начальное значение;
- имя (необязательно).

Форма переменной (variable shape) представляет собой одномерный вектор формата `[x,y,z,...]`. Каждое значение в списке указывает, насколько велико соответствующее измерение или ось. Например, если в качестве переменной вам нужен двухмерный тензор с 50 строками и 10 столбцами, его форма будет иметь вид `[50,10]`.

Размерность переменной, т. е. длина вектора `shape`, в TensorFlow называется *ранг тензора* (tensor rank). Не путайте его с рангом матрицы.

Ранг тензора в TensorFlow указывает размерность тензора; для двумерной матрицы `rank = 2`.

Тип данных (data type) играет важную роль в определении размера переменной. Существует много различных типов данных, включая обычно используемые `tf.bool`, `tf.uint8`, `tf.float32` и `tf.int32`. Каждому типу данных соответствует количество битов, необходимых для представления одного значения этого типа. Например, для `tf.uint8` требуется 8 бит, а для `tf.float32` требуется 32 бита. Для вычислений следует использовать данные одинакового типа, иначе может возникнуть ошибка несоответствия типов данных. Если у вас есть два тензора разного типа, перед выполнением вычислений вы должны явно привести тип одного тензора к типу другого тензора, используя операцию `tf.cast(...)`. Например, если у вас есть переменная `x` типа `tf.int32`, для преобразования ее в `tf.float32` используйте операцию `tf.cast(x, dtype=tf.float32)`.

Переменную следует инициализировать – присвоить ей *начальное значение* (initial value). Для нашего удобства TensorFlow предоставляет несколько различных *инициализаторов*, в том числе константы и нормальное распределение. Вот несколько популярных инициализаторов TensorFlow, которые вы можете использовать для инициализации переменных:

```
tf.zeros
tf.constant_initializer
tf.random_uniform
tf.truncated_normal
```

Наконец, *имя* переменной используется для идентификации этой переменной в вычислительном графе. В описании вычислительного графа переменная присутствует в виде аргумента, ассоциированного с ключевым словом `name`. Если вы не укажете имя переменной, TensorFlow будет использовать схему именования по умолчанию.

Обратите внимание, что переменная Python, которой присвоено значение операцией `tf.Variable`, неизвестна вычислительному графу и не входит в пространство имен переменных TensorFlow. Допустим, вы объявляете переменную TensorFlow следующим образом:

```
a = tf.Variable(tf.zeros([5]), name='b')
```

В данном случае граф TensorFlow будет знать эту переменную по имени `b`, а не `a`.

Объявление выходных данных TensorFlow

Выходные данные TensorFlow обычно являются результатом преобразования либо входных данных, либо переменных, либо тех и других вместе. В нашем примере `h` является выходом, где `h = tf.nn.sigmoid(tf.matmul(x,W+b)`. Выходные данные можно передавать другим операциям, образуя цепочку. Кроме того, это не обязательно должны быть операции TensorFlow. Вы можете использовать стандартную арифметику Python, например:

```
x = tf.matmul(w,A)
y = x + B
z = tf.add(y,C)
```

Объявление операций TensorFlow

Если вы ознакомитесь с описанием TensorFlow API по адресу **https://www.tensorflow.org/api_docs/python/**, то увидите, что TensorFlow имеет обширную коллекцию операций. Здесь мы рассмотрим лишь некоторые из множества операций TensorFlow.

Операции сравнения

Операции сравнения TensorFlow применяются для сравнения двух тензоров. Полный список операторов сравнения можно найти по адресу **https://www.tensorflow.org/api_docs/python/tf/math/equal**. Чтобы лучше понять работу этих операций, рассмотрим пример сравнения тензоров `x` и `y`:

```
# Пусть тензорам x и y присвоены следующие значения:
# x (2-D tensor) => [[1,2],[3,4]]
# y (2-D tensor) => [[4,3],[3,2]]
x = tf.constant([[1,2],[3,4]], dtype=tf.int32)
y = tf.constant([[4,3],[3,2]], dtype=tf.int32)

# Поэлементное сравнение тензоров на равенство.
# Результатом является тензор в булевом (логическом) формате.
# x_equal_y => [[False,False],[True,False]]
x_equal_y = tf.equal(x, y, name=None)
```

```
# Поэлементная проверка условия x < y.
# Результатом является тензор в булевом (логическом) формате.
# x_less_y => [[True,True],[False,False]]
x_less_y = tf.less(x, y, name=None)

# Поэлементная проверка условия x >= y.
# Результатом является тензор в булевом (логическом) формате.
# x_great_equal_y => [[False,False],[True,True]]
x_great_equal_y = tf.greater_equal(x, y, name=None)

# Извлечение элементов из x или y в зависимости от условия.
# Если условие выполняется, извлекаются элементы из тензора x.
# Если условие не выполняется – из тензора y.
condition = tf.constant([[True,False],[True,False]],dtype=tf.bool)
# x_cond_y => [[1,3],[3,2]]
x_cond_y = tf.where(condition, x, y, name=None)
```

Математические операции

TensorFlow позволяет выполнять как простые, так и сложные математические операции над тензорами. Мы рассмотрим лишь некоторые математические операции, представленные в TensorFlow. Полное описание набора операций доступно по адресу **https://www.tensorflow.org/api_docs/python/tf/math**.

```
# Пусть тензорам x и y присвоены следующие значения:
# x (2-D tensor) => [[1,2],[3,4]]
# y (2-D tensor) => [[4,3],[3,2]]
x = tf.constant([[1,2],[3,4]], dtype=tf.float32)
y = tf.constant([[4,3],[3,2]], dtype=tf.float32)
# Поэлементное сложение тензоров x и y
# x_add_y => [[5,5],[6,6]]
x_add_y = tf.add(x, y)

# Матричное перемножение
# x_mul_y => [[10,7],[24,17]]
x_mul_y = tf.matmul(x, y)

# Поэлементное вычисление натурального логарифма x
# Эквивалент вычисления ln(x)
# log_x => [[0,0.6931],[1.0986,1.3863]]
log_x = tf.log(x)

# Уменьшение размерности по заданным осям
# x_sum_1 => [3,7]
x_sum_1 = tf.reduce_sum(x, axis=[1], keepdims=False)
# x_sum_2 => [[4],[6]]
x_sum_2 = tf.reduce_sum(x, axis=[0], keepdims=True)

# Сегментируем тензор соответственно segment_ids, т. е. шаблону
# (элементы тензора с одинаковым ID шаблона образуют сегмент)
# и вычисляем суммы элементов тензора посегментно.
data = tf.constant([1,2,3,4,5,6,7,8,9,10], dtype=tf.float32)
segment_ids = tf.constant([0,0,0,1,1,2,2,2,2,2 ], dtype=tf.int32)
# x_seg_sum => [6,9,40]
x_seg_sum = tf.segment_sum(data, segment_ids)
```

Операции scatter и gather

Операции scatter (разбрасывать) и gather (собирать) принципиально важны при выполнении матричных вычислений, поскольку эти два варианта являются (до последнего времени) единственным способом индексирования тензоров в TensorFlow. Другими словами, вы не можете получить доступ к элементам тензоров в TensorFlow, как это делается в NumPy (например, x[1,0], где x – двумерный массив numpy.ndarray). Операция scatter позволяет назначать значения определенным индексам данного тензора, тогда как операция gather дает возможность извлекать срез (или отдельные элементы) данного тензора. Следующий пример показывает несколько вариантов операций scatter и gather:

```
# Одномерная операция scatter.
ref = tf.Variable(tf.constant([1,9,3,10,5],dtype=tf.float32),name='scatter_update')
indices = [1,3]
updates = tf.constant([2,4],dtype=tf.float32)
tf_scatter_update = tf.scatter_update(ref, indices, updates, use_locking=None, name=None)

# n-мерная операция scatter.
indices = [[1],[3]]
updates = tf.constant([[1,1,1],[2,2,2]])
shape = [4,3]
tf_scatter_nd_1 = tf.scatter_nd(indices, updates, shape, name=None)

# n-мерная операция scatter.
indices = [[1,0],[3,1]] # 2 x 2
updates = tf.constant([1,2]) # 2 x 1
shape = [4,3] # 2
tf_scatter_nd_2 = tf.scatter_nd(indices, updates, shape, name=None)

# Одномерная операция gather.
params = tf.constant([1,2,3,4,5], dtype=tf.float32)
indices = [1,4]
tf_gather = tf.gather(params, indices, validate_indices=True, name=None) #=> [2,5]

# n-мерная операция gather.
params = tf.constant([[0,0,0],[1,1,1],[2,2,2],[3,3,3]],dtype=tf.float32)
indices = [[0],[2]]
tf_gather_nd = tf.gather_nd(params, indices, name=None) #=>[[0,0,0],[2,2,2]]

params = tf.constant([[0,0,0],[1,1,1],[2,2,2],[3,3,3]],dtype=tf.float32)
indices = [[0,1],[2,2]]
tf_gather_nd_2 = tf.gather_nd(params, indices, name=None) #=>[[0,0,0],[2,2,2]]
```

Операции, связанные с нейронными сетями

Рассмотрим несколько полезных операций, связанных с нейронными сетями, которые мы будем интенсивно использовать в следующих главах. Операции, которые мы здесь обсудим, варьируются от простых поэлементных преобразований (т. е. активаций) до вычисления частных производных набора параметров по другому значению. В качестве упражнения реализуем простую нейронную сеть.

Нелинейная активация

Между уровнями нейронной сети, как правило, выполняются нелинейные преобразования. Наличие нелинейности помогает нейронной сети моделировать

шаблоны данных, поведение которых определяется нелинейной функцией. Это очень полезно для решения сложных реальных задач, где данные часто имеют сложные нелинейные зависимости. Без нелинейных активаций между слоями глубокая нейронная сеть останется лишь пакетом последовательных линейных преобразований. Кроме того, пакет линейных слоев можно сжать до одного линейного слоя. Короче говоря, без нелинейных активаций мы не сможем создать нейронную сеть с более чем одним слоем.

Давайте рассмотрим важность нелинейной активации на примере. Сначала вспомним вычисления, которые мы применяли в примере с сигмоидой. Если мы игнорируем свободный член `b`, операция будет выглядеть так:

```
h = sigmoid(W*x)
```

Предположим, что у нас есть трехслойная нейронная сеть с весами слоев `W1`, `W2` и `W3`, где каждый слой выполняет нелинейное преобразование выхода предыдущего слоя. Тогда мы можем записать полное вычисление следующим образом:

```
h = sigmoid(W3*sigmoid(W2*sigmoid(W1*x)))
```

Однако если мы удалим нелинейную активацию (т. е. `sigmoid`), то получим следующее вычисление:

```
h = (W3*(W2*(W1*x))) = (W3*W2*W1)*x
```

Таким образом, без нелинейных активаций три слоя можно свести к одному линейному слою.

В нейронных сетях для нелинейной активации наиболее часто используют сигмоиду и ReLU. В TensorFlow имеются соответствующие операции:

```
# Sigmoid реализует функцию вида 1/(1+exp(-x)).
tf.nn.sigmoid(x,name=None)
# ReLU реализует функцию вида max(0,x).
tf.nn.relu(x,name=None)
```

Операция свертки

Операция свертки широко применяется в различных технологиях обработки сигналов. В частности, свертка используется для обработки изображений и создания различных эффектов. Пример обнаружения кромки объекта при помощи операции свертки показан на рис. 2.6. Задача решается путем перемещения *сверточного фильтра*[1] (convolutional filter) поверх изображения и получения различного выходного сигнала в отдельных местах растрового поля (рис. 2.7). В частности, в каждом месте изображения мы выполняем поэлементное перемножение сверточного фильтра с фрагментом изображения, который он накрывает. Результаты перемножения суммируются в одном пикселе.

В следующем листинге показана реализация операции свертки:

```
x = tf.constant(
    [[
        [[1],[2],[3],[4]],
```

[1] Фильтр для одного канала, например одного цвета в палитре RGB, называется *ядром свертки*. В общем случае фильтр – это коллекция ядер. В случае монохромного изображения понятия ядра и фильтра взаимозаменяемы. – *Прим. перев.*

```
        [[4],[3],[2],[1]],
        [[5],[6],[7],[8]],
        [[8],[7],[6],[5]]
    ]],
    dtype=tf.float32)

x_filter = tf.constant(
    [
        [
            [[0.5]],[[1]]
        ],
        [
        [[0.5]],[[1]]
        ]
    ],
    dtype=tf.float32)

x_stride = [1,1,1,1]
x_padding = 'VALID'

x_conv = tf.nn.conv2d(
    input=x, filter=x_filter,
    strides=x_stride, padding=x_padding
)
```

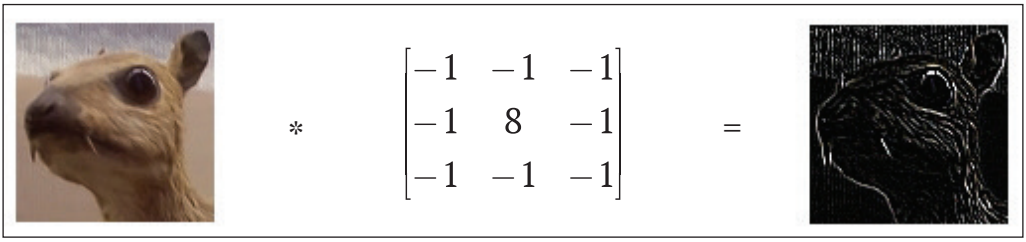


Рис. 2.6 ❖ Определение границ изображения при помощи операции свертки.
Источник: **https://en.wikipedia.org/wiki/Kernel_(image_processing)**

В данном случае обилие квадратных скобок может заставить вас думать, что листинг можно значительно упростить, избавившись от «лишних» скобок. К сожалению, это не так. TensorFlow требует, чтобы в операции `tf.conv2d(input, filter, strides, padding)` аргументы имели точный формат. Рассмотрим каждый аргумент операции более подробно:

- **input** – обычно это 4-мерный тензор, где размеры должны быть упорядочены как `[batch_size, height, width, channel]`;
  - **batch_size** – объем данных (например, таких как изображения и слова) в одном пакете. Обычно мы обрабатываем данные партиями, так как для обучения часто используются большие наборы данных. На каждом шаге обучения мы случайным образом выбираем небольшой пакет данных, который приблизительно представляет полный набор данных. Выполнение таких выборок в течение многих шагов позволяет нам достаточно хорошо аппроксимировать полный набор данных. Параметр `batch_size` мы уже обсуждали в примере входного конвейера TensorFlow;

  - **height** и **width** – высота и ширина ввода;
  - **channel** – глубина входного сигнала (например, у изображения RGB 3 канала – по одному для каждого цвета);
- **filter** – это 4-мерный тензор, который представляет окно операции свертки, т. е. ядро. Размеры фильтра упорядочены как `[height, width, in_channels, out_channels]`:
  - **height** и **width** – высота и ширина фильтра (обычно меньше, чем у входа);
  - **in_channels** – количество каналов на входе;
  - **out_channels** – количество каналов, которые будут созданы на выходе;
- **strides** (шаги) – список из четырех аргументов `[batch_stride, height_stride, width_stride, channel_stride]`. Аргумент `strides` указывает, сколько элементов пропустить за один сдвиг окна свертки на входе. Если вы не совсем понимаете, что такое `strides`, то можете использовать значение по умолчанию `1`;
- **padding** (заполнение) – может принимать одно из значений `['SAME', 'VALID']`, определяющее, как выполнять операцию свертки вблизи границ ввода. Операция `VALID` выполняет свертку без заполнения. Если мы свернем вход n длины с окном свертки размера h, это приведет к выводу размера ($n - h + 1 < n$). Уменьшение размера вывода может серьезно ограничить глубину нейронных сетей. `SAME` добавляет нули вокруг границы поля входных данных так, чтобы выходные данные имели ту же высоту и ширину, что и входные.

Чтобы лучше понять, что такое размер фильтра, шаг и заполнение, обратитесь к рис. 2.7.

Операция подвыборки

Операция *подвыборки* (pooling) работает аналогично операции свертки, но конечный результат отличается. Вместо того чтобы выводить сумму поэлементного умножения фильтра и области изображения, теперь мы берем максимальный элемент из текущей области изображения (рис. 2.8).

```
x = tf.constant(
    [[
        [[1],[2],[3],[4]],
        [[4],[3],[2],[1]],
        [[5],[6],[7],[8]],
        [[8],[7],[6],[5]]
    ]],
    dtype=tf.float32)
x_ksize = [1,2,2,1]
x_stride = [1,2,2,1]
x_padding = 'VALID'

x_pool = tf.nn.max_pool(
    value=x, ksize=x_ksize,
    strides=x_stride, padding=x_padding
)
# Returns (out) =>
[[[[ 4.]
   [ 4.]],
  [[ 8.]
   [ 8.]]]]
```

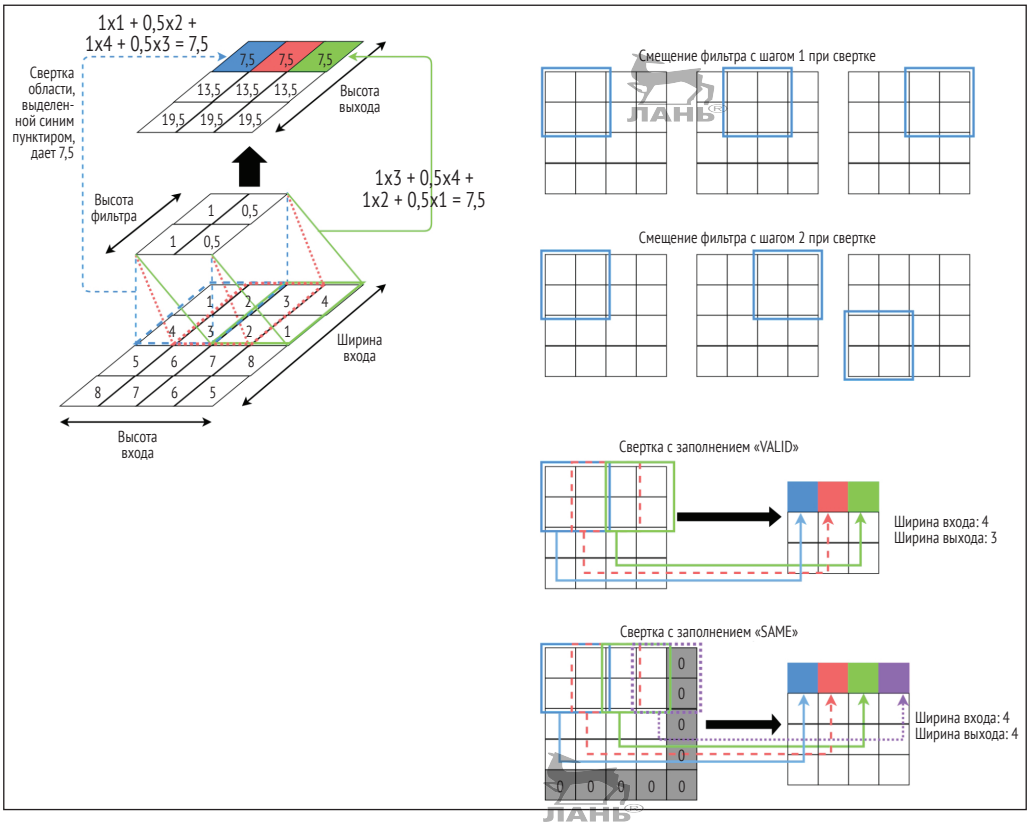


Рис. 2.7 ❖ Операция свертки

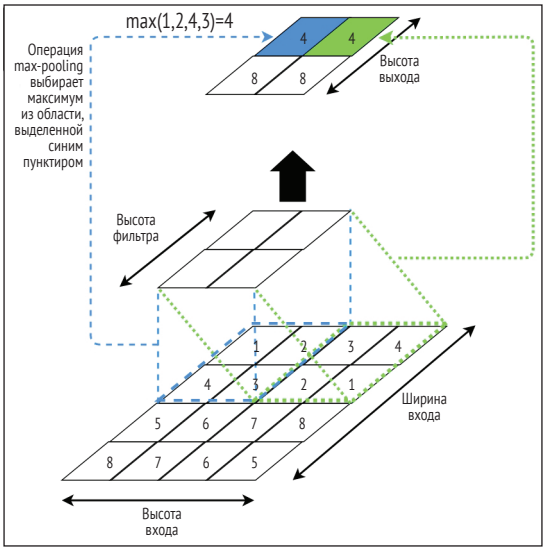


Рис. 2.8 ❖ Операция подвыборки max_pool

Определение потерь

Чтобы нейронная сеть научилась чему-то полезному, нужно уметь определять потери на каждом шаге обучения. В TensorFlow есть несколько функций для автоматического расчета потерь, две из которых показаны в следующем коде. Функция `tf.nn.l2_loss` – это среднеквадратичная ошибка, а `tf.nn.softmax_cross_entropy_with_logits_v2` – это другой тип потерь, который на самом деле лучше работает в задачах классификации. Здесь под *логитом* (logit) мы подразумеваем ненормализованный вывод нейронной сети (т. е. линейный вывод последнего слоя нейронной сети):

```
# Возвращает результат вычисления sum(t**2)/2
x = tf.constant([[2,4],[6,8]],dtype=tf.float32)
x_hat = tf.constant([[1,2],[3,4]],dtype=tf.float32)
# MSE = (1**2 + 2**2 + 3**2 + 4**2)/2 = 15
MSE = tf.nn.l2_loss(x-x_hat)

# Типовая функция потерь, применяемая для оптимизации нейросети.
# Вычисляет перекрестную энтропию с логитами,
# а не с выходами, приводящими к лучшей численной устойчивости.

y = tf.constant([[1,0],[0,1]],dtype=tf.float32)
y_hat = tf.constant([[3,1],[2,5]],dtype=tf.float32)
# Эта функция не усредняет потери перекрестной энтропии
# по всем точкам данных.
# Вы должны сделать это при помощи функции reduce_mean.
CE = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_
v2(logits=y_hat,labels=y))
```

Оптимизация нейронных сетей

После определения потери нейронной сети наша цель – минимизировать эту потерю в процессе обучения. Для этого предназначена процедура оптимизации. Другими словами, оптимизатор должен найти параметры нейронной сети, т. е. весовые коэффициенты и значения смещения, обеспечивающие минимальные потери на всем множестве входных данных. К счастью, наш любимый TensorFlow предоставляет несколько различных оптимизаторов, поэтому нам не нужно беспокоиться об их реализации с нуля.

Рисунок 2.9 иллюстрирует процесс постепенной оптимизации. На нем изображена так называемая *кривая потерь* (в случае больших размерностей мы говорим о *поверхности потерь*), где x – это параметр нейронной сети с одним весом, а y – потеря. У нас есть первоначальное предположение о том, что $x = 2$. С этого момента мы используем оптимизатор для достижения минимального значения y, которое в данном случае получается при $x = 0$. Мы делаем небольшие шаги в направлении, противоположном градиенту в данной точке, и постепенно спускаемся ниже и ниже. Однако в реальных задачах кривая или поверхность потерь будет не такой гладкой, как на рис. 2.9. На самом деле она будет выглядеть намного сложнее.

В следующем примере мы используем `GradientDescentOptimizer`. Параметр `learning_rate` обозначает размер шага в направлении минимизации потери (расстояние между двумя красными точками):

```
# Оптимизаторы играют важную роль в настройке параметров нейросети,
# чтобы минимизировать потери при решении задачи.
```

```
# Например, в задачах классификации это потеря перекрестной энтропии.
tf_x = tf.Variable(tf.constant(2.0,dtype=tf.float32),name='x')
tf_y = tf_x**2
minimize_op = tf.train.GradientDescentOptimizer(learning_rate=0.1).
minimize(tf_y)
```

Рис. 2.9 ❖ Процесс оптимизации

Каждый раз, когда вы выполняете операцию минимизации потерь с помощью `session.run (minimal_op)`, вы приближаетесь к значению `tf_x`, которое дает минимальное значение `tf_y`.

Операции управления потоками

Операции *управления потоком*, как следует из названия, контролируют порядок выполнения вычислений в графе. Допустим, нам нужно выполнить следующие вычисления в заданном порядке:

$$x = x+5$$
$$z = x*2$$

В таком случае, если `x=2`, мы должны получить `z=14`. Сначала попробуем добиться этого самым простым способом:

```
session = tf.InteractiveSession()

x = tf.Variable(tf.constant(2.0), name='x')
x_assign_op = tf.assign(x, x+5)
z = x*2

tf.global_variables_initializer().run()
print('z=',session.run(z))
print('x=',session.run(x))
session.close()
```

В идеале, мы бы хотели увидеть вывод `x=7` и `z=14`, но вместо этого TensorFlow может вывести `x=2` и `z=4`. Это не тот ответ, который вы ожидали. Запомните: TensorFlow не заботится о порядке выполнения вычислений, если вы не укажете это явно. Чтобы исправить предыдущий код, воспользуемся операцией управления потоками:

```
session = tf.InteractiveSession()

x = tf.Variable(tf.constant(2.0), name='x')
```

```
with tf.control_dependencies([tf.assign(x, x+5)]): z = x*2

tf.global_variables_initializer().run()
print('z=',session.run(z))
print('x=',session.run(x))
session.close()
```

Теперь мы точно получим `x=7` и `z=14`. Операция `tf.control_dependencies(...)` гарантирует, что переданные ей в качестве аргументов операции будут выполнены перед выполнением вложенной операции.

Повторное использование переменных с областью видимости

До сих пор мы рассматривали архитектуру TensorFlow и основы, необходимые для реализации базового клиента. Тем не менее TensorFlow может гораздо больше. Как вы уже убедились, TensorFlow ведет себя совершенно иначе, чем типичный скрипт на Python. Например, вы не можете отлаживать код TensorFlow «на ходу», как привыкли это делать со скриптами Python в среде разработки, поскольку в TensorFlow вычисления не выполняются в режиме реального времени, – если только вы не используете метод Eager Execution, который относительно недавно появился в TensorFlow 1.7: **https://research.googleblog.com/2017/10/eager-execution-imperative-define-by.html**. Другими словами, TensorFlow сначала определяет полный вычислительный граф, выполняет все вычисления на устройстве и, наконец, извлекает результаты. Следовательно, отладка клиента TensorFlow может быть довольно утомительной и раздражающей. Это подчеркивает важность внимания к деталям при реализации клиента TensorFlow. Поэтому рекомендуется придерживаться определенной практики кодирования, принятой для TensorFlow.

Один из подходов известен как определение области видимости и облегчает повторное использование переменных – это распространенный сценарий, который часто встречается в клиентах TensorFlow. Чтобы осознать ценность ответа, мы должны сначала понять вопрос. А что может быть лучше для понимания вопроса, чем ошибочный код? Допустим, вам нужна функция, которая выполняет определенные вычисления. Зная `w`, вам нужно вычислить `x*w+y**2`. Вы создаете клиента TensorFlow, выполняющего вычисления:

```
import tensorflow as tf
session = tf.InteractiveSession()
def very_simple_computation(w):
  x = tf.Variable(tf.constant(5.0, shape=None, dtype=tf.float32), name='x')
  y = tf.Variable(tf.constant(2.0, shape=None, dtype=tf.float32), name='y')
  z = x*w + y**2
  return z
```

Затем последовательно вызываете команды

```
tf.global_variable_initializer().run()
session.run(very_simple_computation(2))
```

и получаете ожидаемый ответ. Но если вы решите запустить этот код несколько раз подряд, то столкнетесь с проблемой. Каждый раз при запуске будут созданы две переменные TensorFlow. Помните, что TensorFlow отличается от Python? Это один из таких случаев. Переменные `x` и `y` не будут обновляться в графе при многократном вызове метода. Точнее, старые переменные сохранятся, а новые переменные будут создаваться в графе до тех пор, пока не закончится память компьютера. Но, разумеется, результат вычислений будет правильным. Чтобы увидеть это в действии, запустите `session.run(very_simple_computation(2))` в цикле `for`, и если вы выведете на печать имена переменных в графе, то увидите более двух значений. Так выглядит вывод, когда вы запускаете вычисления 10 раз:

```
'x:0', 'y:0', 'x_1:0', 'y_1:0', 'x_2:0', 'y_2:0', 'x_3:0', 'y_3:0',
'x_4:0', 'y_4:0', 'x_5:0', 'y_5:0', 'x_6:0', 'y_6:0', 'x_7:0',
'y_7:0', 'x_8:0', 'y_8:0', 'x_9:0', 'y_9:0', 'x_10:0', 'y_10:0'
```

Каждый раз, когда вы запускаете код, создается пара переменных. Например, если вы запустите функцию `session.run()` 100 раз, в графе окажется 198 устаревших переменных (99 переменных `x` и 99 переменных `y`).

В таких случаях помогает *область видимости* (scope). Она позволяет вам повторно использовать переменные, а не создавать их каждый раз, когда вызывается функция. Теперь, чтобы добавить возможность многократного использования переменных, мы изменим код на следующий:

```
def not_so_simple_computation(w):
  x = tf.get_variable('x', initializer=tf.constant (5.0, shape=None,
                      dtype=tf.float32))
  y = tf.get_variable('y', initializer=tf.constant(2.0, shape=None,
                      dtype=tf.float32))
  z = x*w + y**2
  return z

def another_not_so_simple_computation(w):
  x = tf.get_variable('x', initializer=tf.constant(5.0, shape=None,
                      dtype=tf.float32))
  y = tf.get_variable('y', initializer=tf.constant(2.0, shape=None,
                      dtype=tf.float32))
  z = w*x*y
  return z

# Поскольку это первый вызов, то будут созданы
# переменные со следующими именами
# x => scopeA/x, y => scopeA/y
with tf.variable_scope('scopeA'):
  z1 = not_so_simple_computation(tf.constant(1.0,dtype=tf.float32))
# scopeA/x и scopeA/y уже созданы, используем повторно
with tf.variable_scope('scopeA',reuse=True):
  z2 = another_not_so_simple_computation(z1)

# Поскольку это первый вызов, то будут созданы
# переменные со следующими именами
# following names x => scopeB/x, y => scopeB/y
with tf.variable_scope('scopeB'):
  a1 = not_so_simple_computation(tf.constant(1.0,dtype=tf.float32))
```

```
# scopeB/x и scopeB/y уже созданы, используем повторно
with tf.variable_scope('scopeB',reuse=True):
  a2 = another_not_so_simple_computation(a1)

# Допустим, мы хотим повторно использовать переменные в области scopeA.
# Поскольку переменные уже созданы, мы должны установить аргумент reuse=True
with tf.variable_scope('scopeA',reuse=True):
  zz1 = not_so_simple_computation(tf.constant(1.0,dtype=tf.float32))
  zz2 = another_not_so_simple_computation(z1)
```

После выполнения команды `session.run([z1,z2,a1,a2,zz1,zz2])` значения `z1`, `z2`, `a1`, `a2`, `zz1`, `zz2` будут выведены в следующем порядке: `9.0`, `90.0`, `9.0`, `90.0`, `9.0`, `90.0`. При просмотре содержимого графа вы должны увидеть только четыре разные переменные: `scopeA/x`, `scopeA/y`, `scopeB/x` и `scopeB/y`. Теперь мы можем запускать вычисления в цикле сколько угодно раз, не беспокоясь о создании избыточных переменных и нехватке памяти.

Вы можете спросить, почему нельзя просто создать четыре переменные в начале кода и использовать их в методах. Однако это нарушает инкапсуляцию вашего кода, потому его выполнение будет зависеть от переменных, которые находятся вне кода и могут быть изменены другим процессом.

Область видимости позволяет повторно использовать код, сохраняя инкапсуляцию. Кроме того, область видимости делает код интуитивно понятным и снижает вероятность ошибок, поскольку мы явно определяем переменную по области и имени вместо использования переменной Python, которая ссылается на переменную TensorFlow.

Реализация нашей первой нейронной сети

Великолепно! Теперь, когда вы изучили архитектуру, основы и определение области деятельности TensorFlow, пришло время двигаться дальше и реализовать умеренно сложную нейросеть. Точно, мы будем реализовывать модель полносвязной нейронной сети, которую обсуждали в главе 1.

Одним из общепринятых этапов изучения нейронных сетей является разработка нейронной сети, способной классифицировать цифры. Для этой задачи мы будем использовать знаменитый набор данных MNIST, доступный по адресу **http://yann.lecun.com/exdb/mnist/**. Вы можете скептически отнестись к идее изучить пример компьютерного зрения в книге про NLP. Но дело в том, что задача компьютерного зрения нуждается в меньшей предварительной обработке данных и проще для понимания.

Поскольку это наша первая встреча с нейронными сетями, мы детально рассмотрим основные этапы решения задачи. Тем не менее обратите внимание, что я пройдусь только по ключевым частям упражнения. Чтобы разобрать упражнение от начала до конца, воспользуйтесь файлом `tenorflow_introduction.ipynb` в папке `ch2`.

Подготовка данных

Во-первых, нам нужно загрузить набор данных с помощью функции `maybe_download(...)` и предварительно обработать его, применив функцию `read_mnist(...)`.

Эти две функции определены в файле упражнения. Вторая функция выполняет два основных шага:

- чтение байтового потока набора данных и преобразование его в правильный объект `numpy.ndarray`;
- стандартизация изображений для получения нулевого среднего и единичной дисперсии, т. е. *отбеливание* (whitening).

В следующем коде показана реализация функции `read_mnist(...)`. Функция принимает в качестве входных данных имя файла, содержащего изображения, и имя файла, содержащего метки. Затем она создает две матрицы NumPy, содержащие все изображения и соответствующие им метки:

```
def read_mnist(fname_img, fname_lbl):
  print('\nReading files  %s and  %s' %(fname_img, fname_lbl))

with gzip.open(fname_img) as fimg:
  magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))
  print(num,rows,cols)
  img = (np.frombuffer(fimg.read(num*rows*cols), dtype=np.uint8).
         reshape(num, rows * cols)).astype(np.float32)
  print('(Images) Returned a tensor of shape ',img.shape)
  # Стандартизация изображений.
  img = (img - np.mean(img))/np.std(img)

with gzip.open(fname_lbl) as flbl:
  # flbl.read(8) читает по 8 байт.
  magic, num = struct.unpack(">II", flbl.read(8))
  lbl = np.frombuffer(flbl.read(num), dtype=np.int8)
  print('(Labels) Returned a tensor of shape:  %s' %lbl.shape)
  print('Sample labels: ',lbl[:10])

return img, lbl
```

Определение графа TensorFlow

Чтобы определить граф TensorFlow, мы сначала объявим заполнители для входных изображений (`tf_inputs`) и соответствующих меток (`tf_labels`):

```
# Определяем входы и выходы.
tf_inputs = tf.placeholder(shape=[batch_size, input_size], dtype=tf.
float32, name = 'inputs')
tf_labels = tf.placeholder(shape=[batch_size, num_labels], dtype=tf.
float32, name = 'labels')
```

Затем напишем функцию Python, которая будет создавать переменные в первый раз. Определим области видимости, чтобы обеспечить возможность повторного использования кода с гарантией обращения к правильным переменным:

```
# Определяем переменные TensorFlow.
def define_net_parameters():
  with tf.variable_scope('layer1'):
    tf.get_variable(WEIGHTS_STRING,shape=[input_size,500],
    initializer=tf.random_normal_initializer(0,0.02))
    tf.get_variable(BIAS_STRING, shape=[500],
    initializer=tf.random_uniform_initializer(0,0.01))
```

```
  with tf.variable_scope('layer2'):
    tf.get_variable(WEIGHTS_STRING,shape=[500,250],
    initializer=tf.random_normal_initializer(0,0.02))
    tf.get_variable(BIAS_STRING, shape=[250],
    initializer=tf.random_uniform_initializer(0,0.01))

  with tf.variable_scope('output'):
    tf.get_variable(WEIGHTS_STRING,shape=[250,10], initializer=tf.
    random_normal_initializer(0,0.02))
    tf.get_variable(BIAS_STRING, shape=[10], initializer=tf.random_
    uniform_initializer(0,0.01))
```

Далее мы объявим процесс вывода для нейронной сети. Обратите внимание, что область видимости обеспечивает интуитивно более понятную последовательность кода по сравнению с использованием переменных без области видимости. Итак, в этой сети у нас есть три слоя:

- полностью связанный слой с активацией ReLU (`layer1`);
- полностью связанный слой с активацией ReLU (`layer2`);
- полностью связанный слой softmax (`output`).

С учетом области видимости мы называем переменные (весовые коэффициенты и смещения) для каждого слоя следующим образом: `layer1/weights`, `layer1/bias`, `layer2/weights`, `layer2/bias`, `output/weights` и `output/bias`. Обратите внимание, что в коде все они имеют одинаковые имена, но разные области видимости:

```
# Определяем вычисления в нейросети
# начиная со входов и до логитов.
# Логиты – это значения до применения softmax и окончательного выхода.

def inference(x):
  # Вычисления для слоя 1.
  with tf.variable_scope('layer1',reuse=True):
    w,b = tf.get_variable(WEIGHTS_STRING),
                          tf.get_variable(BIAS_STRING)
    tf_h1 = tf.nn.relu(tf.matmul(x,w) + b, name = 'hidden1')

# Вычисления для слоя 2.
with tf.variable_scope('layer2',reuse=True):
  w,b = tf.get_variable(WEIGHTS_STRING),
                        tf.get_variable(BIAS_STRING)
  tf_h2 = tf.nn.relu(tf.matmul(tf_h1,w) + b, name = 'hidden1')

# Вычисления для выходного слоя.
with tf.variable_scope('output',reuse=True):
  w,b = tf.get_variable(WEIGHTS_STRING),
                        tf.get_variable(BIAS_STRING)
  tf_logits = tf.nn.bias_add(tf.matmul(tf_h2,w), b, name = 'logits')

return tf_logits
```

Теперь мы определим функцию потерь, а затем операцию минимизации потерь. Упомянутая операция «подталкивает» параметры нейросети в направлении уменьшения потерь. В TensorFlow есть обширная коллекция оптимизаторов. В данном случае мы будем использовать `MomentumOptimizer`, который дает лучшую конечную точность и конвергенцию, чем `GradientDescentOptimizer`:

```
# Определение потери.
tf_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_
v2(logits=inference(tf_inputs), labels=tf_labels))
# Определение функции оптимизации.
tf_loss_minimize = tf.train.MomentumOptimizer(momentum=0.9,learning_
rate=0.01).minimize(tf_loss)
```

Наконец, мы определим операцию извлечения предсказанных вероятностей для данной партии входных данных. Они, в свою очередь, будут применяться для расчета точности вашей нейронной сети:

```
# Определяем предсказания
tf_predictions = tf.nn.softmax(inference(tf_inputs))
```

Запуск нейронной сети

Теперь у нас есть все основные операции, необходимые для запуска нейронной сети и проверки ее способности обучаться классификации цифр:

```
for epoch in range(NUM_EPOCHS):
  train_loss = []

  # Фаза обучения
  for step in range(train_inputs.shape[0]//batch_size):

    # Создаем унитарные коды меток.
    # Например, унитарный код цифры 3 в 10-цифровом наборе MNIST
    # имеет следующий вид:
    # [0,0,0,1,0,0,0,0,0,0]
    labels_one_hot = np.zeros((batch_size, num_labels),
                              dtype=np.float32)
    labels_one_hot[np.arange(batch_size),train_labels[
    step*batch_size:(step+1)*batch_size]] = 1.0

    # Выполняем процесс оптимизации
    loss, _ = session.run([tf_loss,tf_loss_minimize],feed_dict={
    tf_inputs: train_inputs[step*batch_size: (step+1)*batch_size,:],
    tf_labels: labels_one_hot})
    train_loss.append(loss)
# Применяется для усреднения потери в одной эпохе

    test_accuracy = []
    # Фаза тестирования.
    for step in range(test_inputs.shape[0]//batch_size):
      test_predictions = session.run(tf_predictions,feed_dict={tf_
inputs: test_inputs[step*batch_size: (step+1)*batch_size,:]})
      batch_test_accuracy = accuracy(test_predictions,test_
labels[step*batch_size: (step+1)*batch_size])
      test_accuracy.append(batch_test_accuracy)

    print('Average train loss for the  %d epoch:  %.3f\n' %(epoch+1,np.
mean(train_loss)))
    print('\tAverage test accuracy for the  %d epoch:
 %.2f\n' %(epoch+1,np.mean(test_accuracy)*100.0))
```

В данном коде `accuracy(test_predictions,test_labels)` – это функция, которая принимает некоторые прогнозы и метки в качестве входных данных и вычисляет точность (сколько прогнозов соответствует фактической метке). Она определена в полном файле упражнения.

В случае успеха поведение нейросети будет аналогично показанному на рис. 2.10. После 50 эпох точность классификации на проверочных данных должна составить примерно 98 %.

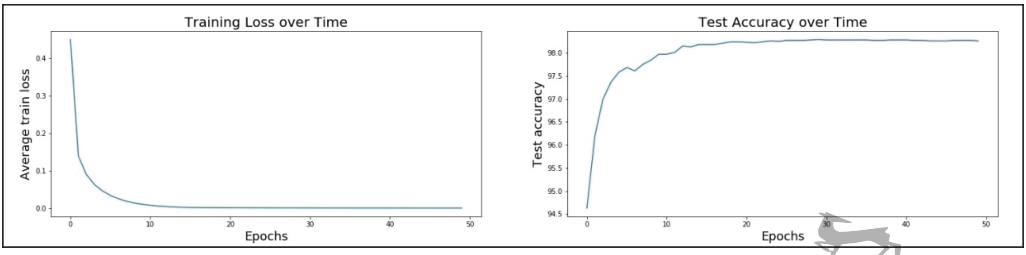


Рис. 2.10 ❖ Потери при обучении и проверка точности в задаче распознавания цифр

Заключение

В этой главе вы сделали первые шаги к решению задач NLP, изучив основы платформы TensorFlow, на которой мы будем реализовывать наши алгоритмы. Сначала мы обсудили основные понятия архитектуры и клиента TensorFlow, затем рассмотрели общую практику кодирования, широко используемую в TensorFlow, – определение областей видимости, – и объединили все полученные знания в примере нейронной сети для классификации набора данных MNIST.

В частности, мы изучали архитектуру TensorFlow, выстраивая объяснение на примере клиента. В клиенте TensorFlow мы определяем граф вычислений. Затем, когда мы создаем сессию, она получает граф, создает представление графа – объект `GraphDef` – и отправляет его распределенному мастеру. Мастер исследует граф, решает, какие компоненты использовать для соответствующих вычислений, и разделяет его на несколько подграфов, чтобы ускорить вычисления. Наконец, исполнители выполняют подграфы и возвращают результат через сессию.

Далее мы обсудили различные элементы, составляющие типичного клиента TensorFlow: входные данные, переменные, выходные данные и операции. Входные данные – это данные, которые мы передаем в алгоритм для обучения и тестирования. Мы рассмотрели три различных способа передачи входных данных: использование заполнителей, предварительная загрузка данных и сохранение данных в качестве тензоров TensorFlow с последующим использованием входного конвейера. Затем обсудили переменные TensorFlow, их отличия от других тензоров, создание и инициализацию. После этого вы узнали, как использовать переменные для создания промежуточных и конечных выходов. Наконец, вы познакомились с базовыми операциями TensorFlow, такими как математические операции, матричные операции, операции, связанные с нейронной сетью, и операции управления потоками, которые пригодятся вам позже.

Вы узнали, как можно использовать область видимости, чтобы избежать определенных ошибок при реализации клиента TensorFlow. Область видимости позволяет легко использовать переменные, сохраняя при этом инкапсуляцию кода.

Наконец, вы реализовали свою первую нейронную сеть, используя все ранее изученные концепции. Это была трехслойная нейросеть для классификации набора рукописных цифр MNIST.

В следующей главе вы увидите, как использовать полносвязную нейронную сеть, реализованную в этой главе, для изучения числового представления слов с учетом семантики.

Глава 3

Word2vec и вектор слова в пространстве смыслов

В этой главе мы обсудим тему первостепенной важности в NLP – Word2vec, *изучение представлений слова* (learning word embeddings)[1] в пространстве смыслов, или, иными словами, методику извлечения *распределенных числовых признаков* (т. е. векторов) слов. Изучение представлений слов лежит в основе многих задач NLP, потому что для их решения требуется хорошее представление признаков слов, сохраняющее семантику слова и его контекст в языке. Например, хорошее представление слова «*лес*» должно сильно отличаться от представления слова «*холодильник*», поскольку эти слова редко используются в сходных контекстах, тогда как представления слов «*лес*» и «*джунгли*» должны быть очень близкими.

Подход Word2vec называется *распределенным представлением*, так как семантика слова представляется шаблоном активации *полного* вектора представления, в отличие от одноэлементного представления в векторе (например, прямого унитарного кодирования).

В этой главе вы шаг за шагом перейдете от классического подхода к современным методам на основе нейронных сетей, обеспечивающим высокое качество поиска хороших представлений слов. Представление слов в пространстве смыслов можно наглядно проиллюстрировать в виде набора слов на двумерном холсте при помощи техники визуализации многомерных данных t-SNE (рис. 3.1). Вглядевшись в рисунок, вы увидите, что похожие понятия расположены близко друг к другу (например, числительные в кластере посередине).

---

[1] Русский перевод термина *Learning Word Embedding* до сих пор является предметом ожесточенных споров среди прикладных специалистов по глубокому обучению, поскольку ни один общепринятый термин на русском языке не передает полный смысл исходного определения. Подход Word2vec предусматривает машинный поиск векторного **представления** слова в пространстве смыслов естественного языка, поэтому в данной книге *Learning Word Embedding* переводится как «*изучение представления слова*». Этот термин применяется рядом авторитетных российских специалистов, например: *Николенко и др.* Глубокое обучение. Погружение в мир нейронных сетей. 2018. – *Прим. перев.*

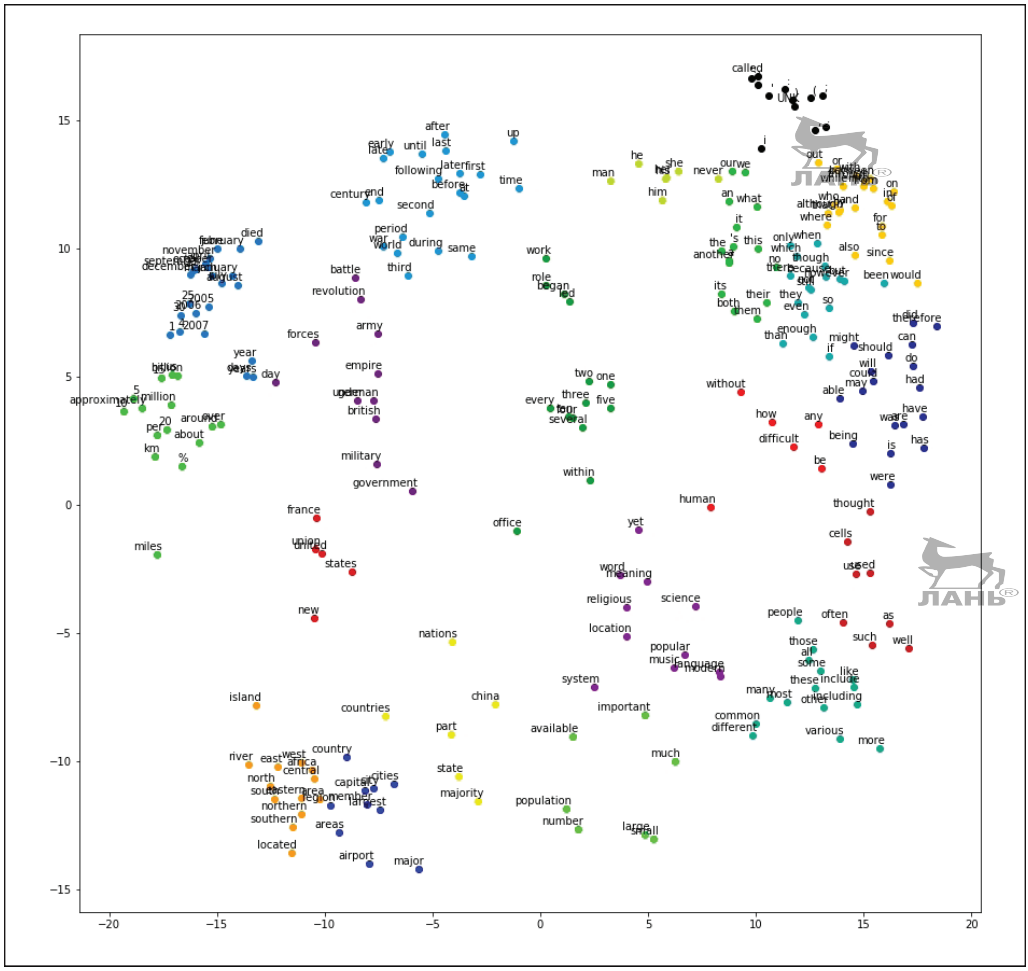


Рис. 3.1 ❖ Пример визуализации представлений слов с использованием t-SNE

t-распределенное представление стохастической близости (t-distributed stochastic neighbor embedding, t-SNE) – это метод уменьшения размерности, который проецирует многомерные данные в двумерное пространство. Он позволяет нам оценить, как данные распределены в многомерном пространстве, и это весьма полезно, поскольку нашему мозгу плохо дается визуализация за пределами трех измерений. Подробнее о t-SNE вы узнаете в следующей главе.

Что такое представление и значение слова?

Что мы подразумеваем под *значением слова*? Это вопрос скорее философский, чем технический. Мы не будем пытаться найти самый правильный ответ, а ограничимся скромным определением, что *значение – это идея или понятие, передаваемое словом и облеченное в его звуковую или символьную оболочку*. Поскольку основной целью NLP является достижение человеческого уровня в решении лингвистиче-

ских задач, необходимо тщательно проработать способы представления слов для машин. Нам нужны алгоритмы, способные анализировать заданный текстовый корпус и выдавать хорошее числовое *представление слова* (word embedding) таким образом, чтобы слова из одного контекста, например «*один*» и «*два*», «*я*» и «*мы*», были представлены близкими векторами, а представления слов из разных контекстов, например «*кошка*» и «*вулкан*», значительно различались.

Сначала мы обсудим некоторые классические подходы к формированию представлений слов, а затем перейдем к освоению более сложных современных методов, использующих нейронные сети для изучения представлений слов.

Классические подходы к представлению слов

В этом разделе мы обсудим некоторые классические подходы, применяемые для числового представления слов. В целом их можно разделить на два класса: подходы, которые используют внешние ресурсы для представления слов, и подходы, которые этого не делают. Сначала мы обсудим WordNet – один из самых популярных методов представления слов на основе внешних ресурсов. Затем перейдем к более локализованным методам (то есть тем, которые не зависят от внешних ресурсов), таким как *унитарное кодирование* (one-hot encoding) и *частота слова – обратная частота документа* (term frequency – inverse document frequency, TF-IDF).

Внешняя лексическая база знаний WordNet для изучения представлений слов

WordNet – один из самых популярных классических подходов или статистических NLP, работающих с представлениями слов. Он опирается на внешнюю *лексическую базу знаний*, содержащую информацию об определении, синонимах, предках и потомках данного слова. WordNet позволяет пользователю выводить различную информацию, такую как аспекты слова, которые фигурируют в предыдущем предложении, и сходство между двумя словами.

Обзорное знакомство с WordNet

Как уже упоминалось, WordNet представляет собой лексическую базу данных, кодирующую отношения тегов части речи между словами, включая существительные, глаголы, прилагательные и наречия.

База данных WordNet была впервые создана на факультете психологии Принстонского университета, США, и в настоящее время размещена на сервере факультета компьютерных наук Принстонского университета. Чтобы оценить связь между словами, WordNet рассматривает наборы синонимов. Английская WordNet в настоящее время содержит более 150 000 слов и более 100 000 наборов синонимов. Кроме того, WordNet не ограничивается только английским языком[1]. К сегодняшнему дню создано множество различных лексических баз данных. Их перечень можно посмотреть по адресу **http://globalwordnet.org/wordnets-in-the-world/**.

[1] Частично переведенная русская версия WordNet доступна по адресу **http://wordnet.ru/**. – *Прим. перев.*

Чтобы понять, как использовать WordNet, следует хорошо усвоить специфическую терминологию. Во-первых, WordNet использует термин *синсет* (synset) для обозначения группы или набора синонимов. Далее, у каждого набора есть *описание* (definition), которое объясняет, что представляет собой набор. Синонимы, содержащиеся в синтаксисе, называются *леммами* (lemma).

В WordNet представления слов моделируются иерархически, что формирует сложный граф между данным набором и ассоциациями с другим набором. Эти ассоциации делятся на две категории: «*является тем-то*...» и «*состоит из*...». Сначала поговорим о первой категории.

Для определенного синсета существует два вида отношений: *гиперонимы* (hypernym) и *гипонимы* (hyponym). Гипероним синсета – это группа синсетов, которая несет более общее (высокоуровневое) значение рассматриваемого синсета. Например, «*транспорт*» является гиперонимом синсета «*автомобиль*». Далее, гипонимы – это синсеты, которые более специфичны, чем соответствующий синсет. Например, «*автомобиль Toyota*» – это гипоним понятия «*автомобиль*».

Теперь рассмотрим категорию отношений «*состоит из*...». *Голоним* (holonym) синсета – это группа синсетов, представляющая всю сущность рассматриваемого синсета. Например, голонимом для «*шин*» является синсет «*автомобили*». *Мероним* (meronym) представляет собой противоположность голониму и означает синсет частей или веществ, который создает соответствующий более общий синсет. Перечисленные отношения схематически изображены на рис. 3.2.

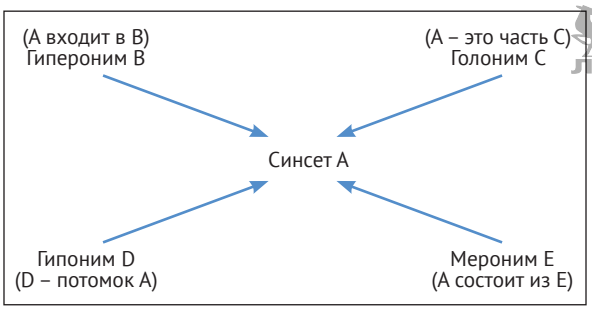


Рис. 3.2 ❖ Различные отношения, вытекающие из синсета

Для изучения метода WordNet и его механизмов мы воспользуемся библиотекой NLTK. Это библиотека Python для обработки текстов на естественном языке. Полный пример доступен в качестве упражнения в файле `ch3_wordnet.ipynb`, расположенном в папке `ch3`.

Установка библиотеки NLTK

Чтобы установить библиотеку NLTK в Python[1], введите следующую команду Python `pip`:

```
pip install nltk
```

[1] Библиотека NLTK входит в состав пакета последних версий дистрибутива Anaconda (см. главу 1) и устанавливается автоматически. Инструкция в этом примечании актуальна для раздельной ручной установки библиотек. – *Прим. перев.*

Кроме того, вы можете использовать IDE (например, PyCharm) для установки библиотеки через графический интерфейс пользователя (GUI). Вы можете найти более подробные инструкции на сайте **http://www.nltk.org/install.html**.
Чтобы импортировать NLTK в Python и загрузить корпус WordNet, сначала импортируйте библиотеку `nltk`:

```
impord nltk
```

Затем можете скачать корпус WordNet, выполнив следующую команду:

```
nltk.download('wordnet')
```

После того как библиотека `nltk` установлена и импортирована, нам нужно импортировать корпус (т. е. базу данных слов) WordNet с помощью команды:

```
from nltk.corpus import wordnet as wn
```

Затем мы можем сделать запрос к корпусу WordNet так:

```
# Запрашиваем все доступные синсеты.
word = 'car'
car_syns = wn.synsets(word)

# Определения каждого синсета в синсете car (автомобиль).
syns_defs = [car_syns[i].definition() for i in range(len(car_syns))]

# Получаем леммы для первого синсета
car_lemmas = car_syns[0].lemmas()[:3]

# Получаем гиперонимы синсета (общий суперкласс).
syn = car_syns[0]
print('\t',syn.hypernyms()[0].name(),'\n')

# Получаем гипонимы синсета (заданный подкласс).
syn = car_syns[0]
print('\t',[hypo.name() for hypo in syn.hyponyms()[:3]],'\n')

# Получаем голонимы для третьего синсета "car".
syn = car_syns[2]
print('\t',[holo.name() for holo in syn.part_holonyms()],'\n')

# Получаем меронимы синсета (заданный подкласс).
syn = car_syns[0]
print('\t',[mero.name() for mero in syn.part_meronyms()[:3]],'\n')
```

После запуска примера мы получим приблизительно следующий вывод:

```
Все доступные синсеты для слова "car"
[Synset('car.n.01'), Synset('car.n.02'), Synset('car.n.03'),
Synset('car.n.04'), Synset('cable_car.n.01')]

Примеры определений доступных синсетов:
car.n.01 : a motor vehicle with four wheels; usually propelled by an
internal combustion engine
car.n.02 : a wheeled vehicle adapted to the rails of railroad
car.n.03 : the compartment that is suspended from an airship and that
carries personnel and the cargo and the power plant

Пример лемм для синсета car.n.03
['car', 'auto', 'automobile']
```

```
Гиперонимы для синсета car.n.01
motor_vehicle.n.01

Гипонимы для синсета car.n.01
['ambulance.n.01', 'beach_wagon.n.01', 'bus.n.04']

Голонимы для синсета car.n.03
['airship.n.01']

Меронимы для синсета car.n.01
['accelerator.n.01', 'air_bag.n.01', 'auto_accessory.n.01']
```

Библиотека позволяет определить сходство между двумя синсетами. В NLTK реализовано несколько различных метрик сходства, и вы можете увидеть их в действии на официальном сайте **www.nltk.org/howto/wordnet.html**. В качестве примера воспользуемся *сходством Ву–Палмера* (Wu–Palmer similarity), которое измеряет сходство между двумя синсетами на основе их глубины в иерархической организации наборов:

```
sim = wn.wup_simility(w1_syns[0], w2_syns[0])
```

Проблемы с WordNet

WordNet – это замечательный общедоступный инструмент для изучения значений слов в задачах NLP. Однако ему присущ ряд недостатков:

- ключевой проблемой в WordNet является утрата *смысловых нюансов*. Существуют как теоретические, так и практические причины, почему это не работает в WordNet. С теоретической точки зрения, тонкая смысловая разница между двумя объектами не поддается прямому определению. На практике определение нюансов субъективно. Например, слова «*просит*» и «*требует*» сходны по значению, но одно из них – «*требует*» – является более напористым. Это и есть нюанс;
- WordNet субъективен по своей природе, поскольку разработан относительно небольшим сообществом. Таким образом, в зависимости от вашей задачи вместо WordNet вам может лучше подойти собственная база определений слов;
- существует также проблема поддержки и развития WordNet. Это весьма трудоемкий процесс. Добавление новых синсетов, определений, лемм и т. д. может стоить очень дорого. Это отрицательно влияет на масштабируемость WordNet, поскольку для поддержания WordNet в актуальном состоянии необходим квалифицированный человеческий труд;
- разработка WordNet для других языков может быть очень затратной. Известны некоторые попытки создать WordNet для других языков и связать его с английской версией WordNet, например MultiWordNet (MWN), но они еще не завершены.

Далее мы обсудим несколько методов представления слов, которые не зависят от внешних ресурсов.

Прямое унитарное кодирование

Более простой способ представления слов – использовать представление слова в виде *унитарного кода* (one-hot encoding). Это означает, что если у нас есть сло-

варь размера V, то каждое i-е слово w_i мы представим в виде вектора длины V и вида [0, 0, 0, ..., 0, 1, 0, ..., 0, 0 , 0], где i-й элемент равен 1, а остальные элементы равны нулю. В качестве примера рассмотрим такое предложение:

Боб и Мэри хорошие друзья.

Унитарное представление каждого слова может выглядеть следующим образом:

Боб: `[1,0,0,0,0]`
и: `[0,1,0,0,0]`
Мэри: `[0,0,1,0,0]`
хорошие: `[0,0,0,1,0]`
друзья: `[0,0,0,0,1]`

Однако вы, возможно, уже поняли, что у этого представления есть много недостатков.

Унитарное представление никоим образом не кодирует сходство между словами и полностью игнорирует контекст, в котором используются слова. Давайте рассмотрим скалярное произведение между векторами слов как меру подобия. Чем больше сходство двух векторов, тем выше скалярное произведение этих векторов. К сожалению, при использовании прямого унитарного кода может случиться так, что слова «*автомобиль*» и «*машина*» будут иметь нулевое сходство, а слова «*автомобиль*» и «*карандаш*» окажутся сходными.

Унитарное представление является крайне неэффективным для больших словарей. Для типичной задачи NLP словарный запас может легко превышать 50 000 слов. Следовательно, попытка построить матрицу представлений для 50 000 слов приведет к очень разреженной матрице с размерностью 50 000×50 000.

Тем не менее унитарное кодирование играет важную роль даже в современных алгоритмах изучения представлений слов. Унитарное кодирование применяют для локального числового представления слов, чтобы улучшить обучение нейросетей и уменьшить числовые представления признаков слов.

Унитарное кодирование также известно как *локализованное представление* (противоположное *распределенному представлению*), поскольку представление признака определяется активацией *одного* элемента в векторе.

Метод TF-IDF

TF-IDF – это частотный метод, учитывающий частоту, с которой слово появляется в корпусе. Он представляет *важность* конкретного слова в данном документе. Интуитивно понятно, что чем выше частота слова, тем важнее это слово в документе. Например, в документе о кошках слово «*кошка*» будет встречаться довольно часто. Однако простое вычисление частоты не сработает, потому что такие слова, как «*этот*», «*который*», встречаются очень часто, но не содержат столько информации. TF-IDF учитывает это и присваивает нулевой вес подобным распространенным словам.

TF означает *частоту термина* (term frequency), а IDF – *обратную частоту документа* (inverse document frequency):

$TF(w_i)$ = количество появлений слова w_i / общее количество слов;
$IDF(w_i)$ = log(общее количество документов / количество документов с w_i);
$TF\text{-}IDF(w_i) = TF(w_i) \times IDF(w_i)$.

Давайте выполним простое упражнение. Рассмотрим два документа:

- doc1: *Это документ про кошек. Кошки отличные домашние компаньоны.*
- doc2: *Это документ про собак. Собаки очень преданные друзья.*

Теперь сделаем вычисления:

*TF-IDF(кошки, doc1) = (2/8) * log(2/1) = 0,075*
*TF-IDF(это, doc2) = (1/8) * log(2/2) = 0,0*

Таким образом, слово «*кошки*» является информативным, а «*это*» – нет. Такой результат применения метода нам вполне подходит с точки зрения измерения важности слов.

Матрица совместной встречаемости

Матрицы совместной встречаемости, или *совместного вхождения* (co-occurrence matrix), в отличие от представления с унитарным кодированием, отражают контекстную информацию слов, но требуют наличия матрицы V×V. В качестве примера возьмем два предложения:

- Джерри и Мэри близкие друзья.
- Джерри покупает цветы для Мэри.

Матрица совместной встречаемости для этих предложений приведена в табл. 3.1. Показан только один треугольник матрицы, так как матрица симметрична.

Таблица 3.1. Матрица совместной встречаемости

	Джерри	**и**	**Мэри**	**близкие**	**друзья**	**покупает**	**цветы**	**для**
Джерри	0	1	0	0	0	1	0	0
и		0	1	0	0	0	0	0
Мэри			0	1	0	0	0	1
близкие				0	1	0	0	0
друзья					0	0	0	0
покупает						0	1	0
цветы							0	1
для								0

Очевидно, что использование такой матрицы сопряжено с большими затратами, поскольку размер матрицы возрастает полиномиально с увеличением размера словаря. Кроме того, сложно использовать значение контекста, превышающее 1. Один из вариантов заключается в том, чтобы иметь взвешенный показатель, где вес слова в контексте уменьшается с расстоянием от интересующего слова.

Все эти недостатки побуждают нас исследовать более строгие, надежные и масштабируемые способы изучения и выведения значений (т. е. представлений) слов.

Word2vec (word to vector, слово в вектор) – это относительно недавно разработанный метод изучения распределенного представления слов, т. е. современный способ автоматического конструирования признаков для многих задач NLP (например, машинного перевода, чат-ботов и генераторов подписей изображений). По сути, Word2vec учится представлять слово в виде вектора, просматривая окружающие слова (т. е. контекст), среди которых используется это слово. Точнее, мы пытаемся предсказать контекст, зная некоторые слова (или наоборот), при помощи нейронной сети, следовательно, нейронная сеть учится находить правильное расположение векторов слов в пространстве смыслов. Мы обсудим этот метод подробно в следующем разделе. Подход Word2vec имеет много преимуществ по сравнению с ранее упомянутыми методами. Они заключаются в следующем:

- подход Word2vec не субъективен и не зависит от конкретных людей, как WordNet;
- размер вектора представления Word2vec не зависит от размера словаря, в отличие от унитарного кодирования или матрицы совместной встречаемости слова;
- Word2vec – это распределенное представление. В отличие от локализованного представления, зависящего от активации одного элемента вектора (например, при унитарном кодировании), распределенное представление зависит от паттерна активации всех элементов в векторе. Это делает Word2vec мощнее и гибче, чем унитарное кодирование.

В следующем разделе вы на примере познакомитесь с изучением представлений слов. Затем научитесь определять функцию потерь, чтобы использовать машинное обучение для изучения представлений слов. Также мы обсудим два алгоритма Word2vec, а именно: *словосочетания с пропуском* (skip-gram) и *непрерывное мультимножество слов* (continuous bag-of-words, CBOW).

Word2vec – нейросетевой подход к изучению представления слова

«Смысл слова раскрывается в контексте».

Дж. Р. Фёрс

Это известное утверждение Джона Руперта Фёрса, высказанное им в 1957 году, лежит в основе Word2vec, поскольку данный метод опирается на контекст слова для изучения его семантики. Word2vec – это новаторский подход, который позволяет изучать значение слов без какого-либо вмешательства человека. Кроме того, Word2vec изучает числовые представления слов, рассматривая слова, окружающие данное слово.

Мы можем удостовериться в обоснованности такого подхода на простом примере. Представьте, что вы сидите на экзамене по лингвистике и в первом вопросе видите такой текст: «*Мария – очень упрямый ребенок. Ее скверный характер всегда доставляет ей неприятности*». Если вы не очень эрудированны, то можете и не знать, что значит слово «*скверный*». В подобной ситуации вы непроизвольно смотрите на фразы, окружающие интересующее вас слово. В нашем примере слово «*скверный*» окружено словами «*упрямый*», «*характер*» и «*неприятности*». Доста-

точно взглянуть на эти три слова, чтобы понять, что в данном контексте «*скверный*» означает плохое свойство характера, связанное с упрямством. Я думаю, что этот пример убедительно демонстрирует важность контекста для понимания значения слова.

Теперь займемся основами Word2vec. Как уже упоминалось, Word2vec изучает значение данного слова, просматривая его контекст и представляя его численно. В контексте мы ссылаемся на фиксированное количество слов перед интересующим словом и за ним. Давайте возьмем гипотетический корпус, состоящий из N слов. Математически это можно представить в виде последовательности $w_0, w_1, \ldots, w_i$ и w_N, где w_i – это i-е слово в корпусе.

Далее, если мы хотим найти хороший алгоритм, который способен изучить значение заданного слова, наш алгоритм должен уметь правильно предсказывать контекстные слова. Это означает, что для любого данного слова w_i должна быть высокой следующая вероятность:

$$P(w_{i-m}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+m} | w_i) = \prod_{j \neq i \wedge j = i-m}^{i+m} P(w_j | w_i).$$

Чтобы перейти к правой части уравнения, нам нужно предположить, что для заданного слова w_i слова контекста не зависят друг от друга (например, w_{i-2} и w_{i-1} независимы). Хотя это и не совсем верно, такое допущение делает проблему обучения решаемой и хорошо работает на практике.

Упражнение: королева = король – он + она?

Прежде чем продолжить, давайте выполним небольшое упражнение, чтобы понять, как максимизация вышеупомянутой вероятности приводит к нахождению хорошего значения (или представлений) слов. Рассмотрим следующий маленький корпус:

Генрих был очень богатым королем. Он женился на прекрасной королеве.
Она была очень добрая.

Теперь давайте выполним ручную предварительную обработку и удалим знаки препинания и неинформативные слова[1]:

быть богатый король он женился прекрасная королева она быть добрая

Затем сформируем набор кортежей для каждого слова с их контекстными словами в формате (*целевое слово → контекстное слово 1*, *контекстное слово 2*). Размер окна контекста примем равным 1 с каждой стороны:

быть → богатый
богатый → быть, король
король → богатый, он
он → король, женился

[1] А также запишем глаголы *был/была* в начальной форме *быть*, чтобы нас не отвлекал признак рода, присущий русскому языку. – *Прим. перев.*

женился → он, прекрасная
прекрасная → женился, королева
королева → прекрасная, она
она → королева, быть
быть → она, добрая
добрая → быть

Помните, что наша цель – правильно предсказывать слова справа, если дано слово слева. Для этого слова в контексте правой части должны иметь высокое числовое или геометрическое сходство со словами в контексте левой части. Другими словами, интересующее слово должно быть передано окружающими словами. Чтобы понять, как это работает, рассмотрим реальные числовые векторы. Для простоты рассмотрим только кортежи, выделенные жирным шрифтом. Давайте начнем с предположения о представлении слова «*богатый*»:

богатый → [*0,0*]

Чтобы из слова «*богатый*» правильно предсказать слова «*быть*» и «*король*», эти слова должны иметь высокое сходство со словом «*богатый*». Давайте примем за меру сходства евклидово расстояние между векторами.

Сначала попробуем следующие значения слов «*король*» и «*быть*»:

король → [*0,1*]
быть → [*-1,0*]

Это правильно, поскольку:

Dist (*богатый*, *король*) = *1,0*
Dist (*богатый*, *быть*) = *1,0*

Здесь *Dist* – евклидово расстояние между двумя словами, как показано на рис. 3.3.

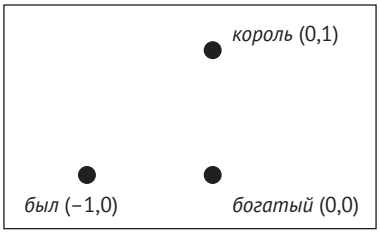


Рис. 3.3 ❖ Расположение векторов слов для слов «*богатый*», «*быть*» и «*король*»

Теперь давайте рассмотрим следующий кортеж:

король → богатый, он

Мы уже установили отношение между словами «*король*» и «*богатый*». Однако мы еще не закончили; чем сильнее отношение, тем ближе должны быть эти два

слова. Итак, сначала настроим вектор слова «*король*» так, чтобы он был немного ближе к вектору слова «*богатый*»:

король → [*0,0.8*]

Далее нам нужно добавить в пространство векторов слово «*он*». Это слово должно быть ближе к слову «*король*». Сейчас мы имеем такую информацию о слове «*он*»:

он → [*0.5,0.8*]

На данный момент расположение векторов слов выглядит как рис. 3.4.

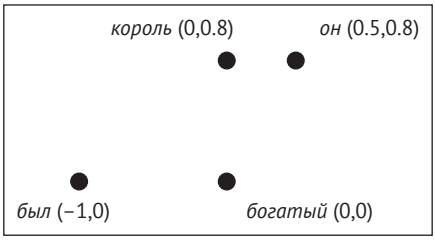


Рис. 3.4 ❖ Расположение векторов слов «*богатый*», «*быть*», «*король*» и «*он*»

Теперь давайте перейдем к следующим двум кортежам: *королева* → *красивая*, *она* и *она* → *королева*, *быть*. Обратите внимание, что я поменял порядок кортежей, так как это облегчает нам понимание примера:

она → *королева*, *быть*

Теперь мы должны использовать логику своего родного языка.

Разумно предположить, что слово «*она*» расположено на таком же расстоянии от слова «*быть*», что и слово «*он*», потому что они одинаково используются в контексте. Поэтому давайте запишем:

она → [*0.5,0.6*]

Далее мы представим слово «*королева*», близкое к слову «*она*»:

королева → [*0.0,0.6*]

Это показано на рис. 3.5.

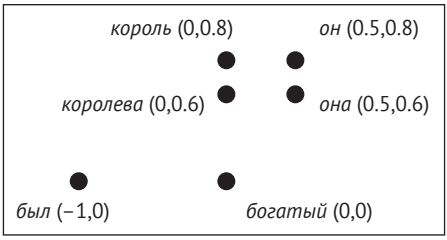


Рис. 3.5 ❖ Координаты векторов слов «*богатый*», «*быть*», «*король*», «*он*», «*она*» и «*королева*»

У нас остался последний кортеж:

королева → *красивая*, *она*

Необходимо найти определение слова «*красивая*». Оно должно располагаться на примерно одинаковом расстоянии от слов «*королева*» и «*она*». Давайте использовать следующий вектор:

красивая → [*0.25,0*]

Теперь мы можем построить новую схему, изображающую отношения между словами. Когда мы смотрим на рис. 3.6, он выглядит интуитивно понятным представлением значений слов.

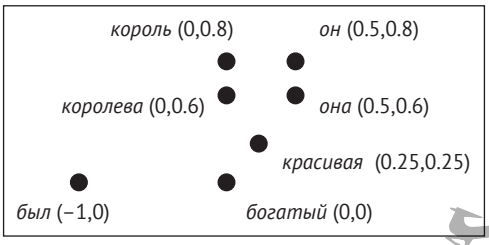


Рис. 3.6 ❖ Координаты векторов слов «*богатый*», «*быть*», «*король*», «*он*», «*она*», «*королева*» и «*красивая*»

Итак, вернемся к вопросу, который скрывался в наших умах с самого начала этого упражнения. Является ли справедливым следующее забавное равенство: *королева* = *король* – *он* + *она*? Теперь мы можем ответить на этот вопрос. Давайте поработаем над правой стороной выражения:

= *король* – *он* + *она*
= [*0.0,0.8*] – [*0.5,0.8*] + [*0.5,0.6*]
= [*0.0,0.6*]

Равенство действительно выполняется. Если вы посмотрите на вектор слова, который мы ранее задали для слова «*королева*», то увидите, что он совпадает с результатом вычислений.

Это был лишь грубый пример, демонстрирующий представление слов, и предложенные векторы могут отличаться от точных представлений слов, изученных с использованием нейросети.

Однако имейте в виду, что это чрезвычайно сокращенный и потому нереальный пример по отношению к тому, как может выглядеть корпус текста в реальном мире. Вы не сможете вручную построить отношения между десятками тысяч слов. Это именно тот случай, когда сложные аппроксиматоры функций, такие как нейронные сети, делают работу за нас. Но чтобы использовать нейронные сети, нам следует математически однозначно сформулировать нашу проблему. Тем не менее это хорошее упражнение, которое на практике показывает мощь и удобство векторного представления слов.

Разработка функции потери для изучения представлений слов

Словарный запас даже для простой задачи в реальном мире может легко превысить 10 000 слов. Поэтому мы не можем подбирать векторы слов вручную для больших текстовых корпусов, и нам необходимо разработать способ автоматического поиска подходящих представлений слов с использованием алгоритмов машинного обучения (например, нейронных сетей) для эффективного выполнения этой трудоемкой задачи. Кроме того, в любом случае – независимо от алгоритма обучения и задачи – необходимо определить функцию потери. Как вы помните, успешное решение задачи означает сведение потери к минимуму. Давайте найдем функцию потери, которая будет служить критерием нахождения хороших векторов представлений слов.

Вспомним уравнение, которое мы обсуждали в начале этого раздела:

$$P(w_{i-m},\ldots,w_{i-1},w_{i+1},\ldots,w_{i+m}|w_i)=\prod_{j\neq i\,\Lambda\,j=i-m}^{i+m}P(w_j|w_i).$$

Исходя из этого уравнения, определим функцию стоимости для нейронной сети:

$$J(\theta)=-(1/N-2m)\sum_{i=m+1}^{N-m}\prod_{j\neq i\,\Lambda\,j=i-m}^{i+m}P(w_j|w_i).$$

Помните, что $J(\theta)$ – это потеря (т. е. стоимость), а не награда. Также мы хотим максимизировать $P(w_j|w_i)$. Поэтому в правой части выражения стоит знак «минус», чтобы преобразовать его в функцию стоимости.

Теперь, вместо того чтобы работать с оператором умножения, давайте перенесем выражение в логарифмическое пространство. Данное преобразование обеспечит неразрывность и численную устойчивость и даст нам следующее уравнение:

$$J(\theta)=-(1/N-2m)\sum_{i=m+1}^{N-m}\prod_{j\neq i\,\Lambda\,j=i-m}^{i+m}\log P(w_j|w_i).$$

Эта формулировка функции стоимости известна как *отрицательное логарифмическое правдоподобие* (negative log-likelihood).

Теперь, когда у нас есть четко сформулированная функция потери, нейронная сеть сможет найти оптимальное значение этой функции, т. е. изучить наилучшие представления слов. А сейчас пришло время познакомиться с прикладными алгоритмами, которые используют функцию потери.

Алгоритм skip-gram

Первый алгоритм, о котором мы поговорим, – skip-gram, или алгоритм *словосочетаний с пропуском*. Данный алгоритм, предложенный Миколовым и др. в 2013 году, использует контекст слов письменного текста для изучения правильных представлений слов. Давайте пошагово изучим этот алгоритм.

Сначала мы разберем процесс подготовки данных, а затем введем обозначения, необходимые для понимания алгоритма. Наконец, обсудим сам алгоритм.

Как вы уже знаете, значение слова может быть извлечено из контекстных слов, окружающих заданное слово. Однако это просто лишь на словах. На самом деле нелегко разработать модель, которая использует этот подход для изучения представлений слов.

От необработанного текста до структурированных данных

Во-первых, нам нужно разработать механизм для извлечения набора данных, которые можно подавать в обучаемую модель. Такие данные должны представлять собой набор кортежей формата (*вход*, *выход*). Более того, процесс подготовки должен выполняться автоматически, т. е. человеку не нужно вручную создавать метки для данных. Таким образом, процесс подготовки данных делает следующее:

- захватывает слова вокруг данного слова;
- работает без присмотра.

Модель skip-gram использует следующий подход для подготовки набора данных.

1. Для данного слова w_i установлен размер m окна контекста. Под размером *окна контекста* понимают количество слов, рассматриваемых как контекст с одной стороны данного слова. Поэтому для w_i контекстное окно (включая целевое слово w_i) будет иметь размер $2m + 1$ и выглядеть так: $[w_{i-m}, \ldots, w_{i-1}, w_i, w_{i+1}, \ldots, w_{i+m}]$.
2. Затем формируются кортежи ввода-вывода $[\ldots, (w_i, w_{i-m}), \ldots, (w_i, w_{i-1}), (w_i, w_{i+1}), \ldots, (w_i, w_{i+m}), \ldots]$; здесь $m + 1 \leqslant i \leqslant N - m$ и N – количество слов в тексте, чтобы получить практическое понимание.

Для примера возьмем следующее предложение и размер окна контекста $m = 1$:

Собака лаяла на почтальона.

Для этого примера набор данных будет следующим:

[(*лаяла*, *собака*), (*лаяла*, *на*), (*на*, *лаяла*), (*на*, *почтальона*)]

Изучение представлений слов с помощью нейронной сети

Представив данные в формате (*вход*, *выход*), можно запускать нейронную сеть для изучения представлений слов. Но сначала определим переменные, которые нам понадобятся в этом процессе. Чтобы хранить представления слов, нам нужна матрица $V \times D$, где V – размер словаря, а D – размерность представления слова (т. е. число элементов в векторе, представляющем одно слово). D определяется пользователем как гиперпараметр нейросети. Чем выше D, тем большую смысловую нагрузку будут нести изученные представления. Такую матрицу называют *пространством представления* (embedding space), или *слоем представления* (embedding layer). Далее следует слой softmax с весами размера $D \times V$ и смещением размера V.

Каждое слово представлено в виде вектора с унитарным кодированием размера V, в котором только один элемент равен 1, а все остальные равны 0. Поэтому

каждое входное слово и соответствующие выходные слова будут иметь размер V. Обозначим i-й вход как x_i, соответствующее представление как z_i и соответствующий выход как y_i.

На данный момент определены все необходимые переменные. Далее для каждого входа x_i ищем векторы представления, соответствующие входу. Эта операция дает нам z_i – вектор представления длины D. После этого вычисляем предсказание для x_i, используя следующее преобразование:

$$\text{logit}(x_i) = z_i W + b$$
$$\hat{y}_i = softmax(\text{logit}(x_i))$$

Здесь $\text{logit}(x_i)$ представляет ненормализованные оценки (то есть логиты), $\hat{y}_i$ – это прогнозируемый результат размера V (вероятность вывода, являющегося словом из словаря размера V), W – матрица весов D×V , b – вектор смещения V×1 и *softmax* – это функция активации softmax. Рассмотрим концептуальную схему (рис. 3.7) и реализацию (рис. 3.8) модели skip-gram. Краткий перечень обозначений:

- V – размер словаря;
- D – размерность слоя представлений;
- x_i – i-е входное слово, представленное в виде вектора с унитарным кодированием;
- z_i – вектор представления (т. е. представление), соответствующий i-му входному слову;
- y_i – выходное слово с унитарным кодированием, соответствующее x_i;
- $\hat{y}_i$ – прогнозируемый результат (вероятность) для x_i;
- $\text{logit}(x_i)$ – ненормализованная оценка входа x_i;
- I_{w_j} – представление слова w_j в унитарном коде;
- W – матрица весов *softmax*;
- b – смещение *softmax*.

Теперь вы можете использовать функцию потери с отрицательным логарифмическим правдоподобием для расчета потери в данной точке данных (x_i, y_i). Далее мы обсудим, как вычислить $P(w_j|w_i)$ из $\hat{y}_i$, а также дадим формальное определение этого объекта.

Почему в оригинальной статье используют два слоя представлений?

В оригинальной статье (Mikolov и др., 2013) применяются два разных пространства представления V×D для обозначения слов в целевом пространстве (слова, используемые в качестве цели) и слов в контекстном пространстве (слова, используемые в качестве контекста). Одна из причин этого состоит в том, что одно и то же слово нечасто встречается в контексте самого себя. Следовательно, мы хотим минимизировать вероятность таких событий. Например, для целевого слова «*собака*» весьма маловероятно, что слово «*собака*» встречается также и в его контексте, т. е. $P(собака \mid собака) \sim 0$. Интуитивно понятно, что если мы вводим точку данных (x_i = *собака* и y_i = *собака*) в нейронную сеть, то ожидаем, что нейронная сеть даст высокую потерю, если предскажет слово «*собака*» как контекст для слова «*собака*». Другими словами, мы хотим, чтобы представление слова «*собака*» находилось на очень большом расстоянии от представления слова «*собака*». Это создает сильное противоречие, так как расстояние между представлениями одного и того же слова должно быть равно 0. Поэтому мы не сможем избежать противоречия, если у нас будет только одно пространство для

представления. Однако наличие двух отдельных пространств для целевых слов и контекстных слов позволяет нам устранить противоречие, потому что у нас есть два отдельных вектора представления для одного и того же слова. Однако на практике, поскольку вы избегаете использования кортежей ввода-вывода, наличие одного и того же слова в качестве ввода и вывода позволяет вам работать с одним пространством представлений и устраняет необходимость в двух различных слоях.

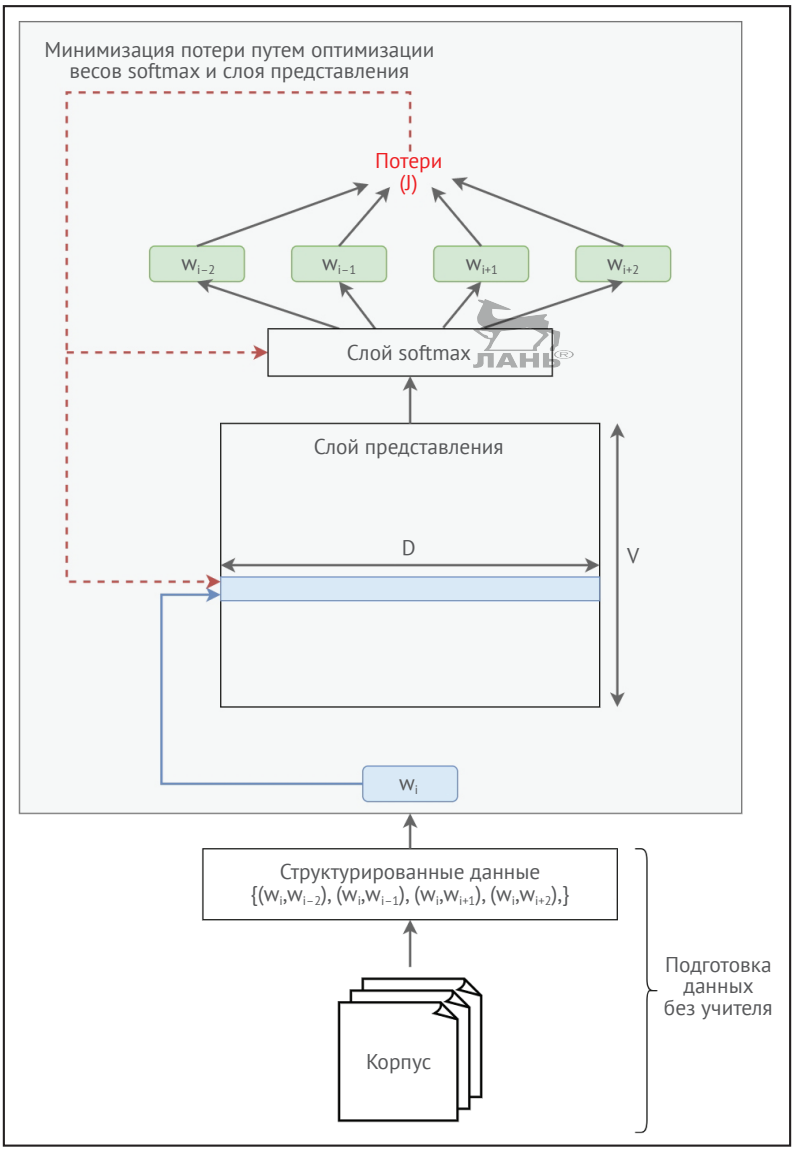


Рис. 3.7 ❖ Концепция модели Skip-Gram

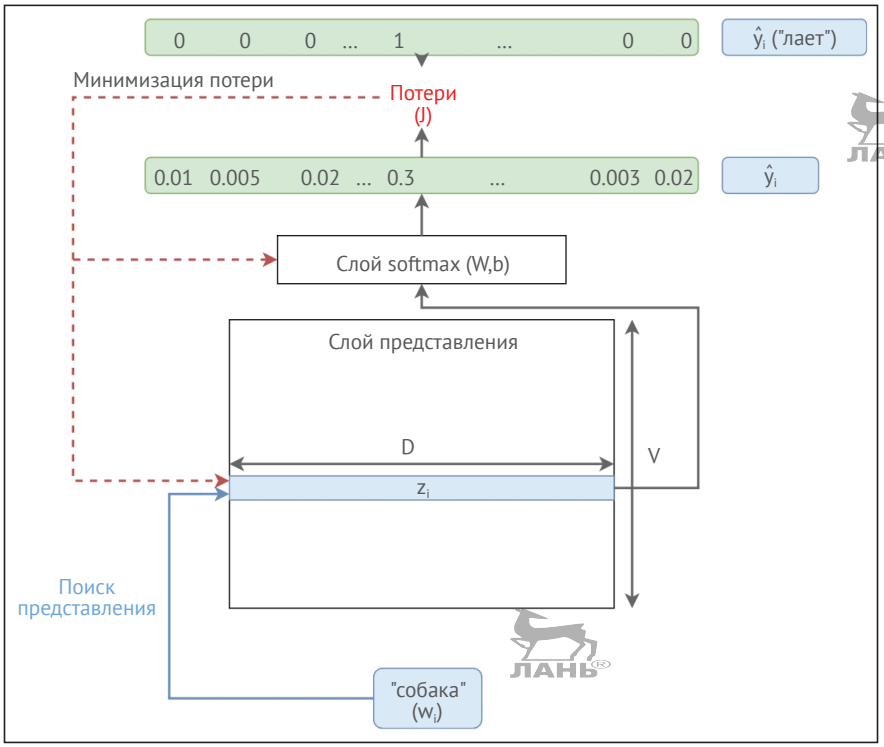


Рис. 3.8 ❖ Реализация модели Skip-Gram

Формулировка прикладной функции потерь

Давайте рассмотрим нашу функцию потерь более внимательно. Мы выяснили, что потеря должна вычисляться следующим образом:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \prod_{j \neq i \wedge j = i-m}^{i+m} \log P(w_j | w_i).$$

Тем не менее рассчитать конкретную потерю от элементов, которые у нас есть на данный момент, не совсем просто.

Сначала разберемся, что представляет собой элемент $P(w_j|w_i)$. Для этого мы перейдем от обозначения отдельных слов к обозначению отдельных точек данных. То есть мы будем говорить, что $P(w_j|w_i)$ задается n-й точкой данных, которая имеет унитарно закодированный вектор w_i в качестве входа (x_n) и унитарно закодированное представление w_j в качестве истинного выхода (y_n). Эту нотацию можно выразить следующим уравнением:

$$P(w_j | w_i) = \frac{\exp(\text{logit}(x_n)_{w_j})}{\sum_{w_k \in vocabulary} \exp(\text{logit}(x_n)_{w_k})}.$$

Член $\text{logit}(x_n)$ обозначает ненормализованный вектор оценки прогноза (т. е. логит) размера V, полученный для данного входного значения x_n, а $\text{logit}(x_n)_{w_j}$ – это

значение оценки, соответствующее ненулевому индексу унитарного представления w_j (мы будем называть это индексом w_j). Затем мы нормализуем значение логита по индексу w_j относительно всех значений логитов, соответствующих всем словам во всем словаре. Этот конкретный тип нормализации известен как softmax-активация, или просто нормализация. Теперь, перейдя к логарифмическому представлению, мы получим следующее уравнение:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i \; j=i-m}^{i+m} \text{logit}(x_n)_{w_j} - \log\left(\sum_{w_k \in vocabulary} \exp(\text{logit}(x_n)_{w_k}) \right).$$

Чтобы эффективно рассчитать функцию logit(...), мы можем развернуть переменные и получить следующую запись:

$$\text{logit}(x_n)_{w_j} = \sum_{l=1}^{V} \mathbb{I}_{w_j} \text{logit}(x_n).$$

Здесь $\mathbb{I}_{w_j}$ – унитарно закодированный вектор w_j. Теперь операция logit сводится к операции сложения и умножения. Поскольку $\mathbb{I}_{w_j}$ имеет только один ненулевой элемент, соответствующий слову w_j, в вычислениях будет использоваться лишь этот индекс вектора. Это более эффективно с вычислительной точки зрения, чем поиск значения в векторе логита, соответствующего индексу ненулевого элемента, путем пролистывания вектора размера словаря.

Теперь, подставляя значение логита в выражение для вычисления потери, мы получаем следующее выражение:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i \; j=i-m}^{i+m} \sum_{l=1}^{V} \mathbb{I}_{w_j} \text{logit}(x_n) - \log\left(\sum_{w_k \in vocabulary} \exp\left(\sum_{l=1}^{V} \mathbb{I}_{w_j} \text{logit}(x_n) \right) \right).$$

Давайте рассмотрим пример, чтобы лучше понять суть вычислений:

Мне нравится NLP

Мы можем создать кортежи ввода-вывода следующим образом:

(*нравится*, *мне*) (*нравится*, *NLP*)

Примем следующие унитарные представления слов:

нравится – 1,0,0
Мне – 0,1,0
NLP – 0,0,1

Далее, давайте рассмотрим кортеж ввода-вывода (*нравится*, *Мне*). Давайте предположим, что когда мы распространили вход «*нравится*» через обучаемую модель skip-gram, то получили такие логиты для слов «*нравится*», «*Мне*» и «*NLP*» соответственно:

2,10,5

Выходные значения softmax для каждого слова в словаре будут следующими:

$P(нравится|нравится) = \exp(2)/(\exp(2)+\exp(10)+\exp(5)) = 0{,}118$
$P(Мне|нравится) = \exp(10)/(\exp(2)+\exp(10)+\exp(5)) = 0{,}588$
$P(NLP|нравится) = \exp(5)/(\exp(2)+\exp(10)+\exp(5)) = 0{,}294$

Предыдущая функция потерь говорит о том, что нам нужно максимизировать $P(Мне|нравится)$, чтобы минимизировать потери. Теперь давайте применим наш пример к этой функции потерь:

$$= -([0,1,0]*([2,10,5]) - log(\exp([1,0,0]*[2,10,5]) + \exp([0,1,0]*[2,10,5]) + \exp([0,0,1]*[2,10,5])))$$
$$= -(10 - log(\exp(2) + \exp(10) + \exp(5))) = 0{,}007$$

При данной функции потерь для члена перед знаком минус в векторе *y* есть только один ненулевой элемент, соответствующий слову «*Мне*». Поэтому мы будем рассматривать лишь вероятность $P(Мне|нравится)$.

Однако это не идеальное решение, которое мы искали. Цель функции потерь с практической точки зрения заключается в том, чтобы максимально увеличить вероятность предсказания контекстного слова по данному слову, при этом сводя к минимуму вероятность «всех» неконтекстуальных слов по отношению к данному слову. Вскоре мы увидим, что четко определенная функция потерь на практике работает недостаточно эффективно. Нам придется разработать более умную функцию приблизительных потерь, чтобы выучить правильные представления слов за приемлемый промежуток времени.

Эффективная аппроксимация функции потерь

Нам повезло иметь функцию потерь, которая является математически и интуитивно понятной. Однако трудная работа на этом только начинается. Если мы попытаемся вычислить функцию потерь в замкнутой форме, которую обсуждали ранее, то неизбежно столкнемся с огромной медлительностью нашего алгоритма.

Причиной тому служит большой словарный запас, ограничивающий производительность. Давайте вспомним полученную ранее функцию стоимости:

$$J(\theta) = -(1/N - 2m)\sum_{i=m+1}^{N-m}\sum_{j\neq i}^{i+m}\sum_{j=i-m} \text{logit}(x_n)_{w_j} - \log\left(\sum_{w_k \in vocabulary} \exp(\text{logit}(x_n)_{w_k})\right).$$

Очевидно, что вычисление потерь для одного примера требует вычисления логитов для всех слов в словаре. В отличие от проблем компьютерного зрения, где нескольких сотен выходных классов более чем достаточно для решения большинства существующих реальных проблем, метод skip-gram не обладает такими свойствами. Поэтому приходится использовать эффективную аппроксимацию функции потерь, не снижающую качество модели.

Мы обсудим два популярных варианта аппроксимации:

- отрицательная выборка;
- иерархический softmax.

Отрицательная выборка слоя softmax

Отрицательная выборка является приближением метода *сопоставительной оценки шума* (noise-contrastive estimation, NCE). При данном подходе хорошая модель отличает данные от шума с помощью логистической регрессии.

Исходя из этого, давайте переформулируем нашу цель изучения представления слов. Нам не требуется *полновероятная* (full-probabilistic) модель, которая имеет точные вероятности всех контекстных слов в словаре по отношению к данному слову. Все, что нам нужно, – это очень точные векторы слов. Следовательно, мы упрощаем задачу, сводя ее к отделению полезных данных (т. е. пары ввода-вывода) от шума (т. е. K-множества воображаемых пар ввода-вывода шума). Под шумом мы подразумеваем ложные пары ввода-вывода, созданные с использованием слов, которые не попадают в контекст данного слова. Мы также избавимся от активации softmax и заменим ее сигмовидной активацией, известной как *логистическая функция*. Она позволяет устранить зависимость функции стоимости от полного словаря, сохраняя вывод в пределах [0, 1]. Процесс отрицательной выборки схематически изображен на рис. 3.9.

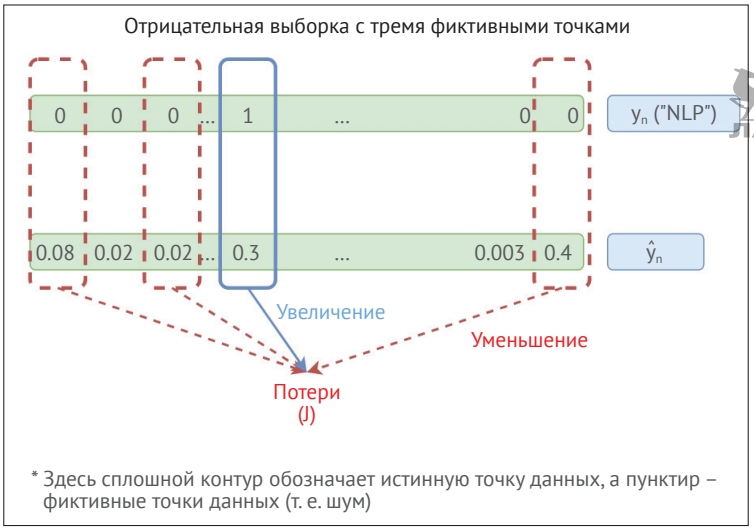


Рис. 3.9 ❖ Процесс отрицательной выборки

Исходное уравнение функции потерь можно записать в следующем виде:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i \; j=i-m}^{i+m} \log(\exp(\mathrm{logit}(x_n)_{w_j}))$$

$$-\log\left(\sum_{w_k \in vocabulary} \exp(\mathrm{logit}(x_n)_{w_k}) \right).$$

В результате аппроксимации уравнение принимает вид:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i}^{i+m} \sum_{j=i-m} \log(\sigma(\text{logit}(x_n)_{w_j}))$$
$$+ \sum_{q=1}^{k} \mathbb{E}_{w_q \sim wocabulary-(w_i, w_j)} \log(1 - \sigma(\text{logit}(x_n)_{w_q})).$$

Здесь σ обозначает сигмовидную активацию, где $\sigma(x) = 1/(1 + \exp(-x))$. Обратите внимание, что для ясности я заменил $\text{logit}(x_n)_{w_j}$ на $\log(\exp(\text{logit}(x_n)_{w_j}))$ в исходном уравнении функции потерь. Вы можете видеть, что новая функция потерь зависит только от расчетов, связанных с k элементами из словаря.

После небольшого упрощения мы приходим к следующему уравнению:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i}^{i+m} \sum_{j=i-m} \log(\sigma(\text{logit}(x_n)_{w_j}))$$
$$+ \sum_{q=1}^{k} \mathbb{E}_{w_q \sim wocabulary-(w_i, w_j)} \log(\sigma(-\text{logit}(x_n)_{w_q})).$$

Давайте на минутку остановимся и подумаем, что означает это уравнение. Для удобства предположим, что $k = 1$. Отсюда следует уравнение:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i}^{i+m} \sum_{j=i-m} \log(\sigma(\text{logit}(x_n)_{w_j})) + \log(\sigma(-\text{logit}(x_n)_{w_q})).$$

Здесь w_j представляет контекстное слово для w_i, а w_q – *не* контекстное слово для w_i. По сути, это уравнение говорит о том, что для минимизации $J(\theta)$ мы должны сделать $\sigma(\text{logit}(x_n)_{w_j}) \approx 1$, т. е. $\text{logit}(x_n)_{w_j}$ должен иметь большое положительное значение. Тогда $\sigma(-\text{logit}(x_n)_{w_q}) \approx 1$ означает, что $\text{logit}(x_n)_{w_q}$ должен иметь большое отрицательное значение. Другими словами, для истинных точек данных, представляющих истинные целевые слова и слова контекста, должны быть получены большие положительные значения, а фиктивные точки данных, представляющие целевые слова, и шум должны получить большие отрицательные значения. Мы получаем то же самое поведение, что при использовании функции softmax, но с большей вычислительной эффективностью.

Здесь σ – сигмовидная активация. По сути, мы выполняем два шага в нашем расчете функции потерь:

- расчет потерь для ненулевого столбца w_j (движение в положительном направлении);
- расчет потерь для K-множества выборок шума (движение в отрицательном направлении).

Иерархический softmax

Иерархический softmax немного сложнее, чем отрицательная выборка, но служит той же цели – аппроксимация, исключающая необходимость вычислять активации для всех слов в словаре для всех обучающих образцов. Однако, в отличие от отрицательной выборки, иерархический softmax использует только истинные данные и не нуждается в выборках шума. Данный подход схематически изображен на рис. 3.10.

Чтобы понять иерархический softmax, давайте рассмотрим пример:

Мне нравится NLP. Глубокое обучение это круто.

Словарь для этого примера выглядит следующим образом:

Мне, *нравится*, *NLP*, *Глубокое*, *обучение*, *это*, *круто*

На основе этого словаря создадим двоичное дерево, в котором все слова в словаре представлены как листья, и добавим специальный токен PAD, чтобы все узлы дерева имели два члена.

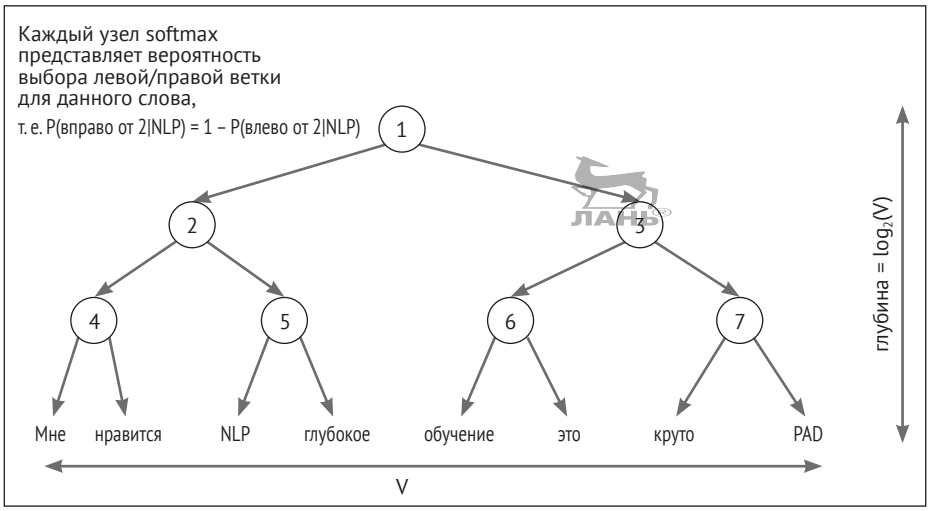


Рис. 3.10 ❖ Многоуровневый softmax

Последний скрытый слой будет полностью связан со всеми узлами в иерархии (рис. 3.11). Обратите внимание, что эта модель имеет одинаковое количество общих весов по сравнению с классическим слоем softmax, но для вычисления используется только их подмножество.

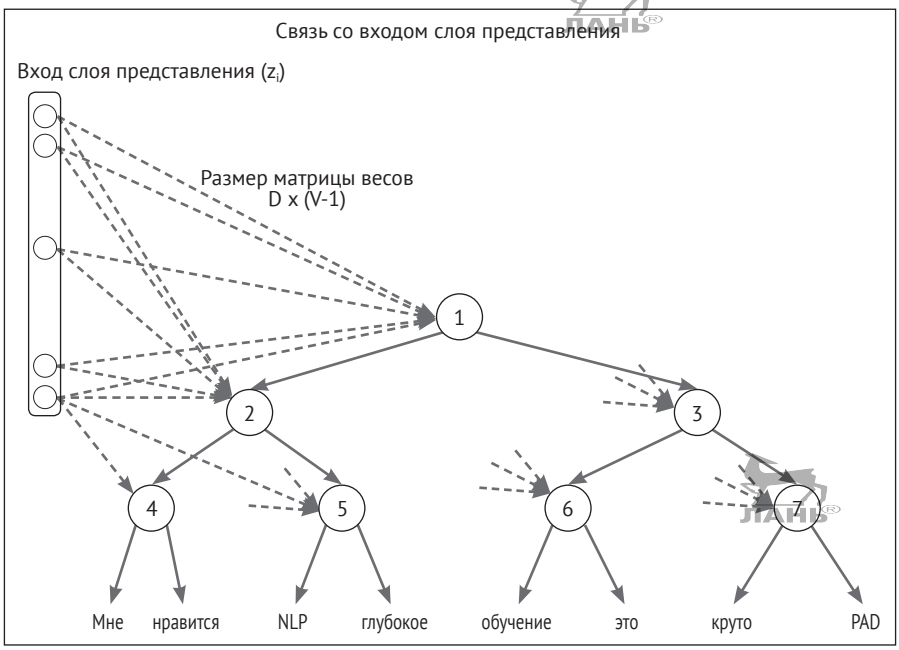


Рис. 3.11 ❖ Как иерархический softmax соединяется со слоем представления

Допустим, нужно вычислить вероятность P(*NLP*|*нравится*), где «*нравится*» – входное слово. Тогда для расчета вероятности нам требуется только подмножество весов, как показано на рис. 3.12.

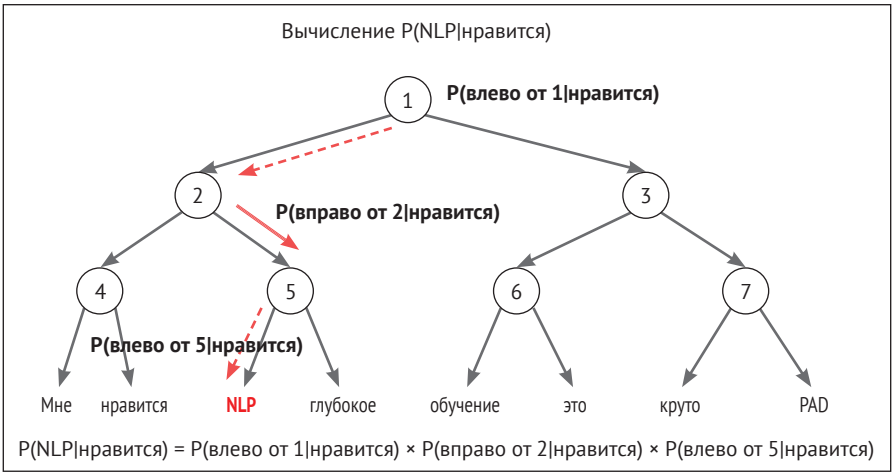


Рис. 3.12 ❖ Вычисление вероятностей с иерархическим softmax

В данном примере вероятность вычисляется следующим образом:

$P(NLP|нравится) = P(влево\ от\ 1|нравится) \times P(вправо\ от\ 2|нравится) \times P(влево\ от\ 5|нравится)$

Поскольку теперь мы знаем, как рассчитать $P(w_j|w_i)$, то можем использовать исходную функцию потерь. Обратите внимание, что этот метод применяет только веса, связанные с узлами в пути для расчета, что обеспечивает высокую вычислительную эффективность.

Изучение иерархии

Хотя иерархический softmax эффективен, один важный вопрос остался без ответа. Как формируется структура дерева? А именно откуда мы знаем, какое слово должно следовать за какой ветвью? Существуют разные варианты построения иерархии:

- **случайная инициализация иерархии**. Этому методу присуще некоторое ухудшение производительности, поскольку при произвольном размещении листьев нельзя гарантировать наилучшее ветвление из возможных вариантов;
- **применение WordNet для построения иерархии**. Для определения подходящего порядка слов в дереве можно использовать WordNet. Этот метод зарекомендовал себя значительно лучше, чем случайная инициализация.

Оптимизация обучаемой модели

Поскольку у нас есть правильно сформулированная функция потерь, оптимизация заключается в вызове правильной функции из библиотеки TensorFlow. Мы будем использовать стохастический процесс оптимизации. Это означает, что мы не передаем полный набор данных сразу, а ограничиваемся случайным пакетом данных на каждом из многочисленных этапов.

Реализация алгоритма skip-gram с TensorFlow

Итак, перейдем к реализации алгоритма skip-gram с использованием библиотеки TensorFlow. В книге мы рассмотрим только определения необходимых операций TensorFlow для изучения представлений, а выполнение операций оставим за кадром. Полное упражнение доступно в файле `ch3_word2vec.ipynb` из каталога `ch3`.

Начнем с определения гиперпараметров модели. Вы можете попробовать менять эти гиперпараметры, чтобы увидеть, как они влияют на конечную производительность, например задать `batch_size = 16` или `batch_size = 256`. Однако можете и не увидеть каких-либо существенных различий на простых учебных задачах, если только не выставите крайние значения, например `batch_size = 1` или `num_sampled = 1`.

```
batch_size = 128
embedding_size = 128 # Размерность вектора представления.
window_size = 4 # Сколько слов рассматривать слева и справа.
valid_size = 16 # Случайный набор слов для оценки схожести.
# Отклонение от вершины распределения.
valid_window = 100
valid_examples = get_common_and_rare_word_ids(valid_size//2,valid_size//2)
num_sampled = 32 # Количество отрицательных точек в выборке.
```

Определяем заполнители TensorFlow для обучающих входных данных, меток и допустимых входных данных:

```
train_dataset = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
```

Затем определяем переменные TensorFlow для слоя внедрения и весов и смещений softmax:

```
embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size],-1.0,1.0))
softmax_weights = tf.Variable(tf.truncated_normal([vocabulary_size, embedding_size],
stddev = 0.5 / math.sqrt(embedding_size)))
softmax_biases = tf.Variable(tf.random_uniform([vocabulary_size],0.0,0.01))
```

Далее определяем операцию просмотра, которая собирает соответствующие представления данного набора обучающих данных:

```
embed = tf.nn.embedding_lookup(embeddings, train_dataset)
```

После этого определим потерю softmax, используя отрицательную выборку:

```
loss = tf.reduce_mean(
  tf.nn.sampled_softmax_loss(weights=softmax_weights,
  biases=softmax_biases, inputs=embed,
  labels=train_labels, num_sampled=num_sampled,
  num_classes=vocabulary_size))
```

Теперь определим оптимизатор для минимизации ранее определенной функции потерь. Не стесняйтесь экспериментировать с другими оптимизаторами, перечисленными по адресу **https://www.tensorflow.org/api_guides/python/train**:

```
optimizer = tf.train.AdagradOptimizer(1.0).minimize(loss)
```

Вычислим сходство между проверочными примерами входных данных и всеми представлениями, используя косинусное расстояние:

```
norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keepdims=True))
normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings, valid_dataset)
similarity = tf.matmul(valid_embeddings, tf.transpose(normalized_embeddings))
```

Определив все переменные и операции TensorFlow, вы можете перейти к выполнению операций, чтобы получить первые результаты. Рассмотрим в общем виде последовательность процедур для выполнения этих операций. Для детального изучения кода обратитесь к полному файлу упражнения.

- Сначала инициализируйте переменные TensorFlow с помощью `tf.global_variables_initializer().run()`.
- Определите общее количество шагов и для каждого шага выполните следующие действия:
  - сгенерируйте пакет данных (`batch_data` – входы, `batch_labels` – выходы) при помощи генератора данных;
  - создайте словарь с именем `feed_dict`, который сопоставляет заполнители обучающего ввода/вывода с данными, сгенерированными генератором данных:

    ```
    feed_dict = {train_dataset : batch_data, train_labels : batch_labels}
    ```

- выполните шаг оптимизации и получите значение потери следующим образом:

```
_, l = session.run([optimizer, loss], feed_dict=feed_dict)
```

Теперь мы обсудим другой популярный алгоритм Word2vec, известный как *непрерывное мультимножество слов*, или *«мешок слов»* (continuous bag-of-words, CBOW).

Алгоритм CBOW

Алгоритм CBOW работает аналогично skip-gram, но с одним существенным изменением в формулировке проблемы. Модель skip-gram предсказывает контекстные слова из целевого слова, а модель CBOW прогнозирует целевое слово из контекстных слов. Давайте сравним, как выглядят данные для skip-gram и CBOW, взяв предложение из предыдущего примера:

Собака лаяла на почтальона.

Кортежи данных (*входное слово*, *выходное слово*) для skip-gram могут выглядеть следующим образом:

(*лаяла*, *собака*), (*лаяла*, *на*), (*на*, *лаяла*), (*на*, *почтальона*)

Кортежи данных для CBOW будут выглядеть следующим образом:

([*собака*, *на*], *лаяла*), ([*лаяла*, *почтальона*], *на*).

Следовательно, вход CBOW имеет размерность $2 \times m \times D$, где m – размер окна контекста, а D – размерность представлений. Концептуальная модель CBOW показана на рис. 3.13.

Мы не будем вдаваться в тонкости CBOW, поскольку они очень напоминают skip-gram. Тем не менее рассмотрим реализацию алгоритма (хотя и не детально, так как он имеет много общего со skip-gram), чтобы вы получили общее представление о том, как правильно реализовать CBOW. Полная реализация CBOW доступна по адресу `ch3_word2vec.ipynb` в папке упражнений `ch3`.

Реализация алгоритма CBOW с TensorFlow

Реализация, как обычно, начинается с объявления переменных:

```
embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
  embedding_size], -1.0, 1.0, dtype=tf.float32))
softmax_weights = tf.Variable(
  tf.truncated_normal([vocabulary_size, embedding_size],
  stddev=1.0 / math.sqrt(embedding_size),
  dtype=tf.float32))
softmax_biases = tf.Variable(tf.zeros([vocabulary_size],dtype=tf.float32))
```

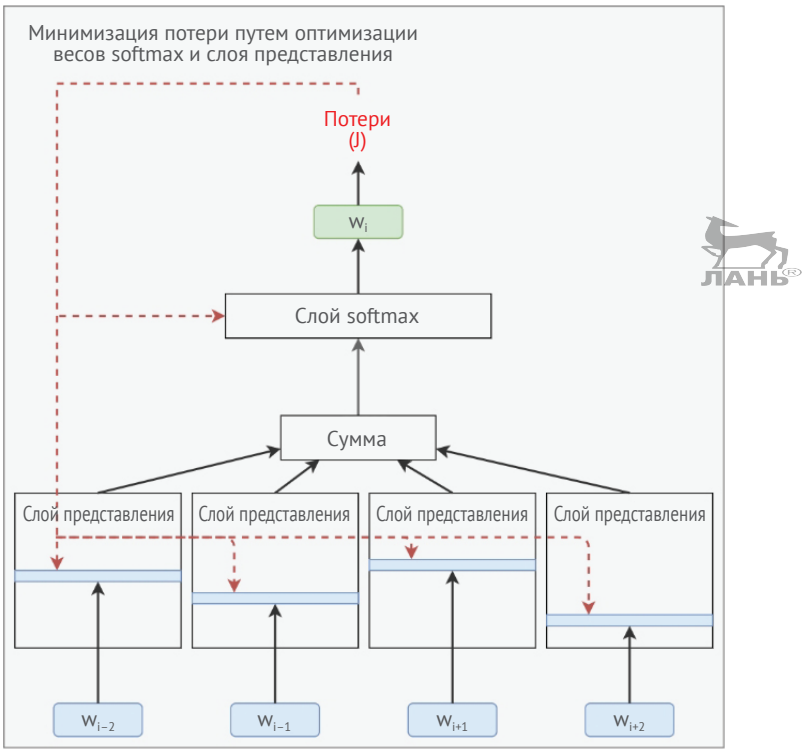


Рис. 3.13 ❖ Модель CBOW

Здесь мы создаем *пакетный набор* (stacked set) представлений, отражающих каждую позицию контекста. Таким образом, у нас будет матрица размера [batch_size, embeddings_size, 2*context_window_size]. Затем мы воспользуемся оператором сокращения, чтобы уменьшить набор до размера [batch_size, embedding_size] путем усреднения *пакетных представлений* (stacked embeddings) по последней оси:

```
stacked_embedings = None
for i in range(2*window_size):
  embedding_i = tf.nn.embedding_lookup(embeddings,
  train_dataset[:,i])
  x_size,y_size = embedding_i.get_shape().as_list()
  if stacked_embedings is None:
    stacked_embedings = tf.reshape(embedding_i,[x_size,y_size,1])
  else:
    stacked_embedings =
    tf.concat(axis=2,
      values=[stacked_embedings,
      tf.reshape(embedding_i,[x_size,y_size,1])]
    )
assert stacked_embedings.get_shape().as_list()[2]==2*window_size
mean_embeddings = tf.reduce_mean(stacked_embedings,2,keepdims=False)
```

Далее определяем потерю `loss` и оптимизатор `optimizer`:

```
loss = tf.reduce_mean(
    tf.nn.sampled_softmax_loss(weights=softmax_weights,
        biases=softmax_biases,
        inputs=mean_embeddings,
        labels=train_labels,
        num_sampled=num_sampled,
        num_classes=vocabulary_size))
optimizer = tf.train.AdagradOptimizer(1.0).minimize(loss)
```

Заключение

Изучение представлений слов стало неотъемлемой частью многих задач NLP и широко используется для таких задач, как машинный перевод, чат-боты, создание подписей к изображениям и моделирование языка. Мало того, что представления слов действуют как метод уменьшения размерности по сравнению с унитарным кодированием, они также дают более богатое представление признаков, чем другие существующие методы. В этой главе мы обсудили два популярных метода обучения представлению слов на основе нейронной сети, а именно skip-gram и CBOW.

Сначала вы узнали про классические подходы и получили представление о том, какие подходы применялись в прошлом. Вы познакомились с различными методами, такими как использование WordNet, построение матрицы совпадений слов и вычисление TF-IDF. Затем вы узнали про ограничения этих подходов.

Это побудило нас изучить методы изучения представлений слов на основе нейронной сети. Во-первых, мы вручную выполнили простой пример, чтобы понять, как можно рассчитать представления, т. е. векторы слов, и убедились, насколько большие возможности у этого подхода.

Далее вы изучили первый алгоритм изучения представлений слов – модель skip-gram и узнали, как подготовить обучающие данные. Позже вы изучили вывод функции потери, использующей слова из контекста целевого слова. Мы обсудили критическое ограничение разработанной нами функции в замкнутой форме. Такая функция потерь не масштабируется для больших словарей. Исходя из этого, мы проанализировали две популярные аппроксимации функции потерь, позволяющие эффективно и действенно рассчитать потери, – отрицательную выборку и иерархический softmax. Наконец, мы кратко рассмотрели реализацию алгоритма skip-gram с помощью TensorFlow.

Далее мы перешли к следующему варианту изучения представлений слов – модели CBOW. Вы узнали, чем CBOW отличается от skip-gram, и ознакомились с реализацией алгоритма CBOW с помощью TensorFlow.

В следующей главе проанализирована производительность уже известных вам методов Word2vec и рассказано про несколько расширений, значительно повышающих производительность. Кроме того, вы освоите новый метод изучения представлений слов, известный как глобальные векторы, или GloVe.

Глава 4

Углубленное изучение Word2vec

В предыдущей главе вы познакомились с Word2vec – основным подходом к изучению представлений слов и двумя распространенными алгоритмами Word2vec: skip-gram и CBOW. В этой главе мы рассмотрим еще несколько тем, связанных с Word2vec, сосредоточив внимание на упомянутых алгоритмах и их расширениях.

Мы начнем главу со сравнения исходного алгоритма skip-gram и его более современного варианта, ранее упомянутого в главе 3. Затем исследуем различия между skip-gram и CBOW и рассмотрим изменение потерь в процессе обучения при использовании этих алгоритмов. Исходя из наших наблюдений и доступных публикаций, попробуем сделать вывод, какой алгоритм работает лучше.

Вы познакомитесь с расширениями методов Word2vec для повышения точности, включая использование более эффективных методов отбора примеров для негативной выборки и игнорирование неинформативных слов в процессе обучения. Вы также изучите новую технику изучения представлений слов GloVe и узнаете про преимущества этой техники по сравнению с skip-gram и CBOW.

Наконец, узнаете, как использовать Word2vec для решения реальной проблемы – классификации документов. Вы научитесь получать представления документов из представлений слов.

Исходный алгоритм skip-gram

Алгоритм skip-gram, ранее обсуждавшийся в этой книге, на самом деле является улучшенным вариантом алгоритма, предложенного в оригинальной статье Миколова и др., опубликованной в 2013 году. В этой статье алгоритм не использует промежуточный скрытый слой, чтобы изучить представления. Напротив, в исходном алгоритме использовались два разных слоя представлений, или *проекции* (входные и выходные представления на рис. 4.1), и определялась функция стоимости, полученная из самих представлений.

Оригинальная *потеря при отрицательной выборке* (negative sampled loss) была определена следующим образом:

$$J(\theta) = -\left(\frac{1}{N-2m}\right)\sum_{i=m+1}^{N-m}\sum_{j\neq i \wedge j=i-m}^{i+m}\log\left(\sigma\left(v_{w_j}'^{T} v_{w_i}\right)\right) + kE_{w_q \sim P_n(w)}\left[\log\sigma\left(-v_{w_q}'^{T} v_{w_i}\right)\right].$$

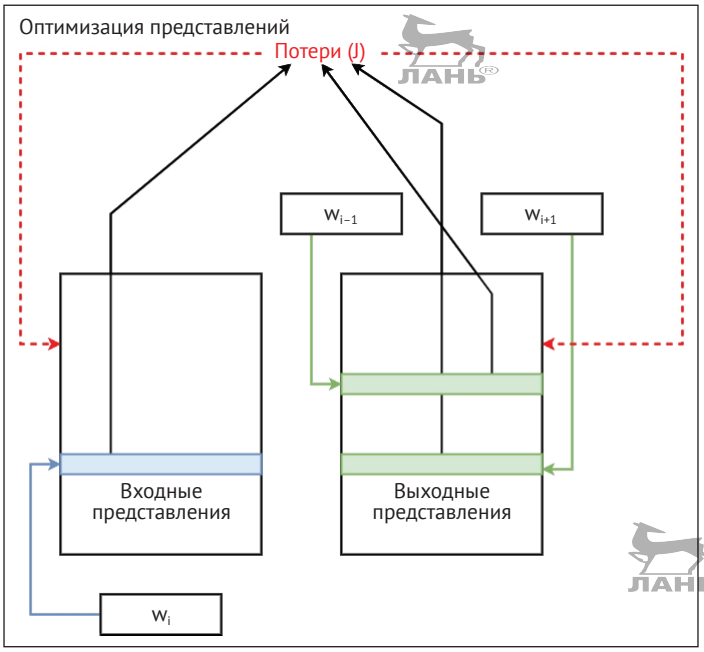


Рис. 4.1 ❖ Оригинальный алгоритм skip-gram без скрытых слоев

Здесь v – уровень входных представлений, v' – уровень представлений выходных слов, v_{w_i} соответствует вектору представления слова w_i во входном слое представлений, а v'_{w_i} – вектору слова w_i в выходном слое представлений, $P_n(w)$ – это распределение шума, из которого мы выбираем выборки шума. Например, это может быть просто равномерная выборка из словаря – $\{w_i, w_j\}$, как мы видели в главе 3. Наконец, E обозначает ожидание (среднее) потерь, полученных от k-отрицательных образцов. Как вы можете видеть, в этом уравнении нет весов и смещений, за исключением самого представления слова.

Реализация исходного алгоритма skip-gram

Реализация исходного алгоритма skip-gram не так проста, как версия, которую мы уже реализовали. Это связано с тем, что функцию потерь необходимо создавать вручную с использованием стандартных функций TensorFlow, поскольку в данном случае нет встроенной функции для расчета потерь, как это было для других алгоритмов.

Во-первых, определим следующие заполнители:

- **входные данные** – заполнитель, содержащий пакет целевых слов с формой `[batch_size]`;
- **выходные данные** – заполнитель, содержащий соответствующие слова контекста для пакета целевых слов и имеющий размер `[batch_size, 1]`:

```
train_dataset = tf.placeholder (tf.int32, shape = [batch_size])
train_labels = tf.placeholder (tf.int64, shape = [batch_size, 1])
```

Определив заполнители ввода и вывода, мы можем использовать встроенную функцию TensorFlow `candidate_sampler` для отбора отрицательных выборок, как показано в следующем коде:

```
negative_samples, _, _ = tf.nn.log_uniform_candidate_sampler(
                                    train_labels, num_true=1,
                                    num_sampled=num_sampled,
                                    unique=True,
                                    range_max=vocabulary_size)
```

Здесь мы выбираем отрицательные слова равномерно, без каких-либо предпочтений. `train_labels` – это истинные выборки, поэтому TensorFlow избегает использовать их в качестве отрицательных выборок. Далее идет число `num_true`, которое обозначает количество истинных классов для данной точки данных, равное 1. За ним следует количество отрицательных выборок `num_sampled`, которые мы хотим получить для пакета данных. Параметр `unique` определяет, должны ли отрицательные образцы быть уникальными. Наконец, `range` определяет максимальный идентификатор слова, чтобы семплер не создавал недопустимых идентификаторов.

Мы избавляемся от весов и смещений softmax и вводим два слоя представлений – один для входных данных, а другой для выходных. Два слоя представлений необходимы для нормальной работы функции стоимости, как было сказано в главе 3.

Затем добавим функции *обхода* (lookup) для входных данных, выходных данных и отрицательных выборок:

```
in_embed = tf.nn.embedding_lookup(in_embeddings, train_dataset)
out_embed = tf.nn.embedding_lookup(out_embeddings, tf.reshape(train_labels,[-1]))
negative_embed = tf.nn.embedding_lookup(out_embeddings, negative_samples)
```

Далее мы определим функцию потерь, и это самая важная часть кода. Этот код реализует функцию потерь, которую мы обсуждали ранее. Однако, как следует из уравнения функции $J(\theta)$, мы не рассчитываем потери для всех слов в документе одновременно. Это связано с тем, что документ может оказаться слишком большим, чтобы полностью поместиться в памяти. Поэтому мы рассчитываем потери для небольших пакетов данных шаг за шагом. Полный код доступен в файле `ch4_word2vec_improvements.ipynb`, расположенном в папке `ch4`.

```
# Вычисление потери для истинных выборок
loss = tf.reduce_mean(
    tf.log(
        tf.nn.sigmoid(
            tf.reduce_sum(
                tf.diag([1.0 for _ in range(batch_size)])*
                tf.matmul(out_embed,tf.transpose(in_embed)),
            axis=0)
        )
    )
)

# вычисление потери для отрицательных выборок
loss += tf.reduce_mean(
```

```
    tf.reduce_sum(
        tf.log(tf.nn.sigmoid(
            -tf.matmul(negative_embed,tf.transpose(in_embed)))),
        axis=0
    )
)
```

Tensorflow реализует `sampled_softmax_loss`, выделяя из полных весов и смещений softmax подмножество весов и смещений, необходимых только для обработки текущего пакета данных. Затем TensorFlow вычисляет потерю по аналогии со стандартным расчетом кросс-энтропии softmax. Однако мы не можем напрямую применить этот подход к расчету потери в исходном алгоритме skip-gram, так как там нет весов и смещений softmax.

Сравнение исходного и улучшенного алгоритмов skip-gram

У нас должна быть веская причина использовать скрытый слой, в отличие от исходного алгоритма skip-gram, где его нет. Поэтому мы сопоставим поведение функции потерь двух алгоритмов – исходного и включающего скрытый слой, как показано на рис. 4.2.

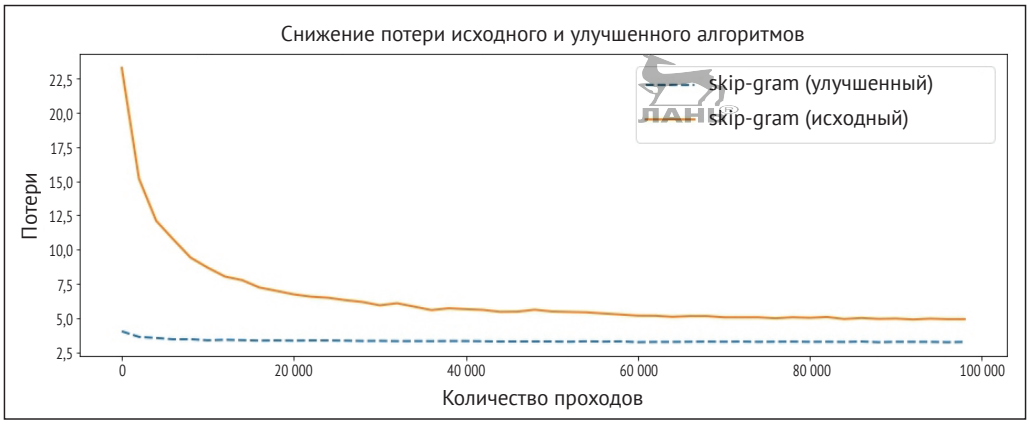


Рис. 4.2 ❖ Сравнение продуктивности исходного и улучшенного алгоритмов skip-gram

Очевидно, что наличие скрытого слоя обеспечивает лучшую продуктивность[1] обучения и точность модели. Отсюда следует, что более глубокие модели Word2vec, как правило, работают лучше.

[1] Под *продуктивностью обучения* здесь и далее мы понимаем количество циклов (эпох) обучения, необходимых для достижения минимальной или приемлемой потери. Чем выше продуктивность, тем быстрее обучается модель и тем ниже вычислительные затраты. – *Прим. перев.*

Сравнение skip-gram и CBOW

Прежде чем рассматривать различия в продуктивности алгоритмов и исследовать причины, давайте вспомним фундаментальное различие между методами skip-gram и CBOW.

Как показано на рис. 4.3 и 4.4, алгоритм skip-gram наблюдает только целевое слово и одно слово контекста в одном кортеже ввода/вывода. С другой стороны, CBOW наблюдает целевое слово и все слова контекста в одной выборке. Например, если мы возьмем фразу «*Собака лает на почтальона*», то skip-gram видит только один кортеж ввода-вывода, например [*собака*, *на*], за один шаг, тогда как CBOW видит кортеж ввода-вывода [[*собака*, *лает*, *почтальона*], *на*]. Следовательно, в одном пакете данных CBOW получает больше информации о контексте данного слова, чем skip-gram. Давайте теперь посмотрим, как эта разница влияет на точность двух алгоритмов.

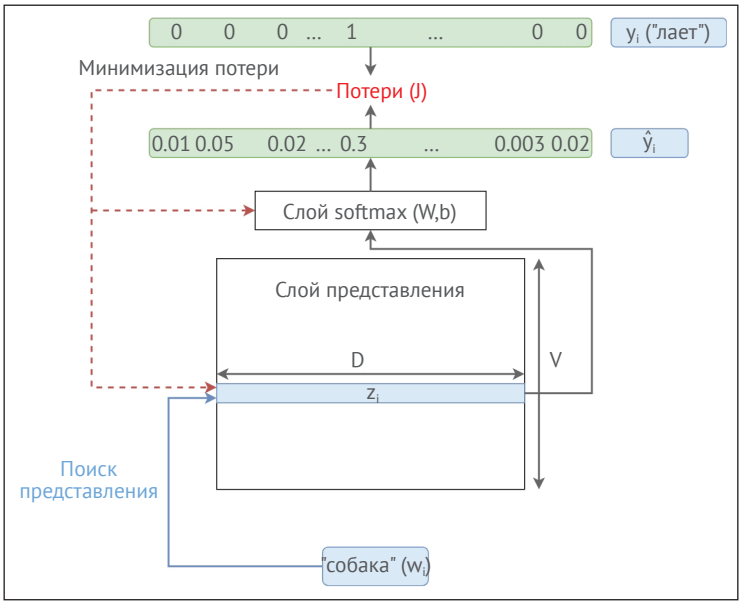


Рис. 4.3 ❖ Блок-схема реализации алгоритма skip-gram

Как показано на рис. 4.3 и 4.4, CBOW имеет доступ к большему количеству информации (входных данных) в заданный момент времени по сравнению со skip-gram, что позволяет CBOW работать лучше при прочих равных условиях.

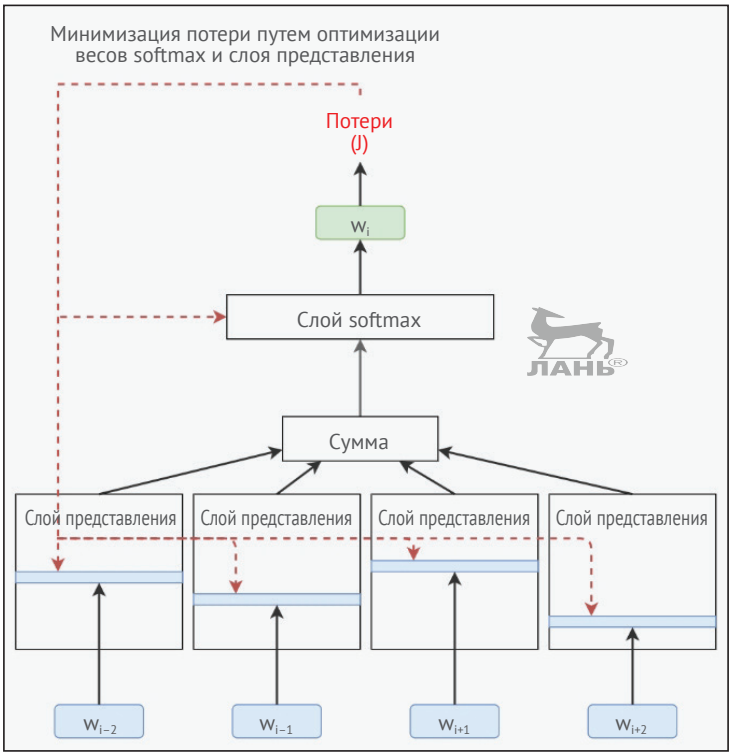


Рис. 4.4 ❖ Блок-схема реализации алгоритма CBOW

Сравнение продуктивности

Теперь давайте построим график поведения потерь для skip-gram и CBOW в задаче, которой мы обучали модели ранее в главе 3 (см. рис. 4.5).

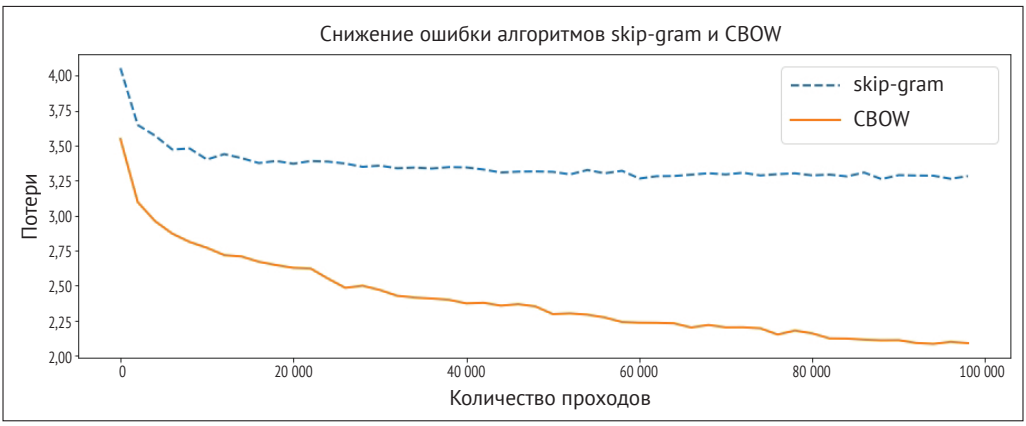


Рис. 4.5 ❖ График поведения потерь – сравнение skip-gram и CBOW

График ясно показывает, что CBOW показывает более быстрое уменьшение потери по сравнению с моделью skip-gram. Однако потеря сама по себе не является исчерпывающей мерой продуктивности, поскольку она может быстро сокращаться из-за переобучения. Хотя существуют контрольные задачи для оценки точности представлений слов (например, задачи на поиск аналогий), мы будем использовать более простой способ проверки. Давайте визуально исследуем изученные представления, чтобы убедиться, что skip-gram и CBOW находят между ними значительную семантическую разницу. Для этого мы используем популярную технику визуализации, известную как t-распределенное представление стохастической близости t-SNE.

Следует отметить, что уменьшение потери не является очень убедительным показателем для оценки точности представления слов, поскольку выборочный softmax, который мы используем для измерения потерь, значительно недооценивает полную потерю. Точность представлений слов часто оценивают при помощи задач на поиск аналогии. Типичное задание на поиск аналогии может выглядеть так:

Осознанная неосознанность – это как заметная _____.

Итак, хороший набор представлений должен дать ответ «*незаметность*». Его можно вычислить с помощью простой арифметической операции:

```
представление (заметная) – [представление (осознанная) – представление
(неосознанность)].
```

Если ближайшим соседом результирующего вектора является вектор слова «*незаметность*», то вы получили правильный ответ.
Существует несколько открытых наборов данных для проверки по аналогиям слов, например:

- набор данных Google: **http://download.tensorflow.org/data/questions-words.txt**;
- большой набор аналоговых тестов (BATS): **http://vsm.blackbird.pw/bats**.

На рис. 4.6 мы видим, что CBOW имеет тенденцию группировать слова плотнее, чем skip-gram, у которого слова, кажется, рассеяны по всему пространству. Можно сказать, что в этом конкретном примере CBOW выглядит визуально более привлекательным по сравнению со skip-gram.

Мы будем использовать алгоритм t-SNE из библиотеки scikit-learn, чтобы вычислить представление с уменьшенной размерностью и затем визуализировать его при помощи Matplotlib. Тем не менее TensorFlow предоставляет гораздо более удобный вариант визуализации представлений с помощью своего фреймворка визуализации TensorBoard. Вы можете найти это упражнение в файле `tensorboard_word_embeddings.ipynb`, расположенном в папке `appendix`.

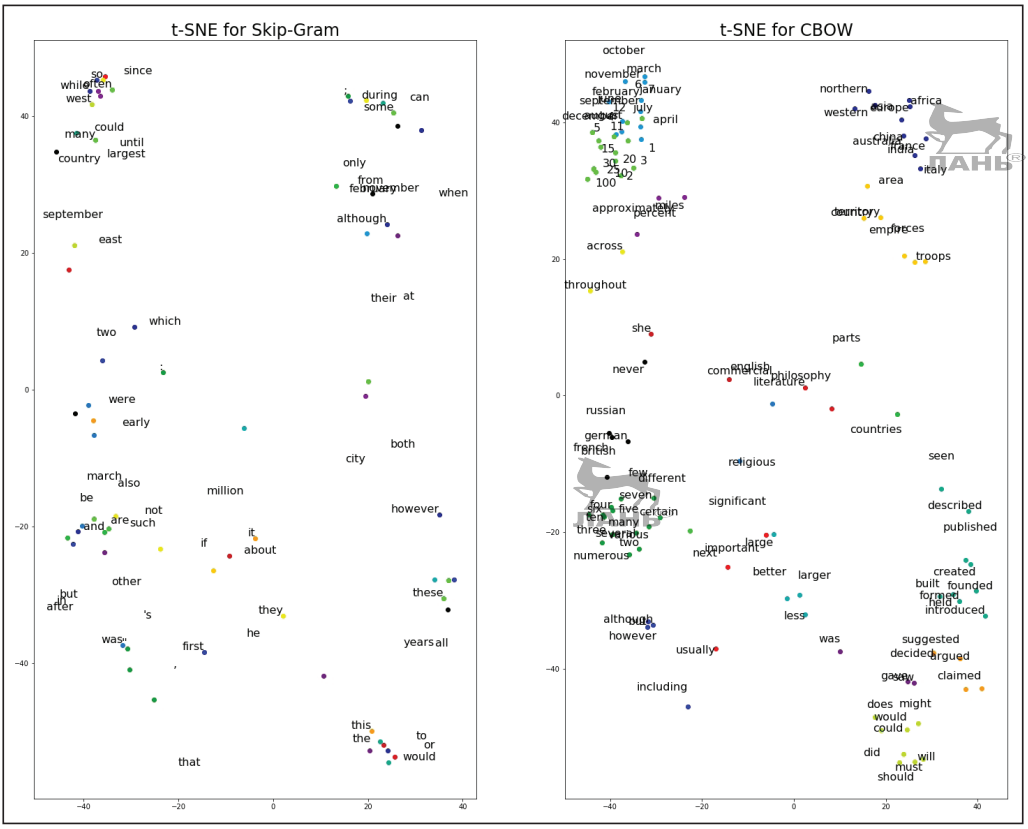


Рис. 4.6 ❖ Визуализация t-SNE для векторов слов, полученных с помощью skip-gram и CBOW

Обзорное знакомство с t-SNE

t-SNE – это метод визуализации, способный визуализировать многомерные данные, например изображения и представления слов, в плоском двумерном пространстве. Мы не будем углубляться в сложные математические приемы, лежащие в основе этого метода, и лишь в общих чертах рассмотрим, как работает алгоритм.

Сначала определим обозначения. Запись $x_i \in R^D$ обозначает D-мерную точку данных, а $X = \{x_i\}$ – входное пространство. Например, это может быть вектор представления слов, подобный тем, которые мы рассмотрели в главе 3, а D – размер представления. Далее, представим гипотетическое двумерное пространство $Y = \{y_i\}$, где y_i обозначает двумерный вектор, соответствующий точке данных x_i; X и Y имеют однозначное сопоставление. Мы будем называть Y *пространством карты* (map space), а y_i – *точками карты* (map point).

Теперь определим условную вероятность $P_{j|i}$, которая определяет вероятность того, что точка данных x_i окажется по соседству с точкой x_j. Значение $P_{j|i}$ должно быть низким, когда точка x_j далека от x_i, и наоборот. Интуитивно понятный выбор для $P_{j|i}$ – это гауссово распределение с центром в точке данных x_i и дисперсией σ_i^2. Дисперсия будет низкой для точек данных, плотно окруженных соседями, и высокой для точек данных с разреженными окрестностями. Если точно, формула для условной вероятности выглядит так:

$$p_{j|i} = \frac{\exp\left(-\left\|x_i - x_j^2\right\| / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\left\|x_i - x_k^2\right\| / 2\sigma_i^2\right)}.$$

Мы можем аналогичным образом определить условную вероятность для точек карты y_i в пространстве Y, $q_{j|i}$.

Чтобы получить хорошее двумерное представление Y многомерного пространства X, вероятности $p_{j|i}$ и $q_{j|i}$ должны демонстрировать одинаковое поведение. То есть если две точки данных похожи в пространстве X, они также должны быть похожими в пространстве Y, и наоборот. Поэтому проблема получения хорошего двумерного представления данных сводится к минимизации несоответствия между $p_{j|i}$ и $q_{j|i}$ для всех $i = 1, \ldots, N$.

Эта проблема может быть формально представлена как минимизация расстояния Кульбака–Лейблера между $p_{j|i}$ и $q_{j|i}$, обозначаемого $\mathrm{KL}(p_{j|i} \| q_{j|i})$.

Поэтому функция стоимости для нашей задачи будет иметь вид:

$$C = \sum_{i=1}^{N} KL(p_{j|i} \| q_{j|i}) == \sum_{i=1}^{N} \sum_{j \neq i} p_{j|i} \log\left(\frac{p_{j|i}}{q_{j|i}}\right).$$

Кроме того, минимизируя стоимость C посредством стохастического градиентного спуска, мы можем найти оптимальное представление Y, которое близко соответствует X.

Мысленно этот процесс можно рассматривать как равновесие, достигаемое набором пружин, прикрепленных между всеми парами точек данных. $p_{j|i}$ – жесткость пружины между точками данных x_i и x_j. Поэтому чем больше x_i и x_j похожи друг на друга, тем ближе они будут располагаться. Следовательно, C для конкретной точки данных действует как общая сила, действующая на эту точку данных, и заставляет ее либо притягивать, либо отталкивать все остальные точки данных в соответствии с общей силой.

Кто же победитель, skip-gram или CBOW?

Когда дело доходит до точности, мы не можем назвать явного победителя. Например, в статье[1] Миколова и др. (2013) «Распределенные представления слов и фраз и их композиционности» предлагается, что skip-gram лучше работает в семантических задачах, тогда как CBOW – в синтаксических. Однако на практике в большинстве реальных задач skip-gram работает лучше, чем CBOW, что противоречит нашим наблюдениям.

Различные эмпирические данные свидетельствуют о том, что skip-gram лучше работает с большими наборами данных, которые обычно используют корпуса из миллиардов слов, и это наблюдение подтверждено в упомянутой статье Миколова, а также в статье[2] Пеннингтона и др. «GloVe: глобальные векторы для представления слов». Однако в нашей задаче было несколько сотен тысяч слов, что сравнительно мало. По этой причине CBOW мог работать лучше.

Теперь позвольте мне объяснить, почему я так думаю. Рассмотрим следующие два предложения:

---

[1] Distributed Representations of Words and Phrases and their Compositionality. Mikolov and others, 2013.

[2] GloVe: Global Vectors for Word Representation. Pennington and others, 2014.

- Мы пришли в дорогой ресторан
- Мы пришли в роскошный ресторан

Для CBOW кортежи ввода-вывода будут выглядеть так:

[[*Мы*, *пришли*, *дорогой*, *ресторан*], *в*]
[[*Мы*, *пришли*, *роскошный*, *ресторан*], *в*]

Кортежи ввода-вывода для skip-gram будут выглядеть так:

[*Мы*, *в*], [*пришли*, *в*], [*дорогой*, *в*], [*ресторан*, *в*]
[*Мы*, *в*], [*пришли*, *в*], [*роскошный*, *в*], [*ресторан*, *в*]

Мы бы хотели, чтобы наша модель понимала, что «*дорогой*» и «*роскошный*» – это несколько разные вещи (то есть «*роскошный*» означает «круче, чем просто дорогой»). Такие слова, имеющие тонкие различия в значении, называются нюансами. Можно предположить, что CBOW с высокой вероятностью воспримет слова «*дорогой*» и «*роскошный*» как одно и то же, потому что их семантика усредняется окружающими словами («*Мы*», «*пришли*» и «*ресторан*»), так как эти слова также являются частью входных данных. В отличие от этого, для skip-gram слова «*дорогой*» и «*роскошный*» выглядят отделенными от «*Мы*», «*пришли*» и «*ресторан*», что позволяет skip-gram уделять больше внимания тонким различиям между словами.

Однако обратите внимание, что в обычной модели есть миллионы параметров. Для обучения таких моделей требуется много данных. CBOW каким-то образом обходит эту проблему, пытаясь сосредоточить внимание не на тонких различиях, а на усреднении всех слов в данном контексте (например, нечто среднее между «*дорогой*» и «*роскошный*»). С другой стороны, skip-gram научится делать более дотошные представления, потому что нет эффекта усреднения, как в CBOW. Разумеется, чтобы изучить дотошные представления, skip-gram требует больше данных. Но при наличии этих данных skip-gram, скорее всего, превзойдет алгоритм CBOW.

Кроме того, обратите внимание, что один вход в модель CBOW приблизительно равен $2\times m$ входам модели skip-gram, где m – размер окна контекста. Это связано с тем, что один вход skip-gram состоит только из одного слова, когда один вход в CBOW состоит из $2\times m$ слов. Таким образом, чтобы сравнение алгоритмов было полностью справедливым, если мы запускаем CBOW на L шагов, то должны запустить алгоритм skip-gram на $2m\times L$ шагов.

Итак, вы узнали, как изначально был реализован skip-gram – у него было два слоя представления, один для просмотра входных слов, а другой для просмотра выходных слов. Мы разобрались, почему алгоритм skip-gram, рассмотренный в главе 3, на самом деле является улучшением исходного алгоритма Миколова, и увидели, что улучшенный skip-gram фактически превосходит исходный алгоритм. Затем мы сравнили характеристики skip-gram и CBOW и увидели, что в нашем примере CBOW работает лучше. Наконец, высказали предположение, почему CBOW выглядит победителем.

Расширения алгоритмов представления слов

В статье Миколова, опубликованной в 2013 г., упомянуты некоторые расширения, способные еще больше повысить точность алгоритмов изучения представлений слов. Хотя они изначально разработаны для использования в skip-gram, они также подойдут и для CBOW. Поскольку мы убедились, что CBOW превосходит алгоритм skip-gram в нашем небольшом примере, мы будем использовать его при изучении всех расширений.

Использование униграммного распределения для отрицательной выборки

Было обнаружено, что результаты отрицательной выборки лучше, когда она выполняется на некоторых определенных распределениях. Одним из них является *униграммное распределение* (unigram distribution). Вероятность униграммы слова w_i определяется следующим уравнением:

$$U(w_i) = \frac{count(w_i)}{\sum_{j \in Corpus} count(w_j)}.$$

Здесь $count(w_i)$ – это количество раз, когда w_i появляется в документе. Когда униграммное распределение *искажено* (distorted) как $U(w_i)^{(3/4)}/Z$ для некоторой константы Z, оно обеспечивает лучшую точность, чем равномерное распределение или стандартное униграммное распределение.

Давайте рассмотрим пример, чтобы лучше понять униграммное распределение. Рассмотрим следующие предложения:

Боб до безумия любит футбол. Он играет в футбол в школьной команде.

Здесь вероятность униграммы слова «*футбол*» будет следующей:

U(*футбол*) = 2/12 = 1/6.

Очевидно, что вероятность униграмм общих слов будет выше. Как правило, общими чаще всего оказываются неинформативные слова, такие как предлоги, артикли, частицы *не/ни* и т. д. Во время оптимизации потери эти слова будут чаще попадать в отрицательную выборку, поэтому более информативные слова будут отбираться менее негативно. Следовательно, униграммы формируют баланс между общими и редкими словами во время оптимизации, что приводит к повышению точности.

Реализация отрицательной выборки на основе униграмм

Рассмотрим пример реализации отрицательной выборки на основе униграмм с помощью TensorFlow:

```
unigrams = [0 for _ in range(vocabulary_size)]
for word,w_count in count:
```

```
    w_idx = dictionary[word]
    unigrams[w_idx] = w_count*1.0/token_count
    word_count_dictionary[w_idx] = w_count
```

Здесь count – это список кортежей, из которых состоит каждый кортеж (идентификатор слова, частота). Этот алгоритм вычисляет вероятности униграммы каждого слова и возвращает их в виде списка, упорядоченного по индексу слова. Данный формат для униграмм предусмотрен в TensorFlow. Полный код упражнения доступен в файле ch4_word2vec_improvements.ipynb, расположенном в папке ch4.

Далее выполняем вычисления, включая обход представлений, как мы обычно делали для CBOW:

```
train_dataset = tf.placeholder(tf.int32, shape=[batch_size,
    window_size*2])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

# Переменные.
# Представления, вектор для каждого слова в словаре.
embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
    embedding_size], -1.0, 1.0, dtype=tf.float32))
softmax_weights =
    tf.Variable(tf.truncated_normal([vocabulary_size,
    embedding_size],
    stddev=1.0 / math.sqrt(embedding_size), dtype=tf.float32))
softmax_biases =
    tf.Variable(tf.zeros([vocabulary_size], dtype=tf.float32))

stacked_embedings = None

for i in range(2*window_size):
    embedding_i = tf.nn.embedding_lookup(embeddings,
    train_dataset[:,i])
    x_size,y_size = embedding_i.get_shape().as_list()
    if stacked_embedings is None:
        stacked_embedings =
            tf.reshape(embedding_i,[x_size,y_size,1])
    else:
        stacked_embedings =
            tf.concat(axis=2,values=[stacked_embedings,
            tf.reshape(embedding_i,[x_size,y_size,1])])
mean_embeddings = tf.reduce_mean(stacked_embedings,2,keepdims=False)
```

Затем мы берем отрицательные выборки, основанные на распределении униграмм. Для этого воспользуемся встроенной функцией TensorFlow tf.nn.fixed_unigram_candidate_sampler:

```
candidate_sampler = tf.nn.fixed_unigram_candidate_sampler(
    true_classes = tf.cast(train_labels, dtype=tf.int64),
    num_true = 1, num_sampled = num_sampled, unique = True,
    range_max = vocabulary_size, distortion=0.75,
    num_reserved_ids=0, unigrams=unigrams, name='unigram_sampler')

loss = tf.reduce_mean(
    tf.nn.sampled_softmax_loss(weights=softmax_weights,
    biases=softmax_biases, inputs=mean_embeddings,
```

```
    labels=train_labels, num_sampled=num_sampled,
    num_classes=vocabulary_size, sampled_values=candidate_sampler))
```

Этот фрагмент кода обеспечивает общий процесс реализации изучения представлений слов с отрицательной выборкой на основе униграмм. Обычно выполняются следующие шаги.

1. Определение переменных, заполнителей и гиперпараметров.
2. Для каждой партии данных делают следующее:
   1) вычисление средней матрицы входных представлений путем поиска представлений для каждого индекса контекстного окна и их усреднения;
   2) расчет потери с помощью отрицательной выборки, отобранной согласно униграммному распределению;
   3) оптимизация нейронной сети с использованием стохастического градиентного спуска.

Следующий однострочный код, извлеченный из предыдущего фрагмента кода, играет наиболее важную роль в этом алгоритме, создавая отрицательные выборки, сгенерированные в соответствии с искаженным униграммным распределением:

```
candidate_sampler = tf.nn.fixed_unigram_candidate_sampler(
    true_classes = tf.cast(train_labels,dtype=tf.int64),
    num_true = 1, num_sampled = num_sampled, unique = True,
    range_max = vocabulary_size, distortion=0.75,
    num_reserved_ids=0, unigrams=unigrams, name='unigram_sampler')
```

Давайте подробно рассмотрим каждый аргумент этой функции:

- `true_classes` – вектор размера `batch_size`, который предоставляет идентификатор целевого слова (целое число) для заданной партии контекстных слов, соответствующих этому целевому слову;
- `num_true` – количество истинных элементов для данного слова (часто 1);
- `num_sampled` – количество отрицательных элементов для выборки на один вход;
- `unique` – указывает брать уникальные отрицательные выборки (без замены);
- `range_max` – размер словаря;
- `distortion` – возвращает униграмму, возведенную в степень, определяемую значением *искажения* (distortion). В нашем примере это 3/4 = 0,75;
- `num_reserved_ids` – это список индексов, обозначающих слова из словаря. Эти индексы не будут выбраны как отрицательные;
- `unigrams` – это вероятности униграмм, упорядоченные по идентификатору слова.

Подвыборка – вероятностное игнорирование общих слов

Подвыборка (subsampling), или игнорирование общих слов, также обеспечивает повышение точности. Смысл подвыборки можно пояснить следующим интуитивно понятным примером: слова ввода-вывода, извлеченные из конечного контекста (*это*, *Франция*), предоставляют меньше информации, чем кортеж (*Париж*, *Франция*). Поэтому лучше игнорировать подобные неинформативные слова (т. е. стоп-слова, такие как слово «*это*»), слишком часто отбираемые из корпуса. Математически это достигается путем игнорирования слова w_i в последовательности слов в корпусе с вероятностью:

$$1-\sqrt{\frac{t}{f(w_i)}}.$$

Здесь t – это константа, которая определяет верхний порог частоты слова, а $f(w_i)$ – частота w_i в корпусе. Подвыборка эффективно уменьшает частоту стоп-слов, улучшая баланс набора данных.

Реализация подвыборки

Простая реализация подвыборки показана в следующем примере фрагмента кода. Мы создаем новую последовательность слов из исходной последовательности, удаляя слова с вероятностью, вычисленной, как показано выше, и будем использовать эту новую последовательность слов для изучения представлений слов. В данном случае мы выбрали t = 10 000:

```
subsampled_data = []
for w_i in data:
    p_w_i = 1 - np.sqrt(1e5/word_count_dictionary[w_i])

    if np.random.random() < p_w_i:
        drop_count += 1
        drop_examples.append(reverse_dictionary[w_i])
    else:
        subsampled_data.append(w_i)
```

Сравнение CBOW и его расширений

На рис. 4.7 изображены графики снижения потери для трех случаев:

- CBOW без дополнений;
- CBOW с отрицательной выборкой на основе униграмм – CBOW (униграмма);
- CBOW с отрицательной выборкой на основе униграмм и подвыборкой – CBOW (униграмма + подвыборка).

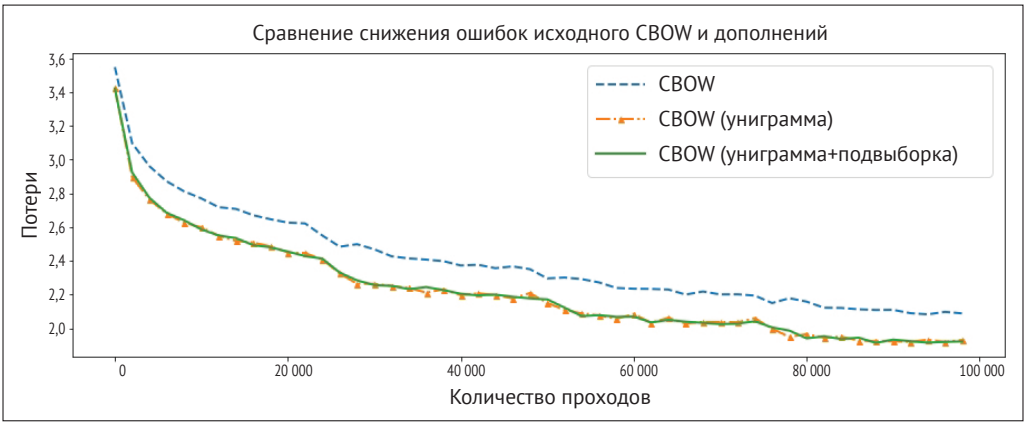


Рис. 4.7 ❖ Сравнение снижения потери у исходного CBOW и алгоритма с различными расширениями

Весьма интересно отметить, что использование комплекса из униграмм и подвыборки дает в целом аналогичные значения потерь по сравнению с использованием только негативной выборки на основе униграмм. Тем не менее это не должно быть неправильно истолковано как отсутствие выгоды от подвыборки в задаче обучения. Причину такого поведения в данном случае нетрудно объяснить. Благодаря использованию подвыборки мы дополнительно избавляемся от многих неинформативных слов, поэтому качество текста увеличивается (с точки зрения качества информации). Это, в свою очередь, усложняет задачу обучения. В предыдущей постановке задачи была возможность использовать обилие неинформативных слов в процессе оптимизации, тогда как в новой постановке задачи такие шансы редки. Это приводит к большей потере (проще говоря, цена ошибки выше), но зато улучшается семантическая точность векторных представлений слов.

Более современные алгоритмы, расширяющие skip-gram и CBOW

Мы уже убедились, что методы Word2vec достаточно мощно выделяют семантику слов. Однако они не лишены ограничений. Например, они не обращают внимания на расстояние между контекстным словом и целевым словом. Однако если контекстное слово находится дальше от целевого слова, его влияние на целевое слово должно быть меньше. Существуют методы, уделяющие отдельное внимание позиционному расположению слов в контексте. Другое ограничение Word2vec заключается в том, что при вычислении вектора слова он обращает внимание только на очень маленькое окно вокруг заданного слова. Однако в действительности полезно рассматривать в совокупности все вхождения слова в корпус. Далее мы рассмотрим метод, который учитывает не только контекст слова, но и глобальную информацию о вхождениях.

Ограничение алгоритма skip-gram

Алгоритм skip-gram и все его варианты игнорируют локализацию контекстных слов в данном контексте. Другими словами, skip-gram не использует точное положение контекстного слова и одинаково обрабатывает все слова. Например, рассмотрим предложение:

Эта собака лаяла на почтальона.

Давайте зададим размер контекстного окна 2 и возьмем целевое слово «*лаяла*». Тогда контекстом для слова «*лаяла*» будут слова «*Эта*», «*собака*», «*на*», «*почтальона*». Кроме того, сформируем четыре точки данных (*лаяла*, *Эта*), (*лаяла*, *собака*), (*лаяла*, *на*) и (*лаяла*, *почтальона*), где первый элемент кортежа является входным словом, а второй – выходным словом. Если мы возьмем две точки данных из этой коллекции (*лаяла*, *Эта*) и (*лаяла*, *собака*), то во время оптимизации оригинальный алгоритм skip-gram будет обрабатывать оба этих кортежа одинаково. Другими словами, skip-gram игнорирует фактическое положение контекстного слова

в контексте. Однако с лингвистической точки зрения очевидно, что кортеж (*лаяла*, *собака*) несет больше информации, чем (*лаяла*, *Эта*). По сути, структурированный skip-gram пытается устранить это ограничение. Давайте посмотрим, как он работает.

Структурированный алгоритм skip-gram

Структурированный алгоритм skip-gram (structured skip-gram algorithm) использует архитектуру, показанную на рис. 4.8, для обхода ограничения исходного алгоритма, рассмотренного в предыдущем разделе.

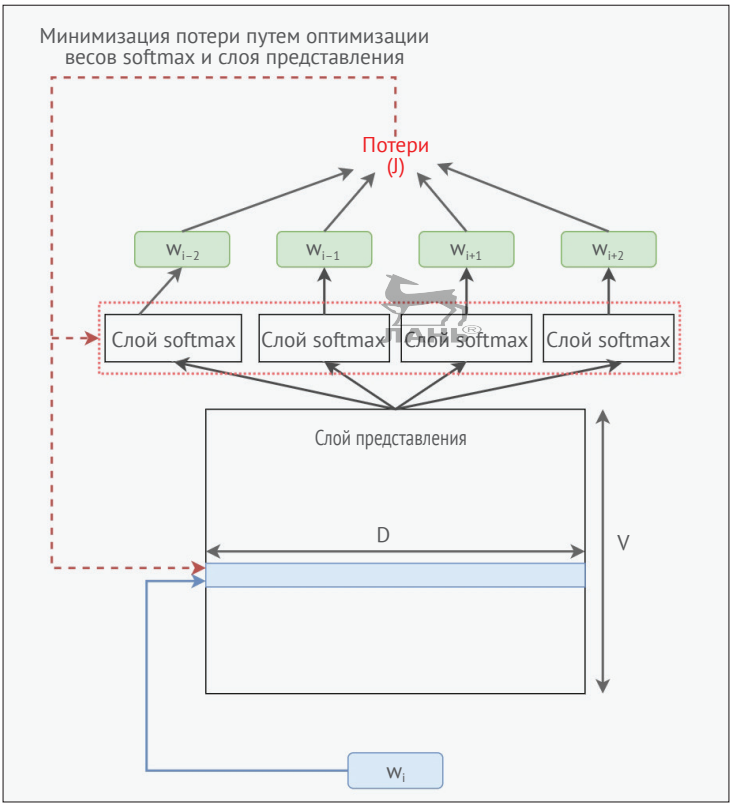


Рис. 4.8 ❖ Схема структурированного алгоритма skip-gram

Структурированный skip-gram сохраняет структуру или локализацию контекстных слов во время оптимизации. С другой стороны, возникают более высокие требования к памяти, так как количество параметров линейно зависит от размера окна. Точнее, при размере окна m (в одну сторону), если исходная модель skip-gram имела P параметров в слое softmax, модель структурированного skip-gram будет иметь $2mP$ параметров, так как у нас есть набор из P параметров для каждой позиции в контекстном окне.

Функция потерь

Оригинальная потеря при отрицательной выборке для модели skip-gram выглядела так:

$$J(\theta) = -\left(\frac{1}{N} - 2m\right) \sum_{i=m+1}^{N-m} \sum_{j \neq i}^{i+m} \sum_{j=i-m} \log(\sigma(\text{logit}(x_n)_{w_j}))$$
$$+ \sum_{q=1}^{k} \mathbb{E}_{w_q \sim vocabulary - \{w_i, w_j\}} \log(\sigma(-\text{logit}(x_n)_{w_q})).$$

Для структурированного skip-gram мы используем следующую потерю:

$$J(\theta) = \sum_{p=1}^{2m} -\left(\frac{1}{N} - 2m\right) \sum_{i=m+1}^{N-m} \sum_{j \neq i}^{i+m} \sum_{j=i-m} \log(\sigma(\text{logit}_k(x_n)_{w_j}))$$
$$+ \sum_{q=1}^{k} \mathbb{E}_{w_q \sim vocabulary - \{w_i, w_j\}} \log(\sigma(-\text{logit}_p(x_n)_{w_j})).$$

Здесь $\text{logit}_p(x_n)_{w_j}$ рассчитывается с использованием p-го набора весов softmax и смещения, соответствующих индексу позиции w_j.

Реализация алгоритма показана в следующем коде, который доступен полностью в файле `ch4_word2vec_extended.ipynb` в папке `ch4`. Как видите, теперь у нас есть $2 \times m$ весов и смещений softmax и векторы представления, соответствующие каждой позиции контекста, распространяются с их собственным весом и смещением.

Сначала мы определяем входные и выходные заполнители:

```
train_dataset = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = [tf.placeholder(tf.int32, shape=[batch_size, 1]) for _
in range(2*window_size)]
```

Затем определяем вычисления, необходимые для расчета потери, начиная с обучающих входов и меток:

```
# Переменные.
embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size],
    -1.0, 1.0))
softmax_weights = [tf.Variable(
    tf.truncated_normal([vocabulary_size, embedding_size],
    stddev=0.5 / math.sqrt(embedding_size))) for _ in range(2*window_
size)]
softmax_biases =
    [tf.Variable(tf.random_uniform([vocabulary_size],0.0,0.01)) for _
in range(2*window_size)]

# Модель.
# Обход представлений входов.
embed = tf.nn.embedding_lookup(embeddings, train_dataset)
# Вычисление потери softmax с использованием
# выборок по отрицательным меткам.
loss = tf.reduce_sum(
    [
        tf.reduce_mean(tf.nn.sampled_softmax_loss(
```

```
            weights=softmax_weights[wi],
            biases=softmax_biases[wi], inputs=embed,
            labels=train_labels[wi], num_sampled=num_sampled,
            num_classes=vocabulary_size))
        for wi in range(window_size*2)
    ]
)
```

Структурированный skip-gram устраняет важное ограничение стандартного алгоритма и учитывает положение контекстных слов во время обучения. Улучшение достигается за счет добавления отдельного набора весов softmax и смещения для каждой позиции контекста. За повышение производительности приходится расплачиваться увеличением объема памяти для хранения дополнительных наборов параметров. Далее мы рассмотрим аналогичное расширение модели CBOW.

Модель непрерывного окна

Модель непрерывного окна расширяет алгоритм CBOW аналогично алгоритму skip-gram. В исходном алгоритме CBOW представления, найденные для всех слов контекста, усредняются перед распространением через слой softmax. Однако в модели непрерывного окна вместо усреднения вложений они объединяются, что приводит к векторам представления длиной $m \times D_{emb}$, где D_{emb} – размер представления в исходном алгоритме CBOW. Рисунок 4.9 иллюстрирует модель непрерывного окна.

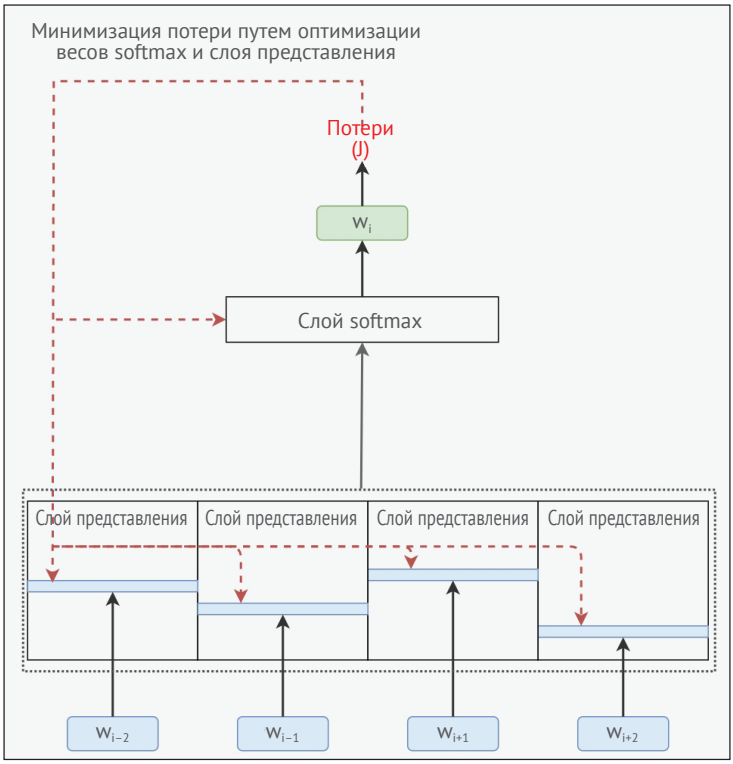


Рис. 4.9 ❖ Модель непрерывного окна

В этом разделе мы обсудили расширения алгоритмов skip-gram и CBOW. Оба варианта, по существу, используют положение слов в контексте вместо одинакового обращения со всеми словами. Далее мы обсудим совсем новый алгоритм изучения представлений слов под названием GloVe. Вы увидите, что GloVe преодолевает некоторые ограничения skip-gram и CBOW.

GloVe – представление на основе глобальных векторов

Методы изучения представлений слов делятся на две категории – методы на основе *факторизации глобальной матрицы* (global matrix factorization) и методы на основе локального контекста. *Латентно-семантический анализ* (latent semantic analysis, LSA) является примером метода, основанного на факторизации глобальной матрицы, а skip-gram и CBOW основаны на окне локального контекста. LSA используется как метод анализа документов, отображающий слова документа в так называемый *концепт* (concept) – общий шаблон слов, которые встречаются в документе. Методы, основанные на факторизации глобальной матрицы, эффективно используют *глобальную* статистику корпуса (например, совместное вхождение слов в *глобальную* область), но исследования показали, что они плохо решают задачи на поиск аналогии. С другой стороны, доказано, что методы локального контекста хорошо работают при поиске аналогий слов, но не используют глобальную статистику корпуса, оставляя место для улучшений. GloVe пытается извлечь лучшее из обоих миров – подход, который эффективно использует глобальную статистику корпуса, оптимизируя модель на основе локального контекста наподобие skip-gram или CBOW.

Знакомство с GloVe

Прежде чем смотреть на детали реализации GloVe, давайте потратим время, чтобы понять основную идею, лежащую в основе этого подхода. Для этого рассмотрим пример.

1. Рассмотрим слова i = *собака* и j = *кошка*.
2. Определим произвольное пробное слово k.
3. Определим P_{ik} – вероятность того, что слова i и k встречаются рядом, и P_{jk} – вероятность того же для слов j и k.

Теперь давайте посмотрим, как отношение P_{ik}/P_{jk} ведет себя с разными значениями k.

Слово k = *лаять* часто встречается вместе с i, таким образом, значение P_{ik} будет высоким. При этом k редко появляется вместе с j, образуя низкое значение P_{jk}. Поэтому мы получаем следующее выражение:

$$P_{ik}/P_{jk} \gg 1.$$

Наоборот, слово k = *мурлыкать* очень редко встречается рядом с i, но с большой вероятностью – рядом со словом j, поэтому значение P_{jk} будет высоким. Следовательно,

$$P_{ik}/P_{jk} \approx 0.$$

Слово k = *хвост* одинаково часто соседствует как с i, так и j. Слово k = *политика* одинаково редко встречается рядом с этими словами. В обоих случаях мы получаем следующее отношение:

$$P_{ik}/P_{jk} \approx 1.$$

Очевидно, что отношение P_{ik}/P_{jk}, вычисляемое путем измерения частоты появления двух слов близко друг к другу, является хорошим средством для измерения взаимосвязи между словами. Иными словами, этот подход можно использовать для изучения представлений слов. Следовательно, хорошей отправной точкой для определения функции потерь будет равенство

$$F(w_i, w_j, \tilde{w}_k) = P_{ik}/P_{jk},$$

где F – некоторая функция. С этого места оригинальная статья проходит через детальные математические рассуждения и приводит нас к следующей функции потерь:

$$J = \sum_{i,j=1}^{V} f(X_{ij})\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log(X_{ij})\right)^2.$$

Здесь функция $f(x) = (x/x_{\max})^{(3/4)}$, если $x < x_{\max}$, иначе $f(x) = 1$. X_{ij} – это частота, с которой слово j появляется в контексте слова i. Параметры w_i и b_i представляют, соответственно, представление слова и смещение для слова i, полученного из входных представлений. $\tilde{w}_j$ и $\tilde{b}_j$ представляют соответственно представление слова и смещение для слова j из выходных представлений. $x_{\max}$ – это установленный нами гиперпараметр. Оба этих представления ведут себя одинаково, за исключением рандомизации при инициализации. На этапе оценки эти представления складываются вместе, что приводит к повышению точности.

Реализация алгоритма GloVe

В этом разделе мы обсудим шаги по реализации GloVe. Полный код доступен в файле упражнения `ch4_glove.ipynb`, расположенном в папке `ch4`.

Сначала объявим входы и выходы:

```
train_dataset = tf.placeholder(tf.int32, shape=[batch_size],name='train_dataset')
train_labels = tf.placeholder(tf.int32, shape=[batch_size],name='train_labels')
```

Далее объявим слои представлений. У нас есть два разных слоя представлений: один для поиска входных слов, а другой для поиска выходных слов. Кроме того, мы определим представление смещения наподобие смещения, которое у нас было для слоя softmax:

```
in_embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size],
    -1.0, 1.0), name='embeddings')
in_bias_embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size],0.0,0.01,
    dtype=tf.float32), name='embeddings_bias')
```

```
out_embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size],
    -1.0, 1.0), name='embeddings')
out_bias_embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size],0.0,0.01,
    dtype=tf.float32), name='embeddings_bias')
```

Теперь объявим соответствующие представления для заданных входов и выходов (меток):

```
embed_in = tf.nn.embedding_lookup(in_embeddings, train_dataset)
embed_out = tf.nn.embedding_lookup(out_embeddings, train_labels)
embed_bias_in = tf.nn.embedding_lookup(in_bias_embeddings, train_dataset)
embed_bias_out = tf.nn.embedding_lookup(out_bias_embeddings, train_labels)
```

Заодно объявим заполнители для $f(X_{ij})$ (`weights_x`) и X_{ij} (`x_ij`) в функции стоимости:

```
weights_x = tf.placeholder(tf.float32, shape=[batch_size], name='weights_x')
x_ij = tf.placeholder(tf.float32, shape=[batch_size], name='x_ij')
```

Наконец, объявим функцию полной потери с предыдущими определенными объектами, которая выглядит следующим образом:

```
loss = tf.reduce_mean(
    weights_x * (tf.reduce_sum(embed_in*embed_out,axis=1) +
    embed_bias_in + embed_bias_out - tf.log(epsilon+x_ij))**2)
```

В этом разделе мы рассмотрели GloVe, еще один метод обучения представлению слов. Основное преимущество GloVe перед ранее описанными методами Word2vec заключается в том, что он уделяет внимание как глобальной, так и локальной статистике корпуса для изучения представлений. Поскольку GloVe способен собирать глобальную информацию о словах, он, как правило, дает лучшую точность, особенно когда размер корпуса увеличивается. Другое преимущество состоит в том, что, в отличие от методов Word2vec, GloVe не аппроксимирует функцию стоимости (как, например, Word2vec с использованием отрицательной выборки), но вычисляет истинную стоимость. Это приводит к лучшей и более простой оптимизации потерь.

Классификация документов с помощью Word2vec

Word2vec – это очень элегантный способ изучения числовых представлений слов. Мы удостоверились в этом, глядя на низкие значения потери и графическое представление t-SNE. Но само по себе изучение представлений слов еще не является решением прикладных задач NLP. Векторы слов используются в качестве цифрового представления входных данных для многих задач, таких как генерация подписей к изображениям и машинный перевод. Однако для решения этих задач объединяют различные подходы, например сверточные нейросети и модели с долгой краткосрочной памятью. О них пойдет речь в следующих главах. Чтобы лучше понять, каким образом представление слов применяется на практике, давайте перейдем к более простой задаче, классификации документов.

Классификация документов – это одна из самых популярных задач в NLP. Все, кому приходится обрабатывать огромные коллекции данных, например для новостных сайтов, издателей и университетов, остро нуждаются в автоматической классификации документов. Поэтому интересно взглянуть, как векторное представление слов может быть адаптировано к реальной задаче классификации посредством представления целых документов вместо слов.

Это упражнение доступно в файле `ch4_document_embedding.ipynb` в папке `ch4`.

Исходный набор данных

Для этой задачи мы будем использовать уже организованный набор текстовых файлов[1]. Это новостные статьи от BBC. Каждый документ в этой коллекции относится к одной из следующих категорий: *Business* (бизнес), *Entertainment* (развлечения), *Politics* (политика), *Sport* (спорт) или *Technology* (технологии). Мы используем 250 документов из каждой категории. Размер нашего словаря составит 25 000 слов.

Кроме того, каждый документ будет представлен тегом `<тип документа>-<id>` для последующей визуализации. Например, 50-й документ раздела *Entertainment* будет представлен как `entertainment-50`. Заметим, что это очень маленький набор данных по сравнению с большими текстовыми корпусами, которые анализируются в реальных приложениях. Тем не менее этого небольшого примера в данный момент достаточно, чтобы познакомиться с применением представлений слов.

Вот пара кратких фрагментов из фактических данных:

Business

Japan narrowly escapes recession

Japan's economy teetered on the brink of a technical recession in the three months to September, figures show.

Revised figures indicated growth of just 0.1 % - and a similar-sized contraction in the previous quarter. On an annual basis, the data suggests annual growth of just 0.2 %, ...

Technology

UK net users leading TV downloads

British TV viewers lead the trend of illegally downloading US shows from the net, according to research.

New episodes of 24, Desperate Housewives and Six Feet Under, appear on the web hours after they are shown in the US, said a report. Web tracking company Envisional said 18 % of downloaders were from within the UK and that downloads of TV programmers had increased by 150 % in the last year. ...

[1] При подготовке перевода не удалось найти в открытом доступе готовый обучающий набор текстов на русском языке. Поэтому воспользуемся англоязычными примерами из оригинала книги. – *Прим. перев.*

Классификация документов при помощи представлений слов

В целом вопрос заключается в том, чтобы выяснить, можно ли расширить методы представления слов, такие как skip-gram или CBOW, для классификации/кластеризации документов. В этом примере применяется алгоритм CBOW, поскольку было показано, что он работает с небольшими наборами данных лучше, чем skip-gram.

В общем виде порядок действий выглядит следующим образом:

1) извлечение данных из всех текстовых файлов и изучение представлений слов, как мы уже делали;
2) извлечение случайной выборки документов из уже изученного набора;
3) расширение изученных представлений на выбранные документы. Точнее, мы сформируем представление документа по среднему значению представлений, принадлежащих всем словам в документе;
4) визуализация найденных представлений документов с помощью техники визуализации t-SNE, чтобы увидеть, могут ли вложения слов быть полезными для кластеризации или классификации документов;
5) наконец, назначение метки для каждого документа при помощи алгоритма кластеризации K-среднее. Вы познакомитесь с кратким определением K-среднего, изучая реализацию.

Реализация – изучение представлений слов

Объявляем несколько заполнителей для обучающих данных, обучающих меток, истинных данных (применяемых для мониторинга представлений слов) и тестовых данных (применяемых для вычисления средних представлений тестовых документов):

```
# Входные данные.
train_dataset = tf.placeholder(tf.int32,
    shape=[batch_size, 2*window_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

test_labels = tf.placeholder(tf.int32,
    shape=[batch_size], name='test_dataset')
```

Далее определяем переменные для представлений весовых коэффициентов и смещений словаря и softmax (применяются для вычисления среднего представления тестовых документов):

```
# Переменные.
# embedding - это вектор для каждого слова в словаре.
embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
    embedding_size], -1.0, 1.0, dtype=tf.float32))
softmax_weights = tf.Variable(
    tf.truncated_normal([vocabulary_size, embedding_size],
    stddev=1.0 / math.sqrt(embedding_size), dtype=tf.float32))
softmax_biases = tf.Variable(
    tf.zeros([vocabulary_size], dtype=tf.float32))
```

Затем определяем функцию потерь softmax для отрицательных выборок по аналогии с предыдущими примерами:

```
loss = tf.reduce_mean(
    tf.nn.sampled_softmax_loss(weights=softmax_weights,
    biases=softmax_biases, inputs=mean_embeddings,
    labels=train_labels, num_sampled=num_sampled,
    num_classes=vocabulary_size))
```

Реализация – от представлений слов к представлениям документов

Чтобы получить хорошее представление документа из представлений слов, мы возьмем среднее представление всех слов, найденных в документе. Однако будем обрабатывать данные пакетами.

Таким образом, для каждого документа необходимо сделать следующее:

1) создать набор данных, в котором каждая точка данных является словом, принадлежащим документу;
2) для мини-пакета, отобранного из набора данных, найти средний вектор представления путем усреднения векторов представлений всех слов в мини-пакете;
3) выполнить обход документа по пакетам и вычислить общее представление документа путем усреднения представлений мини-пакетов.

Получаем среднее значение мини-пакета представлений:

```
mean_batch_embedding = tf.reduce_mean(tf.nn.embedding_
lookup(embeddings, test_labels), axis=0)
mean_embeddings = tf.reduce_mean(stacked_embeddings, 2,
keepdims=False)
```

Затем собираем эти средние вложения в список для всех пакетов в документе и получим среднее представление полного документа. Это очень простой метод получения представлений документов, но в то же время очень мощный, как вы увидите в ближайшее время.

Кластеризация документов и визуализация представлений

На рис. 4.10 визуализирован пример представлений документов, изученных алгоритмом CBOW. Как видите, алгоритм достаточно хорошо научился кластеризовать документы с одинаковой темой. Мы использовали префикс документов (разные цвета для разных категорий документов) для добавления цвета к точкам данных, чтобы разделение было более очевидным. Как мы уже говорили выше, этот простой метод оказался очень эффективным способом классификации/кластеризации документов.

Проверка некоторых выбросов

На рис. 4.10 видно, что некоторые документы являются выбросами, например `tech-42` и `sport-50`. Интересно взглянуть на содержание этих документов, чтобы расследовать возможные причины такого поведения.

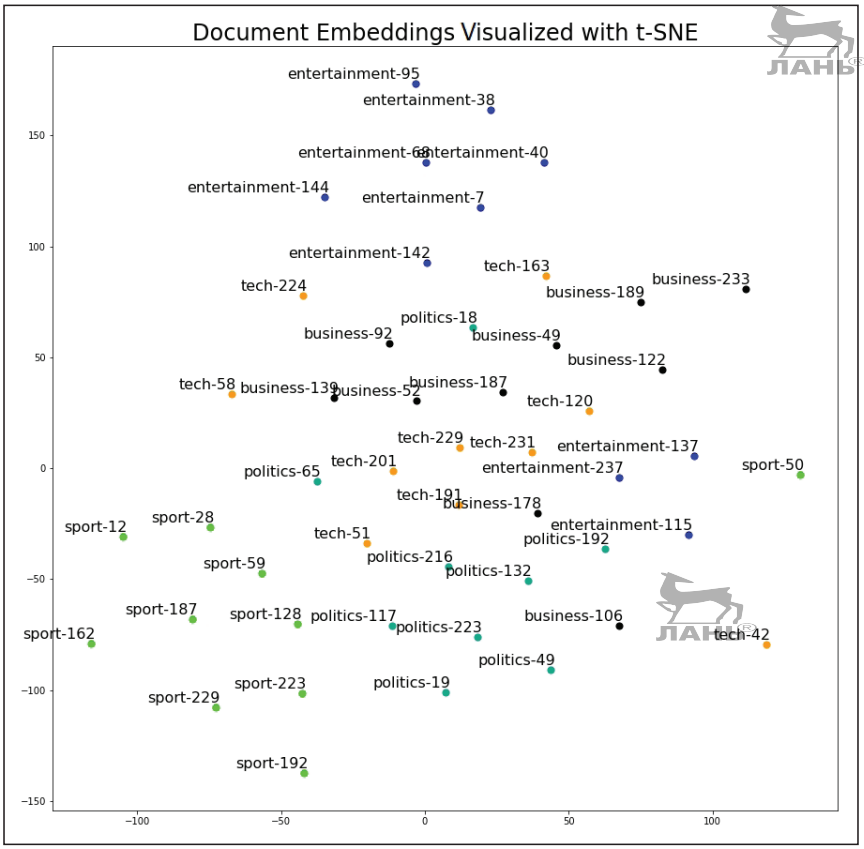


Рис. 4.10 ❖ Визуализация представлений документов при помощи t-SNE

Ниже приведен фрагмент документа `tech-42`:

Tech-42

Hotspot users gain free net calls

People using wireless net hotspots will soon be able to make free phone calls as well as surf the net.

Users of the system can also make calls to landlines and mobiles for a fee. The system is gaining in popularity and now has 28 million users around the world. Its paid service - dubbed Skype Out - has so far attracted 940,000 users. ...

(Пользователи точек доступа получили возможность бесплатных звонков по сети

Благодаря беспроводным точкам доступа люди скоро смогут совершать бесплатные телефонные звонки столь же просто, как посещать сайты.

Пользователи этой системы так же могут совершать платные звонки на обычные проводные или сотовые телефоны. Система набирает популярность

и сегодня объединяет 28 млн пользователей по всему миру. Эта платная услуга – именуемая Skype Out – на сегодняшний день привлекла 940 000 пользователей.)

Этот документ хоть и относится к группе «Технологии», однако написан таким образом, чтобы подчеркнуть ценность Skype для людей, а не углубляться в технические детали системы. Это, в свою очередь, может привести к тому, что документ окажется ближе к темам, более связанным с людьми, таким как «Развлечения» или «Политика».

Теперь рассмотрим фрагмент документа `sport-50`:

Sport-50

IAAF awaits Greek pair's response

Kostas Kenteris and Katerina Thanou are yet to respond to doping charges from the International Association of Athletics Federations (IAAF).

The Greek pair were charged after missing a series of routine drugs tests in Tel Aviv, Chicago and Athens. They have until midnight on 16 December and an IAAF spokesman said: "We're sure their responses are on their way." If they do not respond or their explanations are rejected, they will be provisionally banned from competition. They will then face a hearing in front of the Greek Federation...

(*ИААФ ждет ответа греческой пары*

Костас Кентерис и Катерина Таноу еще не сдали допинговые пробы Международной ассоциации федераций легкой атлетики (ИААФ).

Греческую пару обвинили в том, что они пропустили серию регулярных тестов на допинг в Тель-Авиве, Чикаго и Афинах. У них есть время до полуночи 16 декабря, и представитель ИААФ заявил: «Мы уверены, что их объяснения скоро поступят». Если они не ответят или их объяснения будут отклонены, они будут временно отстранены от участия в соревнованиях. Затем они предстанут перед Греческой Федерацией...)

Мы можем пролить некоторый свет на то, почему документ `sport-50` оказался далеко от иных статей, связанных со спортом. Давайте внимательно посмотрим на другой документ, близкий к `sport-50`, то есть `entertainment-115`:

Entertainment-115

Rapper Snoop Dogg sued for 'rape'

US rapper Snoop Dogg has been sued for $25m (≤13m) by a make-up artist who claimed he and his entourage drugged and raped her two years ago.

The woman said she was assaulted after a recording of the Jimmy Kimmel Live TV show on the ABC network in 2003. The rapper's spokesman said the allegations were "untrue" and the woman was "misusing the legal system as a means of extracting financial gain". ABC said the claims had "no merit". The star has not been charged by police.

(*На рэпера Снуп Догга подали иск за «изнасилование»*

Американский рэпер Снуп Догг получил иск на 25 млн долларов (13 млн фунтов) от визажистки, которая утверждала, что рэпер и его помощники накачали наркотиками и изнасиловали ее два года назад.

Женщина заявила, что на нее напали после записи шоу Джимми Киммела в прямом эфире в сети ABC в 2003 году. Представитель рэпера заявил, что обвинения «не соответствуют действительности», а женщина «злоупотребляет правовой системой как средством извлечения финансовой выгоды». ABC заявила, что претензии не имеют «никаких оснований». Полиция не предъявила звезде обвинения.)

Судя по всему, документы в этом кластере связаны с различными обвинениями в уголовных или административных правонарушениях, а не со спортом или развлечениями. Следовательно, эти документы вполне обоснованно оказались вдалеке от типичных документов, касающихся спорта или развлечений.

Кластеризация/классификация документов с K-средним

До сих пор нам удавалось визуально проверять кластеры документов. Однако этого недостаточно, потому что если у нас есть еще тысяча документов, которые мы хотели бы объединить в кластеры, нам придется визуально проверять расположение тысячи точек. Поэтому нам нужны более автоматизированные способы для проверки распределения документов.

Для кластеризации документов можно воспользоваться методом *K-средних*. Это простой, но мощный метод, используемый для разбиения данных на группы (кластеры) на основе сходства данных, так что схожие данные окажутся в одной группе, а разные данные – в разных группах. Метод K-средних работает следующим образом.

1. Определите K, т. е. количество кластеров, которые будут сформированы. В данном случае зададим значение 5, поскольку известно, что есть пять категорий.
2. Сформируйте K случайных центроидов, которые являются центрами кластеров.
3. Затем ассоциируйте каждую точку данных с ближайшим центроидом кластера.
4. После присвоения всех точек данных для некоторого кластера пересчитайте центроид кластера (то есть среднее значение точек данных).
5. Продолжайте в том же духе, пока смещение центроида после пересчета не станет меньше некоторого порога.

Для реализации алгоритма K-средних мы будем использовать библиотеку scikit-learn. В коде это выглядит следующим образом:

```
kmeans = KMeans(n_clusters=5, random_state=43643, max_iter=10000, n_init=100,
algorithm='elkan')
```

Самый важный гиперпараметр – это `n_clusters`, то есть количество кластеров, которые мы хотим сформировать. Вы можете поиграть с другими гиперпарамет-

рами, чтобы увидеть, какое влияние они оказывают на производительность. Описание гиперпараметров доступно по адресу: **http://scikit-learn.org/stable/modules/Generated/sklearn.cluster.KMeans.html**.

Теперь мы можем классифицировать документы, которые использовали для обучения (или любые другие документы), по классам. Мы получим следующие классы:

Метка	Документы
0	'entertainment-207', 'entertainment-14', 'entertainment-232', 'entertainment-49', 'entertainment-191', 'entertainment-243', 'entertainment-240'
1	'sport-145', 'sport-228', 'sport-141', 'sport-249'
2	'sport-4', 'sport-43', 'entertainment-54', 'politics-214', 'politics-12', 'politics-165', 'sport-42', 'politics-203', 'politics-87', 'sport-33', 'politics-81', 'politics-247', 'entertainment-245', 'entertainment-22', 'tech-102', 'sport-50', 'politics-33', 'politics-28'
3	'business-220', 'business-208', 'business-51', 'business-30', 'business-130', 'business-190', 'business-34', 'business-206'
4	'business-185', 'business-238', 'tech-105', 'tech-99', 'tech-239', 'tech-227', 'tech-31', 'tech-131', 'tech-118', 'politics-10', 'tech-150', 'tech-165'

Результат не идеален, но в целом документы, относящиеся к разным категориям, достаточно точно привязаны к различным меткам. Мы видим, что документы, относящиеся к развлечениям, имеют метку 0, документы, относящиеся к спорту, – 1, документы, относящиеся к бизнесу, – 3 и т. д.

В этом разделе вы узнали, как расширить представление слов для классификации/кластеризации документов. Сначала уже известным способом изучают представления слов. Затем создают представление документа, усредняя представления всех слов, найденных в этом документе. В примере из этого раздела векторные представления документов использовались для кластеризации/классификации набора новостных статей BBC, условно разделенных на следующие категории: развлечения, технологии, политика, бизнес и спорт. После кластеризации документов мы увидели, что они достаточно разумно сгруппированы, т. е. документы, принадлежащие к одной категории, расположены близко друг к другу. Однако обнаружились и выбросы в виде посторонних документов. Но, проанализировав текстовое содержание этих документов, мы увидели, что существуют веские причины, по которым эти документы ведут себя именно таким образом.

Заключение

В этой главе мы исследовали различия между алгоритмами skip-gram и CBOW. Для визуального сравнения применялась популярная технология двумерной визуализации t-SNE. Заодно вы изучили теоретические основы этого метода.

Далее вы познакомились с несколькими расширениями алгоритмов Word2vec, повышающими производительность, а затем с новыми алгоритмами, основанными на алгоритмах skip-gram и CBOW. Структурированный skip-gram расширяет исходный алгоритм, сохраняя положение слов контекста во время оптимизации,

и позволяет алгоритму обрабатывать ввод-вывод в зависимости от расстояния между словами. То же самое расширение можно применить к алгоритму CBOW, и получится алгоритм непрерывного окна.

Затем вы изучили GloVe – еще один метод изучения представлений слов. GloVe продвигает текущие алгоритмы Word2vec на шаг вперед за счет использования глобальной статистики документа. Наконец, вы рассмотрели пример реального использования векторных представлений слов – кластеризацию/классификацию документов. Вы убедились, что представления слов очень эффективны и позволяют достаточно хорошо объединять документы по содержанию.

В следующей главе мы перейдем к обсуждению другого семейства глубоких сетей, которые более сильны в использовании пространственной структуры данных, – сверточных нейронных сетей. Точнее, вы увидите, как сверточные нейросети используют пространственную структуру предложений, чтобы отнести их к различным классам.

Глава 5

Классификация предложений с помощью сверточных нейронных сетей

В этой главе мы обсудим *сверточные нейронные сети* (convolutional neural network, CNN). Они существенно отличаются от полносвязных нейронных сетей и достигли высокого уровня точности в многочисленных задачах. Эти задачи включают классификацию изображений, обнаружение объектов, распознавание речи и, конечно, классификацию предложений. Одно из главных преимуществ CNN заключается в том, что по сравнению с полностью связанным слоем сверточный слой в CNN имеет гораздо меньшее количество параметров. Это позволяет нам строить более глубокие модели, не беспокоясь о переполнении памяти. Кроме того, более глубокие модели обычно лучше работают.

В этой главе вы познакомитесь со сверточными нейросетями, подробно изучите их компоненты и разберетесь в отличиях CNN от их полносвязных аналогов. Затем вы познакомитесь с различными операциями, такими как свертка и объединение, а также с некоторыми гиперпараметрами этих операций. Попутно вы изучите математические основы вычислительных операций в сверточных сетях. Разобравшись с теоретическими основами, вы перейдете к практическим упражнениям по реализации CNN с TensorFlow. Сначала вы создадите CNN для классификации объектов, а затем примените CNN для классификации предложений.

Знакомство со сверточными нейронными сетями

В этом разделе вы познакомитесь с основными понятиями в области CNN. В частности, получите представление о видах операций, присутствующих в CNN, таких как слои свертки, объединяющие слои и полностью связанные слои. Далее вы узнаете, как эти слои связаны между собой, образуя сквозную модель. Наконец, ознакомитесь с математическими определениями операций и узнаете, как различные гиперпараметры, связанные с этими операциями, влияют на результат.

Основы CNN

Давайте рассмотрим основную идею CNN, не вдаваясь пока в технические детали. В сущности, CNN – это стек слоев трех типов, таких как сверточные слои, объединяющие слои и полностью связанные слои. У каждого типа слоев своя специфическая работа.

Входные данные располагаются в *сверточном слое* (convolution layer). По этому слою двумерных входных данных скользит небольшая матрица весовых коэффициентов – иногда ее называют *сверточным фильтром* (convolution filter), или *окном*. Между весами матрицы и теми элементами данных, которые «накрывает» матрица, выполняется операция поэлементного перемножения и суммирования. Результатом операции является одиночное значение, т. е. пиксель. Иными словами, в отличие от полносвязных нейронных сетей, в сверточной сети применяется небольшое количество весов, организованных так, чтобы покрывать только маленький участок ввода во входном слое, и размерность матрицы весов обычно кратна определенным измерениям (например, ширине и высоте изображения). Пример применения простейшей операции свертки к изображению показан на рис. 5.1. В данном случае матрица свертки работает как своего рода фильтр. Если блок изображения, над которым расположена матрица, имеет высокую схожесть с матрицей, свертка выдаст высокое значение для этого местоположения; если нет, будет выведено низкое значение. Выполняя свертку полного изображения, мы получаем выходную матрицу, указывающую, присутствует ли заданный шаблон в определенном месте изображения.

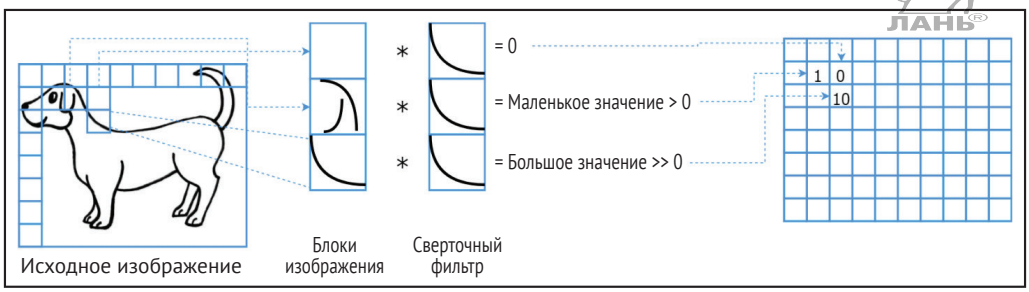


Рис. 5.1 ❖ Простейшая операция свертки изображения

Кроме того, сверточные слои могут чередоваться с *объединяющими слоями* (pooling layer) и *слоями подвыборки* (subsampling layer), уменьшающими размерность входных данных. Уменьшая размерность, мы делаем сеть более инвариантной, а также заставляем CNN учиться с меньшим количеством информации, что приводит к лучшему обобщению и регуляризации модели. Размерность уменьшается за счет разделения поверхности входных данных на участки – *патчи* (patch) и преобразования каждого патча в один элемент. Наиболее популярные преобразования включают в себя выбор максимального элемента патча (`max pool`) или усреднение всех значений в патче (`average`). На рис. 5.2 показан пример того, как операция объединения делает выход CNN инвариантным.

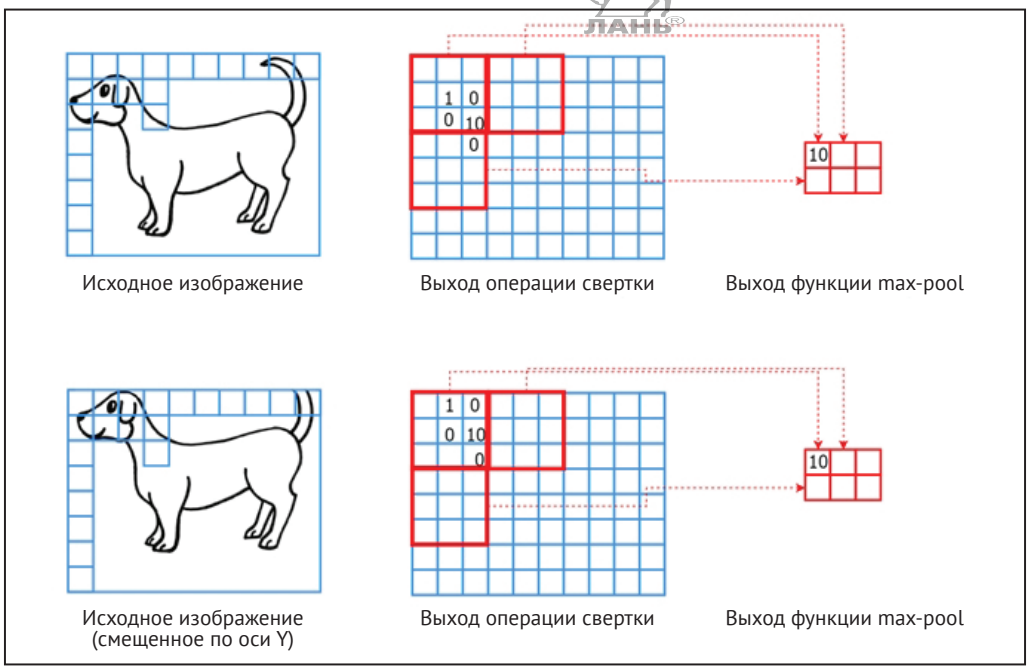


Рис. 5.2 ❖ Как операция объединения помогает сделать обработку данных инвариантной

У нас есть исходное изображение и изображение, слегка смещенное вверх по оси Y. Для этих изображений получены соответствующие выходы свертки, и вы можете видеть, что значение 10 появляется в немного разных местах матрицы. Однако, используя операцию объединения по максимуму (max pooling), которая находит максимальное значение внутри каждой выборки, в итоге мы получаем одинаковый результат. Мы подробно обсудим эту операцию позже.

Наконец, сверточные данные передаются на набор полностью связанных слоев, с выхода которых данные поступают на окончательный уровень классификации/регрессии, например для классификации предложений или изображений. Полностью связанные слои содержат значительную долю общего количества весов CNN, поскольку сами сверточные слои обходятся небольшим количеством весов. Однако обнаружено, что CNN лучше работают с полностью связанными слоями, чем без них. Это может быть связано с тем, что сверточные слои, обладающие весовой матрицей небольшого размера, изучают локализованные объекты, тогда как полностью связанные слои обеспечивают общее представление о том, как эти локализованные объекты соединяются вместе для получения желаемого конечного результата. На рис. 5.3 изображена структура типичной CNN, применяемой для классификации изображений.

Как следует из схемы, CNN сохраняют пространственную структуру входов во время обучения. Другими словами, для двумерного ввода большинство слоев CNN будут тоже двумерными, так как полностью связанные слои располагаются лишь поблизости от выхода. Сохранение пространственной структуры позволяет CNN использовать ценную пространственную информацию входов и изучать входы

с меньшим количеством параметров. Значение пространственной информации показано на рис. 5.4.

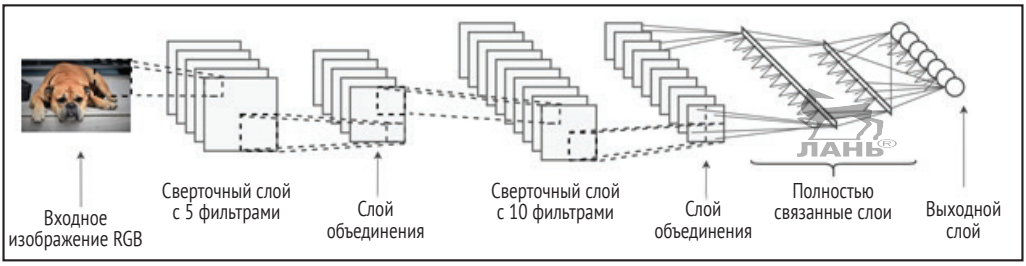


Рис. 5.3 ❖ Обычная архитектура сверточной нейросети

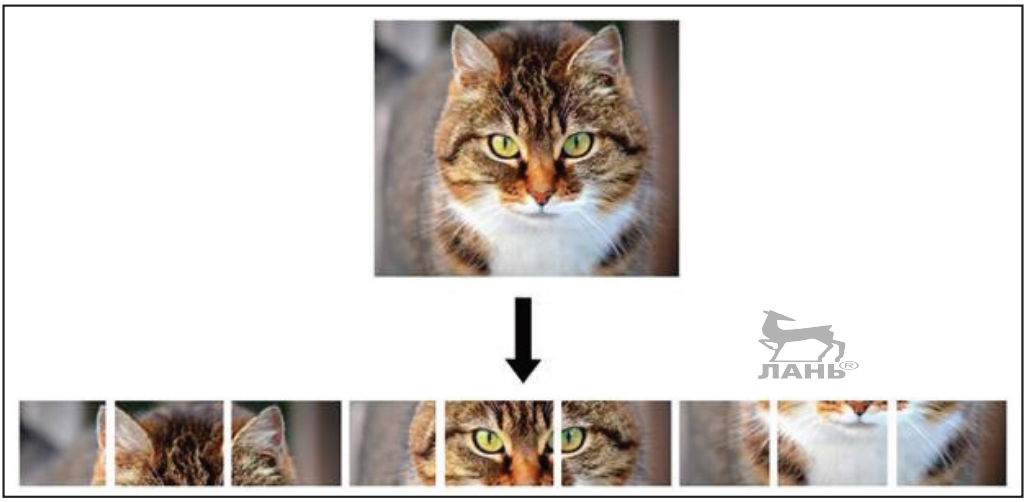


Рис. 5.4 ❖ Развертывание изображения в одномерный вектор приводит к потере важной пространственной информации

Когда двумерное изображение кошки на рис. 5.4 разворачивается в одномерный вектор, уши теряют близость к глазам, да и нос тоже оказывается далеко от глаз. Это означает, что во время развертывания мы разрушили полезную пространственную информацию.

Возможности сверточных нейросетей

CNN – это очень универсальное семейство моделей, обладающих замечательной точностью при решении разнообразных задач. Такая универсальность объясняется способностью CNN выполнять извлечение признаков и обучение одновременно, что обеспечивает высокую эффективность и обобщаемость.

В конкурсе ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2016, который включал классификацию изображений, обнаружение объектов и локализацию объектов в изображении, CNN продемонстрировали невероятно точные

результаты. Например, в задаче классификации изображений удалось достичь точности около 98 % для 1000 объектов различных классов. Это означает, что сверточная нейросеть смогла правильно идентифицировать около 980 различных объектов из 1000.

CNN также с успехом применяют для сегментации изображения, т. е. выделения различных объектов на изображении. Например, в изображении городской улицы, которое включает здания, дорогу, транспортные средства и пассажиров, разделение дороги и зданий является задачей сегментации. Кроме того, сверточные нейросети добились невероятной эффективности в таких задачах NLP, как классификация предложений, генерация текста и машинный перевод.

Устройство сверточных нейросетей

Давайте немного углубимся в технические детали устройства CNN. Прежде всего рассмотрим операцию свертки и введем некоторую терминологию, такую как *размер фильтра*, *шаг* (stride) и *отступ* (padding). Вкратце, размер фильтра означает размер окна операции свертки, шаг – расстояние между двумя положениями окна свертки, а отступ – способ обработки границ входных данных. Затем вы познакомитесь с операциями обратной (транспонированной) свертки и объединения. Наконец, узнаете, как соединить полностью связанные слои и двумерные выходные данные, создаваемые слоями свертки/объединения, и научитесь использовать выходные данные для классификации или регрессии.

Операция свертки

В этом разделе мы подробно обсудим операцию свертки. Сначала рассмотрим стандартную операцию свертки без шага и заполнения, а затем операции *свертки с шагом* и *свертки с заполнением*. Наконец, мы рассмотрим так называемую обратную, или транспонированную, свертку. Для всех операций в этой главе применяется индекс, начинающийся с единицы, а не с нуля.

Стандартная операция свертки

Операция свертки является центральной частью CNN. Пусть у нас есть вход размером $n \times n$ и матрица весов (фильтр) $m \times m$, где $n \geqslant m$. Операция свертки перемещает фильтр по входному пространству. Обозначим вход буквой X, матрицу весов – W и выход – H. Кроме того, в каждой позиции i, j выход рассчитывается следующим образом:

$$h_{i,j} = \sum_{k=1}^{m} \sum_{l=1}^{m} w_{k,l} x_{i+k-1, j+l-1},$$

где $1 \leqslant i, j \leqslant n - m + 1$.

Здесь $x_{i,j}$, $w_{i,j}$ и $h_{i,j}$ обозначают значение в (i, j)-й точке X, W и H соответственно. Как следует из уравнения, хотя вход имеет размерность $n \times n$, размерность выхода в этом случае будет $(n - m + 1) \times (n - m + 1)$. Параметр m известен как *размер ядра*.

Пример стандартной операции свертки визуально представлен на рис. 5.5.

Выходные данные, полученные в результате операции свертки (верхний прямоугольник на рис. 5.5), иногда называют *картой признаков* (feature map).

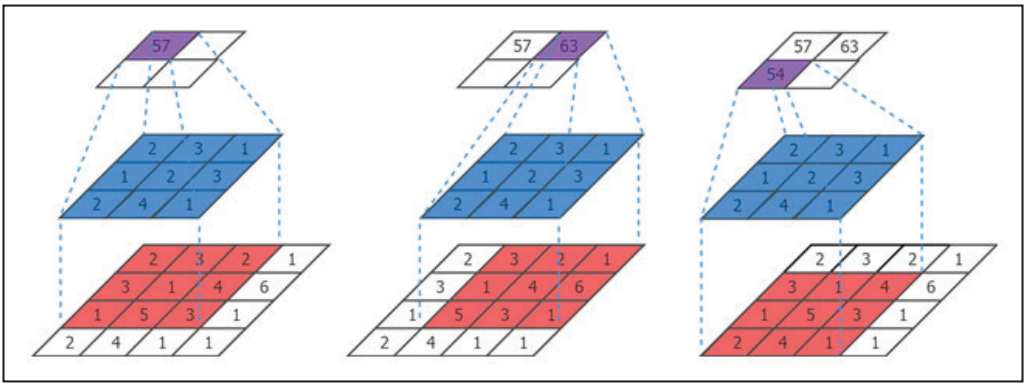


Рис. 5.5 ❖ Операция свертки с размером ядра m = 3, шагом 1 и без отступа

Свертка с шагом

В предыдущем примере мы смещали фильтр на одну позицию. Однако это не обязательно; мы можем использовать и большее смещение. Величина смещения обозначается термином *шаг*. Давайте изменим предыдущее уравнение, добавив в него шаги s_i и s_j:

$$h_{i,j} = \sum_{k=1}^{m}\sum_{l=1}^{m} w_{k,l} x_{(i-1)\times s_i+k,(j-1)\times s_j+l},$$

где $1 \leqslant i \leqslant floor[(n-m)/s_i] + 1$ и $floor[(n-m)/s_j] + 1$.

В этом случае размер выхода будет уменьшаться по мере увеличения шага s_i и s_j. Сравнение рис. 5.5 (шаг = 1) и рис. 5.6 (шаг = 2) иллюстрирует влияние различных шагов.

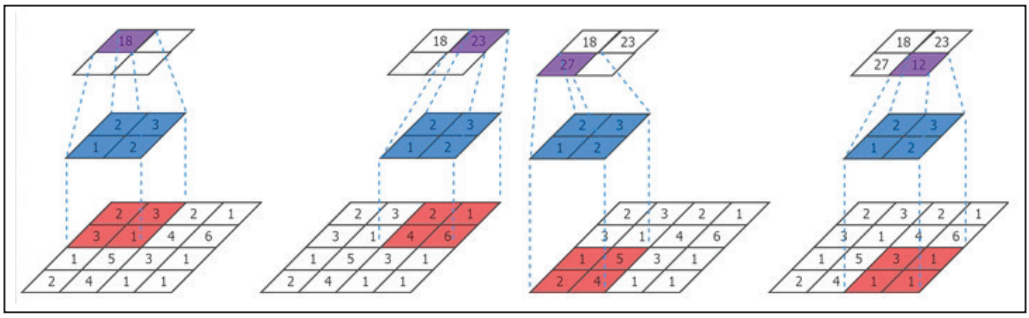


Рис. 5.6 ❖ Операция свертки с размером фильтра m = 2, шаг = 2 и без отступа

Как видите, выполнение свертки с шагом помогает уменьшить размерность входа, аналогично объединяющему слою. Поэтому иногда вместо объединения в CNN используется свертка с шагом, поскольку это уменьшает вычислительную сложность.

Свертка с заполнением

Если фильтр не может выходить за края области данных, то даже при единичном шаге размерность выхода неизбежно уменьшается. Это нежелательное свойство, сильно ограничивающее количество слоев нейросети. Ведь доказано, что глубокие сети работают лучше. Но не путайте естественное снижение размерности со снижением вследствие увеличенного шага. Вы всегда можете отказаться от увеличенного шага, просто задав параметр s =1. Для борьбы с естественной убылью размерности применяют так называемое *заполнение*, т. е. добавление нулей вокруг границы области входа, чтобы размерность выхода совпадала с размерностью входа. Давайте предположим шаг 1:

$$h_{i,j} = \sum_{k=1}^{m}\sum_{l=1}^{m} w_{k,l} x_{i+k-(m-1),j+l-(m-1)},$$

где $x_{i,j} = 0$, если $i, j < 1$ или $i, j > n$.

На рис. 5.7 показано, как благодаря заполнению сохраняется размерность выхода.

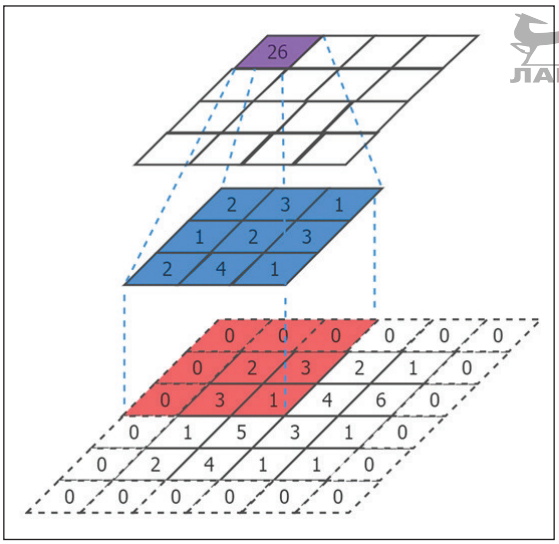


Рис. 5.7 ❖ Операция свертки с размером фильтра m = 3, шагом s = 1 и заполнением нулями (zero-padding)

Транспонированная свертка

Хотя операция свертки выглядит сложной с точки зрения математики, ее можно упростить до умножения матриц. По этой причине мы можем определить операцию транспонированной свертки, или *деконволюции* (deconvolution), как ее иногда называют. Однако будем использовать термин *транспонированная свертка* (transposed convolution), поскольку он звучит более естественно. Кроме того, строго говоря, деконволюция относится к другой математической концепции. Операция транспонированной свертки играет важную роль в CNN для сохранения градиентов во время обратного распространения. Давайте рассмотрим пример.

Пусть у нас есть вход размером $n \times n$ и матрица весов, или фильтр, $m \times m$, где $n \geqslant m$, и операция свертки перемещает матрицу весов по входу. Обозначим вход буквой X, матрицу весов – W, а выход – H. Выход h можно рассчитать как матричное умножение следующим образом.

Давайте для ясности примем $n = 4$ и $m = 3$ и развернем вход X слева направо и сверху вниз, что приведет к следующему представлению:

$$x^{(16,1)} = x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}, x_{2,1}, x_{2,2}, x_{2,3}, x_{2,4}, \ldots, x_{4,1}, x_{4,2}, x_{4,3}, x_{4,4}.$$

Далее определим новую матрицу A из W:

$$A^{(4,16)} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix}.$$

Выполнив перемножение матриц, мы получаем H:

$$H^{(4,1)} = A^{(4,16)} X^{(16,1)}.$$

Изменив выход $H^{(4,1)}$ на $H^{(2,2)}$, мы получим свернутый результат. Теперь давайте спроецируем этот результат обратно на n и m.

Развертывая вход $X^{(n,n)}$ в $X^{(n^2,1)}$ и создавая матрицу $A^{((n-m+1)^2,n^2)}$ из w, как показано выше, мы получаем $H^{((n-m+1)^2,1)}$, который затем преобразуем в $H^{(n-m+1,n-m+1)}$.

Наконец, чтобы получить транспонированную свертку, мы просто транспонируем A и получаем:

$$\hat{X}^{(n^2,1)} = (A^T)^{(n^2,(n-m+1)^2)} H^{((n-m+1)^2,1)},$$

где $\hat{X}$ – результат операции транспонирования свертки.

На этом мы заканчиваем обсуждение операции свертки. Мы обсудили стандартную операцию свертки, операцию свертки с шагом, операцию свертки с отступом и вычисление транспонированной свертки. Далее более подробно обсудим операцию объединения.

Операция субдискретизации

Операция *субдискретизации* была введена в CNN главным образом для уменьшения размера промежуточных выходов, а также для того, чтобы сделать преобразование CNN инвариантным. Эта операция предпочтительнее естественного уменьшения размерности, вызванного сверткой без отступа, так как мы можем решить, где уменьшить размер вывода, с помощью объединения, в отличие от того, чтобы это происходило каждый раз помимо нашей воли. Как мы уже говорили, принудительное уменьшение размерности при свертке без отступа будет строго ограничивать количество слоев нейросети.

Мы определим операцию объединения математически в следующих разделах. Точнее, мы обсудим два типа объединения: максимальное объединение и среднее объединение. Сначала, однако, мы определим обозначения. Для входа размера

$n \times n$ и ядра (аналогично фильтру слоя свертки) размера $m \times m$, где $n \geqslant m$, операция свертки перемещает набор весов по входу. Обозначим ввод x, патч весов w и вывод h.

Операция max-pooling

Операция max-pooling[1] выбирает максимальный элемент в пределах определенного ядра ввода для получения вывода. По сути, max-pooling – это окна над входом (синие квадраты 3×3 на рис. 5.8), в которых каждый раз выбирают максимальное значение. Математически операцию max-pooling определяют следующим уравнением:

$$h_{i,j} = \max(\{x_{i,j}, x_{i,j+1}, \ldots, x_{i,j+m-1}, x_{i+1,j}, \ldots, x_{i+1,j+m-1}, \ldots, x_{i+m-1,j}, \ldots, x_{i+m-1,j+m-1}\}),$$

где $1 \leqslant i, j \leqslant n - m + 1$.

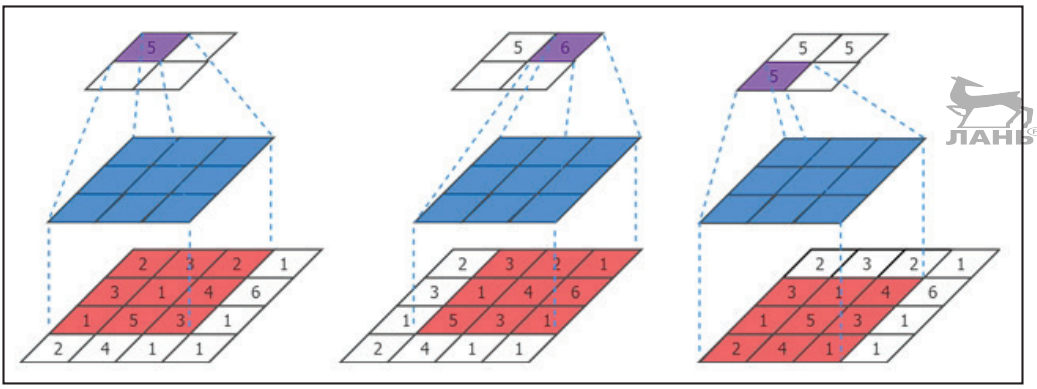


Рис. 5.8 ❖ Операция max-pooling с размером фильтра 3, шагом 1 и без заполнения

Операция max-pooling с шагом

Операция max-pooling с шагом похожа на свертку с шагом. Вот ее уравнение:

$$h_{i,j} = \max\left(\left\{\begin{array}{c} x_{(i-1)\times s_i+1,(j-1)\times s_j+1}, x_{(i-1)\times s_i+1,(j-1)\times s_j+2}, \ldots, \\ x_{(i-1)\times s_i+1,(j-1)\times s_j+m}, x_{(i-1)\times s_i+2,(j-1)\times s_j+1}, \ldots, \\ x_{(i-1)\times s_i+2,(j-1)\times s_j+m}, \ldots, x_{(i-1)\times s_i+m,(j-1)\times s_j+1}, \ldots, \\ x_{(i-1)\times s_i+m,(j-1)\times s_j+m} \end{array}\right\}\right),$$

где $1 \leqslant i \leqslant floor[(n - m)/s_i] + 1$ и $1 \leqslant j \leqslant floor[(n - m)/s_i] + 1$.

Результат изображен на рис. 5.9.

[1] На русский язык обычно не переводится и применяется как имя собственное. – *Прим. перев.*

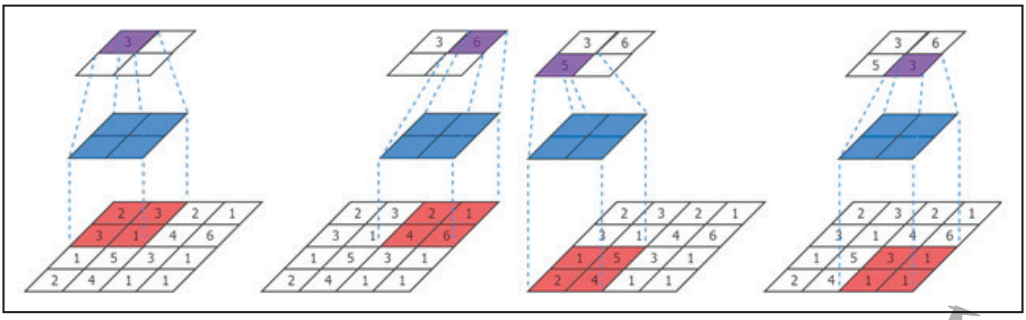


Рис. 5.9 ❖ Операция max-pooling
для входа $n = 4$, фильтра $m = 2$, шага $s = 2$ и без заполнения

Операция average pooling

Операция average pooling работает аналогично max-pooling, за исключением того, что вместо максимума берется среднее значение всех входных данных, попадающих в ядро. Рассмотрим следующее уравнение:

$$h_{i,j} = \frac{x_{i,j}, x_{i,j+1}, \ldots, x_{i,j+m-1}, x_{i+1,j}, \ldots, x_{i+1,j+m-1}, \ldots, x_{i+m-1,j}, \ldots, x_{i+m-1,j+m-1}}{m \times m},$$

$$\forall i \geq 1,\ j \leq n - m + 1.$$

Операция average pooling изображена на рис. 5.10.

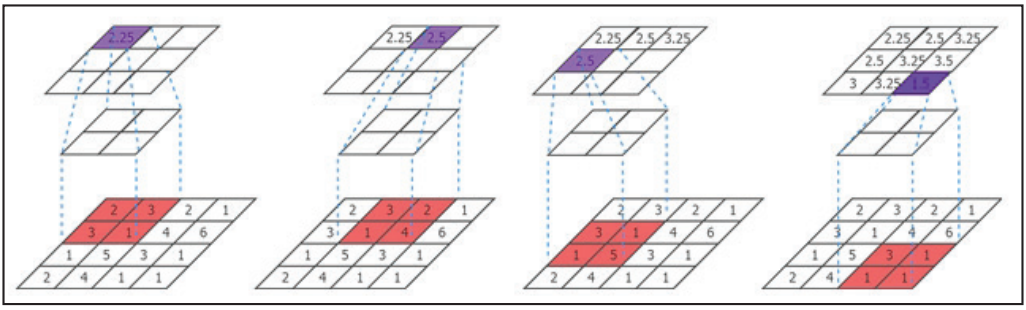


Рис. 5.10 ❖ Операция average pooling
для входа $n = 4$, фильтра $m = 2$, шага $s = 1$ и без заполнения

Полностью связанные слои

Полностью связанные слои (fully connected layers) – это полностью связанный набор весов на протяжении от входа до выхода. Составленная из таких слоев полносвязная сеть способна изучать глобальную структуру данных, поскольку информация распространяется от каждого входа к каждому выходу. Кроме того, наличие полностью связанных слоев позволяет нам в глобальном масштабе комбинировать признаки, извлеченные предшествующими малоразмерными слоями свертки, и благодаря этому достигать поразительных результатов.

Определим размерность выхода последнего сверточного или объединяющего слоя $p \times o \times d$, где p – высота входа, o – ширина, а d – глубина. В качестве примера рассмотрим изображение RGB, которое будет иметь фиксированную высоту, фиксированную ширину и глубину 3 (один канал глубины для каждого компонента RGB).

Тогда для начального полностью связанного слоя, расположенного сразу за последним сверточным или объединяющим слоем, матрица весов будет $w^{(m,p \times o \times d)}$, где высота, ширина и глубина выхода слоя – это количество единиц вывода, произведенных этим последним слоем, а m – количество скрытых единиц в полностью связанном слое. Затем, во время вывода (или прогнозирования), мы приводим размерность выхода последнего сверточного/объединяющего слоя к виду ($p \times o \times d$, 1) и выполняем следующее перемножение матриц, чтобы получить h:

$$h^{(m \times 1)} = w^{(m,p \times o \times d)} x^{(p \times o \times d,1)}.$$

Получающиеся полностью связанные слои будут вести себя как в обычной полносвязной нейронной сети, где у вас есть несколько полностью связанных слоев и выходной слой. Выходной слой может быть классификатором softmax для задачи классификации или линейным слоем для задачи регрессии.

Собираем CNN из компонентов

Теперь мы обсудим, как сверточный, объединяющий и полностью связанные слои собираются воедино, чтобы сформировать CNN.

Как показано на рис. 5.11, слои свертки, объединения и полностью связанные слои формируют сквозную обучаемую модель, которая принимает необработанные данные – в общем случае многомерные, например изображения RGB, – и выдает значимый вывод, например класс объекта. Первым делом сверточные слои изучают пространственные особенности изображений. Нижние сверточные слои изучают низкоуровневые элементы, такие как по-разному ориентированные грани, присутствующие на изображениях, а более высокие уровни изучают более высокоуровневые элементы, такие как формы, образованные гранями (например, круги и треугольники), или более крупные части объекта (например, морда со-

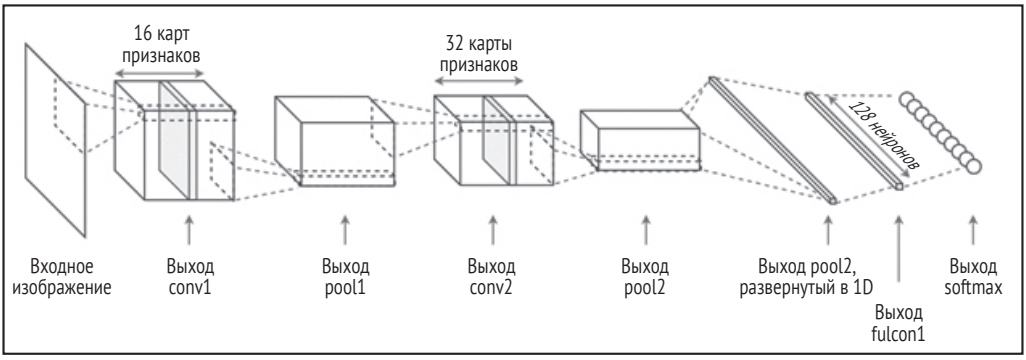


Рис. 5.11 ❖ Комбинация слоев свертки, объединяющих слоев и полностью связанных слоев, составляющих CNN

баки, хвост собаки или капот автомобиля). Промежуточные объединяющие слои повышают инвариантность выхода. Это означает, что если в новом изображении изученный признак появится в другом месте, CNN все равно обнаружит наличие этого признака. Наконец, полностью связанные слои объединяют изученные признаки высокого уровня в глобальные представления, необходимые конечному слою для классификации объекта.

Упражнение – классификация изображений из набора MNIST

Это наш первый пример использования CNN для решения реальной задачи машинного обучения. Мы начнем с классификации изображений. Причина, по которой не следует начинать с задачи NLP, заключается в том, что применение CNN к задачам обработки естественного языка не столь простое и основано на некоторых хитростях. Тем более изначально CNN были созданы для работы с изображениями. Давайте и мы начнем с изображений, а затем разберемся, как CNN применяются к задачам NLP.

Источник данных

В этом упражнении мы будем использовать источник данных, хорошо известный в сообществе компьютерного зрения, – набор данных MNIST. Он представляет собой базу данных размеченных изображений рукописных цифр от 0 до 9. Набор данных содержит три различных набора данных – *обучающий* (training set), *проверочный* (validation set) и *тестовый* (test set). Мы будем обучать модель на обучающем наборе и оценивать ее производительность на незнакомом тестовом наборе. Проверочный набор применяется для текущего мониторинга и оптимизации модели в процессе обучения. Детали процесса мы обсудим ниже. Это одна из самых простых задач в классификации изображений, решаемая с помощью простой CNN. Вы убедитесь, что можно достичь точности распознавания тестового набора почти 98 % без какой-либо специальной регуляризации или уловок.

Реализация CNN

Здесь мы рассмотрим только некоторые важные фрагменты кода из реализации CNN при помощи TensorFlow. Полный код доступен в файле `image_ification_mnist.ipynb` в папке `ch5`. Сначала объявим заполнители TensorFlow для ввода входных данных (изображений) и выходных данных (меток). Затем объявим глобальный шаг, который впоследствии будет использован для снижения скорости обучения:

```
# Входные и выходные заполнители.
tf_inputs = tf.placeholder(shape=[batch_size, image_size, image_size,
n_channels],dtype=tf.float32,name='tf_mnist_images')
tf_labels = tf.placeholder(shape=[batch_size, n_classes],dtype=tf.
float32,name='tf_mnist_labels')
# Глобальный шаг для снижения скорости обучения.
global_step = tf.Variable(0,trainable=False)
```

Далее объявим переменные TensorFlow для весов свертки, смещений и весов полностью связанных слоев. Зададим размер фильтра, шаг и заполнение для каждого слоя свертки, размер ядра, шаг и заполнение для каждого объединяющего слоя, а также размерность выхода для каждого полностью связанного слоя в словаре Python под названием `layer_hyperparameters`:

```
# Инициализация переменных.
layer_weights = {}
layer_biases = {}
for layer_id in cnn_layer_ids:
    if 'pool' not in layer_id:
        layer_weights[layer_id] =
tf.Variable(initial_value=tf.random_normal(shape=layer_
hyperparameters[layer_id]['weight_shape'],
stddev=0.02,dtype=tf.float32),name=layer_id+'_weights')
layer_biases[layer_id] = tf.Variable(initial_value=tf.random_
normal(shape=[layer_hyperparameters[layer_id]['weight_shape'][-1]],
stddev=0.01,dtype=tf.float32), name=layer_id+'_bias')
```

Мы также объявим вычисление логита. *Логиты* – это значение выходного слоя перед применением функции активации softmax. Чтобы вычислить логит, мы будем проходить через каждый слой.

Выполняем свертку входных данных для каждого сверточного слоя при помощи следующего кода:

```
h = tf.nn.conv2d(h,layer_weights[layer_id],layer_
hyperparameters[layer_id]['stride'],
layer_hyperparameters[layer_id]['padding']) + layer_biases[layer_id]
```

Здесь вход `h` в `tf.nn.conv2d` заменяется на `tf.inputs` для самой первой свертки. Мы уже подробно обсуждали аргументы `tf.nn.conv2d` в главе 2. Тем не менее давайте кратко повторим назначение аргументов. Итак, `tf.nn.conv2d(input,filter, strides,padding)` принимает следующие значения аргументов в указанном порядке:

- `input` – это вход для свертки, имеющий форму `[batch size, input height, input width, input depth]`;
- `filter` – это сверточный фильтр, с которым мы сворачиваем вход, имеющий форму `[filter height, filter width, input depth, output depth]`;
- `strides` – обозначает шаг перемещения по каждому измерению входных данных и имеет форму `[batch stride, height stride, width stride, depth stride]`;
- `padding` – обозначает тип заполнения (может быть `'SAME'` или `'VALID'`).

Применяем следующую активацию:

```
h = tf.nn.relu(h)
```

Затем для каждого объединяющего слоя выполняем подвыборку входных данных со следующими параметрами:

```
h = tf.nn.max_pool(h, layer_hyperparameters[layer_id]['kernel_
shape'],layer_hyperparameters[layer_id]['stride'],
layer_hyperparameters[layer_id]['padding'])
```

Функция `tf.nn.max_pool(input, ksize, strides, padding)` получает следующие аргументы в указанном порядке:

- `input` – вход для подвыборки, имеющий форму `[batch size, input height, input width, input depth]`;
- `ksize` – размер ядра в каждом измерении для операции max-pooling `[batch kernel size, height kernel size, width kernel size, depth kernel size]`;
- `strides` – шаг по каждому измерению ввода `[batch stride, height stride, width stride, depth stride]`;
- `padding` – заполнение, принимает значение `'SAME'` или `'VALID'`.

Меняем размерность выхода первого полностью подключенного слоя:

```
h = tf.reshape(h,[batch_size,-1])
```

Затем выполняем перемножение весов, добавление смещения и последующую нелинейную активацию:

```
h = tf.matmul(h,layer_weights[layer_id]) + layer_biases[layer_id]
h = tf.nn.relu(h)
```

Теперь мы можем вычислить логиты:

```
h = tf.matmul(h,layer_weights[layer_id]) + layer_biases[layer_id]
```

Присваиваем самое последнее значение h (выход самого последнего слоя) переменной `tf_logits`:

```
tf_logits = h
```

Определяем функцию потерь. В данном случае это перекрестная энтропия, популярная в задачах обучения с учителем:

```
tf_loss = tf.nn.softmax_cross_entropy_with_logits_v2(logits=tf_logits,labels=tf_labels)
```

Нам также необходимо задать скорость обучения, причем мы будем уменьшать скорость обучения в 0,5 раза всякий раз, когда точность при валидации не увеличивается в течение заранее определенного числа эпох (эпоха – это один проход по полному набору данных). Этот прием известен как *снижение скорости обучения* (learning rate decay):

```
tf_learning_rate = tf.train.exponential_decay(learning_
rate=0.001,global_step=global_step,decay_rate=0.5,decay_
steps=1,staircase=True)
```

Далее мы определяем минимизацию потери при помощи оптимизатора RMSPropOptimizer, работающего лучше, чем *стохастический градиентный спуск* (stochastic gradient descent, SGD), особенно в компьютерном зрении:

```
tf_loss_minimize = tf.train.RMSPropOptimizer(learning_rate=tf_
learning_rate, momentum=0.9).minimize(tf_loss)
```

Наконец, для вычисления точности прогнозов путем сравнения предсказанных и фактических меток мы определяем следующую прогнозную функцию:

```
tf_predictions = tf.nn.softmax(tf_logits)
```

Итак, вы научились использовать функции для реализации структуры CNN, определения потерь, минимизации потерь и получения прогнозов для незнакомых данных. В итоге вы создали свою первую сверточную нейросеть для клас-

сификации рукописных изображений, достигающую точности более 98 %. Далее мы проанализируем результаты тестирования и разберемся, почему нейросеть не смогла правильно распознать некоторые изображения.

Анализ прогнозов, сделанных CNN

На рис. 5.12 показаны случайно выбранные правильно и неправильно классифицированные образцы из тестового набора, на основании которых мы можем оценить качество обучения CNN. Мы видим, что в случае правильно классифицированных образцов нейросеть практически однозначно уверена в своем выборе, что можно рассматривать как хорошее свойство алгоритма обучения. Однако если присмотреться к ошибочно классифицированным образцам, становится очевидно, что они на самом деле трудны, и даже человек может неправильно распознать некоторые из них (например, третье изображение слева во втором ряду). Сделав неправильный выбор, нейросеть уже не так уверена, что опять же является хорошей характеристикой. Кроме того, даже если нейросеть выдала неправильный прогноз, правильная метка зачастую не полностью игнорируется и получает определенный рейтинг в составе прогноза.

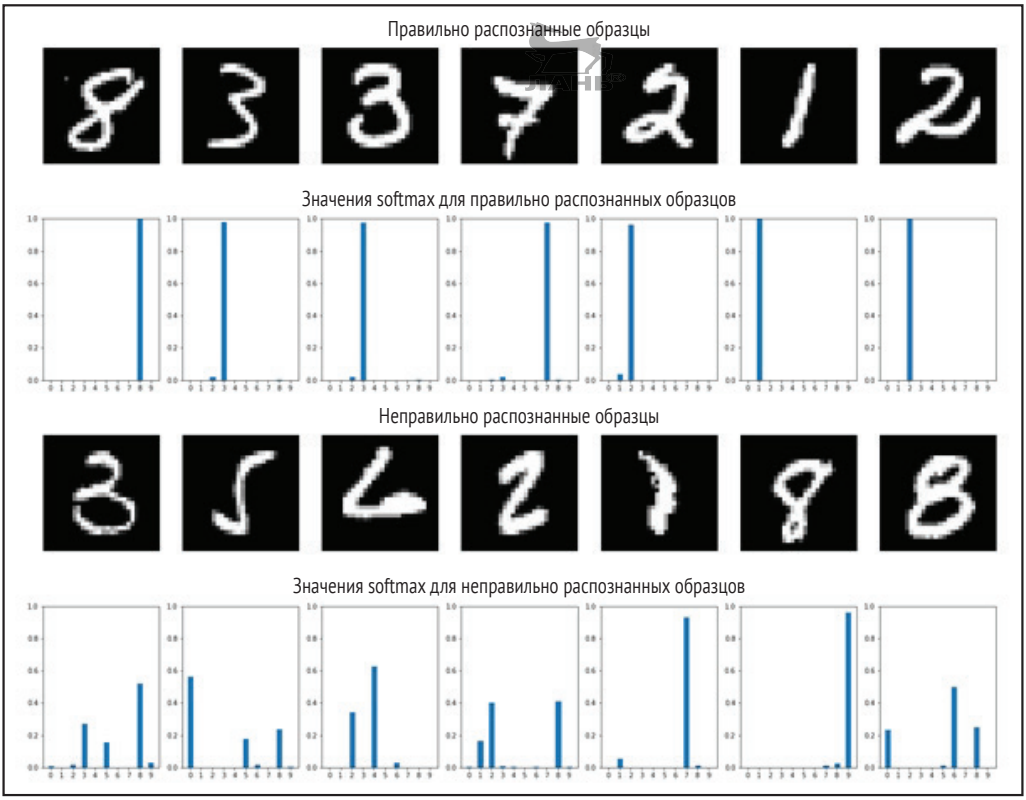


Рис. 5.12 ❖ Примеры правильных и ошибочных прогнозов на наборе MNIST

Классификация предложений с помощью сверточной нейросети

Хотя сверточные нейросети традиционно использовались для задач компьютерного зрения, ничто не мешает их использовать в приложениях NLP. Одним из таких приложений, где эффективно работают нейросети, является классификация предложений.

В следующем учебном упражнении мы воспользуемся обучающей базой данных вопросов, где каждому вопросу присвоена метка класса в соответствии с тем, о чем идет речь.

Например, предложение «Кем был Авраам Линкольн?» является вопросом и снабжено меткой «личность». Для обучения и тестирования нейросети будем использовать размеченный набор предложений на английском языке[1], доступный по адресу **http://cogcomp.org/Data/QA/QC/**. Здесь вы найдете 1000 обучающих предложений и соответствующих им меток и 500 тестовых предложений.

Далее мы рассмотрим реализацию нейросети, представленной в статье Юна Кима[2] «Сверточные нейронные сети для классификации предложений». Однако использование сверточных нейросетей для классификации предложений несколько отличается от примера MNIST, который мы обсуждали выше, потому что теперь все операции выполняются в одном измерении, а не в двух. Кроме того, операции субдискретизации также имеют определенные особенности, о которых вы скоро узнаете. Полный код этого упражнения хранится в файле `cnn_terminсe_classification.ipynb` в папке `ch5`.

Структура нейросети

Далее мы обсудим технические детали реализации сверточной нейросети для классификации предложений. Во-первых, вы узнаете, как данные или предложения преобразуются в подходящий формат для последующей обработки нейросетью. Затем мы рассмотрим объединение операций свертки и субдискретизации в задаче классификации предложений и, наконец, объединим все компоненты в нейросеть.

Преобразование данных

Предположим, что у нас есть предложение из p слов. Если длина предложения меньше заданного значения n, мы дополним его специальным словом, чтобы получить предложение длиной в n слов, где $n \geqslant p$. Далее мы представим каждое слово в предложении вектором размера k, т. е. унитарным кодом или вектором слова Word2vec, полученным с помощью skip-gram, CBOW или GloVe. В свою очередь, пакет предложений размером b может быть представлен матрицей $b \times n \times k$.

---

[1] К сожалению, не удалось найти в открытом доступе равноценный обучающий набор на русском языке. Поэтому далее в этой главе фигурируют исходные примеры на английском. – *Прим. перев.*

[2] *Yoon Kim*. Convolutional Neural Networks for Sentence Classification.

В качестве примера рассмотрим следующие три предложения:

- *Боб и Мэри близкие друзья.*
- *Боб любит футбол.*
- *Мэри обожает читать хорошие книги по вечерам.*

В этом примере в третьем предложении больше всего слов, поэтому назначим $n = 7$, то есть по количеству слов в третьем предложении. Далее, составим унитарное представление каждого слова. В данном случае есть 13 разных слов. Таким образом, мы получаем следующие коды:

Боб: 1,0,0,0,0,0,0,0,0,0,0,0,0
и: 0,1,0,0,0,0,0,0,0,0,0,0,0
Мэри: 0,0,1,0,0,0,0,0,0,0,0,0,0 и т. д.

Кроме того, $k = 13$ по той же причине. С помощью унитарного кода мы можем представить наши предложения в виде трехмерной матрицы размером 3×7×13, как изображено на рис. 5.13.

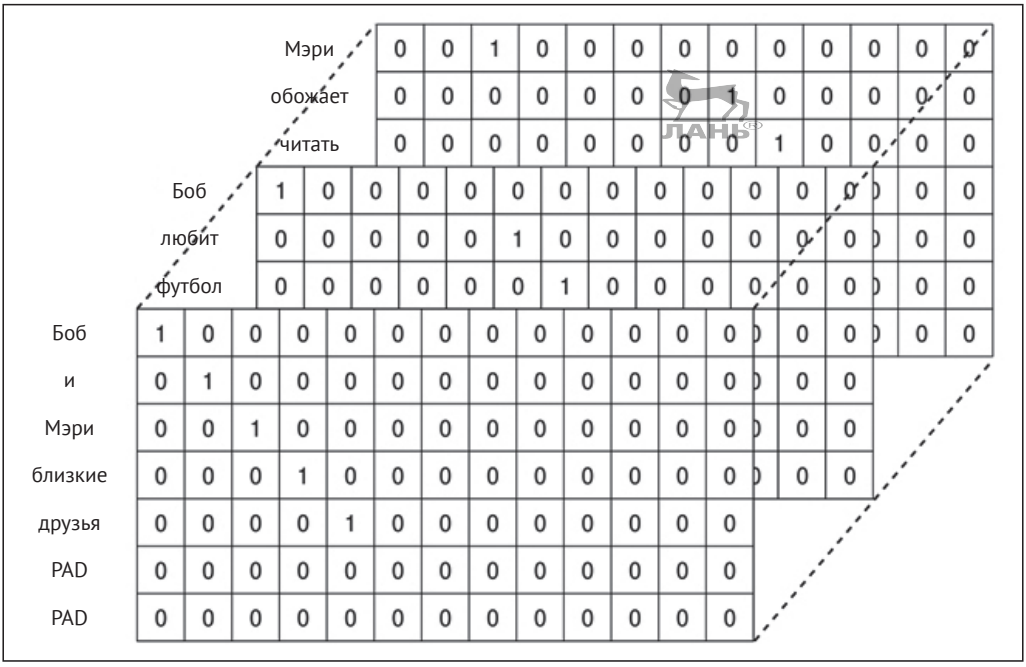


Рис. 5.13 ❖ Матрица предложений

Операция свертки

Если мы проигнорируем размер пакета, т. е. если предположим, что мы обрабатываем только одно предложение за один раз, наши данные будут представлять собой матрицу $n \times k$, где n – количество слов в предложении после дополнения, а k – размерность вектора одного слова. В нашем примере это будет матрица 7×13.

Теперь определим матрицу весов свертки размером $m \times k$, где m – размер фильтра для одномерной операции свертки. Свернув вход x размера $n \times k$ с весовой матрицей W размера $m \times k$, мы получим вывод h размера $1 \times n$ согласно следующему уравнению:

$$h_{i,1} = \sum_{j=1}^{m} \sum_{l=1}^{k} w_{j,l} x_{i+j-1,l}.$$

Здесь $w_{i,j}$ – (i, j)-й элемент W, и мы будем дополнять x нулями, так что h будет иметь размер $1 \times n$. Мы можем записать эту операцию проще:

$$h = W * x + b.$$

Здесь символ * определяет операцию свертки с заполнением, и мы прибавляем дополнительное скалярное смещение b. Операция схематически изображена на рис. 5.14.

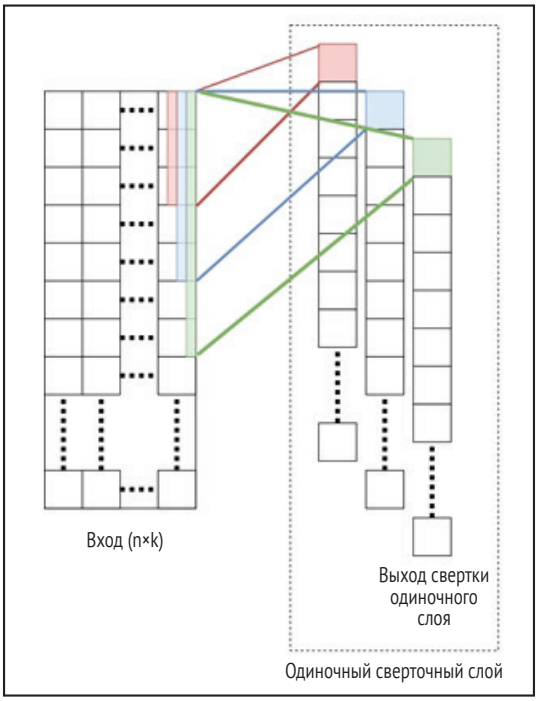


Рис. 5.14 ❖ Операция свертки в задаче классификации предложений

Чтобы изучить богатый набор признаков, у нас имеются параллельные слои с различными размерами сверточного фильтра. Каждый сверточный слой выводит скрытый вектор размером $1 \times n$, и мы объединяем эти выходы, чтобы сформировать входные данные с размерностью $q \times n$ для следующего слоя, где q – количество параллельных слоев. Чем больше q, тем лучше качество модели.

Суть свертки можно продемонстрировать на примере. Допустим, стоит задача научить модель разделять отзывы о фильмах на два класса – положительные и отрицательные, и у нас есть следующие предложения:

- *Мне нравится этот фильм, весьма неплохо*
- *Мне не нравится этот фильм, плохо*

Теперь представьте окно свертки размером 5. Давайте скомпонуем слова в соответствии с движением окна свертки.

Из первого предложения получаются следующие блоки:

[*Мне*, *нравится*, *этот*, *фильм*, ',']
[*нравится*, *этот*, *фильм*, ',', *весьма*]
[*этот*, *фильм*, ',', *весьма*, *неплохо*]

Из второго предложения получаются блоки:

[*Мне*, *не*, *нравится*, *этот*, *фильм*]
[*не*, *нравится*, *этот*, *фильм*, ',']
[*нравится*, *этот фильм*, ',', *плохо*]

В первом предложении все три блока указывают на положительную оценку.

Тем не менее во втором предложении следующие окна передают негативную оценку:

[*Мне*, *не*, *нравится*, *этот*, *фильм*]
[*не*, *нравится*, *этот*, *фильм*, ',']

Нетрудно заметить, что данные блоки помогают классифицировать отзывы благодаря сохранению пространственного распределения. Например, если для вычисления представлений предложений вы используете такой метод, как CBOW, теряющий пространственную информацию, то представления могут оказаться очень близкими. Операция свертки играет важную роль в сохранении пространственной информации предложений.

Имея q разных слоев с разными размерами фильтров, сеть учится извлекать оценки из фраз разного размера, что приводит к повышению точности.

Растянутая субдискретизация

Операция субдискретизации предназначена для подвыборки выходных данных, созданных рассмотренными ранее параллельными сверточными слоями. Это достигается следующим образом.

Предположим, выход последнего слоя h имеет размер $q \times n$. *Слой растянутой субдисретизации* (pooling over time layer) даст выход h' с размером $q \times 1$. Точное вычисление выполняют следующим образом:

$$h'_{i,1} = \{\max(h^{(i)}), \text{ где } 1 \leqslant i \leqslant q\}.$$

Здесь $h^{(i)} = W^{(i)} * x + b$ и $h^{(i)}$ – выходной сигнал, создаваемый i-м слоем свертки, а $W^{(i)}$ – набор весов, принадлежащих этому слою. Проще говоря, операция растянутой субдискретизации создает вектор путем объединения максимального элемента каждого слоя свертки. Операция схематически изображена на рис. 5.15.

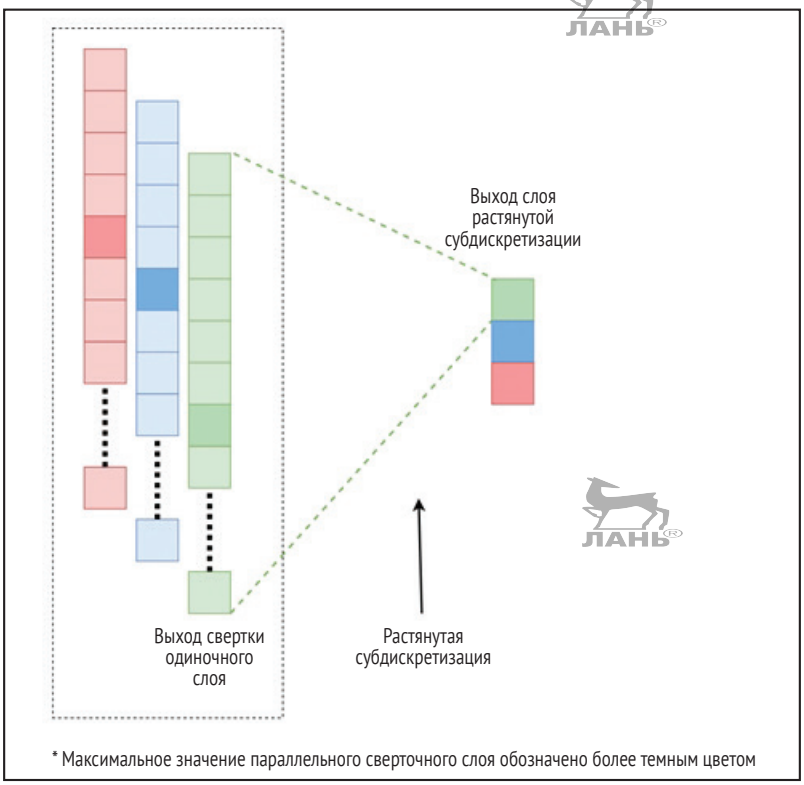


Рис. 5.15 ❖ Операция растянутой субдискретизации в задаче классификации предложений

Комбинируя вышеупомянутые операции, мы приходим к архитектуре, изображенной на рис. 5.16.

Реализация классификации предложений

Сначала определим входы и выходы. На вход будет поступать пакет предложений, в котором слова представлены векторами с унитарным кодированием. Вообще-то, векторное представление слов обеспечит лучшую точность, чем унитарный код, но для простоты мы будем использовать первый вариант:

```
sent_inputs = tf.placeholder(shape=[batch_size,sent_length,vocabulary_
size],dtype=tf.float32,name='sentence_inputs')
sent_labels = tf.placeholder(shape=[batch_size,num_classes],dtype=tf.
float32,name='sentence_labels')
```

Далее определим три разных одномерных сверточных слоя с тремя разными размерами фильтров 3, 5 и 7 (предоставленными в виде списка в `filter_sizes`) и их соответствующими смещениями:

```
w1 = tf.Variable(tf.truncated_normal([filter_sizes[0],vocabulary_
size,1],stddev=0.02,dtype=tf.float32),name='weights_1')
```

```
b1 = tf.Variable(tf.random_uniform([1],0,0.01,dtype=tf.float32),name='bias_1')

w2 = tf.Variable(tf.truncated_normal([filter_sizes[1],vocabulary_
size,1],stddev=0.02,dtype=tf.float32),name='weights_2')
b2 = tf.Variable(tf.random_uniform([1],0,0.01,dtype=tf.float32),name='bias_2')

w3 = tf.Variable(tf.truncated_normal([filter_sizes[2],vocabulary_
size,1],stddev=0.02,dtype=tf.float32),name='weights_3')
b3 = tf.Variable(tf.random_uniform([1],0,0.01,dtype=tf.float32),name='bias_3')
```

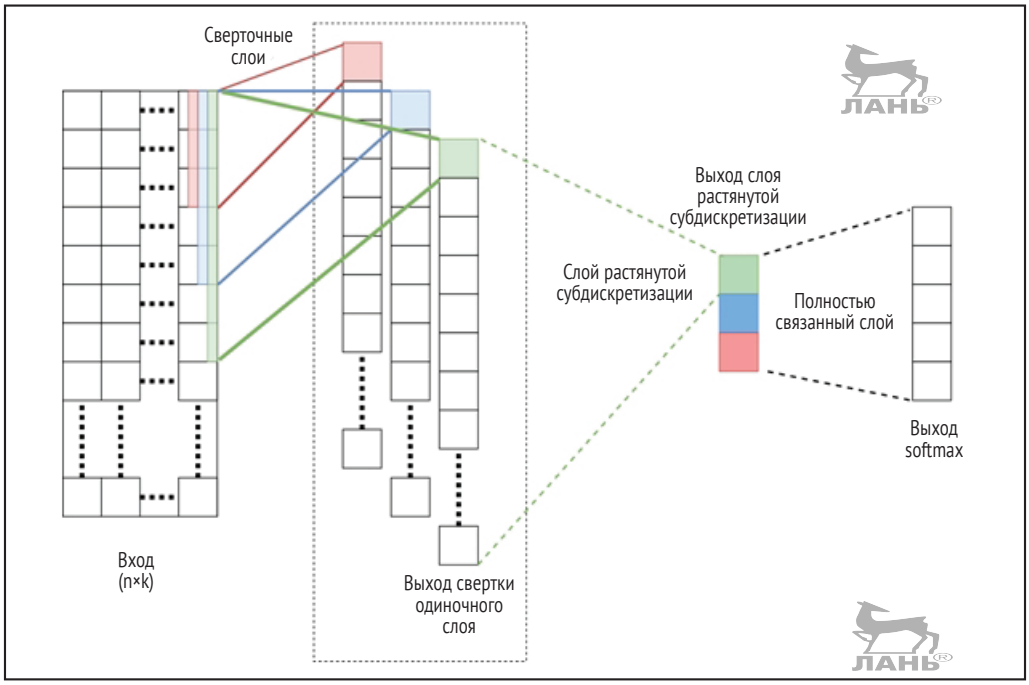


Рис. 5.16 ❖ Архитектура сверточной нейросети для классификации предложений

Теперь рассчитаем три выхода, каждый из которых принадлежит одному из только что определенных слоев свертки. Их можно легко вычислить с помощью функции `tf.nn.conv1d`, предоставленной в библиотеке TensorFlow. Мы используем шаг 1 и заполнение нулями, чтобы гарантировать, что выходы будут иметь тот же размер, что и вход:

```
h1_1 = tf.nn.relu(tf.nn.conv1d(sent_inputs,w1,stride=1,padding='SAME') + b1)
h1_2 = tf.nn.relu(tf.nn.conv1d(sent_inputs,w2,stride=1,padding='SAME') + b2)
h1_3 = tf.nn.relu(tf.nn.conv1d(sent_inputs,w3,stride=1,padding='SAME') + b3)
```

Для расчета max-pool, т. е. выборки максимального значения растянутой субдискретизации, нам придется самостоятельно написать элементарные функции, поскольку TensorFlow не имеет подходящей встроенной функции. Однако написать эти функции довольно просто.

Сначала мы рассчитаем максимальное значение каждого скрытого вывода, производимого каждым слоем свертки, и получим по одному скаляру для каждого слоя:

```
h2_1 = tf.reduce_max(h1_1,axis=1)
h2_2 = tf.reduce_max(h1_2,axis=1)
h2_3 = tf.reduce_max(h1_3,axis=1)
```

Затем объединяем полученные выходные данные по оси 1 (ширина), чтобы получить выходные данные с размерностью (*размер пакета* × q):

```
h2 = tf.concat([h2_1,h2_2,h2_3],axis=1)
```

Далее определяем полностью связанные слои, которые будут полностью подключены к полученному выше выходу. В данном случае есть только один полностью связанный слой, и он же будет являться выходным слоем:

```
w_fc1 = tf.Variable(tf.truncated_normal([len(filter_sizes),num_
classes],stddev=0.5,dtype=tf.float32),name='weights_fulcon_1')
b_fc1 = tf.Variable(tf.random_uniform([num_classes],0,0.01,dtype=tf.
float32),name='bias_fulcon_1')
```

Следующая функция будет генерировать логиты, которые затем используются для расчета потери:

```
logits = tf.matmul(h2,w_fc1) + b_fc1
```

Далее, применяя к логитам активацию softmax, получаем прогнозы:

```
predictions = tf.argmax(tf.nn.softmax(logits),axis=1)
```

Опеределяем функцию потерь, вычисляемой как перекрестная энтропия:

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_
v2(labels=sent_labels,logits=logits))
```

Для оптимизации сети будем использовать встроенный оптимизатор TensorFlow под названием `MomentumOptimizer`:

```
optimizer = tf.train.MomentumOptimizer(learning_
rate=0.01,momentum=0.9).minimize(loss)
```

Выполнение упомянутых выше операций для оптимизации нейросети и оценки тестовых данных дает нам точность на проверочных данных, близкую к 90 % (500 проверочных предложений).

На этом мы заканчиваем обсуждение использования сверточных нейросетей для классификации предложений. Вы узнали о том, как одномерные операции свертки в сочетании со специальной операцией, так называемой растянутой субдискретизацией, могут применяться для реализации классификатора предложений на основе сверточной нейросети. Наконец, вы выполнили упражнение по реализации сверточного классификатора на основе библиотеки TensorFlow и убедились, что он действительно хорошо работает при классификации предложений.

Давайте подумаем, как решение проблемы классификации предложений может быть полезно в реальном мире. Предположим, что вы располагаете большим документом об истории Рима и хотите узнать о Юлии Цезаре, не читая весь документ. В этой ситуации было бы удобно воспользоваться классификатором предложений в качестве удобного инструмента поиска предложений, соответствующих только нужному человеку, поэтому вам не пришлось бы читать весь документ.

Классификация предложений может применяться и для многих других задач. Одним из распространенных применений является классификация обзоров фильмов для автоматического расчета рейтингов. Еще одно важное применение классификация предложений нашла в медицине. Нейросети извлекают клинически значащие предложения из документов, содержащих большое количество текста.

Заключение

В этой главе мы обсудили сверточные нейросети и их различные применения. Во-первых, вы узнали о том, почему они так эффективны в задачах машинного обучения. Затем мы разложили сверточную нейросеть на несколько компонентов, таких как слои свертки и субдискретизации, и подробно обсудили, как выполняются эти операции. Кроме того, мы упомянули несколько важных гиперпараметров, таких как размер фильтра, шаг и заполнение. Затем, чтобы продемонстрировать сверточную нейросеть в действии, мы рассмотрели простой пример классификации изображений рукописных цифр и провели небольшой анализ, чтобы понять, почему нейросеть не может правильно распознать некоторые изображения. Наконец, перешли к применению сверточных нейросетей в задачах NLP. Вы изучили модифицированную архитектуру сверточной нейросети для классификации предложений. Затем выполнили еще одно упражнение и проверили сверточную нейросеть на реальной задаче классификации предложений.

В следующей главе вы познакомитесь с одной из самых популярных разновидностей нейросетей, применяемых для многих задач NLP, – рекуррентными нейронными сетями.

Глава 6

Рекуррентные нейронные сети

Рекуррентные нейронные сети (Recurrent Neural Networks, RNN) – это особое семейство нейронных сетей, предназначенных для работы с последовательными данными, т. е. данными временных рядов, такими как текстовая последовательность (например, предложение переменной длины или документ) или цены фондового рынка. RNN обладают переменной состояния, которая фиксирует различные шаблоны, присутствующие в последовательных данных; следовательно, они могут моделировать последовательные данные. Например, обычные нейронные сети прямого распространения не имеют такой возможности, если только данные не представлены в особой форме, позволяющей фиксировать важные признаки, присущие последовательности. Однако придумать такие представления развернутых во времени признаков крайне сложно. Другой альтернативой для моделей прямого распространения при моделировании последовательных данных является наличие отдельного набора параметров для каждой позиции во времени/последовательности. Таким образом, набор параметров, назначенных определенной позиции, выявляет шаблоны этого фрагмента последовательности. Подобный подход значительно увеличивает требования к памяти модели.

В отличие от сетей прямого распространения, которым нужен отдельный набор параметров для каждой позиции, рекуррентные нейросети используют один и тот же набор общих *сквозных параметров* (parameters over time). Наличие таких параметров фактически является одним из ключевых факторов, позволяющих рекуррентным нейросетям изучать временны́е схемы. Переменная состояния постоянно обновляется с поступлением каждого входа, наблюдаемого в последовательности. Исходя из общих параметров в сочетании с текущим вектором состояния, рекуррентные нейросети могут предсказывать следующее значение последовательности с учетом ранее наблюдаемых значений. Кроме того, поскольку мы обрабатываем один элемент последовательности за раз (например, одно слово в документе), рекуррентные нейросети могут обрабатывать данные произвольной длины без дополнения набора данных специальными токенами.

В этой главе вы изучите устройство и принцип работы рекуррентных нейросетей. Сначала узнаете, как можно создать рекуррентную нейросеть, начав с простой модели прямого распространения. После этого мы обсудим основные функциональные возможности RNN. Вы изучите математические основы и познакомитесь с различными вариантами устройства RNN: один-к-одному, один-ко-многим

и многие-ко-многим. Мы рассмотрим пример использования рекуррентной нейросети для генерации нового текста на основе набора обучающих данных, а также обсудим некоторые функциональные ограничения. Наконец, разберем улучшенную версию рекуррентной нейросети под названием RNN-CF, которая обладает более долгой памятью.

Знакомство с рекуррентными нейронными сетями

Как вы уже знаете, рекуррентная нейросеть поддерживают переменную состояния, отражающую текущее состояние нейросети. Поскольку с течением времени нейросеть видит все больше и больше данных, это дает возможность моделировать последовательные данные. В частности, переменная состояния с течением времени обновляется за счет рекуррентных связей. Наличие таких связей является основным структурным различием между рекуррентной нейросетью и сетью прямого распространения. Под рекуррентными связями можно понимать связи между сохраненными ранее в памяти элементами последовательности и текущей переменной состояния нейросети. Другими словами, рекуррентные связи обновляют текущее значение переменной состояния, исходя из содержимого памяти нейросети, что позволяет ей делать прогноз на основе текущего входа, но с учетом предыдущих входов.

Далее вы узнаете, как можно представить нейросеть прямого распространения в виде графа вычислений, и увидите, почему такая сеть перестает работать при последовательных данных. Затем мы адаптируем граф модели прямого распространения к последовательным данным, что даст нам базовый граф вычислений рекуррентной нейросети. Мы также обсудим технические детали, в том числе правила обновления состояния и подходы к обучению рекуррентных нейросетей.

Проблема с нейросетью прямого распространения

Чтобы оценить предел возможностей нейронных сетей прямого распространения и понять, как рекуррентные нейросети преодолевают это ограничение, возьмем две последовательности данных:

$$x = \{x_1, x_2, \dots x_T\}, \quad y = \{y_1, y_2, \dots y_T\}.$$

Далее, давайте предположим, что в реальном мире x и y связаны следующими отношениями:

$$h_i = g_1(x_t, h_{t-1}),$$
$$y_t = g_2(h_t),$$

где g_1 и g_2 – некоторые функции. Это означает, что текущий выходной сигнал y_t зависит от текущего состояния h_t для некоторого состояния, принадлежащего модели, которая выводит x и y. Кроме того, h_t рассчитывается исходя из текущего ввода x_t и предыдущего состояния h_{t-1}. Состояние кодирует информацию о предыдущих входных данных, наблюдаемых моделью в истории работы.

Теперь давайте возьмем простую нейронную сеть с прямым распространением, которую представим следующим образом:

$$y_t = f(x_t; \theta),$$

где y_t – предсказанный выход при некотором входе x_t.

Если для решения этой задачи мы используем нейронную сеть с прямым распространением, она должна будет производить элементы $\{y_1, y_2, \dots y_T\}$ по одному, принимая $\{x_1, x_2, \dots x_T\}$ в качестве входных данных. Теперь давайте рассмотрим проблему, с которой мы столкнемся, решая задачу с последовательными данными.

Прогнозируемый выходной сигнал x_t в момент времени t нейронной сети с прямой связью зависит только от текущего входного сигнала x_t. Другими словами, он не имеет никаких сведений о входных данных, которые привели к x_t, т. е. $\{x_1, x_2, \dots x_{t-1}\}$. По этой причине нейронная сеть прямого распространения потерпит неудачу при выполнении задачи, где текущий выход зависит не только от текущего входа, но и от предыдущих входов. Давайте рассмотрим простой пример.

Допустим, нам нужно обучить нейронную сеть, чтобы угадывать пропущенные слова. У нас есть исходная фраза, и мы хотели бы предсказать недостающее слово:

У Джеймса есть кошка, и она любит пить _____.

Если мы возьмемся обрабатывать по одному слову за раз, используя нейронную сеть с прямой связью, у нас будет только слово «*пить*», и этого совсем недостаточно для понимания фразы или хотя бы для понимания контекста – слово «*пить*» может встречаться в разных контекстах. Вы можете возразить, что можно достичь неплохих результатов, обрабатывая полное предложение за один раз. Даже если так, этот подход быстро станет непрактичным в случае длинных предложений.

Моделирование с помощью рекуррентных нейронных сетей

С другой стороны, для решения этой проблемы мы можем использовать рекуррентную нейросеть. Начнем с данных, которые у нас есть:

$$x = \{x_1, x_2, \dots x_T\}, \quad y = \{y_1, y_2, \dots y_T\}.$$

Допустим, выполняются следующие отношения:

$$h_t = g_1(x_t, h_{t-1}),$$
$$y_t = g_2(h_t).$$

Теперь давайте заменим g_1 аппроксиматором функции $f_1(x_t, h_{t-1}; \theta)$ с параметрами θ, который принимает текущий вход x_t и предыдущее состояние системы h_{t-1} в качестве входных данных и создает текущее состояние h_t. Затем мы заменим g_2 на $f_2(h_t; \varphi)$, который принимает текущее состояние системы h_t для получения y_T. Это дает нам следующее:

$$h_t = f_1(x_t, h_{t-1}; \theta),$$
$$y_t = f_2(h_t; \varphi).$$

Мы можем думать о $f_1 \circ f_2$ как об аппроксимации истинной модели, которая генерирует x и y. Для большей ясности понимания давайте развернем уравнение следующим образом:

$$y_t = f_2(f_1(x_t, h_{t-1}; \theta); \varphi).$$

Например, мы можем представить y_4 выражением:

$$y_4 = f_2(f_1(x_4, h_3; \theta); \varphi).$$

Развернем это выражение, опустив для упрощения θ и φ:

$$y_4 = f_2\left(f_1\left(x_4, f_2\left(f_1\left(x_3, f_2\left(f_1\left(x_2, f_2\left(f_1(x_1, h_0)\right)\right)\right)\right)\right)\right)\right).$$

Данное выражение можно представить в виде графа, изображенного на рис. 6.1.

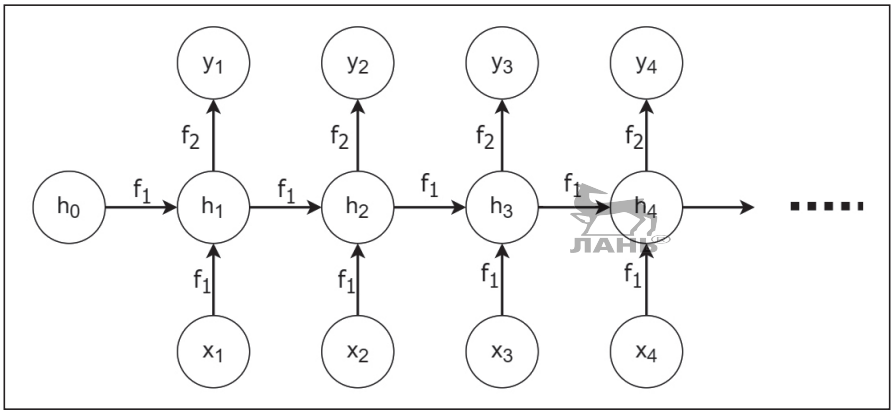


Рис. 6.1 ❖ Развернутая схема отношений между x_i и y_i

Мы можем обобщить схему на рис. 6.1 до схемы одного блока t, как показано на рис. 6.2.

Тем не менее следует понимать, что h_{t-1} фактически является тем, чем h_t был до получения x_t. Другими словами, h_{t-1} – это h_t на один шаг назад. Следовательно, мы можем представить вычисление h_t с помощью периодического соединения, как показано на рис. 6.3.

Возможность суммировать цепочку уравнений, отображающую $\{x_1, x_2, \ldots x_T\}$ в $\{y_1, y_2, \ldots y_T\}$, как на рис. 6.3, позволяет нам записать любой y_t в терминах x_t, h_{t-1} и h_t. Это ключевая идея, лежащая в основе рекуррентной нейросети.

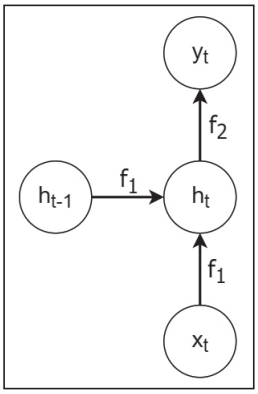


Рис. 6.2 ❖ Схема одного блока вычислений в структуре рекуррентной сети

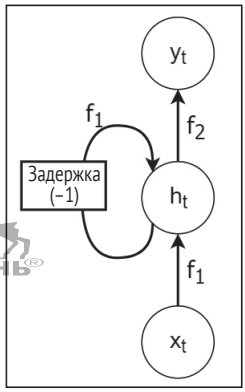


Рис. 6.3 ❖ Схема одного блока вычислений с рекуррентной связью

Устройство рекуррентной нейронной сети в деталях

Теперь давайте еще подробнее рассмотрим, как работает рекуррентная нейросеть, и определим математические уравнения для вычислений внутри нейросети. Давайте начнем с двух функций, которые известны нам как аппроксиматоры функций для обучения y_t по x_t:

$$h_t = f_1(x_t, h_{t-1}; \theta),$$
$$y_t = f_2(h_t; \varphi).$$

Как мы уже видели, нейронная сеть состоит из набора весов и смещений и некоторой нелинейной функции активации. Поэтому можем записать предыдущее отношение в таком виде:

$$h_1 = tanh(Ux_t + Wh_{t-1}).$$

Здесь *tanh* – это функция активации (гиперболический тангенс), U – матрица весов с размерностью $m \times d$, где m – это количество скрытых объектов и d – размерность входа, W – это матрица весов с размерностью $m \times m$, образующая рекуррентную связь от h_{t-1} к h_t. Отношение y_t задается следующим уравнением:

$$y_t = softmax(Vh_t).$$

Здесь V – это весовая матрица размера $c \times m$, а c – это размерность выходных данных (может быть числом выходных классов). На рис. 6.4 мы иллюстрируем, как эти веса образуют рекуррентную нейросеть.

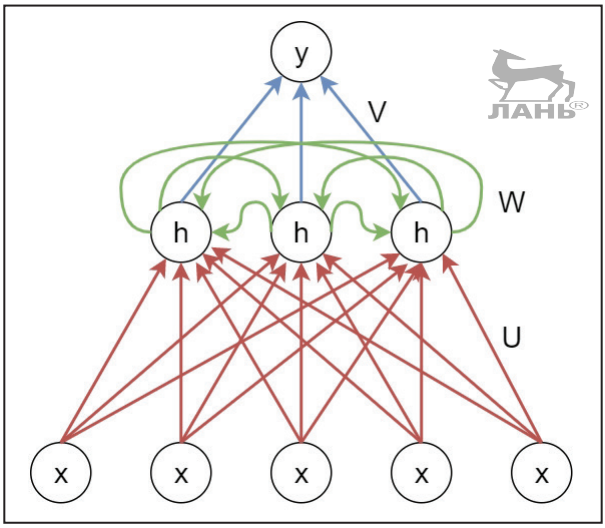


Рис. 6.4 ❖ Структура рекуррентной нейросети

До сих пор речь шла о том, как представить рекуррентную нейросеть в виде графа вычислений с ребрами, обозначающими вычисления. Кроме того, мы рассмотрели базовую математику, лежащую в основе RNN. Давайте теперь посмотрим, как выполняется оптимизация весов рекуррентной нейросети для обучения на последовательных данных.

Обратное распространение во времени

Для обучения рекуррентных сетей используется специальная форма обратного распространения, известная как *обратное распространение во времени* (backpropagation through time, BPTT). Однако чтобы понять BPTT, сначала нам нужно понять, как работает обычное *обратное распространение* (backpropagation, BP). Затем мы обсудим, почему обратное распространение нельзя напрямую использовать в рекуррентных нейросетях и как перестроить BP, чтобы получить BPTT. Наконец, мы рассмотрим две основные проблемы, присущие обратному распространению во времени.

Как работает обратное распространение

Обратное распространение – это метод, который используется для обучения нейронной сети. При обратном распространении выполняются следующие операции:

1) вычисляется прогноз для данного входа;
2) путем сравнения прогноза с фактической меткой входных данных вычисляются потери E (например, среднеквадратичная ошибка или перекрестная энтропия);
3) обновляются веса сети прямого распространения, чтобы минимизировать потери, рассчитанные на шаге 2. Для этого делают небольшой шаг в проти-

воположном направлении градиента $\partial E/\partial w_{ij}$ для всех w_{ij}, где w_{ij} – это j-й вес i-го слоя.

Для более ясного понимания рассмотрим сеть прямой связи, изображенную на рис. 6.5. Она имеет два одинарных веса, w_1 и w_2, и рассчитывает два выходных сигнала h и y, как показано на рис. 6.5. Для простоты мы не вводим в эту модель нелинейность.

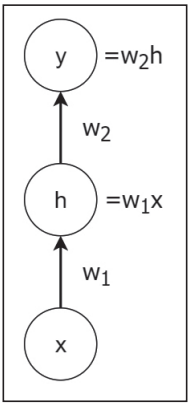


Рис. 6.5 ❖ Вычисления в сети с обратным распространением

Мы можем вычислить $\frac{\partial E}{\partial w_1}$ по цепному правилу следующим образом:

$$\frac{\partial E}{\partial w_1} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial w_1}.$$

Выполнив подстановки, получаем следующее уравнение:

$$\frac{\partial E}{\partial w_1} = \frac{\partial (y-l)^2}{\partial y}\frac{\partial (w_2 h)}{\partial h}\frac{\partial (w_1 x)}{\partial w_1}.$$

Здесь l – правильная метка для точки данных x. Кроме того, мы принимаем среднеквадратичную ошибку в качестве функции потерь. Здесь все определено, и вычислить $\frac{\partial E}{\partial w_1}$ довольно просто.

Почему нельзя использовать простое обратное распространение

Давайте попробуем применить обратное распространение в нейросети, изображенной на рис. 6.6. Теперь у нас есть дополнительный рекуррентный вес w_3. Мы упустили временны́е компоненты входов и выходов для ясности проблемы, которую пытаемся подчеркнуть.

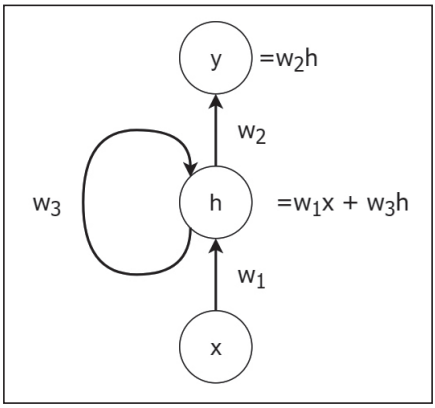


Рис. 6.6 ❖ Вычисления в рекуррентной нейросети

Посмотрим, что произойдет, если мы применим цепное правило для расчета $\frac{\partial E}{\partial w_3}$:

$$\frac{\partial E}{\partial w_3} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial w_3}.$$

Отсюда следует:

$$\frac{\partial E}{\partial w_3} = \frac{\partial (y-l)^2}{\partial y}\frac{\partial (w_2 h)}{\partial h}\left(\frac{\partial (w_1 x)}{\partial w_3} + \frac{\partial (w_3 h)}{\partial w_3}\right).$$

Член $\frac{\partial (w_3 h)}{\partial w_3}$ в данном случае создает проблему, поскольку он рекурсивный. Мы получим бесконечное количество производных, т. к. h является рекурсивным, т. е. вычисление h включает в себя сам h, где h не является константой и зависит от w_3. Проблема решается путем разворачивания входной последовательности x во времени – создания копии RNN для каждого входа x_t и вычисления производных для каждой копии отдельно, а затем сведения их обратно, суммируя градиенты, чтобы вычислить обновление веса. Детали этого процесса будут показаны ниже.

Обратное распространение во времени и обучение RNN

Хитрость в расчете обратного распространения для рекуррентных нейросетей состоит в том, чтобы рассматривать не один вход, а полную последовательность входов. Затем, если мы рассчитаем $\frac{\partial E}{\partial w_3}$ на шаге 4 во времени, получим следующее уравнение:

$$\frac{\partial E}{\partial w_3} = \sum_{j=1}^{3}\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial h_4}{\partial h_j}\frac{\partial h_j}{\partial w_3}.$$

Это означает, что нам нужно вычислить сумму градиентов для всех временных шагов до четвертого шага. Другими словами, мы сначала разворачиваем последовательность, чтобы вычислить $\frac{\partial h_4}{\partial h_j}$ и $\frac{\partial h_j}{\partial w_3}$ для каждого временного шага j. Это делается путем создания четырех копий рекуррентной нейросети. В общем случае для вычисления $\frac{\partial h_t}{\partial h_j}$ нам понадобится $t - j + 1$ копий нейросети. Затем сворачиваем копии в одну нейросеть, суммируя градиенты по всем предыдущим временны́м шагам, чтобы получить градиент, и обновляем ее с градиентом $\frac{\partial E}{\partial w_3}$. Однако эти манипуляции становятся дорогостоящими по мере увеличения количества временных шагов. Для большей вычислительной эффективности мы можем использовать для оптимизации рекуррентных моделей *усеченное обратное распространение во времени* (truncated backpropagation through time, TBPTT), являющееся приближением к BPTT.

Усеченное обратное распространение во времени

В TBPTT мы рассчитываем градиенты только для фиксированного числа T временных шагов (в отличие от вычисления до самого начала последовательности, как в BPTT). Точнее, при вычислении $\frac{\partial E}{\partial w_3}$ для шага t мы вычисляем производные только до $t - T$, т. е. мы не вычисляем производные до самого начала:

$$\frac{\partial E}{\partial w_3} = \sum_{j=t-T}^{t-1} \frac{\partial L}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_j} \frac{\partial h_j}{\partial w_3}.$$

Это намного эффективнее в вычислительном отношении, чем в случае стандартного BPTT, когда для каждого шага t мы вычисляем производные до самого начала последовательности. Но это быстро становится невозможным в вычислительном отношении, поскольку длина последовательности становится все больше и больше (например, обработка текстового документа слово за словом). Однако в усеченном BPTT мы вычисляем производные только для фиксированного числа шагов назад, и, как вы понимаете, вычислительные затраты не меняются по мере увеличения последовательности.

Ограничения BPTT – исчезающие и взрывающиеся градиенты

Наличие способа вычисления градиентов для повторяющихся весов и наличие вычислительно эффективного приближения, такого как TBPTT, еще не решает все проблемы обучения рекуррентной нейросети. Что же еще может пойти не так? Чтобы понять, в чем дело, давайте развернем член $\frac{\partial E}{\partial w_3}$ следующим образом:

$$\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial w_3}=\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial(w_1x+w_3h_3)}{\partial h_1}\frac{\partial(w_1x+w_3h_0)}{\partial w_3}.$$

Поскольку мы знаем, что проблемы обратного распространения возникают из-за рекуррентных связей, давайте проигнорируем члены w_1x и запишем правую часть равенства в виде:

$$=\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial(w_3h_3)}{\partial h_1}\frac{\partial(w_3h_0)}{\partial w_3}.$$

Простым разложением h_3 и выполнением простых арифметических операций мы можем показать следующее:

$$=\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}h_0w_3^3.$$

Мы видим, что всего за четыре временных шага у нас есть член w_3^3. Таким образом, на n-м шаге он станет w_3^{n-1}. Допустим, если бы мы инициализировали w очень маленьким значением (например, 0,00001), то после шага n = 100 градиент был бы бесконечно малым – $0{,}1^{500}$. Кроме того, поскольку компьютеры имеют ограниченную точность в представлении чисел, они просто проигнорируют столь малое значение, принимая его за ноль. Это явление называется *исчезающим градиентом* (vanishing gradient). Решить проблему исчезающего градиента не очень просто. Нет простых способов изменить масштаб градиентов, чтобы они правильно распространялись во времени. Несколько методов, позволяющих частично решить проблему исчезающих градиентов, – это тщательная инициализация весов (например, инициализация Ксавье) или *оптимизация на основе импульса* (momentum-based optimization), т. е. в дополнение к текущему обновлению градиента мы добавляем компонент, известный как *член скорости* (velocity term) и являющийся накоплением всех прошлых градиентов. Однако существуют и более принципиальные подходы к решению этой проблемы, такие как различные структурные модификации стандартной рекурсивной нейросети. Вы познакомитесь с ними в главе 7.

С другой стороны, допустим, мы инициализировали w_3 очень большим значением, скажем, 1000. Тогда после шага n = 100 градиенты будут огромными, порядка 10^{300}. Это приведет к вычислительной неустойчивости, и Python выдаст такие значения, как `Inf` или `NaN` (Not a Number, не является числом). Это явление *взрывающегося градиента* (exploding gradient).

Взрыв градиентов может произойти из-за сложной поверхности функции потерь. Сложные невыпуклые поверхности очень распространены в глубоких нейронных сетях как из-за размерности входных данных, так и из-за большого количества параметров (весов), присутствующих в моделях. Рисунок 6.7 изображает поверхность функции потерь рекуррентной нейросети и демонстрирует наличие стен с очень высокой кривизной. Если метод оптимизации вступает в контакт с такой стеной, то градиенты будут взорваны или перескочат через оптимум, как показано сплошной линией. Это может привести либо к очень плохой минимизации потерь, либо к вычислительной неустойчивости, либо к тому и другому.

Простое решение, позволяющее избежать взрыва градиента в таких ситуациях, состоит в том, чтобы ограничить градиенты до разумного малого значения, когда они превышают некоторый порог. Пунктирная линия на рисунке показывает, что происходит, когда мы обрезаем градиент при небольшом значении. *Отсечение градиента* хорошо освещено в статье[1] «О сложности обучения рекуррентных нейронных сетей» под авторством Пашкану, Миколова и Бенджи.

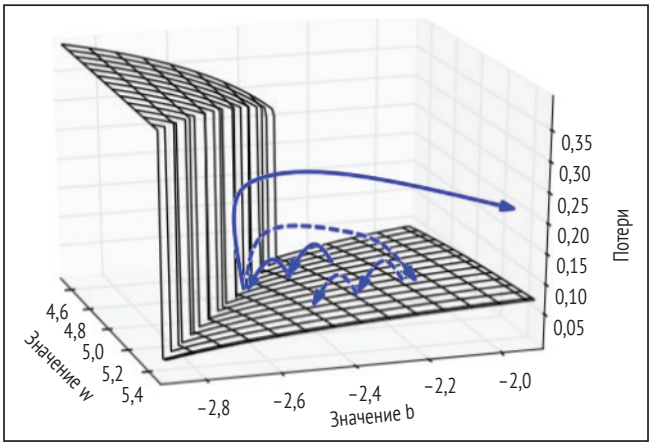


Рис. 6.7 ❖ Явление взрывающегося градиента.
Источник: статья «О сложности обучения рекуррентных нейронных сетей» Пашкану, Миколова и Бенджи

Далее мы обсудим различные способы прикладного применения рекуррентных нейросетей. Эти применения включают классификацию предложений, подписи к изображениям и машинный перевод. Но сначала вы познакомитесь с разделением рекуррентных сетей по нескольким категориям – один-к-одному, один-ко-многим, многие-к-одному и многие-ко-многим.

Применение рекуррентных нейросетей

До сих пор во время обсуждения рекуррентных нейросетей мы говорили о категории *один-к-одному* (one-to-one), где текущий выход зависит от текущего входа, а также от ранее наблюдаемой истории входов. Однако в действительности возможны ситуации, когда имеется только один выход для последовательности входов, последовательность выходов для одного входа и последовательность выходов для последовательности входов, причем последовательности могут иметь разную длину. В этом разделе мы рассмотрим несколько таких ситуаций.

[1] *Pascanu*, *Mikolov*, *and Bengio*. On the difficulty of training recurrent neural networks. International Conference on Machine Learning (2013): 1310–1318.

Один-к-одному

В сетях типа *один-к-одному* (один вход, один выход) текущий вход зависит от ранее наблюдаемых входов (рис. 6.8). Рекуррентные нейросети этого типа подходят для задач, где каждый вход дает выход, но выход зависит как от текущего входа, так и от истории входов, которые привели к текущему входу. Примером такой задачи является прогнозирование фондового рынка, где мы выводим значение для текущего входа, и этот выход также зависит от того, как вели себя предыдущие входы. Другим примером может быть *классификация сцены*, где каждый пиксель в изображении снабжен меткой, например *автомобиль*, *дорога* и *пешеход*. В некоторых случаях x_{t+1} может быть таким же, как y_t. Например, в задачах генерации текста ранее спрогнозированное слово становится входом для прогнозирования следующего слова. На рис. 6.8 изображена рекуррентная нейросеть типа один-к-одному.

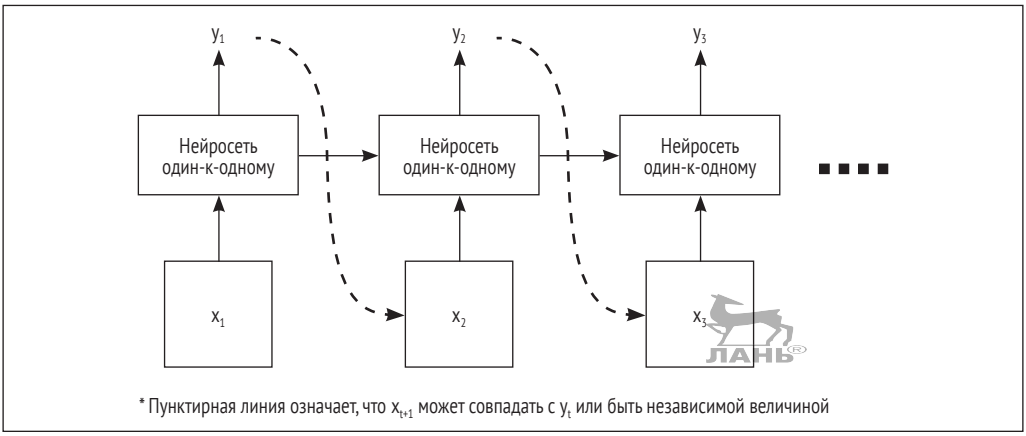


Рис. 6.8 ❖ Рекуррентная нейросеть один-к-одному, имеющая временны́е зависимости

Один-ко-многим

Рекуррентная нейросеть типа *один-ко-многим* (один вход, последовательность выходов) будет принимать один вход и выводить последовательность (рис. 6.9). Здесь мы предполагаем, что входные данные не зависят друг от друга. То есть нам не нужна информация о предыдущих входных данных, чтобы сделать прогноз о текущих входных данных. Тем не менее без рекуррентных связей не обойтись, потому что, хотя мы обрабатываем один вход, выход представляет собой последовательность значений, зависящих от предыдущих выходных значений. Примером практической задачи является генерация подписей к рисункам. Например, для заданного входного изображения будет составлена подпись из пяти или десяти слов. Другими словами, нейросеть будет продолжать предсказывать слова, пока не выведет значимую фразу, описывающую изображение.

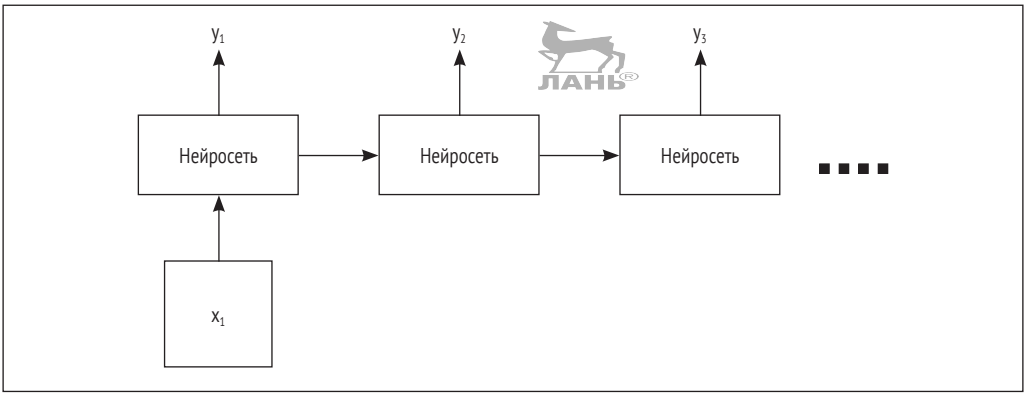


Рис. 6.9 ❖ Рекуррентная нейросеть типа *один-ко-многим*

Многие-к-одному

Нейросети типа *многие-к-одному* (последовательность на входе, один выход) принимают входную последовательность произвольной длины в качестве входа и создают один выход (рис. 6.10). Классификация предложений – одна из задач, успешно решаемых рекуррентными нейросетями такого типа. Предложение – это последовательность слов произвольной длины, поступающая на вход нейросети. На выходе получается прогноз принадлежности предложения к одному из заранее объявленных классов. Вот некоторые конкретные примеры классификации предложений:

- классификация рецензий на фильмы по уровню положительного или отрицательного отзыва (т. е. *анализ настроений*);
- классификация предложения в зависимости от того, что оно описывает (например, человек, объект или местоположение).

Другое применение нейросетей такого типа – классификация крупномасштабных изображений путем обработки только небольшого фрагмента изображения за раз и перемещения окна по всему изображению.

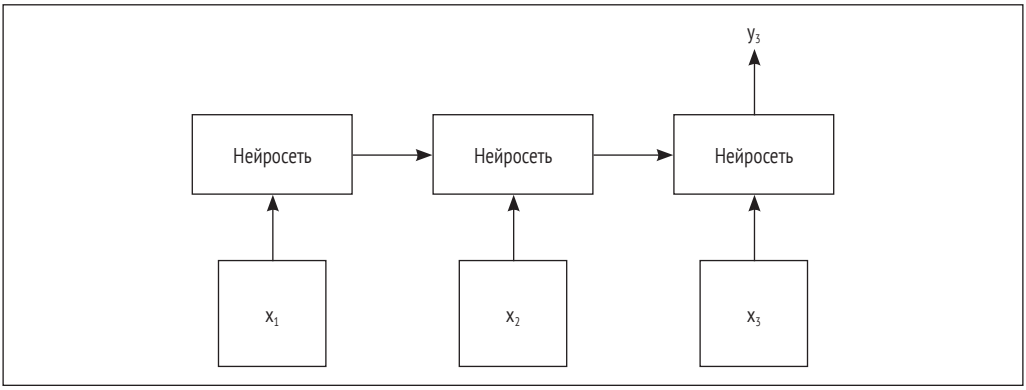


Рис. 6.10 ❖ Рекуррентная нейросеть типа *многие-к-одному*

Многие-ко-многим

Нейросети типа *многие-ко-многим* часто генерируют выходные последовательности произвольной длины из входов произвольной длины (рис. 6.11). Другими словами, входы и выходы не обязательно должны быть одинаковой длины. Это особенно полезно в машинном переводе, где мы переводим предложение с одного языка на другой. Как вы понимаете, одно предложение на определенном языке не всегда совпадает по длине с предложением на другом языке. Еще одним примером являются чат-боты, где нейросеть читает последовательность слов (т. е. запрос пользователя) и выводит последовательность слов – ответ.

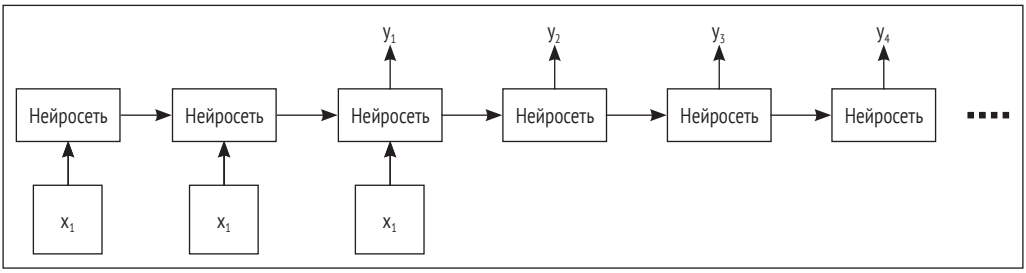


Рис. 6.11 ❖ Рекуррентная нейросеть типа *многие-ко-многим*

Отличительные особенности различных типов рекуррентных сетей перечислены в табл. 6.1.

Таблица 6.1. Различные типы рекуррентных сетей и их применение

Алгоритм	Описание	Применение
Один-к-одному	Принимают один вход и дают один выход. Текущий вход зависит от ранее наблюдаемых входных данных	Предсказания на фондовом рынке, классификация событий, генерация текста
Один-ко-многим	Принимают один вход и дают выход, состоящий из произвольного числа элементов	Генерация подписей к рисункам
Многие-к-одному	Принимают последовательность входных элементов и дают один выход	Классификация предложений (рассматривая одно слово как один вход)
Многие-ко-многим	Принимают последовательность произвольной длины в качестве входных данных и выводят последовательность произвольной длины	Машинный перевод, чат-боты

Генерация текста с помощью рекуррентной нейросети

Итак, давайте рассмотрим первый пример использования рекурсивной нейросети для интересной задачи. В этом упражнении мы будем использовать нейросеть для создания сказки! Это задача типа *один-к-одному*. Мы обучим однослойную нейросеть на коллекции сказок и попросим ее создать новый текст. Для обуче-

ния будем использовать небольшой текстовый корпус из 20 разных сказок. Это упражнение также продемонстрирует одно из важнейших ограничений RNN: отсутствие постоянной долговременной памяти. Полный код упражнения хранится в файле `rnn_language_bigram.ipynb` в папке `ch6`.

Определение гиперпараметров

Прежде всего определим несколько важных гиперпараметров нейросети:

- `num_unroll` – это количество предыдущих шагов, для которых развернут вход. Мы упоминали данный параметр в описании метода TBPTT. Чем выше это значение, тем больше предыдущих состояний учитывает нейросеть. Однако из-за эффекта исчезающего градиента выгода от запоминания предыдущих значений быстро теряется при высоких значениях `num_unroll` (скажем, выше 50). Обратите внимание, что увеличение `num_unroll` повышает требования к памяти;
- размеры пакетов обучающих, валидационных и тестовых данных. Увеличение размера пакетов часто приводит к лучшим результатам, так как нейросеть видит больше данных на каждом этапе оптимизации, но, как и в предыдущем случае, повышает требования к памяти;
- размерность входных, выходных и скрытых слоев. Увеличение размерности скрытого слоя обычно приводит к лучшей производительности. Однако обратите внимание, что в то же время происходит увеличение всех трех наборов весов (т. е. U, W и V), что приводит к высокой вычислительной нагрузке.

Сначала зададим длину развертывания и размеры пакета для тестирования:

```
num_unroll = 50
batch_size = 64
test_batch_size = 1
```

Затем зададим количество единиц в скрытом слое (мы будем использовать один скрытый слой RNN) и размеры входных и выходных данных:

```
hidden = 64
in_size,out_size = vocabulary_size,vocabulary_size
```

Распространение входов во времени для усеченного BPTT

Распространение входов во времени является важной частью процесса оптимизации TBPTT. Наш следующий шаг – определение того, как входы развернуты во времени.

Давайте рассмотрим пример, чтобы понять, как происходит развертывание:

Боб и Мэри пошли покупать цветы.

Предположим, что мы обрабатываем данные на уровне детализации символов. Кроме того, рассмотрим одну порцию данных и количество шагов для развертывания `num_unroll = 5`.

Сначала разобьем предложение на символы:

'Б', 'о', 'б', ' ', 'и', ' ', 'М', 'э', 'р', 'и', ' ', 'п', 'о', 'ш', 'л', 'и', ' ', 'п', 'о', 'ш', 'л', 'и', ' ', 'п', 'о', 'к', 'у', 'п', 'а', 'т', 'ь', ' ', 'ц', 'в', 'е', 'т', 'ы'

Если мы возьмем первые три пакета входов и выходов с распространением 5, это будет выглядеть так:

Вход	Выход
'Б','о','б',' ','и'	'о','б',' ','и',' '
' ','М','э','р','и'	'М','э','р','и',' '
' ','п','о','ш','л'	'п','о','ш','л','и'

В данном случае нейросеть видит относительно длинную последовательность данных за раз, в отличие от посимвольной обработки. Следовательно, она может хранить более длинные воспоминания о последовательности:

```
train_dataset, train_labels = [],[]
for ui in range(num_unroll):
    train_dataset.append(tf.placeholder(tf.float32,
        shape=[batch_size,in_size],name='train_dataset_ %d' %ui))
    train_labels.append(tf.placeholder(tf.float32,
        shape=[batch_size,out_size],name='train_labels_ %d' %ui))
```

Определение набора данных для валидации

Определим набор валидационных данных для измерения точности нейросети. Прогнозы, полученные на этом наборе данных, применяются в качестве показателя качества нейросети:

```
valid_dataset = tf.placeholder(tf.float32,
    shape=[1,in_size],name='valid_dataset')
valid_labels = tf.placeholder(tf.float32,
    shape=[1,out_size],name='valid_labels')
```

Мы формируем валидационный набор, выбирая более длинные тексты и извлекая часть сказки с самого конца. Вы без труда поймете детали реализации, поскольку код тщательно задокументирован.

Определение весов и смещений

Определим параметры весов и смещений нейросети:

- W_xh – веса между входами и скрытым слоем;
- W_hh – веса рекуррентных связей скрытого слоя;
- W_hy – веса между скрытым слоем и выходами.

```
W_xh = tf.Variable(tf.truncated_normal(
                    [in_size,hidden],stddev=0.02,
                    dtype=tf.float32),name='W_xh')
W_hh = tf.Variable(tf.truncated_normal([hidden,hidden],
                    stddev=0.02,
                    dtype=tf.float32),name='W_hh')
W_hy = tf.Variable(tf.truncated_normal(
                    [hidden,out_size],stddev=0.02,
                    dtype=tf.float32),name='W_hy')
```

Определение переменных состояния

Далее мы определим один из наиболее важных объектов, отличающих рекуррентные нейросети от нейронных сетей прямого распространения, – переменные состояния. Они, по сути, представляют собой память рекуррентной нейросети. Кроме того, они смоделированы как *необучаемые переменные* (untrainable variables) TensorFlow.

Сначала мы определим переменные состояния `prev_train_h` и `prev_valid_h`. Одна переменная сохраняет состояние нейросети во время обучения, а другая – во время проверки:

```
prev_train_h = tf.Variable(tf.zeros([batch_size,hidden],
               dtype=tf.float32),name='train_h',trainable=False)
               name='prev_h1',trainable=False)
prev_valid_h = tf.Variable(tf.zeros([1,hidden],dtype=tf.float32),
               name='valid_h',trainable=False)
```

Вычисление скрытых состояний и выходов с развернутыми входами

Далее мы определим вычисления скрытого слоя для каждого *развернутого входа* (unrolled input), ненормализованные оценки и прогнозы. Чтобы рассчитать выходные данные для каждого скрытого слоя, мы обрабатываем выходы скрытого состояния (т. е. `outputs` в коде), представляющие каждый развернутый элемент. Затем для всех шагов `num_unroll` рассчитываются ненормализованные прогнозы (также называемые логитами, или оценками) и прогнозы softmax:

```
# Присоединение вычисленного выхода RNN для
# каждого шага из num_unroll шагов.
outputs = list()

# Эта переменная итеративно используется для num_unroll шагов вычислений.
output_h = prev_train_h

# Вычисление выхода RNN для num_unroll шагов
# (как требуется для усеченного BPTT)
for ui in range(num_unroll):
        output_h = tf.nn.tanh(
            tf.matmul(tf.concat([train_dataset[ui],output_h],1),
                tf.concat([W_xh,W_hh],0))
        )
        outputs.append(output_h)
```

Затем рассчитаем ненормализованные прогнозы `y_scores` и нормализованные прогнозы `y_predictions` следующим образом:

```
# Получаем оценки и прогнозы для всех выходов RNN,
# которые имеются для num_unroll шагов.
y_scores = [tf.matmul(outputs[ui],W_hy) for ui in range(num_unroll)]
y_predictions = [tf.nn.softmax(y_scores[ui]) for ui in range(num_unroll)]
```

Расчет потерь

После того как получены прогнозы, мы вычисляем потерю `rnn_loss` как перекрестную энтропию между прогнозируемым и фактическим выходами. Обратите внимание, что мы сохраняем последний выход нейросети `output_h` в переменную `prev_train_h` с помощью операции `tf.control_dependencies(...)`, чтобы на следующей итерации начать с ранее сохраненного выхода в качестве исходного состояния:

```
# Убедимся, что перед вычислением ошибки
# переменная состояния обновлена значением последнего выхода RNN.
with tf.control_dependencies([tf.assign(prev_train_h,output_h)]):
    # Вычисляем перекрестную энтропию для всех прогнозов,
    # полученных за num_unroll шагов.
    rnn_loss = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=tf.concat(y_scores,0),
        labels=tf.concat(train_labels,0)
    ))
```

Сброс состояния в начале нового сегмента текста

Нам необходимо объявить операции сброса скрытого состояния. Сброс особенно необходим перед созданием нового фрагмента текста во время тестирования. В противном случае нейросеть продолжит создавать текст в зависимости от ранее созданного выхода, что приведет к получению сильно коррелированных результатов. Это плохо, потому что в конечном итоге нейросеть будет выводить одно и то же слово снова и снова. Однако все еще нет единого мнения по поводу полезности сброса состояния во время обучения. Тем не менее мы определим операции TensorFlow для сброса состояния как при валидации, так и при обучении:

```
# Сброс скрытых состояний
reset_train_h_op = tf.assign(prev_train_h,tf.zeros(
                            [batch_size,hidden],
                            dtype=tf.float32))
reset_valid_h_op = tf.assign(prev_valid_h,tf.zeros(
                            [1,hidden],dtype=tf.float32))
```

Расчет результата проверки

Далее, по аналогии с вычислением состояния, потери и прогноза при обучении, мы вычисляем состояние, потерю и прогноз при валидации:

```
# Вычисляем следующее состояние (только для 1 шага)
next_valid_state = tf.nn.tanh(tf.matmul(valid_dataset,W_xh) +
                              tf.matmul(prev_valid_h,W_hh))
# Вычисляем прогноз, используя выход состояния RNN,
# но перед этим присваиваем последний выход состояния RNN
# переменной состояния на фазе валидации.
# Поэтому вам следует убедиться, что вы выполнили операцию
# valid_predictions для обновления состояния
```

```
with tf.control_dependencies([tf.assign(prev_valid_h,next_valid_
state)]):
    valid_scores = tf.matmul(next_valid_state,W_hy)
    valid_predictions = tf.nn.softmax(valid_scores)
```

Расчет градиентов и оптимизация

Итак, мы определили потерю, теперь надо рассчитать градиенты и применить их. Мы воспользуемся методами стохастического градиентного спуска. В частности, используем TBPTT. В этом методе мы разворачиваем нейросеть во времени (аналогично тому, как развернули входные данные) и вычисляем градиенты, а затем сворачиваем рассчитанные градиенты, чтобы обновить веса нейросети. Кроме того, мы будем использовать адаптивный алгоритм Adam (AdamOptimizer), который сходится гораздо быстрее, чем стандартный стохастический градиентный спуск. При использовании алгоритма Adam обязательно применяйте небольшую скорость обучения (например, от 0,001 до 0,0001). Для предотвращения возможных градиентных взрывов применим отсечение градиентов:

```
rnn_optimizer = tf.train.AdamOptimizer(learning_rate=0.001)

gradients, v = zip(*rnn_optimizer.compute_gradients(rnn_loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
rnn_optimizer = rnn_optimizer.apply_gradients(zip(gradients, v))
```

Вывод сгенерированного фрагмента текста

Воспользуемся обученной моделью для генерации нового текста. Мы предсказываем слово, используем это слово в качестве следующего ввода, предсказываем другое слово и продолжаем так делать в течение нескольких временных шагов:

```
# Сохраняем предыдущее скрытое состояние на этапе тестирования.
prev_test_h = tf.Variable(tf.zeros([test_batch_size,hidden],
                                   dtype=tf.float32),name='test_h')

# Тестовый набор
test_dataset = tf.placeholder(tf.float32, shape=[test_batch_size,
                              in_size],name='test_dataset')

# Вычисляем скрытый выход для тестовых данных
next_test_state = tf.nn.tanh(tf.matmul(test_dataset,W_xh) +
                             tf.matmul(prev_test_h,W_hh)
                  )

# Обязательно обновляем скрытое состояние на этапе тестирования
# каждый раз после выдачи прогноза
with tf.control_dependencies([tf.assign(prev_test_h,next_test_state)]):
    test_prediction = tf.nn.softmax(tf.matmul(next_test_state,W_hy))

# Обратите внимание, что мы используем небольшие начальные значения
# при сбросе состояния на этапе тестирования,
# поскольку это добавляет вариации в сгенерированный текст
reset_test_h_op = tf.assign(prev_test_h,tf.truncated_normal(
                            [test_batch_size,hidden],stddev=0.01,
                            dtype=tf.float32))
```

Оценка качества текста

Познакомьтесь с примерами текста, сгенерированного нашей нейросетью. Мы запускали генерацию дважды, без развертывания ввода во времени и с развертыванием.

Без развертывания входа после 10 шагов (эпох) обучения мы получили такой результат:

```
    he the the the the the the the the the the the the the the the the the
the the the the the the the the the the the the the the the the the the
the the the the the the the the the the the the the the the the the the
the the the the the the the the the the the the the
    o the the the the the the the the the the the the the the the the the
the the the the the the the the the the the the the the the the the the
the the the the the the the the the the the the the the the the the the
the the the the the the the the the the the the the t
```

Используя развертывание входа во времени, после 10 шагов мы получаем следующий текст[1]:

```
god grant that our sister may be here, and then we shall be free.
when the maiden,who was standing behind the door watching, heard that
wish,
she came forth, and on this all the ravens were restored to their
human form again. and they embraced and kissed each other,
and went joyfully home whome, and wanted to eat and drink, and
looked for their little plates and glasses. then said one after
the other, who has eaten something from my plate. who has drunk
out of my little glass. it was a human mouth. and when the
seventh came to the bottom of the glass, the ring rolled against
his mouth. then he looked at it, and saw that it was a ring
belonging to his father and mother, and said, god grant that our
sister may be here, and then we shall be free.
```

```
(дай бог, чтобы наша сестра может быть здесь, и тогда мы будем свободны.
когда девушка, которая стояла за дверью и смотрела, услышала, что
пожелает,
она вышла, и на этом все вороны были возвращены к своему
человеческому облику снова. и они обнялись и поцеловали друг друга,
и радостно пошел домой, и хотел есть и пить, и
искал свои маленькие тарелки и стаканы. затем сказал один после
другой, который съел что-то из моей тарелки. кто выпил
из моего маленького стакана. это был человеческий рот. и когда
седьмое подошло к дну стакана, кольцо свернуло
его рот. затем он посмотрел на него и увидел, что это было кольцо,
принадлежащий его отцу и матери, и сказал, дай бог, чтобы наша
сестра может быть здесь, и тогда мы будем свободны.)
```

[1] Здесь и далее мы будем давать неуклюжий дословный перевод примеров на русский язык, чтобы сохранить стиль машинного текста «как есть», без редактирования человеком. – *Прим. перев.*

Нетрудно заметить, что развертывание ввода во времени работает намного лучше. Тем не менее даже в этом случае встречаются некоторые грамматические и орфографические ошибки. Это приемлемо, поскольку мы обрабатываем два символа одновременно.

Другое очевидное наблюдение заключается в том, что нейросеть пытается создать новый текст, комбинируя разные сказки, изученные ранее. Вначале говорится о воронах, а затем история превращается в нечто, похожее на сказку «Златовласка и три медведя», и речь идет о тарелках и о том, кто ел из тарелки. Затем повествование переходит к кольцу.

Это означает, что нейросеть научилась объединять истории и придумывать новые. Тем не менее мы можем улучшить эти результаты, введя лучшие модели обучения (например, LSTM) и лучшие методы поиска (например, лучевой поиск). Вы познакомитесь с ними в следующих главах.

Из-за сложности языка и меньшей репрезентативной мощности нейросети маловероятно, что в процессе обучения вы будете получать результаты в виде более-менее осмысленного текста. Поэтому мы выбрали несколько заранее сгенерированных текстов, чтобы донести свою мысль.

Приглядевшись к сгенерированному тексту, вы заметите, что нейросеть пытается повторять один и тот же фрагмент текста снова и снова. В продемонстрированном фрагменте первое предложение идентично последнему, и весь последующий текст тоже будет повторяться. Эта проблема возникает по причине исчезающего градиента и, как вы скоро увидите, становится более заметной с увеличением размера набора данных.

Поэтому вскоре мы поговорим об усовершенствованном варианте нейросети, менее подверженном этому недостатку.

Перплексия – измерение качества созданного текста

Недостаточно просто сгенерировать текст – нам также нужен способ измерить качество созданного текста. Один из способов заключается в измерении так называемой *перплексии*[1]. Образно говоря, перплексия показывает, насколько озадаченной (perplexed) была нейросеть при генерации выходной последовательности с учетом входных данных. То есть если потеря кросс-энтропии для входа x_i и соответствующего ему выхода y_i равна $l(x_i, y_i)$, то перплексия будет следующей:

$$p(x_i, y_i) = e^{l(x_i, y_i)}.$$

Используя это уравнение, мы можем вычислить среднюю перплексию для учебного набора данных размера N:

[1] *Перплексия* (perplexity – озадаченность, замешательство) – специфичная метрика, применяемая в языковом моделировании и отражающая степень неоднозначности при генерации слова. Например, при общем словаре в 100 000 слов первая модель выбирает следующее слово из 100 вариантов, а вторая – из 1000. Это означает, что первая модель «умнее», а ее перплексия меньше в 10 раз. Очевидно, что среднее качество текста в первом случае будет выше. – *Прим. перев.*

$$p(D_{train}) = (1/N)\sum_{i=1}^{N} p(x_i, y_i).$$

На рис. 6.12 изображены графики изменения перплексии по эпохам при обучении и валидации. Как видите, при обучении перплексия неуклонно снижается, а при валидации заметно колеблется. Это ожидаемо, потому что при валидации мы оцениваем прежде всего способность модели предсказывать новый текст, исходя из обучающих данных. Неудивительно, что в процессе валидации модель периодически «впадает в замешательство», оказываясь в ситуации выбора из относительно большого набора слов.

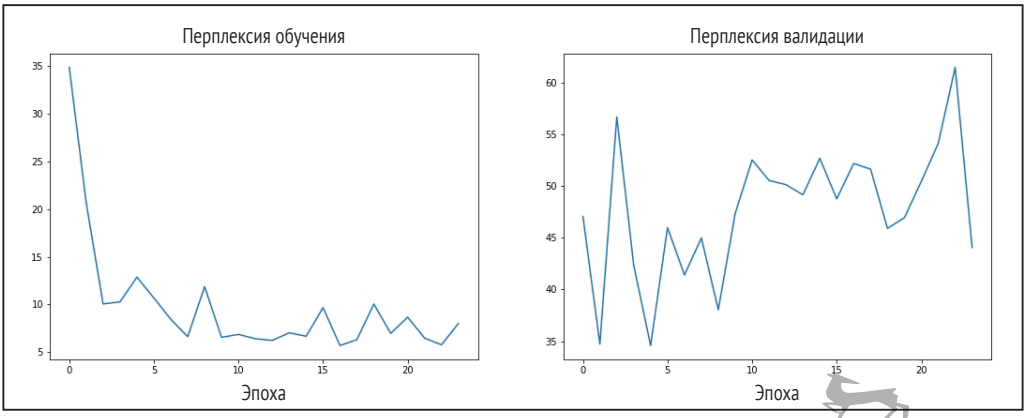


Рис. 6.12 ❖ Графики уровня перплексии при обучении и проверке

Один из способов улучшить результаты – добавить в рекуррентную нейросеть больше скрытых слоев, поскольку более глубокие модели дают лучшие результаты. В файле `rnn_language_bigram_multilayer.ipynb` в папке `ch6` представлен код готовой трехслойной рекуррентной нейросети. Мы предлагаем читателям исследовать его самостоятельно.

Теперь мы возвращаемся к вопросу, существуют ли варианты RNN, которые работают еще лучше. Например, существуют ли рекуррентные нейросети, которые более эффективно решают проблему исчезающего градиента? Давайте поговорим об одном таком варианте под названием RNN-CF.

Рекуррентные нейронные сети с контекстными признаками

Ранее мы обсуждали две важные проблемы, возникающие при обучении простой рекуррентной нейросети, – исчезновение градиента и взрыв градиента. Вы уже знаете, что можно предотвратить взрыв градиента и стабилизировать обучение при помощи такого простого трюка, как отсечение градиента. Однако решение проблемы исчезающего градиента требует гораздо больше усилий, потому что простого механизма масштабирования/отсечения в этом случае не существует. Следовательно, нам нужно изменить структуру нейросети, предоставив ей воз-

можность запоминать более длинные шаблоны в последовательностях данных. В статье Миколова и др. «Проблематика более долгой памяти в рекуррентных нейронных сетях» предложена рекуррентная нейросеть с *контекстными признаками* (RNN with Context Features, RNN-CF), позволяющая дольше запоминать данные.

RNN-CF компенсирует явление исчезающего градиента путем введения нового состояния и нового набора прямых и рекуррентных соединений. Другими словами, RNN-CF имеет два вектора состояния по сравнению со стандартной рекуррентной нейросетью, имеющей только один вектор. Идея состоит в том, что один вектор состояния меняется медленно, обеспечивая более долгую память, в то время как другой вектор может быстро изменяться, выполняя роль краткосрочной памяти.

Особенности устройства RNN-CF

По сути, мы добавляем в обычную RNN несколько параметров, чтобы помочь сохранению памяти в течение более длительного времени. Модификации включают введение нового вектора состояния в дополнение к обычному вектору состояния, присутствующему в стандартной модели RNN. В результате этого также появляется несколько прямых и рекуррентных наборов весов. На рис. 6.13 представлено сравнение RNN-CF и простой RNN на абстрактном уровне.

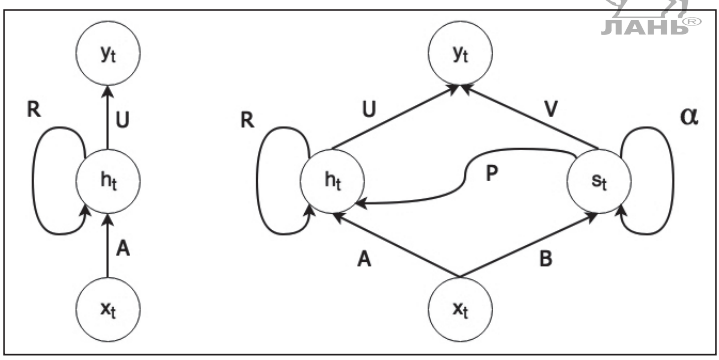


Рис. 6.13 ❖ Сравнение RNN и RNN-CF на абстрактном уровне

Как видно по рис. 6.13, RNN-CF имеет несколько дополнительных весов и слоев. Давайте внимательно разберемся, что они делают.

Во-первых, входные данные принимаются *двумя* скрытыми слоями, подобно обычному скрытому слою, существующему в RNN. Мы убедились, что использование только одного скрытого слоя не способствует сохранению долговременной памяти. Тем не менее можно продлить память, заставляя рекуррентную матрицу быть близкой к единице и удаляя нелинейность. Когда рекуррентная матрица близка к тождественной, без нелинейностей, любое изменение, происходящее с h, всегда должно быть вызвано изменениями в текущих входных данных. Другими словами, предыдущее состояние будет меньше влиять на изменение текущего состояния. Это приводит к изменению состояния медленнее, чем с плотной матрицей весов и нелинейностями. Таким образом, данное устройство нейросети помогает дольше сохранять память. Другая причина, по которой рекуррентная

матрица должна быть близка к 1, заключается в том, что когда веса близки к 1, такие элементы, как w^{n-1}, которые появляются при дифференцировании, не исчезают и не взрываются. Однако если мы попытаемся обойтись совсем без скрытого слоя с нелинейностью, градиент никогда не уменьшится. Уменьшая градиент, мы опираемся на тот факт, что градиенты, создаваемые более старыми входными данными, должны оказывать меньшее влияние, чем более новые. Но при этом нам нужно распространять градиенты во времени до начала ввода. Это дорого в вычислительном отношении. Чтобы получить лучшее из обоих миров, мы оставляем оба этих слоя: стандартный слой состояния RNN h_t, который может быстро изменяться, а также слой контекстных признаков s_t, который изменяется медленнее. Новый слой называется *слоем контекста* и помогает сохранять долговременную память. Правила обновления для RNN-CF следующие. Обратите внимание, что вы не видите, как s_{t-1} умножается на единичную матрицу, потому что $Is_{t-1} = s_{t-1}$:

$$s_t = (1 - \alpha)Bx_t + \alpha s_{t-1},$$
$$h_t = \sigma(Ps_t + Ax_t + Rh_{t-1}),$$
$$y_t = softmax(Uh_t + Vs_t).$$

Специфические обозначения, относящиеся к RNN-CF, приведены в табл. 6.2.

Таблица 6.2. Условные обозначения параметров RNN-CF

Обозначение	Расшифровка
x_t	Текущий вход
h_t	Текущий вектор состояния
y_t	Текущий выход
s_t	Текущий вектор контекстных признаков
A	Матрица весов между x_t и h_t
B	Матрица весов между x_t и s_t
R	Рекуррентные связи h_t
α	Константа, определяющая вклад s_{t-1} в s_t
P	Веса связей h_t и s_t
U	Матрица весов между h_t и y_t
V	Матрица весов между s_t и y_t

Реализация RNN-CF

Итак, вы знаете, что RNN-CF содержит дополнительный вектор состояния, помогающий предотвратить исчезновение градиентов. Далее мы обсудим реализацию RNN-CF. В дополнение к `hidden` (h_t), `W_xh` (A), `W_hh` (R) и `W_hy` (U), которые были в обычной реализации RNN, теперь нам нужно еще три дополнительных набора весов – B, P и V. Кроме того, мы объявим новую переменную `hidden_context`, которая будет также содержать s_t.

Определение гиперпараметров RNN-CF

Начнем с определения как новых, так и уже знакомых гиперпараметров. Один новый гиперпараметр определяет количество нейронов в слое контекстных признаков s_t, где `alpha` представляет параметр α в уравнении.

```
hidden_context = 64
alpha = 0.9
```

Входные и выходные заполнители

Как и в случае стандартного RNN, мы сначала объявляем заполнители, которые принимают входные и выходные данные для обучения, входные и выходные данные для валидации и входные данные для тестирования:

```
# Обучающий набор данных.
# Мы используем развертывание во времени
train_dataset, train_labels = [],[]
for ui in range(num_unroll):
    train_dataset.append(tf.placeholder(tf.float32,
                         shape=[batch_size,in_size],
                         name='train_dataset_ %d' %ui))
    train_labels.append(tf.placeholder(tf.float32,
                        shape=[batch_size,out_size],
                        name='train_labels_ %d' %ui))

# Валидационный набор данных.
valid_dataset = tf.placeholder(tf.float32,
                               shape=[1,in_size],name='valid_dataset')
valid_labels = tf.placeholder(tf.float32,
                              shape=[1,out_size],name='valid_labels')

# Тестовый набор данных.
test_dataset = tf.placeholder(tf.float32,
                              shape=[test_batch_size,in_size],
                              name='save_test_dataset')
```

Объявление весов RNN-CF

Здесь мы определяем веса, необходимые для вычислений в RNN-CF. Как следует из табл. 6.2, нам требуется шесть наборов весов (A, B, R, P, U и V). Как вы помните, в обычной реализации RNN у нас было только три набора весов.

```
# Веса между входами и h.
A = tf.Variable(tf.truncated_normal([in_size,hidden],
                stddev=0.02,dtype=tf.float32),name='W_xh')
B = tf.Variable(tf.truncated_normal([in_size,hidden_context],
                stddev=0.02,dtype=tf.float32),name='W_xs')

# Веса между h и h.
R = tf.Variable(tf.truncated_normal([hidden,hidden],
                stddev=0.02,dtype=tf.float32),name='W_hh')
P = tf.Variable(tf.truncated_normal([hidden_context,hidden],
                stddev=0.02,dtype=tf.float32),name='W_ss')

# Веса между h и y.
U = tf.Variable(tf.truncated_normal([hidden,out_size],
                stddev=0.02,dtype=tf.float32),name='W_hy')
V = tf.Variable(tf.truncated_normal([hidden_context,
                                     out_size],stddev=0.02,
                                     dtype=tf.float32),
                                     name='W_sy')
```

```
# Переменные состояния для обучающих данных
prev_train_h = tf.Variable(tf.zeros([batch_size,hidden],
                           dtype=tf.float32),
                           name='train_h',trainable=False)
prev_train_s = tf.Variable(tf.zeros([batch_size,hidden_context],
                           dtype=tf.float32),name='train_s',
                           trainable=False)

# Переменные состояния для валидационных данных
prev_valid_h = tf.Variable(tf.zeros([1,hidden],dtype=tf.float32),
                           name='valid_h',trainable=False)
prev_valid_s = tf.Variable(tf.zeros([1,hidden_context],
                           dtype=tf.float32),
                           name='valid_s',trainable=False)

# Переменные состояния для тестовых данных
prev_test_h = tf.Variable(tf.zeros([test_batch_size,hidden],
                          dtype=tf.float32),
                          name='test_h')
prev_test_s = tf.Variable(tf.zeros([test_batch_size,hidden_context],
                          dtype=tf.float32),name='test_s')
```

Переменные и операции для поддержки скрытых и контекстных состояний

Здесь мы объявляем переменные состояния RNN-CF. В дополнение к состоянию h_t, которое мы имели в обычной RNN, нам нужно иметь отдельное состояние s_t для контекстных признаков. Всего у нас будет шесть переменных состояния. Три переменные состояния будут хранить вектор состояния h_t во время обучения, валидации и тестирования, а три другие переменные будут хранить на тех же этапах вектор состояния s_t:

```
# Переменные состояния для обучающих данных
prev_train_h = tf.Variable(tf.zeros([batch_size,hidden],
                           dtype=tf.float32),
                           name='train_h',trainable=False)
prev_train_s = tf.Variable(tf.zeros([batch_size,hidden_context],
                           dtype=tf.float32),name='train_s',
                           trainable=False)

# Переменные состояния для данных валидации
prev_valid_h = tf.Variable(tf.zeros([1,hidden],dtype=tf.float32),
                           name='valid_h',trainable=False)
prev_valid_s = tf.Variable(tf.zeros([1,hidden_context],
                           dtype=tf.float32),
                           name='valid_s',trainable=False)

# Переменные состояния для тестовых данных
prev_test_h = tf.Variable(tf.zeros([test_batch_size,hidden],
                                   dtype=tf.float32),
                                   name='test_h')
prev_test_s = tf.Variable(tf.zeros([test_batch_size,hidden_context],
                                   dtype=tf.float32),name='test_s')
```

Далее мы объявляем операции сброса состояний между этапами:

```
reset_prev_train_h_op = tf.assign(prev_train_h,tf.zeros([batch_size,
                        hidden], dtype=tf.float32))
reset_prev_train_s_op = tf.assign(prev_train_s,tf.zeros([batch_size,
                        hidden_context],dtype=tf.float32))

reset_valid_h_op = tf.assign(prev_valid_h,tf.zeros([1,hidden],
                   dtype=tf.float32))
reset_valid_s_op = tf.assign(prev_valid_s,tf.zeros([1,hidden_context],
                   dtype=tf.float32))

# Вводим тестовые состояния с добавлением шума
reset_test_h_op = tf.assign(prev_test_h,tf.truncated_normal(
                           [test_batch_size,hidden],
                           stddev=0.01,
                           dtype=tf.float32))
reset_test_s_op = tf.assign(prev_test_s,tf.truncated_normal(
                           [test_batch_size,hidden_context],
                           stddev=0.01,dtype=tf.float32))
```

Вычисление выхода

Объявив все входные данные, переменные и векторы состояния, мы теперь можем вычислить выход RNN-CF в соответствии с уравнениями в предыдущем разделе. Сначала мы инициализируем векторы состояний нулевыми значениями. Затем разворачиваем наши входные данные для фиксированного набора временных шагов (в соответствии с требованиями BPTT) и отдельно вычисляем ненормализованные выходные данные (иногда называемые логитами, или оценками) для каждого из этих развернутых шагов. Потом мы объединяем все значения *y*, относящиеся к каждому развернутому временному шагу, и вычисляем среднюю потерю всех этих записей относительно истинной метки:

```
# Значения оценок (ненормализованные) и прогнозов (нормализованные).
y_scores, y_predictions = [],[]

# Итеративно используются в течение
# num_unroll шагов вычислений.
next_h_state = prev_train_h
next_s_state = prev_train_s

# Добавляем вычисленные выходы состояния RNN
# для каждого шага на протяжении num_unroll шагов.
next_h_states_unrolled, next_s_states_unrolled = [],[]

# Вычисление выхода RNN для num_unroll шагов
# (как требуется для TBPTT)
for ui in range(num_unroll):
    next_h_state = tf.nn.tanh(
        tf.matmul(tf.concat([train_dataset[ui],prev_train_h,
                  prev_train_s],1),
                  tf.concat([A,R,P],0))
    )
    next_s_state = (1-alpha)*tf.matmul(train_dataset[ui],B) +
                   alpha * next_s_state
```

```
    next_h_states_unrolled.append(next_h_state)
    next_s_states_unrolled.append(next_s_state)

# Получаем оценки и прогнозы для всех выходов RNN,
# произведенных за num_unroll шагов
y_scores = [tf.matmul(next_h_states_unrolled[ui],U) +
            tf.matmul(next_s_states_unrolled[ui],V)
             for ui in range(num_unroll)]
y_predictions = [tf.nn.softmax(y_scores[ui]) for ui in range(num_unroll)]
```

Вычисление потери

Здесь мы определяем вычисление потери RNN-CF. Эта операция идентична той, которую мы определили выше для варианта обычной RNN, и выглядит следующим образом:

```
# Перед вычислением потери обновляем переменную состояния
# последним значением состояния, которое мы получили.
with tf.control_dependencies([tf.assign(prev_train_s, next_s_state),
                              tf.assign(prev_train_h,next_h_state)]):
    rnn_loss = tf.reduce_mean(
               tf.nn.softmax_cross_entropy_with_logits_v2(
               logits=tf.concat(y_scores,0),
               labels=tf.concat(train_labels,0)
    ))
```

Расчет выхода на этапе валидации

Аналогично вычислению выходных данных во время обучения, мы также рассчитываем выходные данные на этапе валидации. Однако не развертываем входные данные во времени, как делали это для обучающих данных, поскольку во время прогнозирования развертывание не требуется:

```
# Логический вывод при валидации
# (очень похож на логический вывод при обучении)
# Вычисляем следующее допустимое состояние (только для 1 шага)
next_valid_s_state = (1-alpha) * tf.matmul(valid_dataset,B) +
alpha * prev_valid_s
next_valid_h_state = tf.nn.tanh(tf.matmul(valid_dataset,A) +
tf.matmul(prev_valid_s, P) +
tf.matmul(prev_valid_h,R))
# Вычисляем прогноз на основе выхода слоя состояния RNN.
# Но перед этим присваиваем последний выход слоя состояния
# переменной состояния этапа валидации.
# Поэтому обязательно надо выполнить операцию rnn_valid_loss
# для обновления состояния при валидации
with tf.control_dependencies([tf.assign(prev_valid_s,
next_valid_s_state),
tf.assign(prev_valid_h,next_valid_h_
state)]):
valid_scores = tf.matmul(prev_valid_h, U) + tf.matmul(
prev_valid_s, V)
valid_predictions = tf.nn.softmax(valid_scores)
```

Вычисление тестового выхода

Теперь мы можем определить вычисления для генерации новых тестовых данных:

```
# Логика вывода, относящаяся к тестовым данным

# Вычисление скрытого выхода для тестовых данных
next_test_s = (1-alpha)*tf.matmul(test_dataset,B)+ alpha*prev_test_s

next_test_h = tf.nn.tanh(
    tf.matmul(test_dataset,A) + tf.matmul(prev_test_s,P) +
    tf.matmul(prev_test_h, R)
                        )

# Скрытый слой состояния необходимо обновить
# каждый раз перед вычислением прогноза
with tf.control_dependencies([tf.assign(prev_test_s,next_test_s),
                              tf.assign(prev_test_h,next_test_h)]):
    test_prediction = tf.nn.softmax(
        tf.matmul(prev_test_h,U) + tf.matmul(prev_test_s,V)
    )
```

Вычисление градиентов и оптимизация

Здесь мы используем оптимизатор, чтобы минимизировать потери, идентичные тому, что мы сделали для обычного RNN:

```
rnn_optimizer = tf.train.AdamOptimizer(learning_rate=.001)

gradients, v = zip(*rnn_optimizer.compute_gradients(rnn_loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
rnn_optimizer = rnn_optimizer.apply_gradients(zip(gradients, v))
```

Текст, созданный с помощью RNN-CF

Давайте сравним тексты, сгенерированные RNN и RNN-CF, как качественно, так и количественно. Сначала мы сравним результаты, полученные с использованием 20 учебных документов. После этого увеличим количество обучающих документов до 100, чтобы проверить, способны ли RNN и RNN-CF усваивать большие объемы данных и помогает ли это выводить текст более высокого качества.

Сначала мы сгенерируем текст после обучения сети RNN-CF только на 20 документах:

```
the king's daughter, who had
no more excuses left to make. they cut the could not off, and her his
first rays of life in the garden,
and was amazed to see with the showed to the grown mighted and the
seart the answer to star's brothers, and seeking the golden apple, we
flew over the tree to the seadow where her
heard that he could not have
discome.

emptied him by him. she himself 'well i ston the fire struck it was
and said the youth, farm of them into the showed to shudder, but here
```

```
and said the fire himself 'if i could but the youth, and thought that
is that shudder.'
'then, said he said 'i will by you are you, you.' then the king, who
you are your
wedding-mantle. you are you are you
bird in wretch me. ah. what man caller streep them if i will bed.
the youth
begged for a hearing, and said 'if you will below in you to be your
wedding-mantle.' 'what.' said he, 'i shall said 'if i hall by you are
you

bidden it i could not have

дочь царя, которая имела
больше никаких оправданий не осталось. они отрезать не могли, а ей его
первые лучи жизни в саду,
и был поражен, увидев с показом взрослым и тому
ищем ответ братьям звезды и ищем золотое яблоко, мы
перелетел через дерево на лужайку, где ее
слышал, что он не мог иметь
discome.

опустошил его им. она сама 'хорошо, я зажег огонь
и сказал молодежи, ферма из них в шоу вздрогнуть, но здесь
и сказал сам огонь 'если бы я мог, кроме молодежи, и подумал, что
это дрожь.'
'Тогда, сказал он, сказал 'я буду с тобой, ты, ты.' тогда король, который
ты твой
свадьба мантии. ты ты ты
Птица в беде меня. ах. какой человек звонил бы им, если я буду спать.
молодежь
попросил о слушании и сказал: 'Если вы будете ниже в вас быть вашим
свадьба мантии.' 'что.' сказал он, 'я должен сказать, 'если я зал за вами
вы

запретил это я не мог иметь
```

С точки зрения качества текста не видно большой разницы по сравнению со стандартной RNN. Давайте подумаем, почему RNN-CF не демонстрирует заметного преимущества. В статье «Проблематика более долгой памяти в рекуррентных нейронных сетях» Миколов и его соавторы предполагают следующее:

> *«Когда количество стандартных скрытых единиц является достаточным для захвата текущих паттернов, изучение саморекуррентных весов больше не кажется важным».*

Таким образом, если количество скрытых блоков достаточно велико, RNN-CF не имеет существенного преимущества перед стандартными RNN. Наверное, все дело именно в этом. Мы используем 64 скрытых нейрона и относительно небольшой корпус, и этого может оказаться достаточно, чтобы сгенерировать историю вполне на уровне способностей RNN.

Давайте проверим, действительно ли увеличение объема данных помогает RNN-CF работать лучше. Для нашего примера мы увеличим количество документов до 100 после обучения в течение примерно 50 эпох.

Так выглядит выход стандартной рекуррентной нейросети:

```
they were their dearest and she she told him to stop crying to the king's son they were their
dearest and she she told him to stop crying to the king's son they were their dearest and she
she told him to stop crying to the king's son they were their dearest and she she told him to
stop crying to the king's son they were their dearest and she she told him to stop
```

```
они были их возлюбленными и она она сказала ему прекратить плакать сыну короля они были их
возлюбленными и она она сказала ему прекратить плакать сыну короля они были их возлюбленными
и она она сказала ему прекратить плакать сыну короля они были их возлюбленными и она она
сказала ему прекратить плакать сыну короля они были их возлюбленными и она она сказала ему
прекратить
```

Можно сделать вывод, что результат RNN стал хуже по сравнению с текстом, полученным на меньших обучающих данных. Наличие большого количества данных и недостаточная пропускная способность модели неблагоприятно влияют на стандартные RNN, что приводит к выводу текста низкого качества.

Далее приведен пример вывода нейросети RNN-CF. Как видите, с точки зрения вариативности текста RNN-CF проделала гораздо лучшую работу, чем стандартная RNN:

```
then they could be the world. not was now from the first for a set out of his pocket, what is
the world. then they were all they were forest, and the never yet not
rething, and took the
children in themselver to peard, and then the first her. then the was in the first, and that
he was to the first, and that he was to the kitchen, and said, and had took the
children in the mountain, and they were hansel of the fire, gretel of they were all the fire,
goggle-eyes and all in the moster. when she had took the
changeling the little elves, and now ran into them, and she bridge away with the witch form,
and their father's daughter was that had neep himselver in the horse, and now they lived them
himselver to them, and they were am the marriage was all they were and all of the marriage was
anger of the forest, and the manikin was laughing, who had said they had not know, and took
the
children in themselver to themselver and they lived them himselver to them
```

```
тогда они могут быть миром. Не было теперь с самого начала для набора из своего кармана, что
такое мир. тогда они были все, что они были лесом, и никогда еще не
и взял
дети в себе сами по себе, а потом первые ее. затем был в первом, и что он был в первом, и что
он был на кухне, и сказал, и взял
дети в горе, и они были ганзелями огня, гретель их всех были огнем, глазами с очками и всем
в материи. когда она взяла
изменил маленьких эльфов, и теперь столкнулся с ними, и она перешла прочь с формой ведьмы,
и у дочери их отца был тот самый сын в лошади, и теперь они жили им самим с ними, и они были
им, брак был всем, чем они были, и весь брак был гневом леса, и манекен смеялся, который
сказал, что они не знали, и взял дети в себе, и они жили в них сами
```

Похоже, что при достаточном количестве данных RNN-CF действительно превосходит стандартные RNN. Если построить график изменений перплексии (рис. 6.14), то можно сделать вывод, что при обучении как RNN-CF, так и стандартная RNN не показывают существенных различий. Наконец, на графике перплексии при валидации мы видим, что RNN-CF демонстрирует меньший размах колебаний по сравнению со стандартной RNN.

Мы можем сделать один важный вывод: когда у нас было меньшее количество данных, стандартная нейросеть, вероятно, подвергалась переобучению. Иными словами, RNN, вероятно, запомнила данные как есть, вместо того чтобы попытаться изучить более общие закономерности, присутствующие в данных. Когда RNN перегружена данными и слишком долго обучается (скажем, около 50 эпох), ее склонность к переобучению становится более заметной. Качество создаваемого текста снижается, и возникают большие колебания перплексии при валидации. С другой стороны, RNN-CF демонстрирует более сбалансированное поведение как с небольшими, так и с большими объемами данных.

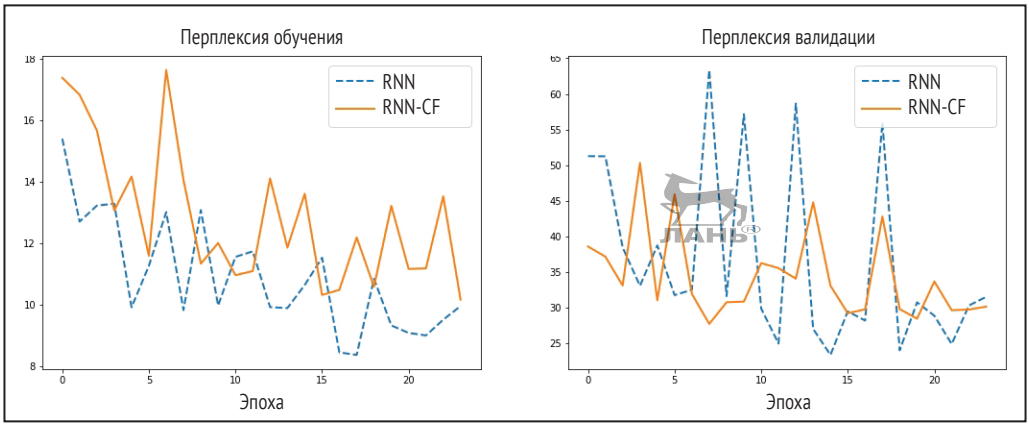


Рис. 6.14 ❖ Графики изменения перплексии при обучении и валидации

Заключение

В этой главе мы рассмотрели рекуррентные нейронные сети, которые отличаются от обычных нейронных сетей с прямой связью и лучше подходят для обработки последовательных данных произвольной длины. Рекуррентные нейросети могут быть реализованы в различных формах: «один-к-одному» (генерация текста), «многие-к-одному» (последовательная классификация изображений), «один-ко-многим» (подписи к изображениям) и «многие-ко-многим» (машинный перевод).

В частности, мы обсуждали, как получить рекуррентную нейросеть из структуры наподобие сетей прямого распространения. Мы определили входные и выходные последовательности и разработали граф вычислений, представляющий последовательность входов и выходов. Этот граф может быть реализован последовательностью копий функций, применяемых к каждому отдельному кортежу ввода-вывода в последовательности. Затем, обобщив эту модель для любого заданного единственного временного шага t в последовательности, мы смогли прийти к базовому вычислительному графу рекуррентной нейросети. Мы рассмотрели точные уравнения и обновили правила, применяемые для вычисления скрытого состояния и выхода.

Далее обсудили проблемы обучения рекуррентной нейросети при использовании обратного распространения во времени. Мы разобрались, как работает BPTT

и почему мы не можем использовать стандартное обратное распространение для обучения рекуррентных нейросетей. Мы рассмотрели две важные практические проблемы, возникающие при обучении по методу BPTT, – исчезающий градиент и взрывающийся градиент – и как их можно решить на базовом уровне.

Затем перешли к практическому применению RNN. Мы обсудили четыре основные категории RNN. Архитектура «один-к-одному» применяется для таких задач, как генерация текста, классификация сцен и маркировка видеокадров. Архитектуры «многие-к-одному» используются для анализа настроений, где мы обрабатываем слова/фразы по одному слову за раз, по сравнению с обработкой предложения целиком, рассмотренной нами в предыдущей главе. Архитектуры «один ко многим» распространены в задачах с подписями, где мы отображаем одно изображение в предложение произвольной длины, описывающее изображение. Архитектура «многие-ко-многим» используется для задач машинного перевода.

Далее мы рассмотрели интересное приложение рекуррентной нейросети – генерацию сказок. Для обучения нейросети мы использовали корпус сказок. В частности, разбили тексты сказок на биграммы, т. е. блоки из двух символов. Мы обучили нейросеть, подавая на вход набор биграмм, выбранных из истории в качестве входных данных, и используя эти же биграммы в качестве выходных данных. Затем мы оптимизировали модель, добиваясь максимальной точности прогнозирования следующей биграммы. Следуя этой процедуре, мы попросили нейросеть создать другую сказку и сделали два важных вывода насчет полученных результатов:

- развертывание ввода во времени фактически помогает получить более долгую память;
- рекуррентные нейросети даже при развертывании могут хранить только ограниченный объем долговременной памяти.

Поэтому мы рассмотрели вариант нейросети, способный захватывать в память более длинные последовательности. Эта улучшенная архитектура под названием RNN-CF имеет два разных уровня: скрытый уровень, т. е. обычный скрытый уровень, встречающийся в простых RNN, и контекстный уровень для сохранения долговременной памяти. Мы обнаружили, что наличие дополнительного контекстного уровня мало помогает при работе с небольшим набором данных, так как у нас был довольно сложный скрытый слой в нашей RNN, и без того работающий достаточно хорошо. В то же время контекстный слой дает немного лучшие результаты, если использовать больше данных.

В следующей главе мы обсудим более мощную модель рекуррентной нейросети, известную как сеть с долгой краткосрочной памятью, которая значительно уменьшает отрицательный эффект исчезающего градиента и дает гораздо лучшие результаты.

Глава 7

Сети с долгой краткосрочной памятью

В этой главе мы обсудим более продвинутый вариант рекуррентной нейросети, известный как *сеть с долгой краткосрочной памятью*[1] (long short-term memory, LSTM). Такие сети широко используются во многих последовательных задачах, включая прогнозирование фондового рынка, языковое моделирование и машинный перевод, и работают лучше, чем другие последовательные модели, особенно при наличии больших объемов данных. LSTM значительно меньше страдают от эффекта исчезающего градиента, который мы упоминали в предыдущей главе.

Основное практическое ограничение, налагаемое исчезающим градиентом, заключается в том, что он не позволяет модели изучать долгосрочные зависимости. Однако LSTM имеют возможность хранить память дольше, чем обычные рекуррентные нейросети, – в течение сотен временных шагов. В отличие от обычных RNN, поддерживающих только одно скрытое состояние, LSTM имеют гораздо больше параметров, а также лучший контроль над тем, какую память хранить и что отбрасывать на данном этапе обучения. Например, RNN не могут решить, какую память хранить, а какую отбрасывать, так как скрытое состояние принудительно обновляется на каждом этапе обучения.

В данной главе мы обсудим, что такое LSTM на абстрактном уровне и как устройство подобной нейросети позволяет хранить долгосрочные зависимости. Затем перейдем к рассмотрению математической структуры LSTM и обсудим пример, демонстрирующий, почему каждое вычисление имеет значение. Мы также сравним LSTM с обычными рекуррентными нейросетями и увидим, что у первых гораздо сложная архитектура, позволяющая превзойти конкурентов в последовательных задачах. Новый взгляд на проблему исчезающего градиента и изучение наглядного примера помогут вам понять, как LSTM решают эту проблему.

Затем мы обсудим несколько методов, которые разработаны для улучшения прогнозов, выдаваемых стандартными LSTM. Например, создание нескольких прогнозов одновременно вместо последовательной выдачи одного за другим может помочь улучшить качество генерируемых прогнозов. Мы также рассмотрим двунаправленные LSTM (BiLSTM), обладающие расширенными возможностями для поиска шаблонов, присутствующих в последовательности.

---

[1] На практике такой длинный термин, особенно в русском переводе, конечно же, не применяется. Обычно ограничиваются аббревиатурой LSTM. – *Прим. перев.*

Наконец, обсудим два последних варианта LSTM. Сначала мы разберем *замочные скважины* (peephole connections), которые улучшают работу LSTM за счет ввода дополнительных параметров и информации в гейты. Далее обсудим *управляемые рекуррентные ячейки* (gated recurrent units, GRU), которые приобретают все большую популярность, поскольку имеют гораздо более простую структуру по сравнению с LSTM, но при этом не снижают качество модели.

Устройство и принцип работы LSTM

Давайте начнем с изучения процессов, происходящих в LSTM. Вы увидите, что помимо состояний в LSTM применяется особый механизм *гейтов* для управления движением информации внутри ячейки. Затем мы проработаем подробный пример и посмотрим, как гейты и состояния работают на различных этапах, в конечном итоге приводя к желаемому результату. Наконец, мы сравним LSTM со стандартными RNN и определим основные отличия.

Что такое LSTM?

LSTM можно рассматривать как усложненную рекуррентную нейросеть, состоящую из пяти основных элементов:

- ***состояние ячейки*** (cell state) – это внутреннее состояние ячейки (то есть памяти) LSTM;
- ***скрытое состояние*** (hidden state) – это внешнее скрытое состояние, используемое для расчета прогнозов;
- ***входной гейт*** (input gate) – определяет, какая часть текущего входа отправляется в состояние ячейки;
- ***забывающий гейт*** (forget gate) – определяет, какая часть предыдущего состояния ячейки отправляется в текущее состояние;
- ***выходной гейт*** (output gate) – определяет, какая часть состояния ячейки выводится в скрытое состояние.

Мы можем представить RNN в виде ячеек следующим образом. Ячейка выводит некоторое состояние, которое нелинейно зависит от предыдущего состояния ячейки и текущего входа. Однако в RNN состояние ячейки обязательно изменяется при каждом новом входе. Это приводит к постоянному изменению состояния ячеек RNN. Такое поведение совершенно нежелательно для хранения долгосрочных зависимостей.

LSTM способны решать, когда заменить, обновить или забыть информацию, хранящуюся в каждом нейроне состояния ячейки. Другими словами, LSTM обладают способностью сохранения состояния ячейки без изменений, что дает им возможность при необходимости хранить долгосрочные зависимости.

Эта способность достигается путем введения управляющего механизма – так называемых *гейтов*. У LSTM есть гейты для каждой операции, которую должна выполнить ячейка. Гейты являются непрерывными (часто сигмоидальные функции) на интервале между 0 и 1, где 0 означает, что информация вообще не проходит через гейт, и 1 означает, что вся информация проходит без ограничений.

LSTM использует один такой гейт для каждого нейрона в ячейке. Как упоминалось выше, от гейтов зависит следующее:

- какая часть текущей информации записывается в состояние ячейки (входной гейт);
- какую часть предыдущего состояния ячейки следует забыть (забывающий гейт);
- какая часть информации выводится в окончательное скрытое состояние из состояния ячейки (выходной гейт).

На рис. 7.1 изображено абстрактное представление LSTM. Каждый гейт решает, какая часть различных данных – например, текущий ввод, предыдущее скрытое состояние или предыдущее состояние ячейки – поступает в окончательное скрытое состояние или состояние ячейки. Толщина каждой линии показывает, сколько информации течет от/к этим гейтам в некоторой гипотетической нейросети. Например, на рис. 7.1 вы можете видеть, что входной гейт допускает больше информации от текущего входа, чем от предыдущего конечного скрытого состояния. В свою очередь, забывающий гейт пропускает больше информации от предыдущего конечного скрытого состояния, чем от текущего входа.

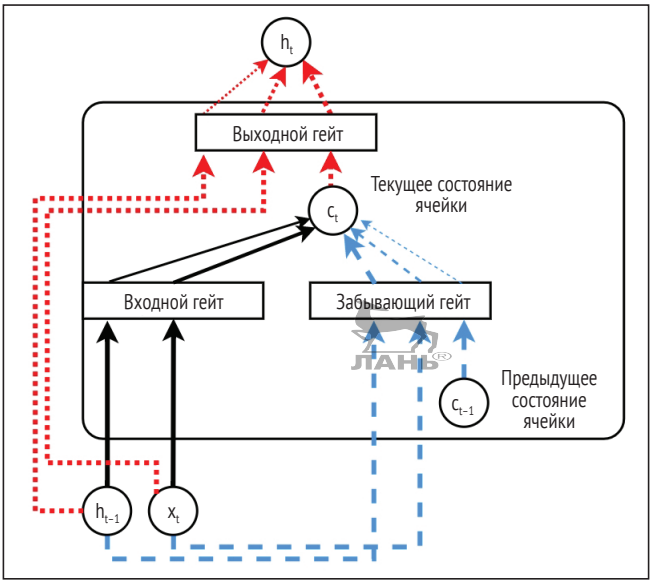


Рис. 7.1 ❖ Обобщенная схема движения данных в LSTM

LSTM в деталях

Сейчас мы приступим к изучению внутреннего механизма LSTM. Сначала кратко обсудим общий вид ячейки, а затем рассмотрим каждую операцию, происходящую в ячейке LSTM, и проработаем практический пример генерации текста.

Как мы уже говорили выше, LSTM состоят в основном из следующих трех элементов:

- **входной гейт** – элемент, который выводит значения между 0 (текущий вход не записывается в состояние ячейки) и 1 (текущий вход полностью записывается в состояние ячейки). Для формирования диапазона выходного сигнала между 0 и 1 обычно применяется сигмоидная активация;
- **забывающий гейт** – сигмоидный элемент, который выводит значения между 0 (предыдущее состояние ячейки полностью забыто при вычислении текущего состояния) и 1 (предыдущее состояние ячейки полностью считывается при вычислении текущего состояния);
- **выходной гейт** – сигмоидный элемент, который выводит значения между 0 (текущее состояние ячейки полностью отбрасывается для вычисления конечного состояния) и 1 (текущее состояние ячейки полностью используется при вычислении окончательного скрытого состояния).

Общий принцип действия LSTM показан на рис. 7.2. Это очень обобщенная схема, на которой скрыты некоторые детали, чтобы избежать путаницы. Для лучшего понимания структура LSTM представлена двумя способами – как с циклами, так и без циклов. На рис. 7.2 справа изображена LSTM с петлями, а слева – та же модель, но с развернутыми петлями.

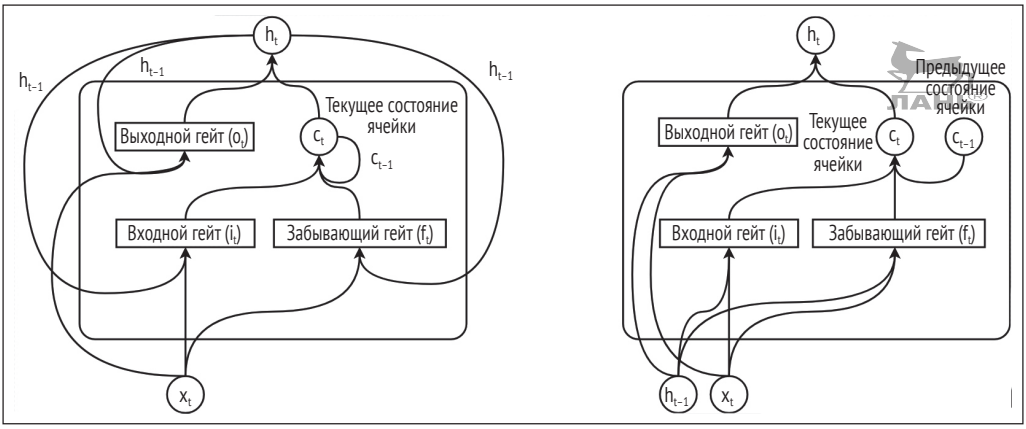


Рис. 7.2 ❖ LSTM с рекуррентными связями, т. е. петлями (справа), и с развернутыми рекуррентными связями (слева)

Давайте рассмотрим небольшой пример для лучшего понимания LSTM. Разбирая пример, мы одновременно будем изучать правила и уравнения. Допустим, у нас есть исходное предложение, начиная с которого нейросеть должна сгенерировать продолжение:

Джон подарил Мэри щенка.

История, которую мы ждем на выходе, должна быть о *Джоне*, *Мэри* и *щенке*. Давайте предположим, что LSTM-сеть должна вывести два предложения следом за исходным:

Джон подарил Мэри щенка. _____. _____.

Допустим, наша LSTM-сеть вывела следующий текст:

Джон подарил Мэри щенка. Он лает очень громко. Они назвали его Бобик.

На самом деле мы еще далеки от вывода таких реалистичных фраз. Однако нейросети могут изучать отношения, такие как между существительными и местоимениями. Например, «*он*» относится к «*щенку*», а «*они*» – к «*Джону*» и «*Мэри*». Затем следует изучить связь между существительным, местоимением и глаголом. Например, в данном случае с местоимением «*они*» глагол должен иметь окончание «*и*». Эти отношения и зависимости схематически показаны на рис. 7.3. Как видите, в этом примере присутствуют как долгосрочные (например, *Бобик → щенок*), так и краткосрочные (например, *Он → лает*) зависимости. Сплошные стрелки показывают связи между существительными и местоимениями, а пунктирные – между существительными/местоимениями и глаголами.

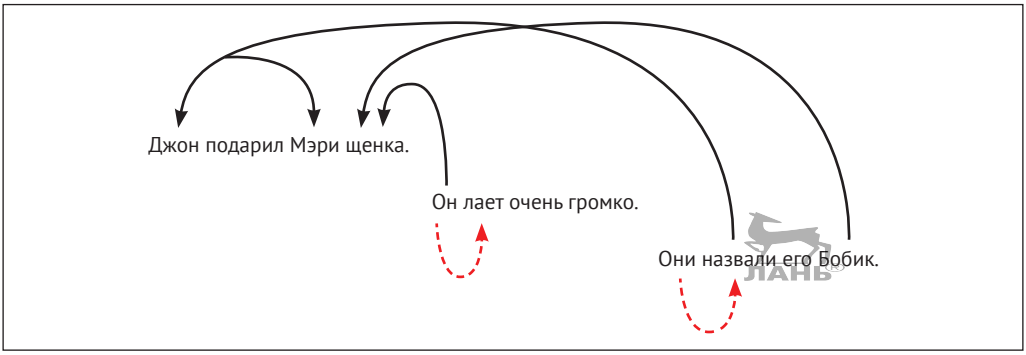


Рис. 7.3 ❖ Исходные и предсказанные предложения со связями между различными словами

Теперь давайте рассмотрим, как LSTM-сети моделируют такие отношения и зависимости для вывода осмысленного текста с учетом начального предложения.

Входной гейт i_t принимает текущий ввод x_t и предыдущее конечное скрытое состояние h_{t-1} в качестве ввода и вычисляет i_t следующим образом:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i).$$

Входной гейт можно понимать как вычисление, выполняемое на скрытом уровне стандартной RNN, обладающей одним скрытым слоем с сигмоидальной активацией. Вспомните, как мы вычисляли скрытое состояние стандартной рекуррентной нейросети:

$$h_t = tanh(Ux_t + Wh_{t-1}).$$

Следовательно, вычисление i_t в LSTM выглядит вполне аналогично вычислению h_t в стандартной RNN, за исключением другой функции активации и наличия смещения.

Если в результате вычислений $i_t = 0$, это означает, что никакая информация с текущего входа не будет поступать в состояние ячейки. И наоборот, $i_t = 1$ означает, что вся информация с текущего входа окажется в состоянии ячейки.

Затем вычисляется так называемое *значение-кандидат*, которое претендует на место в памяти и позже применяется для вычисления текущего состояния ячейки:

$$\tilde{c}_t = tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c).$$

Вычисления схематически представлены на рис. 7.4.

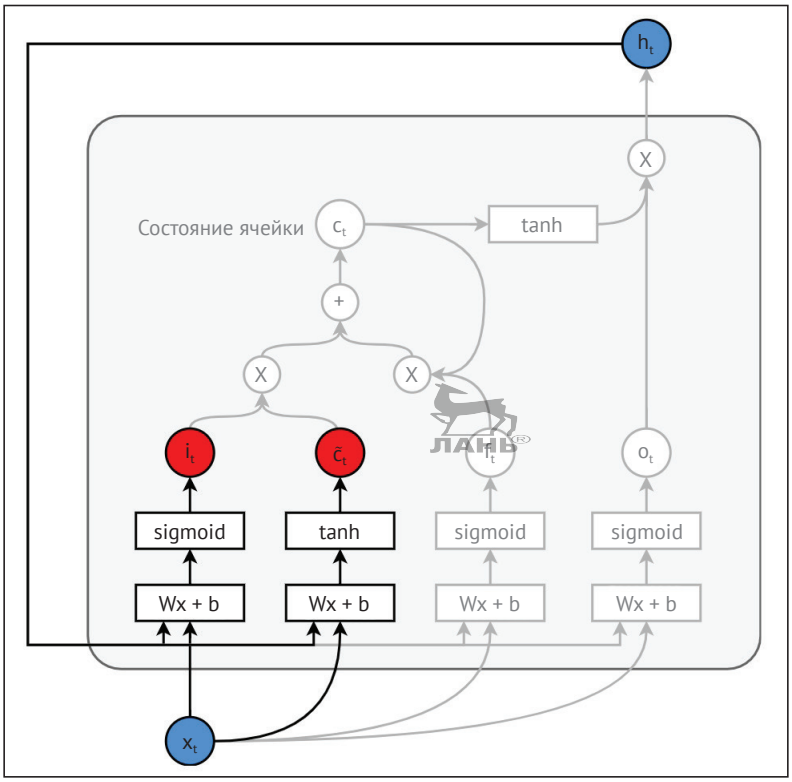


Рис. 7.4 ❖ Вычисление i_t и $\tilde{c}_t$ в контексте всех вычислений, происходящих в LSTM (остальные вычисления показаны серым)

В нашем примере, в самом начале обучения, входной гейт должен быть сильно активирован. Первое слово, которое выводит LSTM, – «*Он*». Чтобы сделать это, LSTM-сеть должна узнать, что «*щенок*» также упоминается как «*он*». Давайте предположим, что в нашей LSTM-нейросети есть пять нейронов для хранения состояния. Мы бы хотели, чтобы нейросеть хранила информацию о том, что «*он*» относится к «*щенку*». А еще мы хотим, чтобы LSTM выучила (в другом нейроне), что глагол заканчивается на «*и*», если он относится к местоимению «*они*». Еще одна вещь, которую следует знать LSTM, – это то, что «*щенок лает громко*». На рис. 7.5 показано, как эти знания могут быть закодированы в состоянии ячейки LSTM. Каждый круг представляет отдельный нейрон (то есть скрытую единицу) состояния ячейки.

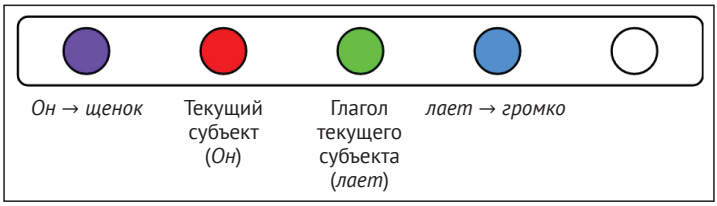


Рис. 7.5 ❖ Знания, которые надо закодировать в состоянии ячейки

С помощью этой информации мы можем сгенерировать первое новое предложение:

Джон дал Мэри щенка. Он лает очень громко.

Далее рассчитывается забывающий гейт:

$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$.

Забывающий гейт делает следующее. Значение $f_t = 0$ означает, что никакая информация из c_{t-1} не будет применяться для вычисления c_t, и наоборот, $f_t = 1$ означает, что вся информация c_{t-1} будет распространяться в вычислении c_t.

Давайте посмотрим, как забывающий гейт помогает предсказать следующее предложение:

Они назвали его Бобик.

Здесь, как видите, присутствуют новые отношения между словами «*Джон*», «*Мэри*» и «*они*». Следовательно, нам больше не нужна информация касаемо местоимения «*он*» и о том, как ведет себя глагол «*лает*», так как субъектами являются «*Джон*» и «*Мэри*». Мы можем использовать забывающий гейт в сочетании с текущим субъектом «*они*» и соответствующим им глаголом «*назвали*», чтобы заменить информацию, хранящуюся в нейронах **текущий субъект** и **глагол для текущего субъекта** (рис. 7.6).

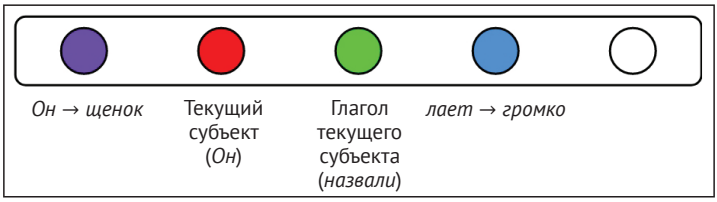


Рис. 7.6 ❖ Знание в третьем нейроне слева (*он* → *лает*) заменено новой информацией (*они* → *назвали*)

Мы можем проиллюстрировать это преобразование на рис. 7.7, используя понятие весов. Мы не изменяем состояние нейрона, сохраняя отношение *Он → щенок*, потому что «*щенок*» появляется как объект в последнем предложении. Это делается путем установки в 1 веса связи от c_{t-1} до c_t для отношения *Он → щенок*. Затем мы заменим содержимое нейронов, поддерживающих текущую информацию о субъекте и глаголе, новым субъектом и глаголом. Это достигается установкой весов забывания f_t для этого нейрона на 0. Затем установим в 1 вес i_t, соединяющего текущий субъект и глагол с соответствующими нейронами состояния. Мы можем думать о $\tilde{c}_t$ как о сущности, содержащей какую-то новую информацию (например, информацию из текущего входного x_t), которую следует перенести в состояние ячейки.

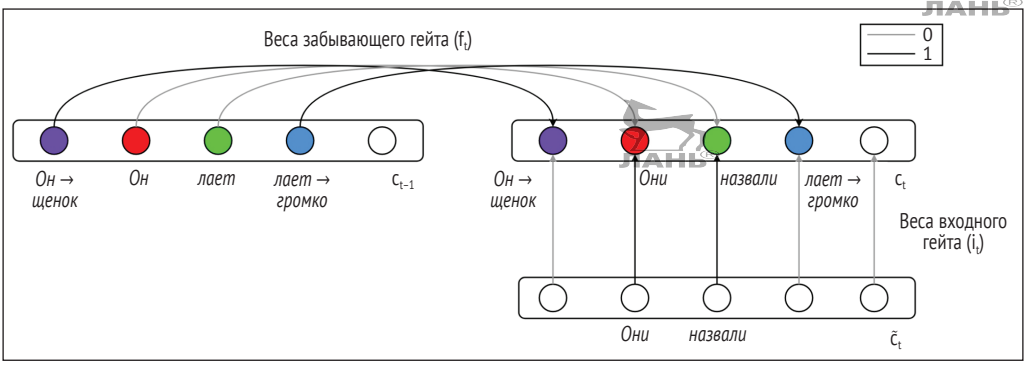


Рис. 7.7 ❖ Вычисление состояния ячейки c_t из предыдущего состояния c_{t-1} и значения-кандидата $\tilde{c}_t$

Текущее состояние ячейки вычисляют следующим образом:

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t.$$

Другими словами, текущее состояние является комбинацией двух гейтов:

- какую информацию забыть/запомнить из предыдущего состояния ячейки;
- какую информацию добавить/отбросить к текущему входу.

Далее на рис. 7.8 выделим черным цветом вычисления внутри LSTM, которые мы рассмотрели к этому моменту.

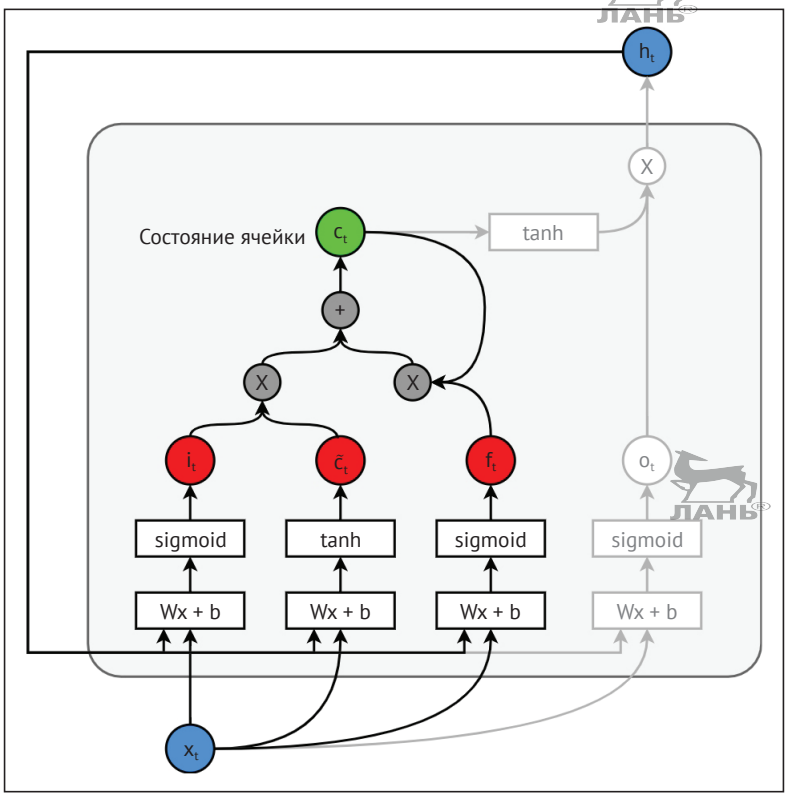


Рис. 7.8 ❖ Вычисления, рассмотренные к этому моменту, включая i_t, f_t, $\tilde{c}_t$ и c_t

После обучения полное состояние ячейки может выглядеть, как на рис. 7.9.

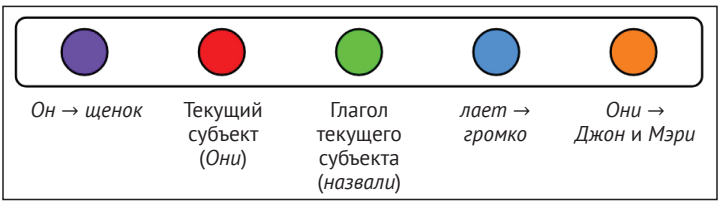


Рис. 7.9 ❖ Полное состояние ячейки после вывода двух предложений

Конечное состояние ячейки LSTM h_t вычисляется следующим образом:

$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$;
$h_t = o_t tanh(c_t)$.

В нашем примере мы хотим получить на выходе предложение:

Они назвали его Бобик.

Для генерации этого предложения нам не нужен предпоследний нейрон, т. к. он содержит информацию о том, как лает щенок. Следовательно, мы можем игнорировать последний нейрон, содержащий отношение *лает* → *громко*, во время предсказания последнего предложения. Это именно то, что делает выходной гейт o_t, – он будет игнорировать ненужную память и извлекать из состояния ячейки только необходимую связанную информацию при вычислении окончательного вывода. На рис. 7.10 показано, как выглядит полная ячейка LSTM.

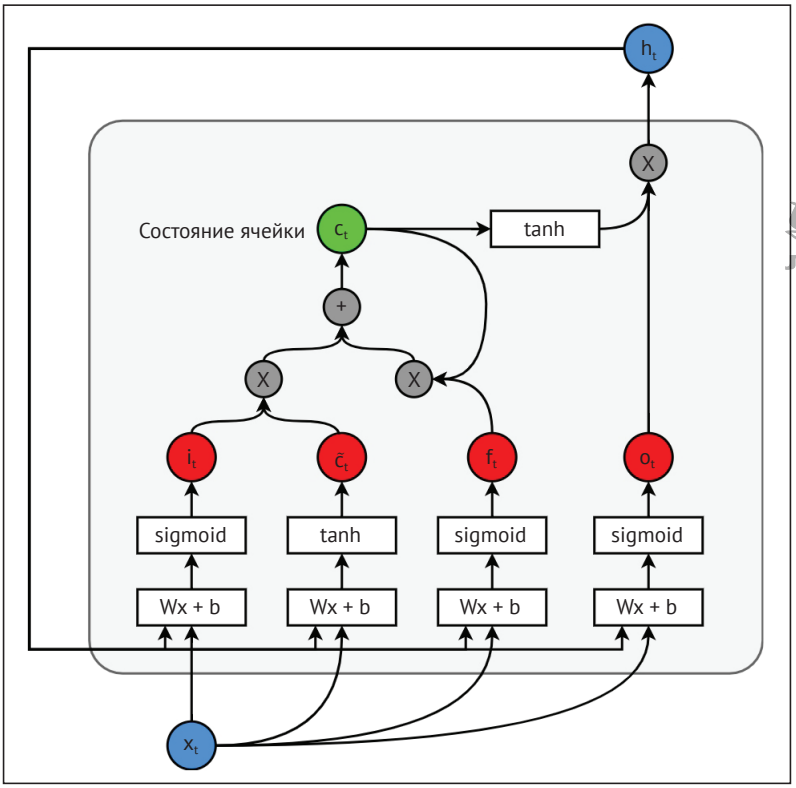


Рис. 7.10 ❖ Так выглядит полная схема ячейки LSTM

Подведем промежуточный итог и сведем вместе уравнения операций, выполняемых в ячейке LSTM:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i);$$
$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f);$$
$$\tilde{c}_t = tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c);$$
$$c_t = f_t c_{t-1} + i_t \tilde{c}_t;$$
$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o);$$
$$h_t = o_t tanh(c_t).$$

Теперь для лучшего понимания общей картины мы можем развернуть состояния ячейки LSTM во времени и показать, как они соединяются в последовательность, передавая предыдущее состояние ячейки для вычисления следующего состояния (рис. 7.11).

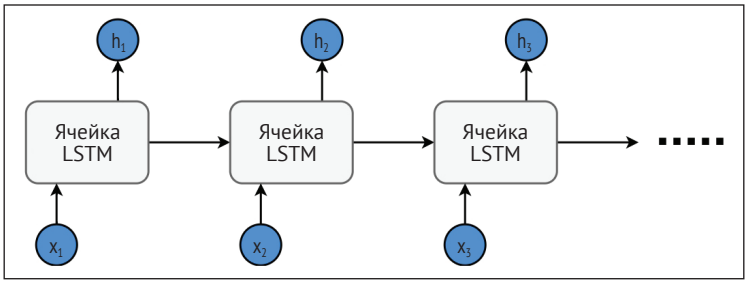


Рис. 7.11 ❖ Последовательная работа LSTM

Однако этого все еще недостаточно для того, чтобы сделать что-то полезное. Как видите, несмотря на то что мы можем создать хорошую цепочку LSTM, действительно способную моделировать последовательность, у нас все еще нет выходных данных или прогноза. Если мы хотим использовать то, чему научилась нейросеть, нам нужен способ извлечь окончательный вывод из LSTM. Для этого мы поместим слой softmax (с весами W_s и смещением b_s) поверх LSTM. Окончательный результат вычисляется в соответствии со следующим уравнением:

$$y_t = softmax(W_s h_t + b_s).$$

Теперь мы можем изобразить LSTM со слоем softmax, как на рис. 7.12.

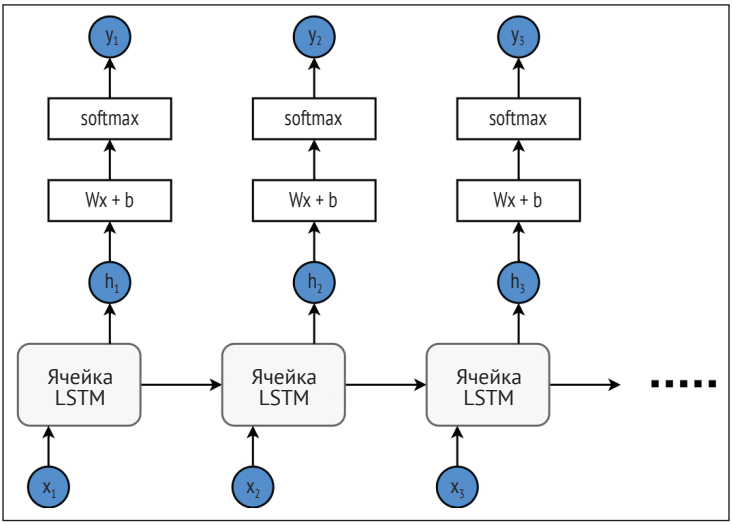


Рис. 7.12 ❖ Развернутая во времени LSTM со слоем softmax

Чем LSTM отличаются от стандартных RNN

Давайте разберемся, в чем заключаются отличия LSTM от обычной RNN. Прежде всего LSTM имеет более сложную структуру. Одно из основных отличий состоит в том, что LSTM имеет два разных состояния – состояние ячейки c_t и окончательное скрытое состояние h_t, в то время как RNN имеет только одно скрытое состояние h_t. Следующее важное отличие состоит в наличии у LSTM трех независимых гейтов, что позволяет намного полнее и гибче управлять обработкой текущего входа и предыдущего состояния при вычислении окончательного скрытого состояния h_t.

Наличие двух разных состояний довольно выгодно. Благодаря этому механизму, даже когда состояние ячейки изменяется быстро, окончательное скрытое состояние все равно будет изменяться медленнее. Таким образом, в то время как состояние ячейки отражает как краткосрочные, так и долгосрочные зависимости, окончательное скрытое состояние может отражать либо только краткосрочные зависимости, либо только долгосрочные зависимости, либо и то, и другое.

Механизм управления перемещением информации состоит из трех гейтов: входных, забывающих и выходных:

- входной гейт контролирует, какая часть текущих данных записывается в состояние ячейки;
- забывающий гейт контролирует, какая часть предыдущего состояния ячейки переносится в текущее состояние ячейки;
- выходной гейт контролирует, какая часть состояния ячейки передается в окончательное скрытое состояние.

Совершенно очевидно, что это гораздо более глубокий подход, особенно по сравнению со стандартными RNN, позволяющий лучше контролировать, как текущий вход и предыдущее состояние ячейки вносят вклад в текущее состояние ячейки. Кроме того, выходной гейт позволяет управлять вкладом ячейки в конечное скрытое состояние. На рис. 7.13 приведено сравнение схем стандартной RNN и LSTM, наглядно отражающее разницу в функциональности двух моделей.

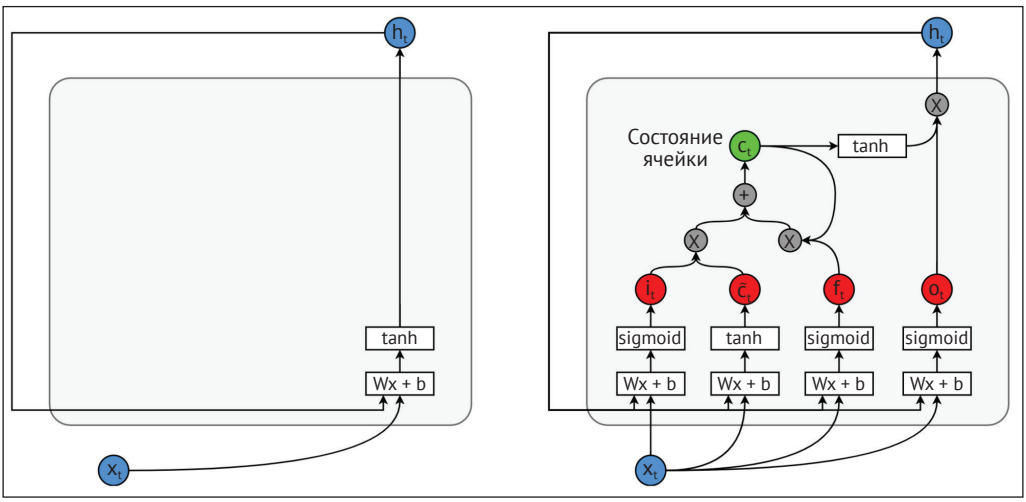


Рис. 7.13 ❖ Пошаговое сравнение ячеек RNN и LSTM

Таким образом, благодаря возможности хранить два разных состояния LSTM может изучать как краткосрочные, так и долгосрочные зависимости, что помогает решить проблему исчезающего градиента. Об этом пойдет речь в следующем разделе.

Как LSTM решает проблему исчезающего градиента

Как мы говорили выше, хотя RNN теоретически надежны, на практике они страдают серьезным недостатком. Когда применяется обратное распространение во времени (BPTT), градиент стремительно уменьшается, что позволяет нам распространять информацию во времени лишь на несколько шагов. Следовательно, RNN обладает лишь кратковременной памятью. Это существенно ограничивает полезность RNN в реальных последовательных задачах.

Полезные и интересные последовательные задачи, такие как прогнозирование фондового рынка или обработка естественного языка, нуждаются в способности изучать и хранить долгосрочные зависимости. Рассмотрим пример предсказания следующего слова:

Джон талантливый ученик. Он хорошо учится и играет в регби и крикет.
Все другие студенты хотят быть как _____.

Для нас это очень простая задача. Ответом будет слово «*Джон*». Однако для RNN это сложная задача. Мы пытаемся предсказать ответ, который лежит в самом начале текста. Для решения этой задачи нам нужен способ хранения долгосрочных зависимостей в состоянии RNN. Это именно тот тип задач, для решения которых предназначены LSTM-сети.

В главе 6 мы обсуждали, как исчезает или взрывается градиент при отсутствии нелинейных функций. Сейчас вы увидите, что это может произойти даже при наличии нелинейного компонента. Давайте рассмотрим производную $\partial h_t/\partial h_{t-k}$ для стандартной сети RNN и сети LSTM ($\partial c_t/\partial c_{t-k}$). Как вы узнали в предыдущей главе, это ключевой член уравнения, который вызывает исчезающий градиент.

Давайте предположим, что скрытое состояние стандартной RNN рассчитывается следующим образом:

$$h_t = \sigma(W_x x_t + W_h h_{t-1}).$$

Чтобы упростить вычисления, мы можем игнорировать текущие входы и сосредоточиться на рекуррентной части, которая даст нам следующее уравнение:

$$h_t = \sigma(W_h h_{t-1}).$$

Вычисляя $\partial h_t/\partial h_{t-k}$ для предыдущего равенства, мы получаем:

$$\partial h_t/\partial h_{t-k} = \prod_{i=0}^{k-1} W_h \sigma(W_h h_{t-k+i})(1-\sigma(W_h h_{t-k+i}));$$

$$\partial h_t/\partial h_{t-k} = W_h^k \prod_{i=0}^{k-1} \sigma(W_h h_{t-k+i})(1-\sigma(W_h h_{t-k+i})).$$

Теперь давайте посмотрим, что происходит, когда $W_h h_{t-k+i} \ll 0$ или $W_h h_{t-k+i} \gg 0$ (что произойдет по мере продолжения обучения). В обоих случаях $\partial h_t / \partial h_{t-k}$ начнет приближаться к 0, что приведет к исчезающему градиенту. Даже если $W_h h_{t-k+i} = 0$, когда градиент является максимальным, после перемножений на большом количестве временных шагов общий градиент становится довольно маленьким. Впрочем, член W_n^k может также вызвать взрыв или исчезновение градиентов – например, из-за неудачной инициализации. Однако по сравнению с исчезновением градиента из-за $W_h h_{t-k+i} \ll 0$ или $W_h h_{t-k+i} \gg 0$ исчезновение или взрыв градиента, вызванное членом W_n^k, относительно легко предотвращается путем тщательной инициализации весов и отсечения градиента.

Теперь давайте посмотрим на ячейку LSTM. Точнее, мы рассмотрим состояние ячейки, заданное следующим уравнением:

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t.$$

Это произведение всех задействованных забывающих гейтов, выполняемое в LSTM. Однако если аналогичным образом вычислить $\partial c_t / \partial c_{t-k}$ для LSTM, т. е. игнорируя члены $W_{fx} x_t$ и b_{ft}, поскольку они не являются рекуррентными, мы получим следующее выражение:

$$\partial c_t / \partial c_{t-k} = \prod_{i=0}^{k-1} \sigma(W_{fh} h_{t-k+i}).$$

В этом случае, хотя градиент продолжает исчезать, если $W_h h_{t-k+i} \ll 0$, с другой стороны, если $W_h h_{t-k+i} \gg 0$, производная будет уменьшаться гораздо медленнее, чем в стандартной RNN. Поэтому у нас появилась альтернатива, когда градиент не исчезает. Кроме того, поскольку используется функция сжатия, градиенты не будут взрываться из-за того, что $\partial c_t / \partial c_{t-k}$ велико (что, вероятно, случится во время взрыва градиента). Кроме того, когда $W_h h_{t-k+i} \gg 0$, мы получаем максимальный градиент, близкий к 1. Это означает, что градиенты не будут быстро уменьшаться, как в случае с RNN. Наконец, в выводе нет такого компонента, как W_h^k. Однако математические выкладки для $\partial h_t / \partial h_{t-k}$ выполняются несколько сложнее. Выполняя дифференцирование, мы получаем:

$$\partial h_t / \partial h_{t-k} = \partial(o_t tanh(c_t)) / \partial h_{t-k}.$$

Далее мы можем получить что-то наподобие этой формы:

$$tanh(.)\sigma(.)[1 - \sigma(.)]w_{oh} + \sigma(.)[1 - tanh^2(.)]\{c_{t-1}\sigma(.)[1 - \sigma(.)]w_{fh} + \sigma(.)[1 - tanh^2(.)]w_{ch} + tanh(.)\sigma(.)[1 - \sigma(.)]w_{ih}\}.$$

Нам безразличен компонент внутри $\sigma(.)$ или $tanh(.)$, потому что вне зависимости от аргумента значение функции будет ограничено диапазоном (0,1) или (–1,1). Если мы еще больше упростим выражение, заменяя члены $\sigma(.)$, $[1 - \sigma(.)]$, $tanh(.)$ и $[1 - tanh^2(.)]$ какой-нибудь общей нотацией, например $\gamma(.)$, то получим следующее выражение:

$$\gamma(.)w_{oh} + \gamma(.)[c_{t-1}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}].$$

Допуская, что внешний член $\gamma(.)$ поглощается каждым членом $\gamma(.)$, расположенным внутри квадратных скобок, получаем:

$\gamma(.)w_{oh} + c_{t-1}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}$.

Отсюда следует:

$$\partial h_t/\partial h_{t-k} \approx \prod_{i=0}^{k-1}\gamma(.)w_{oh} + c_{t-1}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}.$$

Это означает, что хотя член $\partial c_t/\partial c_{t-k}$ защищен от любых компонентов W_h^k, этого нельзя сказать про $\partial h_t/\partial h_{t-k}$. Поэтому мы должны соблюдать осторожность при инициализации весов LSTM и применять градиентное отсечение.

Однако в случае LSTM подверженность h_t исчезающему градиенту не так важна, как для RNN. Дело в том, что c_t по-прежнему может хранить долгосрочные зависимости без влияния исчезающего градиента, а h_t может извлекать долгосрочные зависимости из c_t, если это необходимо.

Улучшение LSTM

Как вы убедились на примере RNN, наличие прочной теоретической базы не всегда гарантирует, что нейросеть будет безупречно работать практике. Это связано с ограничениями вычислительной точности компьютеров и относится в том числе к LSTM-сетям. Наличие сложной архитектуры – позволяющей лучше моделировать долгосрочные зависимости в данных – само по себе не означает, что LSTM будет выдавать совершенно реалистичные прогнозы. Поэтому было разработано множество расширений, помогающих LSTM лучше работать на этапе прогнозирования. Здесь мы обсудим несколько таких улучшений: жадная выборка, лучевой поиск, использование векторов слов вместо унитарного кодирования и использование двунаправленных LSTM.

Жадная выборка

Если мы попытаемся всегда предсказывать слово с наибольшей вероятностью, LSTM будет склонна давать очень монотонные результаты. Например, она будет многократно повторять одно и то же слово, прежде чем переключиться на другое.

Один из способов решить проблему – использовать *жадную выборку* (greedy sampling), когда мы формируем наилучший набор n и делаем выборку из этого набора. Это помогает нарушить монотонный характер прогнозов.

Давайте рассмотрим первое предложение предыдущего примера:

Джон подарил Мэри щенка.

Скажем, мы начинаем с первого слова и хотим предсказать следующие три слова:

Джон _____ _____ _____.

Если мы будем действовать строго детерминистически, LSTM может вывести что-то наподобие:

Джон подарил Мэри подарил Джон.

Однако, выбирая следующее слово из подмножества наиболее вероятных слов в словаре, LSTM-сеть вынуждена изменять прогноз и может выдать правильную последовательность:

Джон подарил Мэри щенка.

Впрочем, она может выдать и такой результат:

Джон подарил щенку щенка.

Хотя жадная выборка помогает добавить больше вариаций к сгенерированному тексту, этот метод не гарантирует, что вывод всегда будет реалистичным, особенно при выводе длинных последовательностей. Давайте рассмотрим улучшенную технику поиска, которая на самом деле работает еще за несколько шагов до прогноза.

Лучевой поиск

Лучевой поиск (beam search) – это способ улучшить качество прогнозов, сделанных LSTM. При этом прогнозы находят путем решения поисковой задачи. Ключевая идея лучевого поиска состоит в том, чтобы производить b выходов (то есть y_t, y_{t+1}, …, y_{t+b}) одновременно вместо одного выхода y_t. Здесь параметр b известен как *длина луча*, а совокупность из b произведенных результатов – как *луч*. С технической точки зрения, мы выбираем луч, который имеет наибольшую общую вероятность $p(y_t, y_{t+1}, \ldots, y_{t+b}|x_t)$ вместо выбора самой высокой вероятности $p(y_t|x_t)$. Прежде чем делать прогноз, мы смотрим в будущее, что обычно приводит к лучшим результатам.

Давайте рассмотрим новый пример:

Джон позвал Мэри в кино.

Допустим, мы предсказываем слово за словом. И изначально у нас есть следующее:

Джон ____ ____ ____ ____.

Предположим гипотетически, что наша LSTM создает предложение при помощи лучевого поиска. Допустим, длина луча $b = 2$, и мы рассмотрим $n = 3$ лучших кандидата на каждом этапе поиска. Тогда вероятности для каждого слова могут выглядеть, как показано на рис. 7.14.

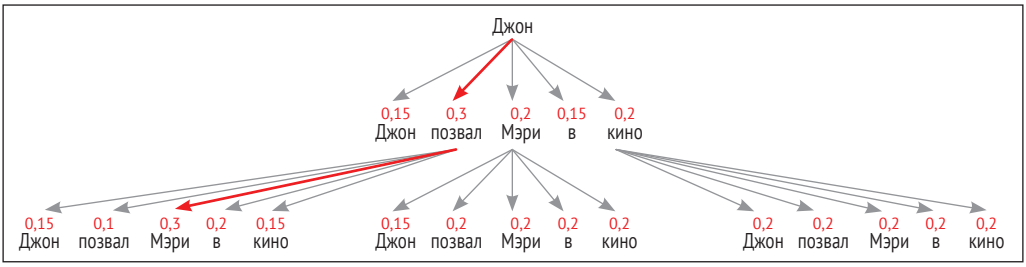


Рис. 7.14 ❖ Пространство лучевого поиска при $b = 2$ и $n = 3$

Мы начинаем со слова Джон и получаем вероятности для всех слов в словаре. В нашем примере при $n = 2$ мы выбираем трех лучших кандидатов для следующего уровня дерева: «*позвал*», «*Мэри*» и «*кино*». Обратите внимание, что это не ситуация в реальной LSTM-сети, а просто поясняющий пример. Затем из этих выбранных кандидатов вырастает следующая ветвь дерева. Мы вновь выбираем три лучших кандидата, и поиск будет повторяться, пока мы не достигнем глубины b в дереве.

Путь, который дает наибольшую совместную вероятность, т. е. P(*позвал*, *Мэри*|*Джон*) = 0,09, выделен более толстыми стрелками. Кроме того, это лучший механизм прогнозирования, поскольку он возвращает более высокую вероятность, или так называемое *вознаграждение* за фразу «*Джон позвал Мэри*», чем за фразы «*Джон Мэри Джон*» или «*Джон Джон позвал*».

В наших примерах жадная выборка и лучевой поиск сработали одинаково эффективно. Однако ситуация меняется, когда мы масштабируем задачу для вывода небольшого эссе. Тогда результаты, полученные при помощи лучевого поиска, будут более реалистичными и грамматически правильными, чем результаты, полученные жадной выборкой.

Использование векторных представлений слов

Другой популярный способ повышения производительности LSTM – это использование в качестве входных данных векторных представлений слов вместо применения векторов с унитарным кодированием. Давайте разберемся в достоинствах этого метода на примере. Допустим, мы хотим генерировать текст, начиная с некоторого случайного слова. В нашем случае это будет так:

Джон ____ ____ ____ ____.

Будем считать, что наша LSTM-сеть обучена следующим предложениям:

Джон дал Мэри щенка. Мэри отправила Бобу котенка.

Предположим также, что у нас есть векторы слов, которые отображаются на двумерное пространство, как показано на рис. 7.15.

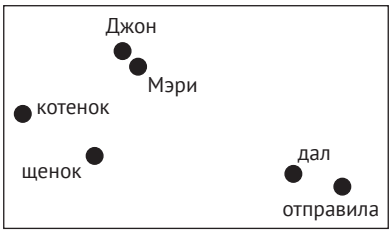


Рис. 7.15 ❖ Предполагаемая топология векторов слов в двумерном пространстве

Представление этих слов в их числовой форме может выглядеть следующим образом:

котенок: [*0.5*, *0.3*, *0.2*]
щенок: [*0.49*, *0.31*, *0.25*]
дал: [*0.1*, *0.8*, *0.9*]

Видно, что *расстояние* (*котенок*, *щенок*) < *расстояние* (*котенок*, *дал*). Однако если мы используем унитарное кодирование, представления слов будут выглядеть следующим образом:

котенок: [*1*, *0*, *0*,...]
щенок: [*0*, *1*, *0*,...]
дал: [*0*, *0*, *1*,...]

В этом случае *расстояние* (*котенок*, *щенок*) = *расстояние* (*котенок, дал*). Как вы уже знаете, представления с унитарным кодированием не фиксируют правильные отношения между словами и представляют текст так, будто все слова одинаково удалены друг от друга. В свою очередь, векторные представления слов способны учитывать такие отношения и являются более подходящими в качестве признаков в LSTM.

Используя векторы слов, LSTM научится лучше использовать отношения между словами. Например, LSTM с векторами слов изучит следующее предложение:

Джон дал Мэри котенка.

Оно довольно близко расположено к предыдущему:

Джон дал Мэри щенка.

Но зато они оба заметно далеки от такого предложения:

Джон дал Мэри дал.

В случае использования унитарного кодирования не получится обнаружить разницу.

Двунаправленные LSTM (BiLSTM)

Создание *двунаправленных LSTM* (bidirectional LSTM, BiLSTM) – это еще один способ улучшить качество прогнозов LSTM. Под двунаправленностью мы подразумеваем обучение LSTM чтению данных от начала до конца и от конца до начала. До сих пор во время обучения LSTM мы просматривали последовательность в одном направлении.

Рассмотрим следующие два предложения:

Джон дал Мэри _____. Он лает очень громко.

Мы хотим, чтобы LSTM-сеть предсказала пропущенное слово.

Если мы просматриваем последовательность от начала до пропущенного слова, это будет выглядеть следующим образом:

Джон дал Мэри _____.

Нам не хватает информации о контексте пропущенного слова, чтобы сделать обоснованное предсказание. Однако если мы организуем просмотр в двух направлениях, то получим следующие блоки:

Джон дал Мэри _____.
_____. Он лает очень громко.

В данном случае, исходя из контекста громкого лая, можно сделать предположение, что речь идет о собаке или щенке. Некоторые задачи могут значительно выиграть от чтения данных с обеих сторон. Кроме того, двунаправленный просмотр увеличивает объем данных, доступных для нейронной сети, и повышает ее производительность.

Другое применение BiLSTM – машинный перевод с исходного языка на целевой язык. Одна из проблем машинного перевода заключается в отсутствии прямого соответствия между языками. Поэтому знание начала и окончания фразы на исходном языке помогает значительно лучше понять контекст, тем самым создавая лучшие переводы. Например, возьмем задачу перевода филиппинского языка на английский. В филиппинском языке предложения обычно пишутся в порядке «глагол – объект – субъект», тогда как в английском принят порядок «субъект – глагол – объект». В подобном задании на перевод будет очень полезно читать предложения вперед и назад.

BiLSTM – это две отдельные LSTM-сети. Одна нейросеть изучает данные от начала до конца, а другая – от конца до начала. На рис. 7.16 изображена обобщенная архитектура сети BiLSTM.

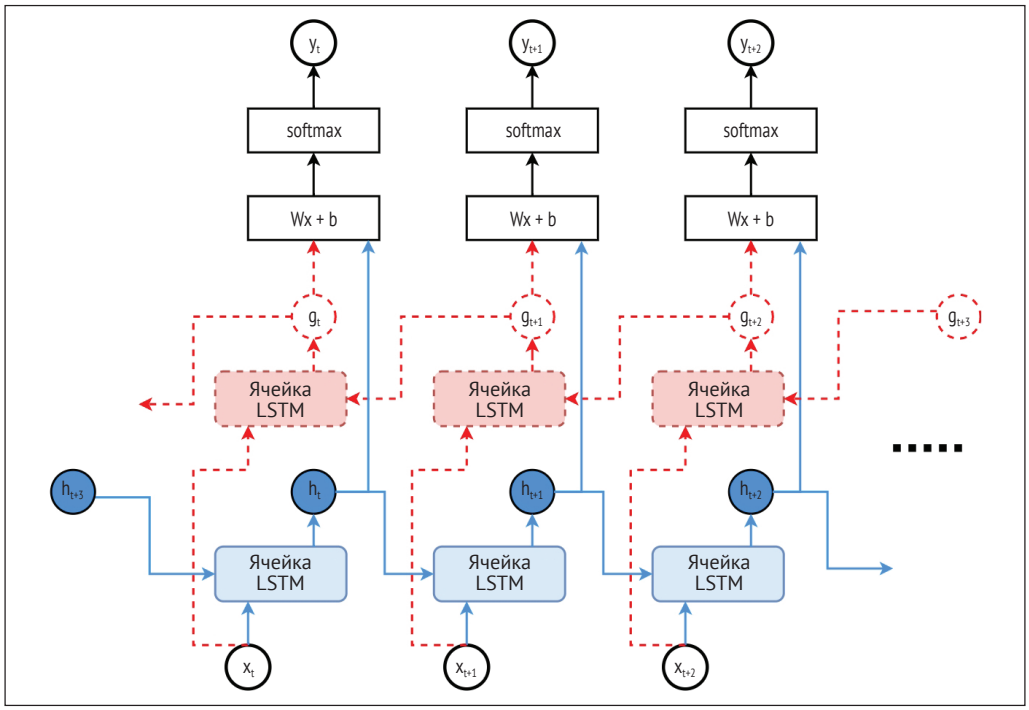


Рис. 7.16 ❖ Архитектура BiLSTM

Обучение происходит в два этапа. Во-первых, сеть, изображенная сплошными линиями, обучается на данных, полученных чтением текста с начала до конца. Это обычная процедура обучения для стандартных LSTM-сетей. Во-вторых, сеть, изображенная пунктиром, обучается на данных, полученных чтением текста в обратном направлении. Затем на этапе вывода объединяют оба состояния и создают вектор для прогнозирования пропущенного слова.

Другие варианты LSTM

Хотя на практике наиболее часто применяется стандартная архитектура LSTM, появилось много модификаций, которые либо упрощают исходную архитектуру, либо повышают производительность, либо и то, и другое. Мы рассмотрим два варианта структурной модификации архитектуры LSTM – «замочные скважины» и GRU.

Замочная скважина

Связи через замочную скважину (peephole connection) позволяют гейтам видеть не только текущий вход и предыдущее конечное скрытое состояние, но также и предыдущее состояние ячейки. Это увеличивает количество весов в ячейке LSTM. Наличие таких связей положительно влияет на результаты. Комплекс уравнений LSTM-сети с замочными скважинами выглядит следующим образом:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + W_{ic}c_{t-1} + b_i);$$
$$\tilde{c}_t = tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c);$$
$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + W_{fc}c_{t-1} + b_f);$$
$$c_t = f_t c_{t-1} + i_t \tilde{c}_t;$$
$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + W_{oc}c_{t-1} + b_o);$$
$$h_t = o_t tanh(c_t).$$

Давайте кратко рассмотрим, как замочные скважины улучшают работу LSTM. В обычной архитектуре гейты видят текущий вход и окончательное скрытое состояние, но не состояние ячейки. Однако в подобной конфигурации, если выходной гейт близок к нулю, даже если состояние ячейки содержит важную информацию, окончательное скрытое состояние все равно будет близко к нулю. Таким образом, гейты не будут учитывать скрытое состояние при расчете. Включение состояния ячейки непосредственно в уравнение вычисления гейта позволяет лучше контролировать состояние ячейки и может эффективно работать даже в ситуациях, когда выходной гейт близок к нулю. Другими словами, гейты могут «подсматривать» текущее состояние ячейки через «замочную скважину».

Архитектура LSTM с замочными скважинами схематически проиллюстрирована на рис. 7.17. Стандартные связи LSTM изображены серым цветом, а вновь добавленные связи – черным.

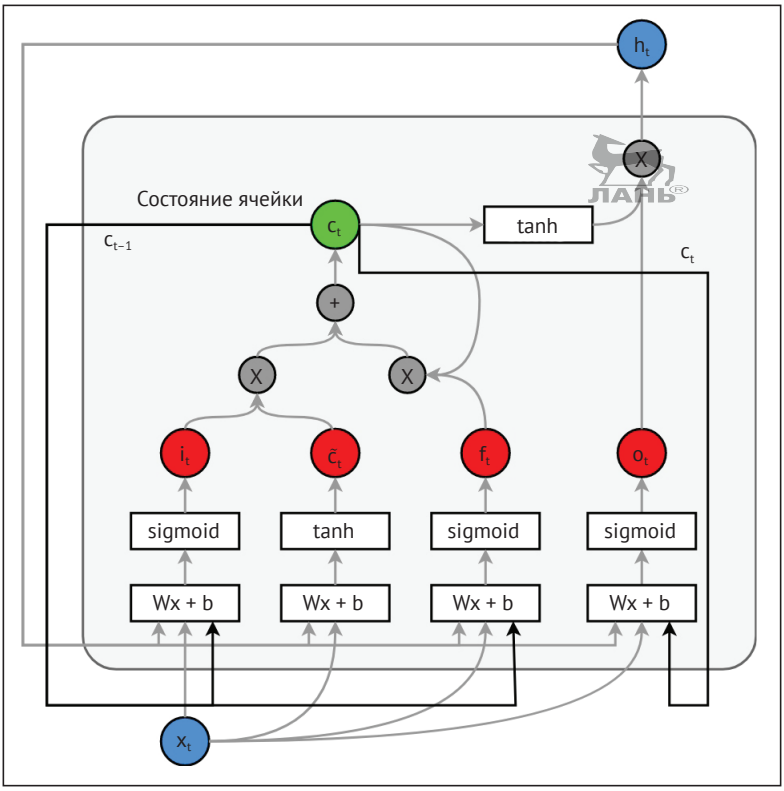


Рис. 7.17 ❖ LSTM с замочными скважинами
(новые связи показаны черным цветом, а стандартные – серым)

Управляемые рекуррентные ячейки (GRU)

Управляемые рекуррентные ячейки (gated recurrent unit, GRU) можно рассматривать как упрощение стандартной архитектуры LSTM. Как вы уже видели, стандартная LSTM имеет три разных гейта и два разных состояния. Это само по себе требует большого количества параметров. Поэтому исследователи старались уменьшить количество параметров, и GRU является одним из вариантов решения проблемы.

Существует несколько важных различий между GRU и LSTM.

Во-первых, GRU объединяет два состояния – состояние ячейки и конечное скрытое состояние, – в одно скрытое состояние h_t. Теперь благодаря этой простой модификации мы можем избавиться от выходного гейта. Помните, выходной гейт просто решал, какая часть состояния ячейки считывается в окончательное скрытое состояние. Отказ от выходного гейта значительно уменьшает количество параметров в ячейке.

$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r).$$
$$\tilde{h}_t = tanh(W_{hx}x_t + W_{hh}(r_t h_{t-1}) + b_n).$$

Затем GRU объединяют входной и забывающий гейты в один *гейт обновления* (update gate). Стандартная LSTM-сеть имеет два гейта, входной и забывающий. Входной гейт решает, какая часть текущего входа считывается в состояние ячейки, а забывающий определяет, какая часть предыдущего состояния ячейки считывается в текущее состояние. Математически это можно показать следующим образом:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i);$$
$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f).$$

GRU объединяют эти две операции в один гейт обновления. Если он равен 0, то полная информация о предыдущем состоянии ячейки помещается в текущее состояние, причем ничто из текущих входных данных не поступает в это состояние. Если гейт обновления равен 1, то, наоборот, все текущие входные данные считываются в текущее состояние ячейки, и ничто из предыдущего состояния ячейки не переходит в текущее состояние. Другими словами, входной гейт i_t становится инверсным к забывающему гейту, то есть $1 - f_t$:

$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z);$$
$$h_t = z_t\tilde{h}_t + (1 - z_t)h_{t-1}.$$

Теперь для наглядности соберем все уравнения GRU в группу:

$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r);$$
$$\tilde{h}_t = tanh(W_{hx}x_t + W_{hh}(r_t h_{t-1}) + b_n);$$
$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z);$$
$$h_t = z_t\tilde{h}_t + (1 - z_t)h_{t-1}.$$

Выглядит намного компактнее, чем LSTM, не так ли? На рис. 7.18 приведено сравнение архитектур ячейки GRU (слева) и ячейки LSTM (справа).

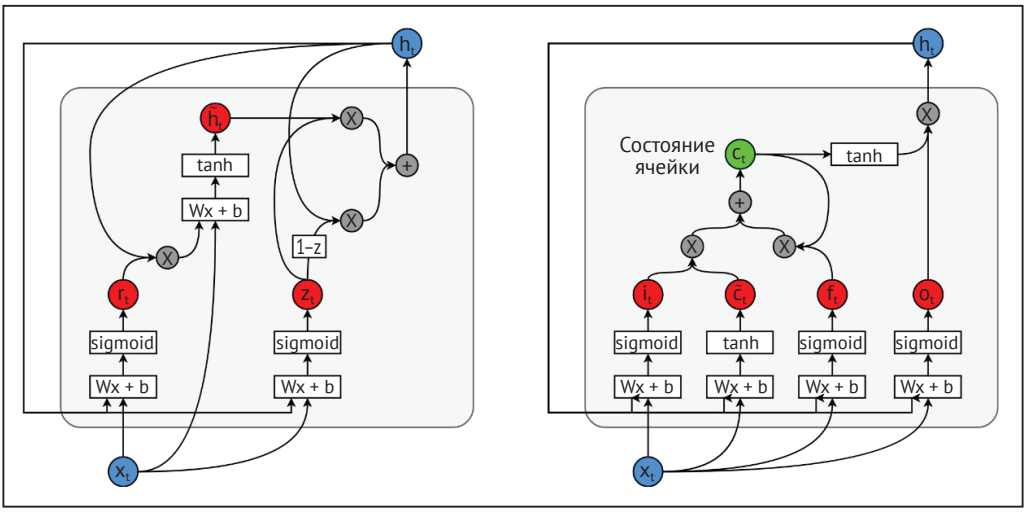


Рис. 7.18 ❖ Ячейка GRU (слева) и стандартная ячейка LSTM (справа)

Заключение

В этой главе вы узнали о LSTM-сетях. Сначала мы рассмотрели архитектуру LSTM на высоком уровне, а затем углубились в подробности вычислений, которые происходят в LSTM, и продемонстрировали ход вычислений на примере.

Вы увидели, что LSTM состоит из пяти основных компонентов:

- состояние ячейки – состояние внутренней ячейки LSTM;
- скрытое состояние – внешнее скрытое состояние, используемое для расчета прогнозов;
- входной гейт – управляет переносом данных из текущего входа в состояние ячейки;
- забывающий гейт – управляет переносом данных из предыдущего состояния ячейки в текущее состояние;
- выходной гейт – определяет, какая часть состояния ячейки выводится в скрытое состояние.

Наличие такой сложной структуры позволяет LSTM достаточно хорошо отслеживать как краткосрочные, так и долгосрочные зависимости.

Мы сравнили LSTM с обычными RNN и увидели, что LSTM-сети действительно способны изучать долгосрочные зависимости благодаря своей структуре, тогда как RNN могут упускать долгосрочные зависимости. Позже мы обсудили, как LSTM решают проблему исчезающего градиента.

Затем мы обсудили несколько расширений, которые улучшают производительность LSTM. Во-первых, это очень простая техника, так называемая жадная выборка, в которой, вместо того чтобы всегда выводить только наилучшего кандидата, мы случайным образом выбираем прогноз из набора лучших кандидатов. Вы узнали, как жадная выборка улучшает разнообразие сгенерированного текста. Далее мы рассмотрели более сложную технику – лучевой поиск. При этом, вместо того чтобы делать прогноз на один временной шаг в будущее, мы прогнозируем несколько временных шагов и выбираем кандидатов, которые дают наилучшую совместную вероятность. Еще одно улучшение заключается в использовании векторных представлений слов. Опираясь на векторные представления, LSTM-сеть может научиться более точно заменять семантически подобные слова во время предсказания, что приводит к большей реалистичности и правильности сгенерированного текста. Последней модификацией, которую мы рассмотрели, были двунаправленные LSTM или BiLSTM. Преимущество BiLSTM наиболее наглядно проявляется при заполнении пропущенного слова в предложении. BiLSTM-сеть читает текст в обоих направлениях, от начала до конца и от конца до начала. Это дает нам более развернутый и полный контекст.

Наконец, мы обсудили две модификации стандартной LSTM – замочные скважины и GRU. Стандартная LSTM-сеть при расчете гейтов смотрит только на текущий ввод и скрытое состояние. Используя замочную скважину, мы делаем расчеты гейта зависимыми от всего – текущего входа, скрытого состояния и состояния ячейки.

GRU – это гораздо более элегантный вариант стандартной LSTM-сети, который упрощает LSTM без ущерба для качества. GRU имеют только два гейта и одно состояние, в то время как стандартные LSTM имеют три гейта и два состояния.

В следующей главе мы рассмотрим различные архитектуры в действии, включая реализацию каждой из них, и проверим, насколько хорошо они работают в задачах генерации текста.

Глава 8

Применение LSTM для генерации текста

Теперь, когда вы хорошо понимаете основные принципы работы LSTM-сетей, в частности знаете решение проблемы исчезающего градиента и правила обновления состояния, мы можем перейти к применению LSTM в задачах обработки естественного языка. LSTM-сети интенсивно используются для генерации текста и создания подписей к изображениям. Например, языковое моделирование очень полезно для задач автоматического реферирования текста или создания увлекательной текстовой рекламы для продуктов, когда создание заголовка или аннотации изображения применяется при поиске товара, или когда пользователь хочет получить изображение по описанию, например «рыжая кошка».

Приложение, которое мы рассмотрим в этой главе, – это генерация нового текста при помощи LSTM. В качестве источника данных мы загрузим переводы с немецкого языка на английский некоторых сказок братьев Гримм. Мы применим эти сказки для обучения LSTM-сети и попросим ее сочинить новый текст. Мы обработаем текст, разбив его на *биграммы* уровня символов (n-граммы, где $n = 2$), и составим словарь из уникальных биграмм. Затем мы исследуем способы реализации ранее описанных методов, таких как жадная выборка и лучевой поиск, а также рассмотрим реализации последовательных моделей, отличных от стандартных LSTM, таких как LSTM с замочными скважинами и GRU.

Далее вы узнаете, как можно генерировать текст с лучшими входными представлениями, превышающими биграммы на уровне символов, например с отдельными словами. Обратите внимание, что очень неудобно представлять признаки слов в виде унитарного кода, так как словарь может очень быстро разрастись по сравнению с биграммами символьного уровня. Один из эффективных методов решения этой проблемы состоит в том, чтобы сначала выучить представления слов (или применять предварительно выученные представления) и использовать их в качестве входных данных для LSTM. Используя представления слов, мы избегаем проклятия размерности. В реальной задаче размер словарного запаса может составлять от 10 000 до 1 000 000 слов. Тем не менее представления слов имеют фиксированную размерность независимо от размера словаря.

Наши данные

Давайте начнем с обсуждения данных, которые мы будем использовать для генерации текста, и различных этапов предварительной обработки для очистки данных.

О наборе данных

Вам стоит внимательно ознакомиться с исходными данными. Увидев сгенерированный текст, вы сможете оценить его качество с учетом обучающих данных. Мы загрузим первые 100 книг с сайта **https://www.cs.cmu.edu/~spok/grimmtmp/**. Аналогичные данные мы использовали в главе 6 для демонстрации качества сетей RNN.

Мы начнем с загрузки 100 книг с веб-сайта с помощью автоматического скрипта следующим образом:

```
url = 'https://www.cs.cmu.edu/~spok/grimmtmp/'

# При необходимости создаем каталог
dir_name = 'stories'
if not os.path.exists(dir_name):
    os.mkdir(dir_name)

def maybe_download(filename):
  """Download a file if not present"""
  print('Downloading file: ', dir_name+ os.sep+filename)

  if not os.path.exists(dir_name+os.sep+filename):
    filename, _ = urlretrieve(url + filename,
                              dir_name+os.sep+filename)
  else:
    print('File ',filename, ' already exists.')

  return filename

  num_files = 100
  filenames = [format(i, '03d')+'.txt' for i in range(1,101)]

  for fn in filenames:
      maybe_download(fn)
```

Теперь можно взглянуть на примеры фрагментов текста, извлеченных из двух случайно выбранных историй.

Вот первый фрагмент[1]:

Then she said, my dearest benjamin, your father has had these coffins made for you and for your eleven brothers, for if I bring a little girl into the world, you are all to be killed and buried in them. And as she wept while she was saying this, the son comforted her and said, weep not, dear mother, we will save ourselves, and go hence. But she said, go forth into the forest with your eleven brothers, and let one sit constantly on the highest tree which can be found, and keep watch, looking towards the tower here in the castle. If I give birth to a little son, I will put up a white flag, and then you may venture to come back. But if I bear a daughter, I will hoist a red flag, and then fly hence as quickly as you are able, and may the good God protect you.

(*«И сказала она, мой дорогой вениамин, твой отец велел приготовить эти гробы для тебя и твоих одиннадцати братьев, если я произведу на свет маленькую девочку, вы все будете убиты и похоронены в них. Она с рыданиями говорила это, а сын утер ей слезы и сказал, не плачь, милая мамочка, мы уж спасемся и уйдем*

[1] Отрывок из сказки «Двенадцать братьев».

отсюда. Тогда иди подальше в лес вместе со своими одиннадцатью братьями, сказала она, и пусть один из вас неотлучно сидит на самом высоком дереве, какое сможете найти, и смотрит на замковую башню. Если у меня родится сынок, я велю поднять белый флаг, и вы можете смело возвращаться. Если же родится дочь, я велю поднять красный флаг, и тогда бегите прочь как можно скорее, и да хранит вас Бог».)

И второй фрагмент текста[1]:

Red-cap did not know what a wicked creature he was, and was not at all afraid of him.

"Good-day, little red-cap," said he.

"Thank you kindly, wolf."

"Whither away so early, little red-cap?"

"To my grandmother's."

"What have you got in your apron?"

"Cake and wine. Yesterday was baking-day, so poor sick grandmother is to have something good, to make her stronger."

"Where does your grandmother live, little red-cap?"

"A good quarter of a league farther on in the wood. Her house stands under the three large oak-trees, the nut-trees are just below. You surely must know it," replied little red-cap.

The wolf thought to himself, what a tender young creature. What a nice plump mouthful, she will be better to eat than the old woman.

(*Красная Шапочка и не подозревала, насколько он был коварен, и нисколечко его не боялась.*

«Здравствуй, маленькая Красная Шапочка», – сказал он.

«Ты так любезен, дорогой волк».

«Куда же ты идешь так рано, маленькая Красная Шапочка?»

«К моей бабушке».

«А что ты ей несешь?»

«Пирог и вино. Вчера мы пекли пироги, и я несу бедной больной бабушке гостинец, который придаст ей силы».

«Далеко ли живет твоя бабушка, маленькая Красная Шапочка?»

«Добрую четверть лиги отсюда, в лесу. Ее домик стоит под тремя дубами, а вокруг растет орешник. Вы наверняка знаете это место», – ответила маленькая Красная Шапочка.

[1] Отрывок из сказки «Красная Шапочка».

Волк подумал про себя, какое нежное юное дитя, словно пышная плюшечка. Она уж точно будет вкуснее, чем старуха.)

Предварительная обработка данных

В процессе предварительной обработки мы представим весь текст строчными буквами и разобьем на символьные *n*-граммы, где *n* = 2. Рассмотрим следующее предложение:

The king was hunting in the forest.
(*Царь охотился в лесу.*)

Это предложение будет разбито на последовательность *n*-грамм следующим образом:

['*th*', '*e*', '*ki*', '*ng*', '*wa*', '*s*', ...]

Мы будем использовать биграммы символьного уровня, потому что это значительно уменьшает размер словарного запаса по сравнению с использованием отдельных слов. Более того, мы заменим все биграммы, которые появляются в корпусе менее 10 раз, специальным токеном *UNK* (unknown, неизвестный). Это поможет нам еще сильнее сократить словарный запас.

Реализация LSTM

Итак, приступим к реализации LSTM-сети. Хотя в TensorFlow есть подбиблиотеки, в которых уже реализованы готовые к использованию LSTM, мы создадим нейросеть с нуля. Это будет очень ценный опыт, так как в реальном мире встречаются ситуации, когда вы не можете использовать готовые компоненты без доработки. Полный код доступен в файле `lstm_for_text_generation.ipynb`, расположенном в папке упражнений `ch8`. Однако мы также выполним упражнение, в котором вы увидите, как использовать существующий RNN API TensorFlow. Исходный код упражнения находится в файле `lstm_word2vec_rnn_api.ipynb`, расположенном в той же папке.

Сначала мы обсудим гиперпараметры LSTM и их назначение. После этого мы рассмотрим рабочие параметры – весовые коэффициенты и смещения, – необходимые для реализации LSTM, и вы узнаете, как эти параметры используются в составе операций, выполняемых в LSTM. Отсюда последует понимание того, как мы будем последовательно загружать данные в LSTM. Далее вы узнаете, как реализовать оптимизацию параметров с помощью отсечения градиента. Наконец, мы перейдем к выводу предсказаний, которые, по сути, являются биграммами и в конечном итоге составят значимую историю.

Объявление гиперпараметров

Прежде всего объявим некоторые гиперпараметры, необходимые для LSTM:

```
# Количество нейронов в переменных скрытого состояния.
num_nodes = 128
```

```
# Количество точек данных в нашем пакете.
batch_size = 64
# Количество шагов развертывания
num_unrollings = 50
dropout = 0.2 # Мы используем дропаут (выборочное обнуление).
```

Поясним назначение гиперпараметров более подробно:

- `num_nodes` – обозначает количество нейронов в памяти состояния ячейки. Когда данных много, увеличение сложности памяти улучшит точность модели, однако замедлит вычисления;
- `batch_size` – объем пакета данных, обработанных за один шаг. Увеличение размера пакета дает лучшую производительность, но предъявляет более высокие требования к памяти;
- `num_unrollings` – количество временных шагов, используемых в усеченном BPTT. Чем больше шагов `num_unrollings`, тем выше качество модели, но это увеличит как требования к памяти, так и время вычислений;
- `dropuot` – мы будем использовать метод регуляризации под названием *дропаут*[1] (dropout), чтобы уменьшить переобучение модели и получить лучшие результаты. Дропаут случайным образом удаляет информацию из переменных входа/выхода/состояния перед передачей значений в последующие операции. Потеря случайных данных во время обучения заставляет нейросеть искать новые признаки и зависимости, что ведет к повышению качества модели.

Объявление параметров

Теперь мы объявим переменные TensorFlow для действующих параметров LSTM. Сначала зададим параметры входного гейта:

- `ix` – веса, соединяющие вход с входным гейтом;
- `im` – веса, соединяющие скрытое состояние с входным гейтом;
- `ib` – смещение.

Здесь мы объявим параметры:

```
# Входной гейт - сколько памяти записывается в состояние ячейки.
# Связывает текущий вход с входным гейтом.
ix = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes],
stddev=0.02))
# Связывает предыдущее скрытое состояние со входным гейтом.
im = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],
stddev=0.02))
# Смещение входного гейта.
ib = tf.Variable(tf.random_uniform([1, num_nodes],-0.02, 0.02))
```

Аналогично мы объявим веса для забывающего гейта, значения-кандидата и выходного гейта.

Забывающий гейт объявим следующим образом:

```
# Забывающий гейт - сколько памяти отбрасывается из состояния ячейки.
# Связывает текущий вход с забывающим гейтом.
```

[1] Этот термин не имеет удачного общепринятого перевода на русский язык. Специалисты используют буквальную кальку с английского. – *Прим. перев.*

```
fx = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes],
stddev=0.02))
# Связывает предыдущее скрытое состояние с забывающим гейтом.
fm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],
stddev=0.02))
# Смещение забывающего гейта.
fb = tf.Variable(tf.random_uniform([1, num_nodes],-0.02, 0.02))
```

Значение-кандидат (используется для вычисления состояния ячейки) объявим следующим образом:

```
# Значение-кандидат (c~t) применяется для вычисления текущего состояния.
# Связывает текущий вход с кандидатом.
cx = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes],
stddev=0.02))
# Связывает предыдущее скрытое состояние с кандидатом.
cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],
stddev=0.02))
# Смещение кандидата.
cb = tf.Variable(tf.random_uniform([1, num_nodes],-0.02,0.02))
```

Выходной гейт объявим следующим образом:

```
# Выходной гейт – сколько данных поступает на выход из состояния ячейки.
# Связывает текущий вход с выходным гейтом.
ox = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes],
stddev=0.02))
# Связывает предыдущее скрытое состояние с выходным гейтом.
om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],
stddev=0.02))
# Смещение выходного гейта.
ob = tf.Variable(tf.random_uniform([1, num_nodes],-0.02,0.02))
```

Далее мы объявим переменные для состояния и выхода. Это переменные TensorFlow, представляющие внутреннее состояние ячейки и внешнее скрытое состояние ячейки LSTM. При объявлении вычислительной операции LSTM мы определяем, что переменные должны обновляться с использованием последних значений состояния ячейки и скрытых состояний, которые мы вычисляем, используя функцию `tf.control_dependencies(...)`.

```
# Переменные сохраняют состояние при развертывании.
# Скрытое состояние.
saved_output = tf.Variable(tf.zeros([batch_size, num_nodes]),
trainable=False, name='train_hidden')
# Состояние ячейки
saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]),
trainable=False, name='train_cell')
# Сохранение переменных на этапе валидации
saved_valid_output = tf.Variable(tf.zeros([1, num_
nodes]),trainable=False, name='valid_hidden')
saved_valid_state = tf.Variable(tf.zeros([1, num_
nodes]),trainable=False, name='valid_cell')
```

Наконец, мы объявим слой softmax, чтобы получить реальные прогнозы:

```
# Веса и смещения классификатора softmax.
w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size],
stddev=0.02))
b = tf.Variable(tf.random_uniform([vocabulary_size],-0.02,0.02))
```

Обратите внимание, что мы используем нормальное распределение с нулевым средним и небольшим стандартным отклонением. Это уместно, так как наша модель представляет собой простую одиночную ячейку LSTM. Однако когда сеть становится глубже (т. е. несколько ячеек LSTM, уложенных друг на друга), требуются более сложные методы инициализации. Один из таких методов известен как *инициализация Ксавье*, предложенная Ксавье Глоро и Йошуа Бенжи в их статье[1] «Проблемы обучения глубоких нейронных сетей с прямым распространением». Метод представлен в TensorFlow в качестве инициализатора переменной, как показано здесь: **https://github.com/tensorflow/docs/blob/r1.14/site/en/api_docs/python/tf/contrib/layers/xavier_initializer.md**.

Объявление ячейки LSTM и ее операций

Объявив веса и смещения, мы можем далее объявить операции в ячейке LSTM. Эти операции включают в себя следующее:

- расчет выходов, производимых входом и забывающим гейтом;
- расчет внутреннего состояния ячейки;
- расчет выхода, производимого выходным гейтом;
- расчет внешнего скрытого состояния.

Ниже приведена реализация нашей ячейки LSTM:

```
def lstm_cell(i, o, state):

    input_gate = tf.sigmoid(tf.matmul(i, ix) +
                            tf.matmul(o, im) + ib)
    forget_gate = tf.sigmoid(tf.matmul(i, fx) +
                             tf.matmul(o, fm) + fb)
    update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
    state = forget_gate * state + input_gate * tf.tanh(update)
    output_gate = tf.sigmoid(tf.matmul(i, ox) +
                             tf.matmul(o, om) + ob)
    return output_gate * tf.tanh(state), state
```

Входные данные и метки

Теперь объявим развернутые обучающие входы и метки. Обучающие входы представляют собой список пакетов данных длиной `num_unrolling` , где каждый пакет данных имеет размер `[batch_size, vocabulary_size]`:

```
train_inputs, train_labels = [],[]

for ui in range(num_unrollings):
```

[1] Understanding the difficulty of training deep feedforward neural networks. Glorot and Bengio: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, 2010.

```
    train_inputs.append(tf.placeholder(tf.float32,
                                shape=[batch_size,vocabulary_size],
                                name='train_inputs_ %d' %ui))
    train_labels.append(tf.placeholder(tf.float32,
                               shape=[batch_size,vocabulary_size],
                               name = 'train_labels_ %d' %ui))
```

Здесь мы также объявляем заполнители для входов и выходов этапа валидации, которые будут использоваться для вычисления перплексии. Обратите внимание, что мы не используем развертывание для вычислений, связанных с валидацией.

```
# Заполнители для данных валидации.
valid_inputs = tf.placeholder(tf.float32, shape=[1,vocabulary_size],
               name='valid_inputs')
valid_labels = tf.placeholder(tf.float32, shape=[1,vocabulary_size],
               name = 'valid_labels')
```

Последовательные вычисления для обработки последовательных данных

Здесь мы последовательно вычисляем результаты, полученные при развертывании обучающих входов. Мы также будем использовать дропаут (см. статью[1] «Дропаут: простой способ предотвратить переобучение нейронных сетей»), поскольку он повышает качество модели. Наконец, мы вычисляем значения логитов для всех скрытых выходных значений, полученных на обучающих данных:

```
# Хранит полученные выходы состояния всех развертываний,
# использованных при вычислении ошибки.
outputs = list()

# Эти две переменные Python итеративно обновляются
# на каждом шаге развертывания.
output = saved_output
state = saved_state

# Вычисляем скрытое состояние (output) и состояние ячейки (state)
# рекурсивно для всех шагов развертывания.
for i in train_inputs:
    output, state = lstm_cell(i, output, state)
    output = tf.nn.dropout(output,keep_prob=1.0-dropout)
    # Append each computed output value
    outputs.append(output)

# Вычисляем оценочные значения (логиты)
logits = tf.matmul(tf.concat(axis=0, values=outputs), w) + b
```

Затем, прежде чем вычислять ошибку, мы должны обновить выход и внешнее скрытое состояние до самого последнего текущего значения, которое вычислили ранее. Для этого добавляем условие `tf.control_dependencies` и сохраняем логит и вычисленную ошибку:

[1] Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Srivastava, Nitish, and others // Journal of Machine Learning Research 15 (2014): 1929–1958.

```
with tf.control_dependencies([saved_output.assign(output),
                              saved_state.assign(state)]):
    # Классификатор.
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=logits, labels=tf.concat(axis=0,
                                        values=train_labels)))
```

Мы также определяем логику прямого распространения для валидационных данных. Обратите внимание, что мы используем дропаут только во время обучения, но не во время валидации:

```
# Внутренняя логика на этапе валидации
# Вычисляем выход ячейки LSTM для валидационных данных
valid_output, valid_state = lstm_cell(
    valid_inputs, saved_valid_output, saved_valid_state)

# Вычисление логитов
valid_logits = tf.nn.xw_plus_b(valid_output, w, b)
```

Выбор оптимизатора

Мы будем использовать один из лучших стохастических оптимизаторов на основе градиента на сегодняшний день, известный под названием Adam. Здесь в коде `gstep` – это переменная, которая используется для уменьшения скорости обучения с течением времени. Мы обсудим этот параметр в следующем разделе. Кроме того, будем использовать отсечение, чтобы избежать взрыва градиента:

```
# Скорость обучения уменьшается при каждом увеличении gstep.
tf_learning_rate = tf.train.exponential_decay(0.001,gstep,
                   decay_steps=1, decay_rate=0.5)
# Оптимизатор Adam Optimizer и отсечение градиента
optimizer = tf.train.AdamOptimizer(tf_learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimizer = optimizer.apply_gradients(
zip(gradients, v))
```

Снижение скорости обучения

Как упоминалось выше, я использую затухающую скорость обучения. Снижение скорости обучения с течением времени является обычной техникой, применяемой в глубоком обучении для достижения лучшего качества и снижения переобучения. Ключевой идеей здесь является понижение скорости обучения, если перплексия на стадии валидации не уменьшается в течение предварительно определенного числа эпох. Давайте посмотрим, как именно это реализовано.

Сначала мы объявляем `gstep` и операцию для увеличения `gstep`, которая называется `inc_gstep`:

```
# Снижение скорости обучения
gstep = tf.Variable(0,trainable=False,name='global_step')
# Выполнение этой операции приведет к увеличению значения gstep
```

```
# и снижению скорости обучения
inc_gstep = tf.assign(gstep, gstep+1)
```

Теперь мы можем написать простой код для вызова операции `inc_gstep` всякий раз, когда перплексия при валидации не уменьшается:

```
# Если перплексия при валидации не уменьшается
# заданное число эпох подряд,
# то снижаем скорость обучения
decay_threshold = 5
# Следим за повышением перплексии
decay_count = 0
min_perplexity = 1e10

# Вычисление снижения скорости обучения.
def decay_learning_rate(session, v_perplexity):
  global decay_threshold, decay_count, min_perplexity
  # Снижение скорости обучения.
  if v_perplexity < min_perplexity:
    decay_count = 0
    min_perplexity= v_perplexity
  else:
    decay_count += 1

if decay_count >= decay_threshold:
  print('\t Reducing learning rate')
  decay_count = 0
  session.run(inc_gstep)
```

Мы обновляем `min_perplexity` всякий раз, когда обнаруживаем новую минимальную перплексию при валидации. Кроме того, `v_perplexity` – текущая перплексия валидации.

Получение прогнозов

Теперь мы можем делать прогнозы, просто применяя активацию softmax к вычисленным раньше логитам. Мы также объявляем операцию прогнозирования для логитов на стадии валидации:

```
train_prediction = tf.nn.softmax(logits)
# Обязательно обновляем переменные состояния
# до перехода к следующему циклу генерации
with tf.control_dependencies([saved_valid_output.assign(valid_output),
                              saved_valid_state.assign(valid_state)]):
    valid_prediction = tf.nn.softmax(valid_logits)
```

Вычисление перплексии

Мы определили, что такое перплексия, в главе 7. Если коротко, перплексия – это мера того, насколько LSTM-сеть «впадает в замешательство» при выборе следующей *n*-граммы из набора потенциальных вариантов. Следовательно, более высокая перплексия означает низкое качество прогнозов, и наоборот:

```
train_perplexity_without_exp = tf.reduce_sum(
    tf.concat(train_labels,0)*-tf.log(tf.concat(
        train_prediction,0)+1e-10))/(num_unrollings*batch_size)
# Вычисляем перплексию валидации
valid_perplexity_without_exp = tf.reduce_sum(valid_labels*-tf.
log(valid_prediction+1e-10))
```

Сброс состояний

Мы используем сброс состояния, так как обрабатываем несколько документов. В начале обработки нового документа мы возвращаем скрытое состояние обратно в ноль. Однако не очень ясно, помогает ли сброс состояния на практике. С одной стороны, выглядит вполне логично сбросить память ячейки LSTM в начале каждого документа на ноль, когда начинаешь читать новую историю. С другой – это создает систематическое смещение переменных состояния в сторону нуля. Мы рекомендуем вам попробовать запустить алгоритм как со сбросом состояния, так и без него, и посмотреть, какой метод работает лучше.

```
# Сброс состояния обучения
reset_train_state = tf.group(tf.assign(saved_state,
                             tf.zeros([batch_size, num_nodes])),
                             tf.assign(saved_output, tf.zeros(
                             [batch_size, num_nodes])))

# Сброс состояния валидации
reset_valid_state = tf.group(tf.assign(saved_valid_state,
                             tf.zeros([1, num_nodes])),
                             tf.assign(saved_valid_output,
                             tf.zeros([1, num_nodes])))
```

Жадная выборка против унимодальности

Это довольно простой метод, в котором стохастически выбирается следующий прогноз из *n* лучших кандидатов, найденных LSTM. Кроме того, мы свяжем вероятность выбора определенного кандидата с вероятностью того, что этот кандидат будет следующей биграммой:

```
def sample(distribution):

  best_inds = np.argsort(distribution)[-3:]
  best_probs = distribution[best_inds]/
  np.sum(distribution[best_inds])
  best_idx = np.random.choice(best_inds,p=best_probs)
  return best_idx
```

Генерация нового текста

Наконец, мы объявляем заполнители, переменные и операции, необходимые для создания нового текста. Это делается по аналогии с подготовкой к стадии обучения. Сначала мы объявляем входной заполнитель и переменные для состояния

и вывода. Далее объявляем операции сброса состояния. Наконец, определяем вычисления ячейки LSTM и прогнозы для нового текста:

```
# Генерация текста: пакет 1, без развертывания.
test_input = tf.placeholder(tf.float32, shape=[1, vocabulary_size],
name = 'test_input')

# Аналогичные переменные для стадии тестирования
saved_test_output = tf.Variable(tf.zeros([1,
                                num_nodes]),
                                trainable=False, name='test_hidden')
saved_test_state = tf.Variable(tf.zeros([1,
                               num_nodes]),
                               trainable=False, name='test_cell')

# Вычисление выхода ячейки LSTM для тестовых данных
test_output, test_state = lstm_cell(
test_input, saved_test_output, saved_test_state)

# Обязательно обновляем переменные состояния перед
# переходом к следующей итерации
with tf.control_dependencies([saved_test_output.assign(test_output),
                              saved_test_state.assign(test_state)]):
    test_prediction = tf.nn.softmax(tf.nn.xw_plus_b(test_output,
                                    w, b))

# Сброс состояния на тестовой стадии
reset_test_state = tf.group(
    saved_test_output.assign(tf.random_normal([1,
                             num_nodes],stddev=0.05)),
    saved_test_state.assign(tf.random_normal([1,
                            num_nodes],stddev=0.05)))
```

Пример сгенерированного текста

Давайте взглянем на текст, сгенерированный LSTM после 50 шагов обучения:

```
they saw that the birds were at her bread, and threw behind him a comb
which
made a great ridge with a thousand times thousands of spikes. that
was a
collier.
the nixie was at church, and thousands of spikes, they were flowers,
however, and had hewn through the glass, the children had formed a
hill of mirrors, and was so slippery that it was impossible for the
nixie to cross it. then she thought, i will go home quickly and
fetch my axe, and cut the hill of glass in half. long before she
returned, however, and had hewn through the glass, the children saw
her from afar,
and he sat down close to it,
and was so slippery that it was impossible for the
nixie to cross it.

они увидели, что птицы были у нее на хлебе, и бросили за ним расческу
которая
```

```
сделала огромный хребет с тысячей раз по тысяче шипов. это
был
угольщик.
никси была в церкви, и тысячи шипов, они были цветами, однако, и прорубили стекло, дети
устроили холм зеркал, и было настолько скользко, что никси не могла пересечь его. Затем она
подумала, я быстро пойду домой, возьму свой топор и рассеку стеклянный холм пополам. однако
задолго до того, как она вернулась и прорубила стекло, дети увидели
ее издалека,
и он сел рядом с ним,
и был настолько скользким, что никси не могла пересечь его.
```

Как видите, этот текст выглядит намного лучше, чем сгенерированный при помощи RNN. На самом деле в нашем учебном корпусе присутствует сказка о водяной фее-никси[1]. Тем не менее наша LSTM-сеть не просто выводит этот текст, но добавляет больше красок к этой истории, вводя новые сущности, такие как разговоры о церкви и цветах, которых нет в оригинальном тексте. Далее мы сравним текст, сгенерированный стандартной LSTM-сетью, с текстами, созданными сетью с замочными скважинами и GRU.

Сравнение качества текстов на выходе разных модификаций LSTM

Давайте попробуем экспериментально сравнить качество сгенерированных текстов у трех моделей – обычной LSTM-сети, модификации с замочными скважинами и GRU. Исходный код полного упражнения хранится в файле в `lstm_extensions.ipynb`, расположенном в папке `ch8`.

Обычная LSTM-сеть

Мы начнем с реализации стандартной модели LSTM, но не будем здесь повторять ее код, поскольку он идентичен тому, что мы обсуждали ранее. Наша цель – получить текст, сгенерированный LSTM-сетью.

Обзор

Давайте вспомним, как устроена стандартная LSTM-сеть. Как мы уже говорили, она состоит из следующих ключевых компонентов:

- **входной гейт** – решает, какая часть текущего ввода записывается в состояние ячейки;
- **забывающий гейт** – решает, сколько из предыдущего состояния ячейки будет записано в текущее состояние;
- **выходной гейт** – решает, сколько информации из состояния ячейки отправляется на вывод во внешнее скрытое состояние.

На рис. 8.1 проиллюстрированы связи между гейтами, входными данными, состоянием ячейки и внешними скрытыми состояниями.

[1] Водяная фея из немецкого фольклора, способная менять облик. – *Прим. перев.*

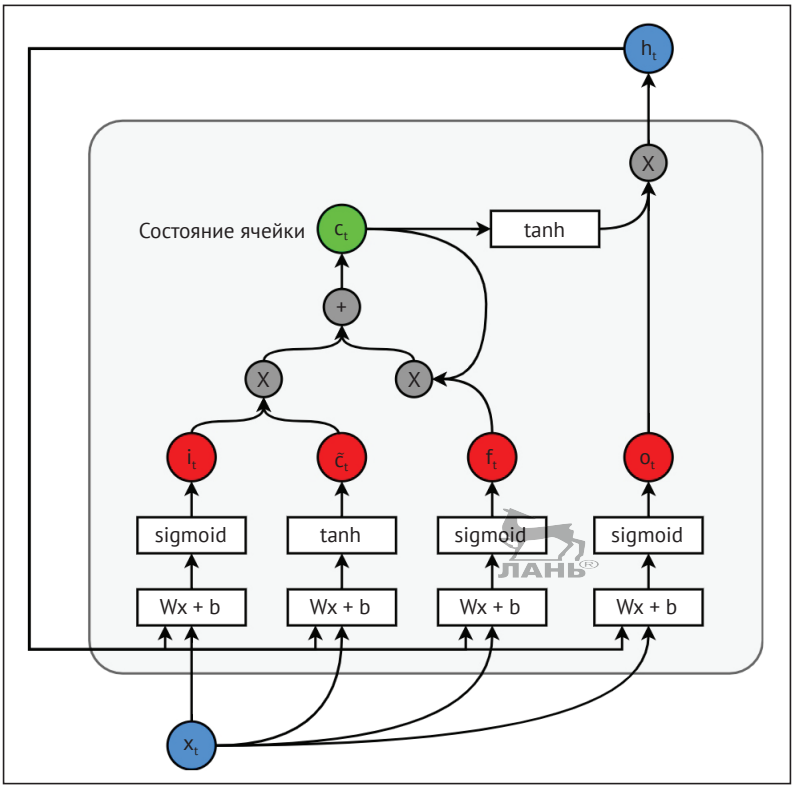


Рис. 8.1 ❖ Ячейка LSTM

Пример сгенерированного текста

Начнем с текстов, созданных стандартной LSTM-сетью после одного шага обучения и после 25 шагов обучения на сказках братьев Гримм.

Текст, созданный на шаге 1:

```
emy that then the to they the the to and and and then there the to
the to the withe there the the to, and ther, and ther tthe the the the
withe the the the the wid the th to e the there to, and the the the
the the wid the the the to, the and to the was and and was the when
hind the whey the the to and was the whe wous thout hit the to hhe was
they his up the was was the wou was and and wout the the ous to hhe
the was and was they hind and and then the the the wit to the wther
thae wid the and the the wit the ther, the there the to the wthe wit
the the the the wit up the they og a and the whey the the ous th the
wthe the ars to and the whey it a and whe was they the ound the was
whe was and and to ther then the and ther the wthe art the the and and
the the the to and when the the wie to the wthe wit up the whe wou
wout hit hit the the the to the whe was aou was to t the out and the
and hit the the the with then the wie the to then the the to, the to a
t to the the wit up he the wit there
```

Текст, созданный на шаге 25:

```
there, said the father for a while, and her trouble she was to carry
the mountain. then they were all the child, and they were once and
only sighed, but they said, i am as
old now as the way and drew the child, and he began and wife looked at
last and said, i have the child, fath-turn, and
hencefore they were to himself, and then they trembled, hand all three
days with him. when the king of the golden changeling, and his wife
looked at last and only one lord, and then he was laughing, wished
himself, and then he said
nothing and only sighed. then they had said, all the changeling
laugh, and he said, who was still done, the bridegroom, and he went
away to him, but he did not trouble to the changeling away, and then
they were over this, he was all to the wife, and she said,
has the wedding did gretel give her them, and said, hans in a place.
in her trouble shell into the father. i am you.
the king had said, how he was to sweep. then the spot on hand but the
could give you doing there,
```

```
(там, сказал отец некоторое время, и ее беда она должна была нести гору. тогда они все были
ребенком, и они однажды вздохнули, но они сказали, я как
старый как, кстати, и нарисовал ребенка, и он начал, и жена наконец посмотрела и сказала
у меня есть ребенок,
следовательно, они были к себе, и затем они дрожали, передавайте ему все три дня. когда король
золотого подменыша и его жена посмотрели наконец-то и только на одного господина, а потом он
засмеялся, пожелал себе, а потом сказал
ничего и только вздохнул. затем они сказали, весь подменышев смех, и он сказал, кто еще был
готов, жених, и он ушел к нему, но он не беспокоился о подменыше, и тогда они были над этим,
он был все для жена, и она сказала,
неужели свадьба дала ей гретель и сказала, что ганс на месте. в ее беде переходит в отца.
я есть ты.
король сказал, как он был подметать. тогда место под рукой, но может дать вам делать там,)
```

Мы можем отметить, что на шаге 25 наблюдается довольно резкое повышение качества текста по сравнению с шагом 1. Кроме того, этот текст выглядит намного лучше, чем текст, который мы видели в главе 6, когда 100 сказок были использованы для обучения RNN.

Пример генерации текста при помощи GRU

Прежде всего кратко освежим в памяти, из чего состоит GRU, и сформируем код для реализации ячейки, а затем познакомимся с примером текста, сгенерированного этой модификацией нейросети.

Обзор

GRU – это элегантное упрощение базовых операций LSTM, основанное на двух ключевых модификациях (рис. 8.2):

- объединение внутреннего состояния ячейки и внешнего скрытого состояния в одно состояние;
- объединение входного гейта и забывающего гейта в один гейт обновления.

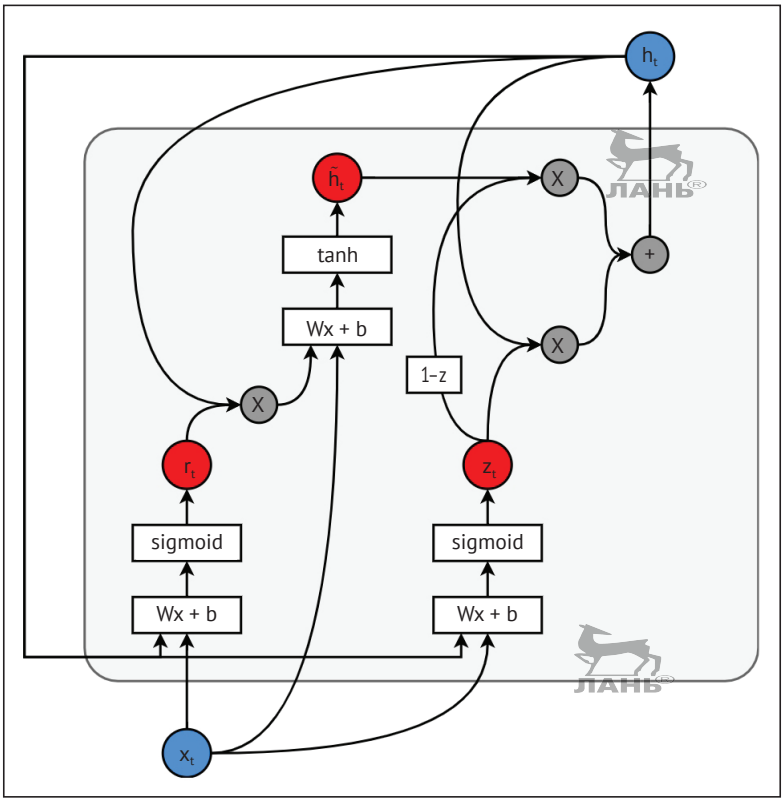


Рис. 8.2 ❖ Ячейка GRU

Исходный код

Объявляем ячейку GRU:

```
def gru_cell(i, o):
    """Create a GRU cell."""
    reset_gate = tf.sigmoid(tf.matmul(i, rx) + tf.matmul(o, rh)
                            + rb)
    h_tilde = tf.tanh(tf.matmul(i,hx) + tf.matmul(
        reset_gate * o, hh) + hb)
    z = tf.sigmoid(tf.matmul(i,zx) + tf.matmul(o, zh) + zb)
    h = (1-z)*o + z*h_tilde

    return h
```

Затем мы будем вызывать этот метод, как это делали ранее в другом примере:

```
for i in train_inputs:
    output = gru_cell(i, output)
    output = tf.nn.dropout(output,keep_prob=1.0-dropout)
    outputs.append(output)
```

Пример сгенерированного текста

Сравним текст, созданный GRU после одного шага обучения и 25 шагов обучения на наборе сказок.

Текст, созданный на шаге 1:

```
hing ther that ther her to the was shen andmother to to her the
cake, and the caked the woked that the wer hou shen her the the the
that her her, and to ther to ther her that the wer the wer ther the
wong are whe was the was so the the caked her the wong an the woked
the wolf the soought and was the was he grandmred the wolf sas shen
that ther to hout her the the cap the wolf so the wong the soor ind
the wolf the when that, her the the wolf to and the wolf sher the the
cap the cap. the wolf so ther the was her her, the the the wong and
whe her the was her he grout the ther, and the cap., and the caked the
the ther the were cap and the would the the wolf the was the whe wher
cad-the cake the was her her, he when the ther, the wolf so the that,
and the wolf so and her the the the cap. the the wong to the wolf,
andmother the cap. the so to ther ther, the woked he was the was the
when the caked her cad-ing and the cake, and
```

Текст, созданный на шаге 25:

```
you will be sack, and the king's son, the king continued, and he was
about to them all, and that she was strange carry them to somether,
and who was there, but when the shole before the king, and the king's
daughter was into such into the six can dish of this wine before the
said, the king continued, and said to the king, when he was into the
castle to so the king.
then the king was stranged the king.
then she said, and said that he saw what the sack, but the king, and
the king content up the king.
the king had the other, and said, it is not down to the king was in
the blower to be took them. then the king sack, the king, and the
other, there, and
said to the other, there, and the king, who had been away, the six
content the six conved the king's strong one, they were not down the
king.
then she said to her, and saw the six content until there, and the
king content until the six convered the
```

```
(вы будете уволены, и сын короля, король продолжал, и он был
о них всех, и что она была странной, неси их кому-то,
и кто был там, но когда башмак перед королем и дочь
царя были в таковых в шесть банок этого вина перед
сказанным, король продолжил и сказал королю, когда он был
в замке так король.
тогда король был задушен королем.
затем она сказала, и сказал, что он видел, что мешок, кроме царя, и
король доволен королем.
у короля был другой, и сказал, что это не так, чтобы король был в
ветродуе чтобы их забрали. тогда король мешок, король и
другое, там и
там сказал другому, и король, который был вдали, шесть
```

```
довольствуясь шестеркой, они одержали верх над силой короля, они не падали ниц
король.
затем она сказала ей, и увидела шесть довольно там, и
король доволен, пока шестерка не объявила)
```

Можно отметить, что с точки зрения качества текста GRU не демонстрируют заметного улучшения по сравнению со стандартными LSTM. При этом выходные данные GRU имеют больше повторов в тексте, чем LSTM. В данном примере бросается в глаза слово «*король*». Возможно, это связано с ухудшением долговременной памяти, вызванной упрощением модели, т. е. наличием только одного состояния по сравнению с двумя состояниями в стандартной LSTM-сети.

LSTM с замочными скважинами

В этом разделе мы обсудим модификацию LSTM с замочными скважинами и отличия от стандартного LSTM. Далее мы обсудим их реализацию, а затем текст, сгенерированный LSTM с замочными скважинами.

Обзор

Замочные скважины – это, по сути, способ, с помощью которого входные, забывающие и выходные гейты могут непосредственно видеть состояние ячейки вместо ожидания внешнего скрытого состояния (рис. 8.3).

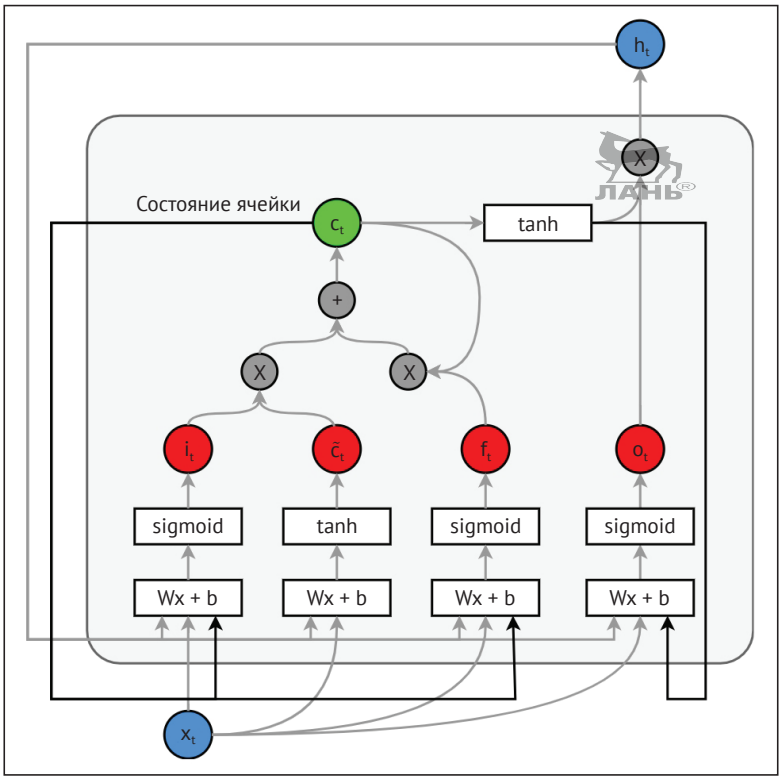


Рис. 8.3 ❖ LSTM с замочной скважиной

Исходный код

Обратите внимание, что я использую диагональные соединения, поскольку заметил, что недиагональные соединения с замочными скважинами, предложенные в статье[1] Джерса и Шмидхубера, в задаче моделирования языка скорее вредят, чем помогают. Поэтому я принял другой вариант, в котором используются диагональные соединения скважин, предложенные в статье[2] «Архитектуры рекуррентных сетей с долгой краткосрочной памятью в задачах масштабного акустического моделирования».

Ниже приведена реализация кода:

```
def lstm_with_peephole_cell(i, o, state):

input_gate = tf.sigmoid(tf.matmul(i, ix) + state*ic +
                                tf.matmul(o, im) + ib)
forget_gate = tf.sigmoid(tf.matmul(i, fx) + state*fc +
                        tf.matmul(o, fm) + fb)
update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
state = forget_gate * state + input_gate * tf.tanh(update)
output_gate = tf.sigmoid(tf.matmul(i, ox) + state*oc +
                        tf.matmul(o, om) + ob)

return output_gate * tf.tanh(state), state
```

Затем мы будем вызывать этот метод для каждой партии входных данных для охвата всех временных шагов (то есть временных шагов `num_unrollings`), как в этом коде:

```
for i in train_inputs:
    output, state = lstm_with_peephole_cell(i, output, state)
    output = tf.nn.dropout(output,keep_prob=1.0-dropout)
    outputs.append(output)
```

Пример сгенерированного текста

Давайте взглянем на тексты, созданные LSTM с замочными скважинами после одного шага обучения и 25 шагов обучения, и сравним их с текстами, полученными при помощи других модификаций LSTM.

Ниже приведен текст, созданный на шаге 1:

```
our oned he the the hed the the the he here hed he he e e and her and
the ther her the then hed and her and her her the hed her and the the
he he ther the hhe the he ther the whed hed her he hthe and the the
the ther the to e and the the the ane and and her and the hed ant and
the and ane hed and ther and and he e the th the hhe ther the the and
the the the the the the hed and ther hhe wher the her he he and he
hthe the the the he the then the he he e and the the the and and the
the the ther to he hhe wher ant the her and the hed the he he the and
ther and he the and and the ant he he e the and ther he e and ther
here th the whed
```

---

[1] Recurrent Nets that Time and Count. Gers and Schmidhuber: Neural Networks, 2000.

[2] Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling. Sak, Senior and Beaufays, Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH, 2014: 338–342.

Следующий текст создан на шаге 25:

```
will, it was there, and it was me, and i trod on the stress and there
is a stone and the went and said, klink, and that the princess and
they said, i will not stare
it, the wedding and that the was of little the sun came in the sun
came out, and then the wolf is took a little coat and i were at little
hand and beaning therein and said, klink, and broke out of the shoes
he had the wolf of the were to patches a little put into the were, and
they said, she was to pay the bear said, "ah, that they come to the
well and there is a stone and the wolf were of the light, and that the
two old were of glass there is a little that his
well as well and wherever a stone
and they were the went to the well, and the went the sun came in the
seater hand, and they said, klink, and broke in his sead, and i were
my good one
the wedding and said, that the two of slapped to said to said, "ah,
that his store once the worl's said, klink, but the went out of a
patched on his store, and the wedding and said, that
```

```
(будет, это было там, и это был я, и я наступил на напряжение и там
это камень, и пошел и сказал, клинк, и что принцесса и
они сказали, я не буду смотреть
это, свадьба и что было мало солнца взошло на солнце
вышел, а затем волк взял маленькое пальто, и я был немного
рука и боб в нем и сказал, клинк, и выскочил из обуви
у него был волк, чтобы повязки были немного вставлены в него, и
они сказали, что она должна была заплатить медведю, сказала: "ах, что они приходят к
ну и есть камень и волк были света, и что
два старых стекла были мало что его
ну и везде где камень
и они пошли к колодцу, и взошло солнце взошло в
руке местного жителя, и они сказали, клинк, и сломал в его sead, и я был
мой хороший
на свадьбу и сказал, что две пощечины сказал, чтобы сказал: "ах,
что его магазин когда-то сказал, клинк, но вышел из
залатали на своем магазине и свадьбу и сказали, что)
```

Текст, созданный с помощью LSTM с замочными скважинами, выглядит грамматически бедным по сравнению с результатом работы стандартной LSTM-сети или GRU. Далее мы выполним количественное сравнение трех методов с точки зрения перплексии.

Обучение нейросети и проверка перплексии

На рис. 8.4 представлено поведение перплексии во времени для стандартной LSTM, модификации с замочными скважинами и GRU. Во-первых, мы видим, что отсутствие дропаута дает значительное снижение перплексии обучения. Тем не менее мы не должны делать вывод, что дропаут отрицательно влияет на качество модели, так как эта привлекательная низкая перплексия связана с переобучени-

ем. Это видно из графика перплексии при валидации. Хотя перплексия при обучении LSTM без дропаута очевидно конкурирует с моделями, которые используют дропаут, перплексия при валидации намного выше, чем у этих моделей. Это доказывает, что дропаут фактически помогает нам в задаче генерации языка.

Кроме того, из всех методов, которые используют дропаут, наилучшее качество обеспечивают LSTM и GRU. Весьма удивительно, что LSTM с замочными скважинами демонстрирует ощутимо худшую перплексию при обучении и немного худшую при валидации. Это означает, что замочные скважины никак не помогают решить нашу задачу, но вместо этого затрудняют оптимизацию, вводя больше параметров в модель. Исходя из этого наблюдения, далее в книге мы будем выполнять упражнения со стандартной LSTM-сетью. Эксперименты с GRU я оставляю в качестве самостоятельной работы для читателей.

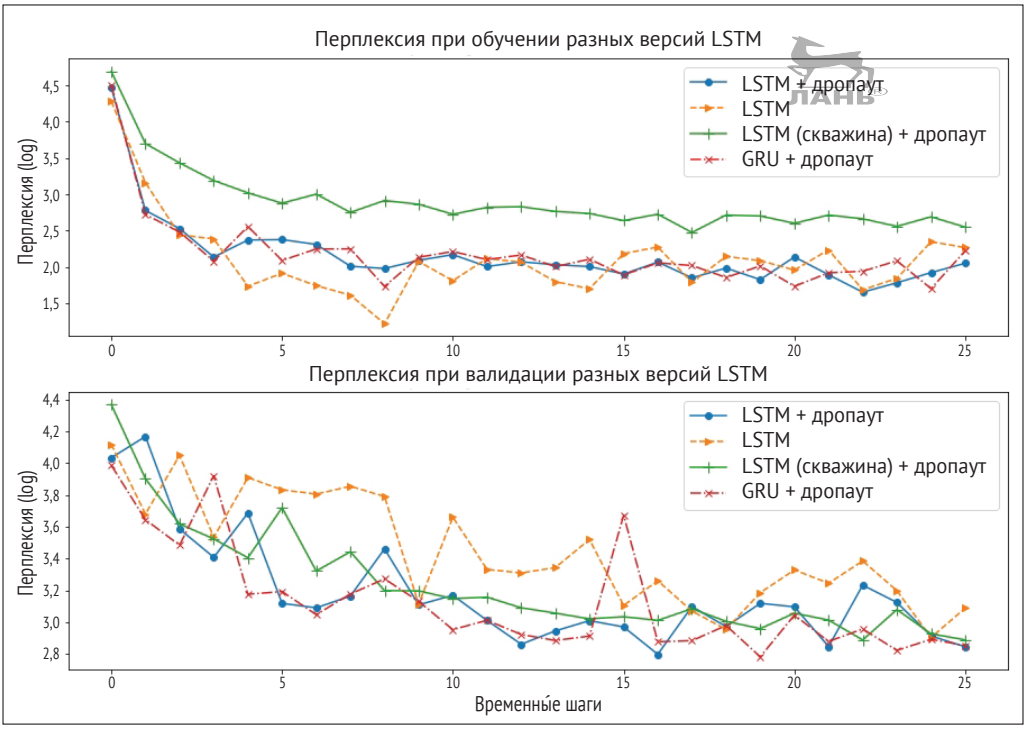


Рис. 8.4 ❖ Изменение перплексии во времени при обучении и валидации

В текущей литературе предполагается, что среди LSTM и GRU нет явного победителя, и многое зависит от задачи (см. статью[1] «Эмпирическая оценка управляемых рекуррентных нейронных сетей в задаче моделирования последовательностей»).

[1] Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, Chung and others, NIPS 2014 Workshop on Deep Learning, December 2014.

Модификация LSTM – лучевой поиск

Как мы видели ранее, некоторые методы заметно улучшают качество текста. Теперь давайте посмотрим, способен ли на это лучевой поиск, который мы обсуждали в главе 7. При лучевом поиске мы рассматриваем ряд шагов (называемых лучом) и выбираем луч (т. е. последовательность биграмм), имеющий наибольшую совместную вероятность среди всех лучей. Совместная вероятность рассчитывается путем умножения вероятностей предсказания каждой биграммы в луче. Обратите внимание, что это жадный поиск, поэтому мы будем выбирать лучших кандидатов итеративно по мере ветвления дерева. Следует отметить, что этот поиск не детерминирован, т. е. не обязательно приведет к лучшему в мире лучу.

Реализация лучевого поиска

Для реализации лучевого поиска нам нужно только изменить технику генерации текста. Операции обучения и проверки остаются прежними. Однако код будет более сложным, чем набор операций генерации текста, который мы видели ранее. Этот код находится в конце файла упражнения `lstm_for_text_generation.ipynb` в папке `ch8`.

Сначала мы определим длину луча `beam_length` – на сколько шагов мы смотрим в будущее, и `beam_neighbors` – количество кандидатов, которые сравниваем на каждом временном шаге:

```
beam_length = 5
beam_neighbors = 5
```

Объявляем заполнители в количестве `beam_neighbor`, чтобы запоминать лучших кандидатов на каждом временном шаге:

```
sample_beam_inputs = [tf.placeholder(tf.float32, shape=[1, vocabulary_
size]) for _ in range(beam_neighbors)]
```

Далее объявляем два заполнителя для хранения наилучшего найденного жадным поиском глобального индекса луча и локального индекса наилучшего кандидата луча, которых мы будем использовать для продолжения построения прогноза на следующем этапе:

```
best_beam_index = tf.placeholder(shape=None, dtype=tf.int32)
best_neighbor_beam_indices = tf.placeholder(shape=[beam_neighbors],
dtype=tf.int32)
```

Объявляем переменные состояния и выхода для каждого кандидата луча, как мы делали ранее для одиночного прогноза:

```
_ in range(beam_neighbors)]
saved_sample_beam_state = [tf.Variable(tf.zeros([1, num_nodes])) for _
in range(beam_neighbors)]
```

Объявляем операции сброса состояния:

```
reset_sample_beam_state = tf.group(
    *[saved_sample_beam_output[vi].assign(tf.zeros([1, num_nodes]))
for vi in range(beam_neighbors)],
```

```
    *[saved_sample_beam_state[vi].assign(tf.zeros([1, num_nodes])) for
vi in range(beam_neighbors)]
)
```

Кроме того, нам понадобятся расчеты для вывода состояния ячейки и прогнозов для каждого луча:

```
# Вычисляем состояние lstm_cell state и выход для каждого луча.
sample_beam_outputs, sample_beam_states = [],[]
for vi in range(beam_neighbors):
    tmp_output, tmp_state = lstm_cell(
        sample_beam_inputs[vi], saved_sample_beam_output[vi],
        saved_sample_beam_state[vi]
    )
    sample_beam_outputs.append(tmp_output)
    sample_beam_states.append(tmp_state)

# Для данного набора лучей выводим список векторов прогнозов
# с размером beam_neighbors.
# Каждый луч содержит предсказания для полного словаря.
sample_beam_predictions = []
for vi in range(beam_neighbors):
    with tf.control_dependencies([saved_sample_beam_output[vi].
assign(sample_beam_outputs[vi]),
                                  saved_sample_beam_state[vi].
assign(sample_beam_states[vi])]):
        sample_beam_predictions.append(tf.nn.softmax(tf.nn.xw_
plus_b(sample_beam_outputs[vi], w, b)))
```

Далее мы определим новый набор операций для обновления состояния и выходных переменных каждого луча индексами лучших лучей-кандидатов, найденными на каждом шаге. Это важно для каждого шага, поскольку лучшие кандидаты в лучи не будут равномерно разветвляться на протяжении всего дерева. Пример изображен на рис. 8.5. Наилучшие лучи-кандидаты выделены жирным шрифтом и стрелками.

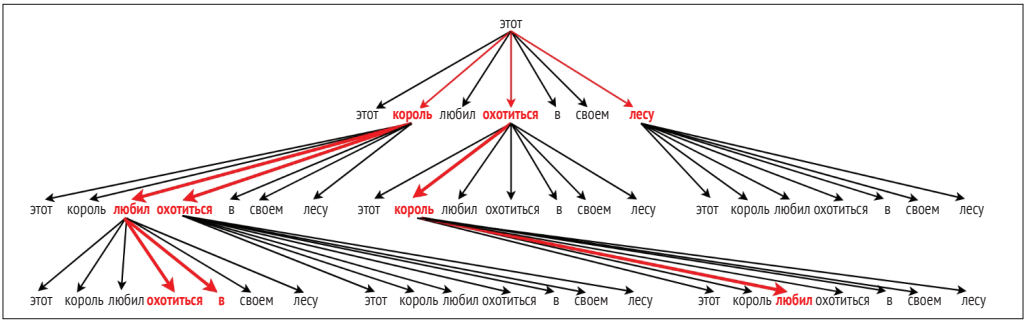


Рис. 8.5 ❖ Лучевой поиск, иллюстрирующий необходимость обновления состояний луча на каждом этапе

Как видите, в данном случае кандидаты не выбираются равномерно, имея всегда одного кандидата из поддерева (набора стрелок, начинающихся с одной и той

же точки) на заданной глубине. Например, на втором уровне нет никаких кандидатов, продолжающих луч *король → охотится*, поэтому обновление состояния, вычисленное для этого пути, становится бесполезным. Таким образом, состояние, которое мы хранили для этого пути, следует заменить состоянием для пути *король → любил*, поскольку теперь есть два пути, совместно использующих данный родительский путь. Мы будем использовать следующий код для таких замен состояний:

```
stacked_beam_outputs = tf.stack(saved_sample_beam_output)
stacked_beam_states = tf.stack(saved_sample_beam_state)

update_sample_beam_state = tf.group(
    *[saved_sample_beam_output[vi].assign(tf.gather_nd(stacked_beam_
outputs,[best_neighbor_beam_indices[vi]])) for vi in range(beam_
neighbors)],
    *[saved_sample_beam_state[vi].assign(tf.gather_nd(stacked_beam_
states,[best_neighbor_beam_indices[vi]])) for vi in range(beam_
neighbors)]
)
```

Пример текста, созданного лучевым поиском

Давайте посмотрим, как лучевой поиск влияет на качество работы LSTM. Пожалуй, результат выглядит лучше, чем раньше:

```
and they sailed to him and said,
oh, queen. where heavens, she went to her, and thumbling
where the whole kingdom likewis, and that she had given him as that
he had to eat, and they gave him the money, hans took his head that
he had been the churchyar, and they gave him the money, hans took his
head that he had been the world, and, however do that, he have begging
his that he was
placed where they were brought in the mouse's horn again. where
have, you come? then thumbling where the world, and when they came to
them, and that he was soon came back, and then the will make that they
hardled the world, and, however do that heard him, they have gone out
through the room, and said the king's son was again and said,
ah, father, i have been in a dream, for his horse again,
answered the door. when they saw
each other that they had been. then they saw they had been.

(и они приплыли к нему и сказали:
о королева. где святые небеса, она пошла к ней, и мальчик-с-пальчик
где все царство точно так же, и что она дала ему все что
он должен был есть, и они дали ему деньги, ганс взял его голову, что
прежде был церковником, и они дали ему деньги, ганс взял его
голову, что прежде был миром, и, как бы то ни было, он просил
его, что он был
поместили туда, где их снова привели в рог мыши. откудова же
ты пришел? затем мальчик-с-пальчик там где мир, и когда они пришли
к ним, и что он скоро вернулся, а затем сделают, что они
покорят мир, и, как бы его ни услышали, они ушли
через комнату, и сказал сын царя был снова и сказал:
```

```
ах, отец, мне снова снилась его лошадь,
отворил дверь. когда они увидели
друг друга, что они были. затем они увидели, что были.)
```

По сравнению с текстом, созданным LSTM, этот текст, вероятно, имеет больше вариаций, сохраняя при этом грамматическую согласованность[1]. Таким образом, лучевой поиск предлагает более качественные прогнозы по сравнению с предсказаниями по одному слову за раз. Кроме того, мы видим, что LSTM интересно объединяет различные элементы из сказок, чтобы придумать любопытные концепции (например, рог мыши, объединение Мальчика-с-пальчика и Ганса, персонажа из другой сказки). Но все же встречаются случаи, когда слова вместе не имеют большого смысла. Давайте подумаем, как мы можем сделать LSTM еще лучше.

Генерация текста на уровне слов вместо *n*-грамм

Продолжаем обсуждать способы улучшения LSTM. Во-первых, вы узнаете, как растет количество параметров модели, если использовать признаки слова в унитарном коде. Это побуждает нас использовать низкоразмерные векторы слов вместо векторов с унитарным кодированием. Наконец, мы рассмотрим пример использования векторного представления слов для генерации более качественного текста по сравнению с использованием биграмм. Исходный код для этого раздела хранится в файле `lstm_word2vec.ipynb` в папке `ch8`.

Проклятие размерности

Одним из основных ограничений, мешающих нам использовать слова вместо *n*-грамм в качестве входных данных LSTM, является резкое увеличение количества параметров в нашей модели. Давайте разберемся в этом на примере. Предположим, что у нас есть вход с размерностью 500 и состояние ячейки с размерностью 100. В результате получается около 240 тыс. параметров (исключая слой softmax):

$$=\sim 4 \times (500 \times 100 + 100 \times 100 + 100) =\sim 240 \text{ тыс.}$$

Давайте теперь увеличим размерность входа до 1000. Теперь общее количество параметров будет примерно 440 тыс.:

$$=\sim 4 \times (1000 \times 100 + 100 \times 100 + 100) =\sim 440 \text{ тыс.}$$

Как видите, при увеличении входной размерности на 500 единиц число параметров возросло на 200 тыс. Это не только увеличивает вычислительную сложность, но также увеличивает риск переобучения из-за большого количества параметров. Итак, нам нужны способы ограничения размерности входа.

[1] Мы приводим дословный перевод сгенерированного текста, не подвергая его правке, и предлагаем читателю самостоятельно оценить уровень смыслового наполнения. Впрочем, грамотность и грамматическая согласованность английского текста действительно стали лучше. – *Прим. перев.*

Word2vec спешит на помощь

Как вы помните, Word2vec не только обеспечивает более низкоразмерное представление слов по сравнению с унитарным кодированием, но также дает семантически обоснованные признаки. Давайте рассмотрим три слова: «*кошка*», «*собака*» и «*вулкан*».

Если мы представим в унитарном коде только эти слова, то получим одинаковое евклидово расстояние между ними:

расстояние (*кошка*, *вулкан*) = *расстояние* (*кошка*, *собака*).

Однако если мы изучим представления слов, то расстояния будут различаться:

расстояние (*кошка*, *вулкан*) > *расстояние* (*кошка*, *собака*).

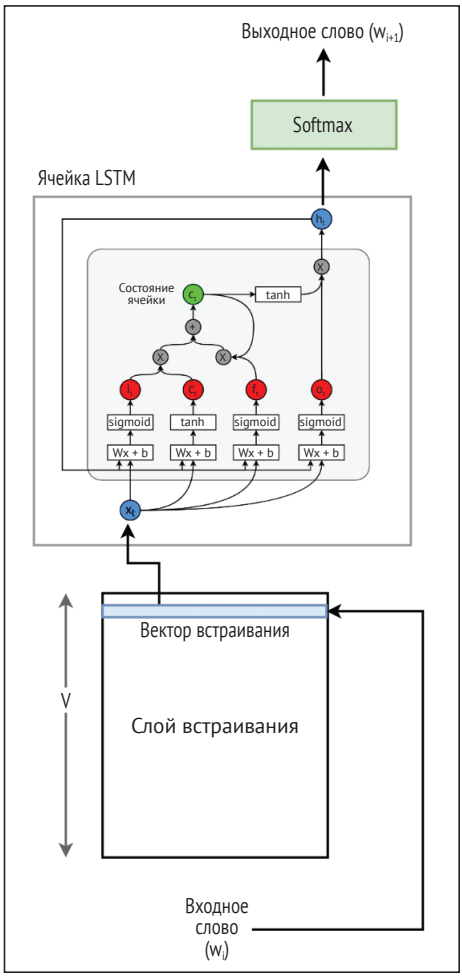


Рис. 8.6 ❖ Структура LSTM-сети с использованием векторных представлений слов

Мы хотели бы, чтобы наши признаки представляли второй случай, где расстояние между семантически похожими вещами меньше, чем между разнородными. Следовательно, модель сможет генерировать более качественный текст.

Генерация текста с помощью Word2vec

Теперь наша LSTM-сеть становится немного сложнее по сравнению со стандартной версией, так как мы вставляем слой представлений между входом и LSTM. Рисунок 8.6 изображает обобщенную архитектуру LSTM–Word2vec. Код реализации доступен в качестве упражнения в файле `lstm_word2vec.ipynb`, расположенном в папке `ch8`.

Сначала мы изучим векторы слов, используя модель Continuous Bag-of-Words (CBOW). Ниже приведены некоторые из лучших отношений, изученных нашей моделью Word2vec:

```
Nearest to which: what
Nearest to not: bitterly, easily, praying, unseen
Nearest to do: did
Nearest to day: evening, sunday
Nearest to two: many, kinsmen
Nearest to will: may, shall, 'll
Nearest to pick-axe: ladder
Nearest to stir: bestir, milk
```

```
Ближайшие к кто: что
Ближайшие к нет: горько, легко, молится, невиданный
```

```
Ближайшие к делать: сделал
Ближайшие к день: вечер, воскресенье
Ближайшие к два: многие, родственники
Ближайшие к воля: может, должен, буду
Ближайшие к кирка: лестница
Ближайшие к перемешивание: встряхивание, молоко
```

Теперь мы можем загрузить представления – а не унитарные коды – в нашу LSTM-сеть. Для этого добавим функцию `tf.nn.embedding_lookup`:

```
for ui in range(num_unrollings):
    train_inputs.append(tf.placeholder(tf.int32, shape=[batch_
size],name='train_inputs_ %d' %ui))
    train_inputs_embeds.append(tf.nn.embedding_
lookup(embeddings,train_inputs[ui]))
```

Для задачи моделирования языка более общего назначения мы можем использовать уже имеющиеся предварительно подготовленные векторы слов. Готовые данные, найденные путем изучения текстового корпуса с миллиардами слов, свободно доступны для скачивания и использования. Здесь мы перечислим несколько таких репозиториев, предоставляющих доступ к готовым векторам слов:

- Word2vec: **https://code.google.com/archive/p/word2vec/**;
- предварительно обученные векторы слов GloVe: **https://nlp.stanford.EDU/projects/glove/**;
- векторы слов fastText: **https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md**.

Однако, поскольку мы работаем со словарем очень ограниченного размера, мы выучим наши собственные векторы слов. Дело в том, что если мы попытаемся использовать эти массивные репозитории векторов слов для скромной задачи со словарным запасом, не превышающим несколько тысяч слов, это приведет к необоснованным вычислительным затратам. Более того, поскольку мы генерируем сказки, некоторые уникальные слова, например эльфы и водяные никси, возможно, даже не встречались во время подготовки репозиториев.

Остальная часть кода аналогичным образом применяется для вычисления ячеек LSTM, потерь, оптимизации и прогнозов, как было показано в предыдущих примерах. Однако помните, что теперь входная размерность LTSM больше не является размерностью словаря, а равна размерности вектора встраивания.

Текст, созданный с помощью LSTM–Word2vec и лучевого поиска

Следующий текст создан нейросетью LSTM–Word2vec и подвергнут простой обработке, включающей удаление лишних пробелов. Теперь текст выглядит довольно реалистично:

```
i am in a great castle. the king's son. the king 's son. "you are mine with the dragon ,
and a glass mountain and she gave it to you. "the king's son. "i have not". "no," said the
king's son , and a great lake, and in its little dish, which was much larger than do you
want to have not. the king. if i had a great lake, but it was not long before it. then the
```

```
king's son. the king's son, however, drank only the milk. then the king 's son said, "you
are not". then the wedding was celebrated, and when she got to the king's son. "you are mine,
and a glass mountain and the king 's son, however. they gave him to see her heart, and went
away, and the old king's son, who was sitting by the town, and when they went to the king's
boy. she was in its little head against it as long as it had strength to do so, until at
last it was standing in the kitchen and heard the crown, which are so big. when she got into
a carriage, and slept in the whole night, and the wedding was celebrated, and when she got
to the glass mountain they thrust the princess remained, the child says, come out. when she
got into a great lake, but the king's son, and there was a great lake before the paddock
came to a glass mountain, and there were full of happiness. when the bride, she got to sleep
in a great castle, and as soon as it was going to be put to her house, but the wedding was
celebrated, and when she got to the old woman, and a glass of wine. when it was evening,
she began to cry in the whole night, and the wedding was celebrated, and after this the
king's boy. and when she had washed up, and when the bride, who came to her, but when it was
evening, when the king 's son. the king 's son. the king 's son. "i will follow it. then the
king". if i had a great lake, and a glass mountain, and there were full dress, i have not.
"thereupon the king's son as the paddock had to put in it. she felt a great lake, so she is
mine. then the king 's son's son".
```

```
(Я в прекрасном замке. сын короля. сын короля. "ты мой с драконом и стеклянной горой,
и она дала тебе это", сын короля. "я не". "Нет", сказал сын короля, и большое озеро,
и в его маленьком блюде, которое было намного больше, чем вы хотели бы не иметь. король.
если бы у меня было отличное озеро, но это было незадолго до этого. потом сын короля. сын
короля, однако, пил только молоко. тогда сын короля сказал: "ты нет". затем свадьба была
отпразднована, и когда она добралась до сына короля. "Ты мой, и стеклянная гора, и сын короля,
однако. Они дали ему увидеть ее сердце и ушли, и сын старого короля, который сидел у города,
и когда они пошли к королю мальчик. она была в его маленькой голове против него, пока у него
были силы, чтобы сделать это, пока, наконец, он не стоял на кухне и не услышал корону, которая
такая большая. когда она села в коляску, и спала целую ночь, и свадьба праздновалась, и когда
она добралась до стеклянной горы, они толкнули оставшуюся принцессу, говорит ребенок, выходи.
когда она попала в большое озеро, но сын короля, и прежде было большое озеро перед загоном
пришел к стеклянной горе, и там было полно счастья. когда невеста, она заснула в большом
замке, и как только ее собирались положить в ее дом, но свадьбу праздновали, и когда она
добралась до старухи и бокал вина. когда был вечер, она начала плакать всю ночь, и свадьба
была отпразднована, и после этого паж короля. и когда она умылась, и когда невеста, которая
пришла к ней, но когда был вечер, когда сын короля. сын короля. сын короля. "я последую за
этим. тогда король". Если бы у меня было отличное озеро и стеклянная гора, и там было полное
платье, я бы не стал. "После этого сын короля, как в загоне. Она почувствовала великое озеро,
поэтому она моя. Тогда сын сына короля".)
```

Как видите, здесь нет циклического повторения текста, как мы видели в примере со стандартной RNN, текст выглядит грамматически правильным в большинстве случаев, и очень мало орфографических ошибок.

Итак, мы проанализировали, как выглядит текст, сгенерированный различными модификациями рекуррентной нейросети – стандартной LSTM-сетью, модификацией с замочными скважинами, GRU, LSTM с лучевым поиском и LSTM с лучевым поиском и использованием векторного представления слов Word2vec. Но это было субъективное визуальное сравнение. Давайте посмотрим, как выполняется количественная оценка и сравнение результатов.

Анализ уровня перплексии

На рис. 8.7 представлены графики изменения перплексии для всех методов, которые мы рассмотрели до сих пор. Чтобы сделать сравнение еще интереснее, мы добавим к сравнению одну из лучших моделей, которую можем придумать: трехслойную LSTM-сеть, использующую векторы слов и дропаут. Можно предположить, что из методов, которые используют дропаут, – т. е. методов, уменьшающих переобучение, – LSTM с функциями Word2vec показывает наиболее многообещающие результаты. Я хочу сказать, что LSTM-сети с Word2vec обеспечивают хорошее качество не только благодаря числовым представлениям, но и потому, что они лучше учитывают сложность задачи. В концепции Word2vec атомарная единица, которую мы используем для обучения, – это слова, в отличие от других моделей, использующих биграммы. Из-за большого объема словаря генерация естественного текста на уровне слов может быть сложной задачей по сравнению с таковой на уровне биграмм. Поэтому достижение перплексии в обучении на уровне слов, сопоставимой с таковой у моделей на основе биграмм, можно рассматривать как признак хорошего качества. Сравнивая перплексию валидации, мы видим, что методы, основанные на векторном представлении слов, демонстрируют более высокий уровень перплексии. Это ожидаемо, так как задача более сложная из-за большого словарного запаса. Еще одно интересное наблюдение, на которое я хотел бы обратить ваше внимание, – это сравнение однослойных и глубоких LSTM-сетей. Заметно, что глубокая LSTM-сеть демонстрирует более низкую и стабильную перплексию при валидации. Следовательно, можно предположить, что глубокие модели часто дают лучшие результаты. Обратите внимание, что мы не рассматриваем результаты использования лучевого поиска, так как он влияет только на прогноз, но не влияет на перплексию при обучении.

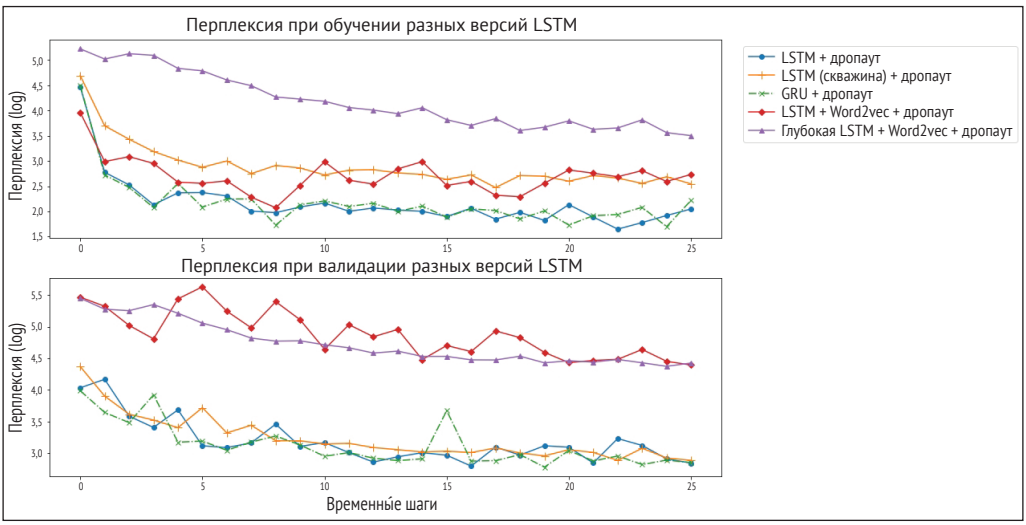


Рис. 8.7 ❖ График изменения перплексии у разных модификаций LSTM

Использование TensorFlow RNN API

Далее мы рассмотрим пример использования TensorFlow RNN API, благодаря чему можем сделать код проще. RNN API содержит множество функций, связанных с рекуррентными нейросетями, которые помогают нам быстрее и проще реализовать RNN. Теперь вы увидите, как знакомый вам пример из предыдущих разделов книги можно реализовать с использованием TensorFlow RNN API. Однако чтобы сделать упражнение интереснее, мы реализуем глубокую сеть LSTM с тремя уровнями, о которой говорили выше. Полный код упражнения доступен в файле `lstm_word2vec_rnn_api.ipynb` в папке `ch8`.

Сначала мы объявим заполнители для хранения входных данных, меток и соответствующих векторов встраивания для входных данных. Я не привожу вычисления, связанные с данными валидации, поскольку мы уже обсуждали их:

```
# Входные обучающие данные.
train_inputs, train_labels = [],[]
train_labels_ohe = []
# Объявляем развернутые обучающие входы.
for ui in range(num_unrollings):
    train_inputs.append(tf.placeholder(tf.int32,
        shape=[batch_size],name='train_inputs_ %d' %ui))
    train_labels.append(tf.placeholder(tf.int32,
        shape=[batch_size], name = 'train_labels_ %d' %ui))
    train_labels_ohe.append(tf.one_hot(train_labels[ui],
        vocabulary_size))

# Объявляем операции перебора встраиваний для всех
# развернутых входов на этапе обучения.
train_inputs_embeds = []
for ui in range(num_unrollings):
    # Используем expand_dims для добавления дополнительных осей,
    # которые понадобятся потом для вычислений ячейки LTSM.
    train_inputs_embeds.append(tf.expand_dims(
                               tf.nn.embedding_lookup(
                               embeddings,train_inputs[ui]),0))
```

Затем объявляем список ячеек LSTM из RNN API:

```
# num_nodes - последовательность размеров скрытых слоев
cells = [tf.nn.rnn_cell.LSTMCell(n) for n in num_nodes]
```

Для всех ячеек LSTM мы также объявим `DropoutWrapper`, который выполняет операцию дропаута на входах/состояниях/выходах ячейки:

```
# Объявляем DropoutWrapper для каждой ячейки LSTM
dropout_cells = [
    rnn.DropoutWrapper(
        cell=lstm, input_keep_prob=1.0,
        output_keep_prob=1.0-dropout, state_keep_prob=1.0,
        variational_recurrent=True,
        input_size=tf.TensorShape([embeddings_size]),
        dtype=tf.float32
    ) for lstm in cells
]
```

Параметры, предоставляемые этой функции:

- `cell` – это тип ячейки RNN, которую мы используем в вычислениях;
- `input_keep_prob` – это количество единиц входа, которые должны оставаться активированными при выполнении дропаута (от 0 до 1);
- `output_keep_prob` – это количество единиц выходных данных, которые должны оставаться активированными при выполнении дропаута;
- `state_keep_prob` – это количество единиц состояния ячейки, которое необходимо сохранить при выполнении дропаута;
- `variational_recurrent` – это специальный тип дропаута для RNN, представленный Галом и Гахрамани в статье[1] «Теоретически обоснованное применение дропаута в рекуррентных нейронных сетях».

Затем мы объявляем инициализированный нулями тензор с именем `initial_state`, который будет содержать итеративно обновляемые состояния ячейки LSTM:

```
# Начальное состояние памяти LSTM.
initial_state = stacked_dropout_cell.zero_state(batch_size, dtype=tf.
float32)
```

Объявив список ячеек LSTM, мы теперь можем объявить объект `MultiRNNCell`, инкапсулирующий список ячеек, следующим образом:

```
# Объявляем объект MultiRNNCell, который
# использует DropoutWrapper при обучении.
stacked_dropout_cell = tf.nn.rnn_cell.MultiRNNCell(dropout_cells)
# Объявляем объект MultiRNNCell, который НЕ использует
# дропаут при валидации и тестировании.
stacked_cell = tf.nn.rnn_cell.MultiRNNCell(cells)
```

Далее вычисляем выход ячейки LSTM с помощью функции `tf.nn.dynamic_rnn` следующим образом:

```
# Объявляем вычисления ячейки LSTM при обучении
train_outputs, initial_state = tf.nn.dynamic_rnn(
    stacked_dropout_cell, tf.concat(train_inputs_embeds,axis=0),
    time_major=True, initial_state=initial_state
)
```

Эта функция нуждается в следующих параметрах:

- `cell` – тип последовательной модели, которая будет использоваться для вычисления выходных данных. В нашем случае это будет ячейка LSTM, которую мы объявили ранее;
- `inputs` – это входы для ячейки LSTM. Входные данные должны иметь форму `[num_unrollings, batch_size, embeddings_size]`. Следовательно, у нас есть все пакеты данных для всех временных шагов в этом тензоре. Мы будем называть этот тип данных *основным по времени* (time major), так как ось времени – это нулевая ось;
- `time_major` – мы говорим, что наши входные данные являются основными по времени;
- `initial_state` – LSTM необходимо иметь начальное состояние.

[1] A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. Gal and Ghahramani, Data-Efficient Machine Learning workshop, ICML (2016).

После расчета окончательного скрытого состояния и состояния ячейки LSTM мы определим логиты – ненормализованные оценки, полученные из слоя softmax для каждого слова, и прогнозы – нормализованные оценки уровня softmax для каждого слова:

```
# Переформатируем окончательный выход в [num_unrollings*batch_size, num_nodes]
final_output = tf.reshape(train_outputs,[-1,num_nodes[-1]])

# Вычисляем логиты
logits = tf.matmul(final_output, w) + b
# Вычисляем прогнозы
train_prediction = tf.nn.softmax(logits)
```

Далее сделаем наши логиты и метки основными по времени. Это необходимо для функции потерь, которую мы будем использовать:

```
# Преобразовываем логиты в основные по времени
# [num_unrollings, batch_size, vocabulary_size].
time_major_train_logits = tf.reshape(logits,[num_unrollings,batch_
size,-1])

# Создаем обучающие метки, основные по времени
# [num_unrollings, batch_size, vocabulary_size]
# поскольку они применяются в функции потерь
time_major_train_labels = tf.reshape(tf.concat(train_
labels,axis=0),[num_unrollings,batch_size])
```

Теперь мы переходим к определению ошибки как разности между выходными данными, вычисленными из LSTM и слоя softmax, и фактическими метками. Для этого мы будем использовать функцию `tf.contrib.seq2seq.sequence_loss`. Эта функция широко используется в задачах машинного перевода для вычисления разницы между выходным переводом модели и фактическим переводом, которые представляют собой две последовательности слов. Следовательно, ту же концепцию можно распространить на нашу проблему, потому что мы, по сути, выводим последовательность слов:

```
# Мы используем функцию потерь sequence-to-sequence.
# Мы вычисляем среднее по пакетам
# и берем сумму по всей длине последовательности.
loss = tf.contrib.seq2seq.sequence_loss(
    logits = tf.transpose(time_major_train_logits,[1,0,2]),
    targets = tf.transpose(time_major_train_labels),
    weights= tf.ones([batch_size, num_unrollings], dtype=tf.float32),
    average_across_timesteps=False,
    average_across_batch=True
)

loss = tf.reduce_sum(loss)
```

Данной функции потерь мы предоставляем следующие аргументы:

- `logits` – ненормализованные оценки прогнозов, которые мы вычислили ранее. Однако эта функция принимает логиты, упорядоченные по следующей форме: `[batch_size, num_unrollings, vocabulary_size]`. Для этого мы используем функцию `tf.transpose`;

- targets – фактические метки для пакета или последовательности входов. Они должны быть в форме [batch_size, num_unrollings];
- weights – это веса, которые мы даем каждой позиции на оси времени, а также на оси пакета. В данном случае мы не различаем входные данные по их позиции, поэтому установим значение 1 для всех позиций;
- average_across_timesteps – мы не усредняем ошибку по временным шагам. Нам нужна сумма по временным шагам, поэтому мы установим параметр в False;
- average_across_batch – нам нужно усреднить ошибки по пакету, поэтому мы установим для этого параметра значение True.

Далее определим оптимизатор, как уже делали раньше:

```
# Применяется для снижения скорости обучения
gstep = tf.Variable(0, trainable=False)

# Выполнение этой операции увеличивает gstep,
# что приводит к снижению скорости обучения.
inc_gstep = tf.assign(gstep, gstep+1)

# Оптимизатор AdamOptimizer и отсечение градиента.
tf_learning_rate = tf.train.exponential_decay(0.001,gstep,decay_
steps=1, decay_rate=0.5)

print('Defining optimizer')
optimizer = tf.train.AdamOptimizer(tf_learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimizer = optimizer.apply_gradients(
    zip(gradients, v))

inc_gstep = tf.assign(gstep, gstep+1)
```

Теперь, когда все функции объявлены, вы можете запустить код, как показано в файле упражнения.

Заключение

В этой главе мы рассмотрели реализации алгоритма LSTM-сетей и различные подходы к улучшению качества LSTM по сравнению с качеством стандартного варианта. В первом упражнении главы мы обучили нашу LSTM-сеть на корпусе, составленном из сказок братьев Гримм, и попросили нейросеть сочинить новую сказку. Мы обсудили детали реализации LSTM на примерах кода, извлеченных из упражнений.

Затем вы узнали о том, как реализовать LSTM с замочными скважинами и GRU. Мы устроили экспериментальное сравнение качества между стандартной LSTM-сетью и ее модификациями и обнаружили, что стандартная LSTM показала лучшие результаты по сравнению с LSTM с замочными скважинами и GRU. Мы сделали удивительное наблюдение, что замочные скважины увеличили перплексию при обучении, но практически не помогли улучшить качество сгенерированного текста.

Далее вы узнали про другие способы повышения качества текстов, генерируемых LSTM. Мы начали с технологии лучевого поиска и рассмотрели пошаговую реализацию этого метода. Затем мы перешли к представлениям слов и убедились, что использование векторных представлений позволяет ощутимо повысить качество текста.

Мы можем сделать вывод, что LSTM – это очень мощные модели машинного обучения, способные охватывать как долгосрочные, так и краткосрочные зависимости. Вариант с лучевым поиском помогает создавать более реалистично выглядящие текстовые фразы по сравнению с предсказанием по одному слову за шаг. Кроме того, передача на вход LSTM векторных представлений слов вместо унитарных кодов позволяет еще больше улучшить качество модели.

В следующей главе мы рассмотрим еще одну интересную задачу, касающуюся как сетей прямого распространения, так и LSTM, – создание подписей к изображениям.

Глава 9

Применение LSTM – генерация подписей к рисункам

В предыдущей главе мы рассмотрели использование LSTM-сетей для генерации текста. В этой главе мы будем использовать LSTM для решения более сложной задачи – *аннотирования*, т. е. создания подходящих подписей для изображений. Эта задача существенно сложнее предыдущей, потому что ее решение включает в себя несколько подзадач, таких как обучение и применение CNN для генерации векторов признаков изображений, изучение представлений слов и обучение LSTM для генерации подписей.

Автоматические подписи или аннотации изображений имеют широкий спектр применений. Одним из наиболее распространенных приложений является поиск изображений. Автоматические подписи могут использоваться для извлечения всех изображений, относящихся к определенному текстовому запросу пользователя. Другим вариантом применения могут быть социальные сети, когда загруженному пользователем изображению автоматически присваивают подпись, чтобы пользователь мог уточнить сгенерированный текст или опубликовать его как есть.

Для создания подписей к изображениям мы будем использовать популярный набор данных, известный как Microsoft Common Objects in Context (MS-COCO). Сначала мы обработаем изображения из набора данных MS-COCO, чтобы получить весторы изображений с помощью предварительно обученной сверточной нейронной сети (CNN), которая уже хорошо классифицирует изображения. CNN получает на вход изображение фиксированного размера и выдает на выход класс, к которому принадлежит изображение, например «*кошка*», «*собака*», «*колесо*» и «*дерево*». При помощи CNN мы можем получить сжатые закодированные векторы, описывающие изображения.

Затем мы обработаем подписи к изображениям, чтобы изучить представления слов, найденных в подписях. Для этой задачи также можно использовать предварительно обученные векторы слов.

Наконец, получив векторы изображений и слов, мы передадим их в LSTM и обучим рекуррентную нейросеть на изображениях и соответствующих им подписях. Затем для проверки попросим нейросеть сгенерировать заголовки для набора незнакомых изображений.

Сначала мы воспользуемся предварительно обученной CNN для генерации векторов изображений. Затем реализуем собственный алгоритм изучения представлений слов и LSTM. И наконец, вы увидите, как для решения этой задачи можно использовать предварительно обученные векторы слов вместе с модулями LSTM, представленными в TensorFlow RNN API. Использование предварительно обученных векторов слов и RNN API значительно сокращает объем кода.

Знакомство с данными

Давайте сначала разберемся с данными, с которыми мы столкнемся как напрямую, так и косвенно. Итак, мы воспользуемся двумя наборами данных:

- ILSVRC ImageNet **http://image-net.org/download**;
- MS-COCO **http://cocodataset.org/#download**.

Мы не будем задействовать первый набор данных напрямую, но зато он важен для изучения подписей. Этот набор данных содержит изображения и соответствующие метки классов. Мы воспользуемся нейросетью, которая уже обучена на данном наборе данных, поэтому нам не придется загружать набор и обучать CNN с нуля. Далее мы воспользуемся набором данных MS-COCO, который содержит изображения и соответствующие подписи к ним. Мы напрямую обучим нейросеть на этом наборе данных, сопоставляя изображение с вектором признаков фиксированного размера, а затем сопоставим полученные векторы с соответствующими заголовками, используя LSTM (мы подробно обсудим этот процесс ниже).

Набор данных ILSVRC ImageNet

ImageNet – это набор данных, состоящий приблизительно из 1 млн изображений и соответствующих им меток. Упомянутые изображения относятся к 1000 различных категорий. Этот набор данных очень представителен и содержит почти все объекты, способные оказаться на изображениях, для которых мы будем создавать подписи. Поэтому я считаю, что ImageNet является хорошим обучающим набором для извлечения признаков изображений, необходимых для генерации титров. Я говорил, что мы используем этот набор данных лишь косвенно, потому что в упражнении задействуем предварительно обученную CNN. Нам не придется самим загружать и обучать CNN на этом наборе данных. На рис. 9.1 показаны примеры некоторых классов из набора данных ImageNet.

Набор данных MS-COCO

Теперь перейдем к набору данных под названием MS-COCO. Мы будем использовать набор данных 2014 года. Как сказано выше, этот набор данных состоит из изображений и их соответствующих описаний (аннотаций). Набор данных

довольно большой (например, обучающий набор данных состоит из ~120 тыс. выборок и может занимать более 15 ГБ). Наборы данных обновляются каждый год, и затем проводится конкурс для выявления команды, которая добивается наилучших результатов. Использование полного набора данных важно, когда целью является обучение и проверка самой продвинутой нейросети. В нашем случае мы хотим обучить достаточно разумную модель, способную высказать предположение об изображении в целом. Поэтому для обучения нашей сквозной модели мы будем использовать меньший набор данных – около 40 тыс. изображений и около 200 тыс. аннотаций. На рис. 9.2 показаны некоторые из доступных выборок[1].

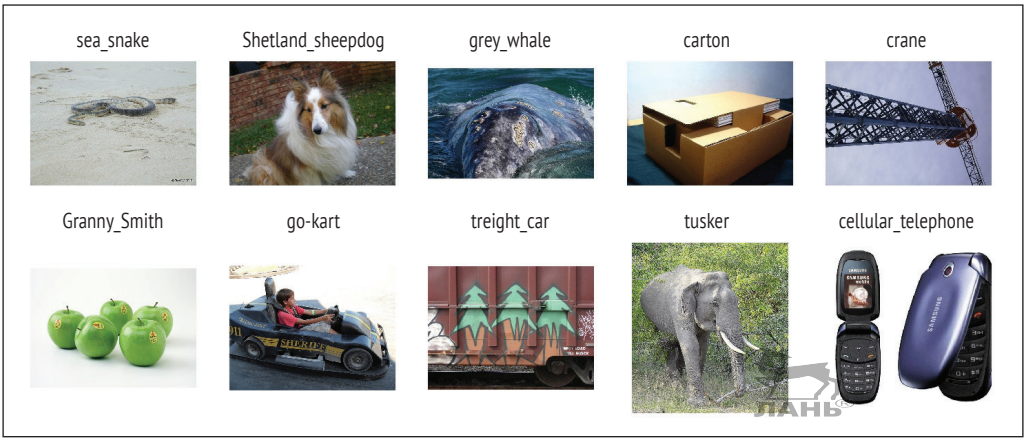


Рис. 9.1 ❖ Пример изображений из набора данных ImageNet

Мы будем использовать начальный набор из 1000 образцов в качестве набора для проверки, а остальные в качестве обучающего набора.

На практике вы должны использовать отдельные наборы данных для тестирования и валидации. Однако поскольку мы используем ограниченные данные, чтобы максимизировать обучение, то применяем один и тот же набор данных как для тестирования, так и для валидации.

На рис. 9.3 приведены примеры некоторых изображений из набора для валидации, представляющих различные объекты и сцены. Мы будем использовать их для визуальной проверки результатов, поскольку невозможно визуально проверить все 1000 образцов в проверочном наборе.

[1] Для большей наглядности описания на рис. 9.2 переведены на русский язык. Описания в оригинальном наборе представлены на английском языке. – *Прим. перев.*

Розовый и зеленый маркер, рядом другой объект.
Красные ножницы на столе.
Крупным планом колечко для пальца у ножниц и тонкие маркеры.
Крупным планом красные ножницы и тонкий зеленый маркер.
Очень крупный план каких-то ножниц и маркеров.

Ванная комната с корзиной для журналов и маленьким шкафом.
Компактный совмещенный санузел, ванна, унитаз, раковина, журналы.
Маленькая ванная, оформленная в тонах натурального дерева.
Холщовая коробочка стоит на унитазе в ванной.
Пример ухоженной ванной комнаты.

Женщина выгуливает коричневую лошадь на манеже для верховой езды.
Женщина в конюшне с коричневой лошадью.
Женщина обучает свою великолепную коричневую лошадь.
Женщина с коричневой лошадью на песчаном манеже.
Женщина и лошадь на манеже в конюшне.

Мужчина высоко подпрыгнул и пытается поймать фрисби.
Мужчина демонстрирует трюк с фрисби.
Некто подпрыгнул с фрисби.
Некто подпрыгнул во время игры в фрисби.
Мужчина пытается поймать фрисби во время прыжка.

Серый мотоцикл на грунтовой дороге, позади здание.
Мотоцикл припаркован на грунтовой дороге перед старой сельской стоянкой грузовиков.
Это мотоцикл на грунтовой дороге.
Мотоцикл стоит на грунтовой дороге перед деревенским зданием с рекламой.
Мотоцикл стоит на грунтовой дороге.

Рис. 9.2 ❖ Пример изображений и аннотаций из набора данных MS-COCO

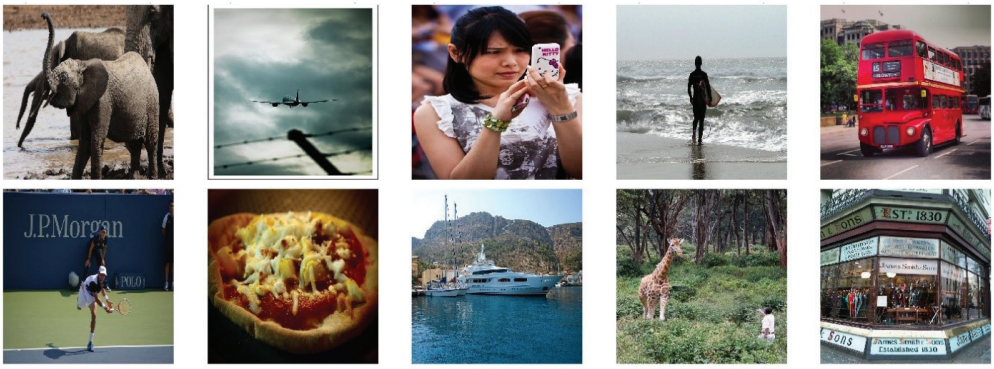


Рис. 9.3 ❖ Незнакомые изображения,
которые мы используем для проверки способности нашей нейросети
создавать подписи к картинкам

Устройство модели для генерации подписей к изображениям

Сначала мы рассмотрим последовательность генерации подписей к изображениям в самом общем виде, а затем разберем ее по частям, чтобы получить полное представление о модели. Каркас модели для создания подписей к изображениям состоит из трех базовых компонентов и одного необязательного:

- CNN, генерирующая закодированные векторы изображений;
- слой представлений, изучающий векторные представления слов;
- не обязательно – адаптирующий слой, способный преобразовать исходную размерность представления в произвольную размерность (подробности будут обсуждаться ниже);
- LSTM-сеть, получающая закодированные векторы изображений и выводящая соответствующую подпись.

Сначала мы получим CNN, генерирующую закодированные векторы изображений. Мы добьемся этого, обучив CNN на большом наборе классификационных данных, например ImageNet, и будем использовать полученные знания для создания сжатых векторизованных представлений изображений.

Кто-то может спросить: почему бы не ввести в LSTM изображение, как оно есть? Давайте вспомним результат простого оценочного вычисления, которое мы сделали в предыдущей главе:

> «*Увеличение на 500 точек данных во входном слое приводит к увеличению на 200 000 параметров*».

Изображения, с которыми мы имеем дело, состоят из $224 \times 224 \times 3 \sim 150$ тыс. точек. Вы можете самостоятельно прикинуть увеличение количества параметров, которое случится в LSTM-сети. Поэтому поиск сжатого представления изображений имеет решающее значение. Другая причина, по которой LSTM не подходят для прямой обработки изображений, заключается в том, что их трудно приспособить для обработки данных подобного рода.

Существуют сверточные варианты LSTM, способные работать с входными данными изображения, используя операцию свертки вместо полностью связанных слоев. Такие сети широко применяются для решения пространственно-временных задач, например прогнозов погоды или категоризации видео, имеющих как пространственные, так и временные измерения данных. Вы можете прочитать больше о сверточных LSTM в статье[1] Джеффа Донахью и др. «Долговременные рекуррентные сверточные сети в задачах визуального распознавания и описания».

Хотя процедура обучения представлениям изображений совершенно иная, мы достигаем той же цели, что и в случае изучения представлений слов. Как вы помните, похожие слова должны иметь близкие векторы (т. е. высокое сходство), а разные слова – отдаленные векторы (т. е. низкое сходство). Другими словами,

[1] Long-term Recurrent Convolutional Networks for Visual Recognition and Description. Jeff Donahue, and others, Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (2015).

если $Image_x$ является векторным представлением изображения x, то должно выполняться условие:

$$Расстояние\ (Image_{кошка}, Image_{вулкан}) > Расстояние\ (Image_{кошка}, Image_{собака}).$$

Затем мы изучим представления слов для текстового корпуса, созданного путем извлечения всех слов из всех заголовков, доступных в наборе данных MS-COCO. Опять же, изучение представлений слов уменьшает размерность входных данных для LSTM и помогает находить более значимые признаки. Однако это служит и еще одной важной цели в архитектуре модели.

Когда мы использовали LSTM для генерации текста, то применяли либо унитарное кодирование, либо векторы представлений слов. Поэтому вход LSTM всегда имел постоянную размерность. Стандартная LSTM-сеть не справится с входными данными переменного размера. Однако нам не приходилось беспокоиться об этом, поскольку мы имели дело только с текстом.

Однако теперь мы работаем как с изображениями, так и с текстом, и нам нужно убедиться, что векторы изображений и представления каждого слова, входящего в подписи этих изображений, имеют одинаковую размерность. Кроме того, с помощью векторов слов мы можем создать представление произвольной фиксированной длины для признаков всех слов. Поэтому мы используем векторы слов, совпадающие по длине с векторами изображений.

Наконец, мы создадим последовательность данных для каждого изображения, где первый элемент последовательности – векторизованное представление изображения, за которым следуют векторы слов для каждого слова в заголовке изображения в указанном порядке. Затем мы используем эту последовательность данных для обучения LSTM, как уже делали ранее.

Упомянутый выше подход аналогичен методике, описанной в работе[1] «Показать, понять и рассказать: нейронная генерация подписей к изображениям со зрительным вниманием». Процесс схематически изображен на рис. 9.4.

Извлечение признаков изображения

Итак, вы получили общее понимание того, в каком порядке работает модель. Теперь мы можем перейти к вопросу использования CNN для извлечения векторов признаков изображений. Чтобы получить хорошие векторы признаков, нам следует обучить CNN с изображениями и соответствующими классами или воспользоваться предварительно обученной нейросетью, свободно доступной в интернете. Мы не станем изобретать велосипед, тренируя CNN с нуля, потому что существуют предварительно обученные модели, доступные для бесплатного скачивания. Мы также должны помнить, что если от CNN ожидают способности описывать много объектов, ее нужно обучать на наборе классов, соответствующих различным объектам. Вот почему важна модель, обученная на большом наборе данных, таком как ImageNet. Как мы видели ранее, ImageNet содержит 1000 категорий объектов. Это более чем адекватно задаче, которую мы пытаемся решить.

[1] Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. Xu and others, Proceedings of the 32nd International Conference on Machine Learning (2015).

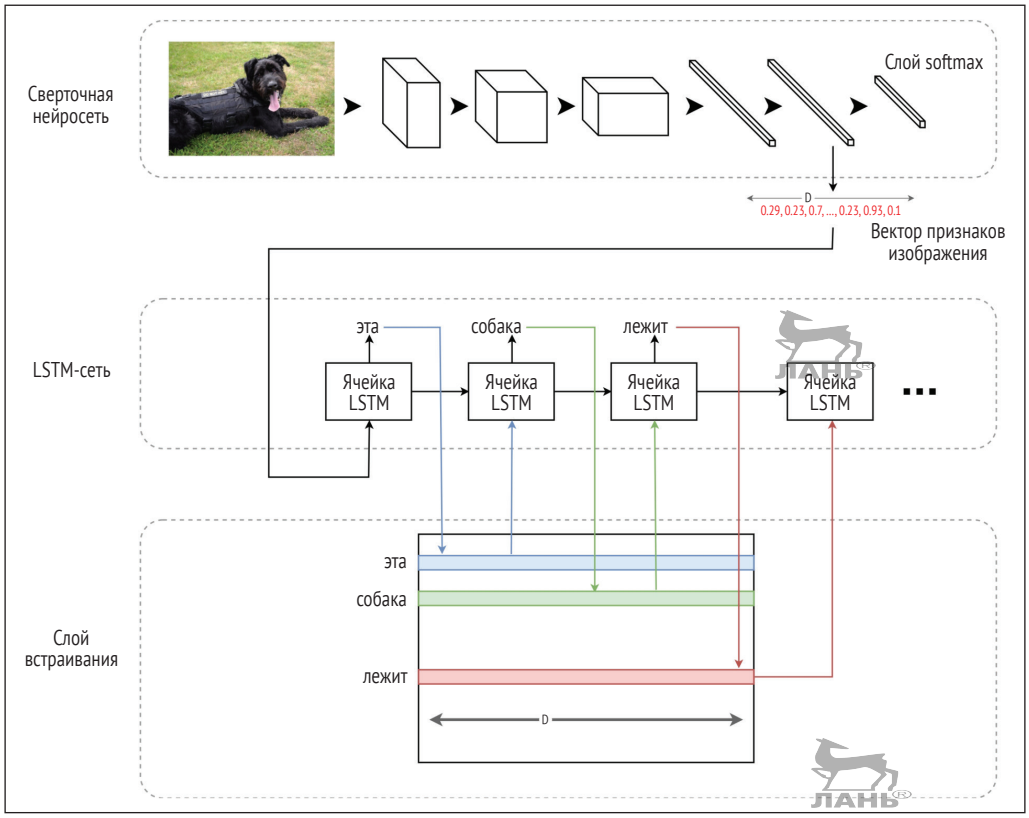


Рис. 9.4 ❖ Архитектура машинного обучения в задаче генерации подписей к изображениям

Имейте в виду, однако, что ImageNet содержит приблизительно 1 млн изображений. Кроме того, поскольку существует 1000 классов, мы не сможем обучить небольшую CNN с простой структурой, т. е. с несколькими слоями. Нам нужна более мощная и глубокая нейросеть, но из-за сложности CNN и самого набора данных обучение такой сети может занять несколько недель. Например, VGG – известная CNN, показавшая исключительно хорошую точность классификации в ImageNet, – обучается в течение двух-трех недель.

Следовательно, нам нужны более разумные способы решения этой проблемы. К счастью, нейросети, такие как VGG, легкодоступны для загрузки, поэтому мы можем использовать их без дополнительного обучения. Это так называемые *предварительно обученные модели* (pretrained models). Использование упомянутых моделей позволяет нам сэкономить несколько недель вычислительного времени. Чтобы воссоздать сеть и немедленно использовать ее для вывода, нам достаточно скачать изученные веса и исходный код, описывающий структуру CNN.

В этом упражнении мы будем использовать сверточную нейросеть VGG, доступную по адресу: **http://www.cs.toronto.edu/~frossard/post/vgg16/**. Архитектура VGG заняла второе место в конкурсе ImageNet 2014 года. VGG имеет несколько вариантов: 13-слойная глубокая сеть (VGG-13), 16-слойная глубокая сеть (VGG-16)

и 19-слойная глубокая сеть (VGG-19). Мы будем использовать 16-слойную нейросеть. На рис. 9.5 схематически изображена архитектура сети VGG-16.

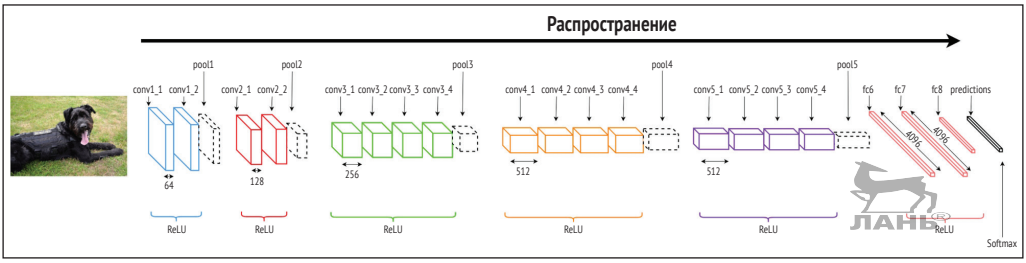


Рис. 9.5 ❖ Архитектура 16-слойной сверточной нейросети VGG

Реализация – загрузка весов и вывод с помощью VGG-16

Веб-сайт **http://www.cs.toronto.edu/~frossard/post/vgg16/** предоставляет весовые коэффициенты в виде словаря массивов NumPy. Существует 16 значений веса и 16 значений смещения, соответствующих 16 слоям VGG-16. Они хранятся под ключами следующего вида:

```
conv1_1_W, conv1_1_b, conv1_2_W, conv1_2_b, conv2_1_W, conv2_1_b…
```

Первым делом загрузите файл с веб-сайта и поместите его в папку данных `ch9/image_caption_data`. Далее мы обсудим реализацию, от загрузки скачанной CNN до использования ее для прогнозирования. Сначала вы узнаете, как создать необходимые переменные TensorFlow и загрузить их с готовыми весами. Затем мы объявим входной конвейер для чтения изображений, а также несколько этапов предварительной обработки. Потом объявим операции вывода для CNN, чтобы получить прогнозы. Наконец, объявим вычисление категории, которая, по мнению CNN, подходит лучше всего для данного изображения. Последняя операция не требуется для создания подписей к изображениям, однако поможет убедиться, что мы правильно настроили предварительно обученную CNN.

Создание и обновление переменных

Сначала мы загружаем в память словарь массивов NumPy, содержащих веса CNN:

```
weight_file = os.path.join('image_caption_data', 'vgg16_weights.npz')
weights = np.load(weight_file)
```

Затем мы создаем переменные TensorFlow и присваиваем им фактические веса. Учтите, что эта операция может занять довольно много памяти. Поэтому, чтобы избежать сбоев, мы специально попросим TensorFlow использовать центральный, а не графический процессор. Далее опишем код для создания и загрузки переменных TensorFlow с правильными весами. Сначала определим все ключи словаря (обозначающие разные идентификаторы уровня CNN) в списке Python

TF_SCOPES. Затем будем перебирать каждый идентификатор слоя, используя соответствующую матрицу весов и вектор смещения в качестве инициализаторов для конкретных переменных TensorFlow, названных в соответствии с соответствующим идентификатором слоя:

```
def build_vgg_variables(npz_weights):
    '''
    Создаем необходимые переменные TenzorFlow,
    чтобы развернуть модель VGG-16
    с предварительно обученными весами.
    :param npz_weights: веса загружаются как словарь
    :return:
    '''

    params = []
    print("Building VGG Variables (Tensorflow)...")

    with tf.variable_scope('CNN'):
        # Перебираем каждый сверточный и полностью подключенный слой
        # и создаем переменные TensorFlow, используя распределение
        # областей видимости.
        for si,scope in enumerate(TF_SCOPES):
            with tf.variable_scope(scope) as sc:
                weight_key, bias_key = TF_SCOPES[si]+'_W',
                                       TF_SCOPES[si]+'_b'

                with tf.device('/cpu:0'):
                    weights = tf.get_variable(TF_WEIGHTS_STR,
                              initializer= npz_weights[weight_key])
                    bias = tf.get_variable(TF_BIAS_STR,
                           initializer = npz_weights[bias_key])

                    params.extend([weights,bias])
    return params
```

Предварительная обработка входов

Далее объявляем входной конвейер для ввода изображения в нейросеть. VGG-16 предъявляет следующие требования к входным изображениям, чтобы прогнозы были правильными:

- входы должны иметь размерность [224,224,3];
- входы должны иметь нулевое среднее (но не единичную дисперсию).

Следующий код создает конвейер, который читает данные прямо из набора файлов с заданными именами, применяет предварительные преобразования и создает пакет преобразованных изображений. Эта процедура определена в функции preprocess_inputs_with_tfqueue в файле упражнения.

Сначала мы определим очередь файлов. Она содержит имена файлов изображений, которые мы должны читать:

```
# Очередь имен файлов типа FIFO.
# Создаем очередь FIFO до тех пор, пока ридер в этом нуждается
filename_queue = tf.train.string_input_producer(filenames,
                 capacity=10, shuffle=False)
```

Далее мы определим ридер, принимающий очередь имен файлов и выдающий буфер, в котором хранятся изображения, соответствующие именам файлов из очереди в любой момент времени:

```
# Ридер, читающий файлы по их именам из очереди
# и выдающий их содержимое по одному.
reader = tf.WholeFileReader()
_, image_buffer = reader.read(filename_queue,
                  name='image_read_op')

# Читает исходные данные изображения и возвращает uint8.
dec_image = tf.image.decode_jpeg(contents=
            image_buffer,channels=3,name='decode_jpg')
# Конвертирует данные uint8 в float32.
float_image = tf.image.convert_image_dtype(dec_image,
              dtype=tf.float32,name= 'float_image')
```

Затем мы выполняем вышеупомянутую предварительную обработку:

```
# Приводим к размерности 224×224×3.
resized_image = tf.image.resize_images(float_
                image,[224,224])*255.0

# Для VGG нужны изображения с нулевым средним
# (но не приведенные к единичной дисперсии)
std_image = resized_image - tf.reduce_mean(resized_
image,axis=[0,1], keepdims=True)
```

После того как конвейер предварительной обработки определен, мы попросим TensorFlow создать пакет предварительно обработанных изображений за раз, без перемешивания:

```
image_batch = tf.train.batch([std_image],
              batch_size = batch_size, capacity = 10,
              allow_smaller_final_batch=False,
              name='image_batch')
```

Распространение данных через VGG-16

На данный момент мы создали CNN и объявили конвейер для чтения изображений и создания пакета путем чтения файлов, сохраненных на диске. Теперь мы хотели бы пропустить через CNN изображения, считанные конвейером. Иными словами, мы хотим передавать в CNN входные данные (т. е. изображения) и получать прогнозы (т. е. вероятности того, что изображение принадлежит к определенному классу) в качестве выходных данных. Для этого мы начнем с первого слоя и будем повторять операцию до достижения уровня softmax. Этот процесс определен в функции `inference_cnn` в файле упражнения.

На каждом слое мы получаем вес и смещение следующим образом:

```
def inference_cnn(tf_inputs, device):

    with tf.variable_scope('CNN'):
        for si, scope in enumerate(TF_SCOPES):
            with tf.variable_scope(scope,reuse=True) as sc:
```

```
                weight, bias = tf.get_variable(TF_WEIGHTS_STR),
                               tf.get_variable(TF_BIAS_STR)
```

Затем вычисляем результат для первого сверточного слоя:

```
h = tf.nn.relu(tf.nn.conv2d(tf_inputs,weight,strides=[1,1,1,1],
               padding='SAME')+bias)
```

Аналогично вычисляем выход остальных сверточных слоев, где вход – это выход предыдущего слоя:

```
h = tf.nn.relu(tf.nn.conv2d(h, weight, strides=[1, 1, 1, 1],
               padding='SAME') + bias)
```

А для слоев субдискретизации результат вычисляется следующим образом:

```
h = tf.nn.max_pool(h,[1,2,2,1],[1,2,2,1],padding='SAME')
```

Затем мы вычисляем выход первого полностью связанного слоя, расположенного сразу за последним сверточным слоем. Нам нужно изменить размерность данных, поступающих с последнего сверточного/объединяющего слоя с `[batch_size, height, width, channel]` на `[batch_size, height*width*channel]`, так как дальше мы работаем с полностью связанным слоем:

```
h_shape = h.get_shape().as_list()
h = tf.reshape(h,[h_shape[0], h_shape[1] * h_shape[2] * h_shape[3]])
h = tf.nn.relu(tf.matmul(h, weight) + bias)
```

Для последующего набора полностью связанных слоев, кроме последнего слоя, вычисляем выходы следующим образом:

```
h = tf.nn.relu(tf.matmul(h, weight) + bias)
```

Наконец, мы не применяем никакую активацию к последнему связанному слою. Это будет представление признаков изображения, которое мы будем загружать непосредственно в LSTM в виде вектора с длиной 1000:

```
out = tf.matmul(h,weight) + bias
```

Извлечение векторизованных представлений изображений

Самая важная информация, которую мы извлекаем из CNN, – это представления признаков изображений. В качестве источника упомянутых представлений мы используем выход самого последнего слоя перед применением softmax. Следовательно, вектор, соответствующий одному изображению, имеет длину 1000:

```
tf_train_logit_prediction = inference_cnn(train_image_batch, device)
tf_test_logit_prediction = inference_cnn(test_image_batch, device)
```

Прогнозирование вероятностей классов с помощью VGG-16

Далее мы объявим операции, необходимые для получения представлений признаков изображений, а также фактических прогнозов softmax, чтобы убедиться, что наша модель действительно верна. Мы объявим их как для тренировочных, так и для тестовых данных:

```
tf_train_softmax_prediction = tf.nn.softmax(tf_train_logit_prediction)
tf_test_softmax_prediction = tf.nn.softmax(tf_test_logit_prediction)
```

Теперь давайте запустим эти операции и посмотрим, насколько правильно они работают (рис. 9.6).

Рис. 9.6 ❖ Прогноз класса для наших тестовых изображений с VGG

Кажется, наша нейросеть неплохо делает свою работу. Конечно, существуют неправильно классифицированные образцы (например, жираф идентифицирован как лама), но в большинстве случаев объект на изображении распознан правильно.

При выполнении объявленных выше операций для получения векторов признаков и прогнозов помните о переменной `batch_size`. Увеличение этого параметра сделает код быстрым. Однако это также может привести к сбою системы, если у вашего компьютера недостаточно большой объем оперативной памяти (желательно иметь больше 8 ГБ). Рекомендуется оставить значение `batch_size` меньше 10, если у вас нет машины высшего класса.

Изучение представлений слов

Далее мы обсудим, как изучить представления слов, найденных в подписях. Сначала мы обработаем подписи, чтобы уменьшить словарный запас:

```
def preprocess_caption(capt):
capt = capt.replace('-',' ')
capt = capt.replace(',','')
capt = capt.replace('.','')
capt = capt.replace('"','')
capt = capt.replace('!','')
capt = capt.replace(':','')
```

```
capt = capt.replace('/','')
capt = capt.replace('?','')
capt = capt.replace(';','')
capt = capt.replace('\' ',' ')
capt = capt.replace('\n',' ')

return capt.lower()
```

Например, рассмотрим следующее предложение:

В гостиной и столовой есть два стола, диваны и несколько стульев.

Оно будет преобразовано следующим образом:

в гостиной и столовой есть два стола диваны и несколько стульев

Затем мы воспользуемся моделью Continuous Bag-of-Words (CBOW) для изучения представлений слов, как мы это делали в главе 3. Важное условие, которое мы должны учитывать на данном этапе, заключается в том, что *размерность вектора представления слов должна совпадать с размерностью представлений признаков изображений*, поскольку стандартные LSTM не могут обрабатывать входные данные переменного размера.

Если мы хотим использовать предварительно изученные представления слов, то, скорее всего, желаемого совпадения размерностей изначально у нас не будет. В этом случае мы можем использовать адаптирующий слой, аналогичный слоям нейронной сети, чтобы сопоставить размерность вектора слова с размерностью представления признаков изображения. Ниже вы увидите, как эта операция выполняется в упражнении.

Теперь давайте взглянем на некоторые представления слов, изученные после выполнения 100 000 шагов:

```
Nearest to suitcase: woman
Nearest to girls: smart, racket
Nearest to barrier: obstacle
Nearest to casings: exterior
Nearest to normal: lady
Nearest to developed: natural
Nearest to shoreline: peninsula
Nearest to eating: table
Nearest to hoodie: bonnet
Nearest to prepped: plate, timetable
Nearest to regular: signs
Nearest to tie: pants, button

Ближайший к чемодан: женщина
Ближайшие к девушка: умница, ракетка
Ближайший к барьер: препятствие
Ближайший к отделка: экстерьер
Ближайший к нормальный: леди
Ближайший к развитой: естественный
Ближайший к побережье: полуостров
Ближайшие к еда: стол
Ближайший к толстовка: капюшон
```

```
Ближайший к школьник: табличка, расписание
Ближайший к упорядоченный: знаки
Ближайший к галстук: штаны, пуговица
```

Подготовка подписей для подачи в LSTM

Теперь, прежде чем вводить векторы слов вместе с векторами признаков изображения, нам нужно выполнить еще несколько шагов предварительной обработки подписей.

Но прежде чем объявить предварительную обработку в коде, давайте рассмотрим несколько основных статистических параметров. Подпись состоит в среднем из десяти слов со стандартным отклонением около двух слов. Эта информация важна для нас, чтобы обрезать заголовки, которые неоправданно длинны.

Исходя из этого соображения, давайте установим максимальную длину подписи, равную 12.

Далее введем два новых токена, SOS и EOS. Первый обозначает начало предложения (Strart Of Sentence), тогда как второй – конец предложения (End Of Sentence). Это поможет LSTM-сети легко определить начало и конец подписи.

Далее мы дополним подписи длиной менее 12 слов токенами EOS таким образом, чтобы их длина составляла ровно 12 слов.

Итак, рассмотрим следующую подпись:

мужчина стоит на теннисном корте с ракеткой

После обработки она будет выглядеть так:

SOS мужчина стоит на теннисном корте с ракеткой EOS EOS EOS EOS

Очень короткая подпись наподобие этой:

кот сидит на столе

будет также дополнена до 12 слов:

SOS кот сидит на столе EOS EOS EOS EOS EOS EOS EOS

Однако следующая чересчур длинная подпись:

в хорошо освещенной и красиво оформленной гостиной видна стеклянная входная дверь

превратится в сокращенный вариант:

SOS в хорошо освещенной и красиво оформленной гостиной видна стеклянная входная EOS

Обратите внимание, что даже после укорачивания подписи контекст изображения в основном сохраняется.

Важно, чтобы все подписи были одинаковой длины, чтобы мы могли обработать пакеты изображений и подписей.

Формирование данных для LSTM

Далее определим, как сформировать пакет данных для обучения LSTM. Всякий раз, когда мы обрабатываем новую порцию данных, первый вход должен быть вектором изображения, а метка должна быть токеном SOS. Мы будем объявлять пакет данных, где если логическое значение `first_sample` равно `True`, тогда входная информация извлекается из векторов признаков изображений, а если `first_sample` имеет значение `False`, входная информация извлекается из вложений слов. Кроме того, после генерации пакета данных мы переместим курсор на единицу, поэтому в следующий раз при создании пакета данных мы получим очередной элемент в последовательности. Таким образом, мы можем развернуть последовательность пакетов данных для LSTM, где первый пакет последовательности – это векторы признаков изображения, за которыми следуют представления слов в подписи, соответствующие этой серии изображений.

```
# Заполняем все индексы
for b in range(self._batch_size):

    cap_id = cap_ids[b] # Текущий индекс подписи
    # Текущий вектор признаков изображения.
    cap_image_vec = self._image_data[self._fname_caption_tuples[
                                    cap_id][0]]
    # Текущая подпись.
    cap_text = self._fname_caption_tuples[cap_id][1]

    # Если курсор вышел за конец подписи, то сброс.
    if self._cursor[b]+1>=self._cap_length:
        self._cursor[b] = 0

    # Если мы обрабатываем новый пакет,
    # то первая выборка должна быть вектором признаков изображения.
    if first_sample:
        batch_data[b] = cap_image_vec
        batch_labels[b] = np.zeros((vocabulary_size),
                          dtype=np.float32)
        batch_labels[b,cap_text[0]] = 1.0
    # Если мы продолжаем обработку текущего пакета,
    # продолжаем использовать текущее слово в качестве входных данных
    # и следующее слово в качестве выхода
    else:
        batch_data[b] = self._word_embeddings[
                        cap_text[self._cursor[b]],:]
        batch_labels[b] = np.zeros((vocabulary_size),
                          dtype=np.float32)
        batch_labels[b,cap_text[self._cursor[b]+1]] = 1.0

        # Сдвигаем курсор
        self._cursor[b] = (self._cursor[b]+1) %self._cap_length
```

Процесс генерации данных визуализирован на рис. 9.7, где `batch_size=1` и `num_unrollings=5`. Чтобы иметь больший размер пакета, вы можете выполнить операции для `batch_size` таких последовательностей параллельно.

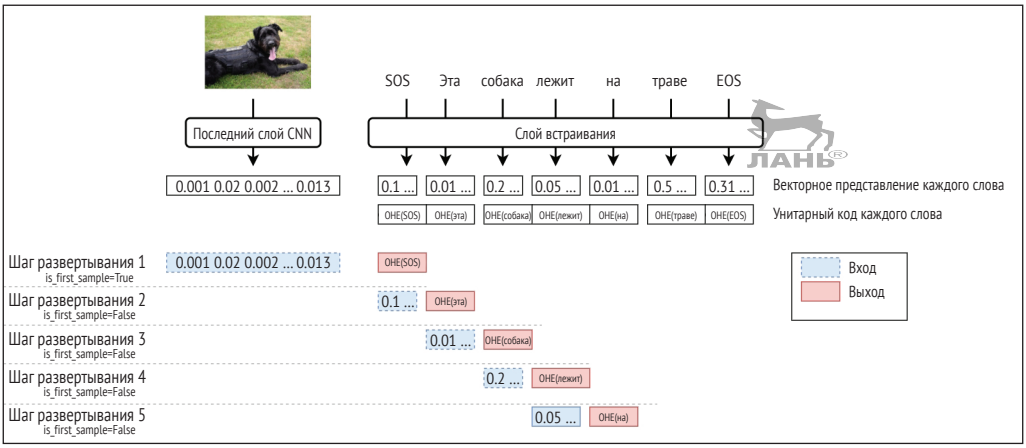


Рис. 9.7 ❖ Визуализация процесса генерации данных

Определение параметров и процедуры обучения LSTM

Теперь, когда мы определили генератор данных для вывода пакета данных, начинающегося с набора векторов признаков изображения, за которым следует слово за словом заголовок для соответствующих изображений, объявим ячейку LSTM. Объявление ячейки и процедуры обучения аналогично тому, что вы изучили в предыдущей главе.

Сначала мы определим параметры ячейки LSTM. Два набора весов и смещение для входного гейта, забывающего гейта, выходного гейта и для вычисления значения-кандидата:

```
# Входной гейт (i_t) - какую часть данных записать в состояние ячейки
# Подключаем текущий вход ко входному гейту
ix = tf.Variable(tf.truncated_normal([embedding_size, num_nodes],
stddev=0.01))
# Подключаем предыдущее скрытое состояние ко входному гейту
im = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],
stddev=0.01))
# Смещение входного гейта
ib = tf.Variable(tf.random_uniform([1, num_nodes],0.0, 0.01))

# Забывающий гейт (f_t) - какую часть данных удалить из состояния ячейки
# Подключаем текущий вход к забывающему гейту
fx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes],
stddev=0.01))
# Подключаем предыдущее скрытое состояние к забывающему гейту
fm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],
stddev=0.01))
# Смещение забывающего гейта
fb = tf.Variable(tf.random_uniform([1, num_nodes],0.0, 0.01))

# Значение-кандидат (c~_t) - применяется для вычисления текущего состояния ячейки
# Подключаем текущий вход к кандидату
```

```
cx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes],
stddev=0.01))
# Подключаем предыдущее скрытое состояние к кандидату
cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],
stddev=0.01))
# Смещение кандидата
cb = tf.Variable(tf.random_uniform([1, num_nodes],0.0,0.01))

# Выходной гейт – какое количество данных направить из состояния ячейки на выход
# Подключаем текущий вход к выходному гейту
ox = tf.Variable(tf.truncated_normal([embedding_size, num_nodes],
stddev=0.01))
# Подключаем предыдущее скрытое состояние к выходному гейту
om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],
stddev=0.01))
# Смещение выходного гейта
ob = tf.Variable(tf.random_uniform([1, num_nodes],0.0,0.01))
```

Объявляем веса softmax:

```
# Веса и смещения для классификатора softmax.
w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size],
stddev=0.01))
b = tf.Variable(tf.random_uniform([vocabulary_size],0.0,0.01))
```

Далее объявляем переменные состояния и выхода, чтобы хранить состояние и выходные данные LSTM при обучении и валидации:

```
# Переменные, сохраняют состояние при развертывании.
# Скрытое состояние
saved_output = tf.Variable(tf.zeros([batch_size, num_nodes]),
trainable=False, name='test_cell')
# Состояние ячейки
saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]),
trainable=False, name='train_cell')

# Переменные скрытого состояния и состояния ячейки для тестовых данных
saved_test_output = tf.Variable(tf.zeros([batch_size, num_
nodes]),trainable=False, name='test_hidden')
saved_test_state = tf.Variable(tf.zeros([batch_size, num_
nodes]),trainable=False, name='test_cell')
Объявляем вычисления ячейки LSTM:
def lstm_cell(i, o, state):
    input_gate = tf.sigmoid(tf.matmul(i, ix) + tf.matmul(o, im) +
                            ib)
    forget_gate = tf.sigmoid(tf.matmul(i, fx) + tf.matmul(o, fm) +
                             fb)
    update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
    state = forget_gate * state + input_gate * tf.tanh(update)
    output_gate = tf.sigmoid(tf.matmul(i, ox) + tf.matmul(o, om) +
                             ob)
    return output_gate * tf.tanh(state), state
```

Затем мы будем циклически вычислять состояние и выход ячейки LSTM для шагов `num_unrollings` на каждом шаге обучения:

```
# Эти две переменные Python циклически обновляются
# на каждом шаге развертывания.
output = saved_output
state = saved_state

# Вычисляем скрытое состояние (output) и состояние ячейки (state)
# рекурсивно для всех шагов развертывания.
for i in train_inputs:
    output, state = lstm_cell(i, output, state)
    # Append each computed output value
    outputs.append(output)

# Вычисляем значения оценки.
logits = tf.matmul(tf.concat(axis=0, values=outputs), w) + b

# Прогнозы.
train_prediction = tf.nn.softmax(logits)
```

После сохранения выходных данных и состояния LSTM в ранее объявленных переменных мы рассчитаем потери путем суммирования по развернутой оси и получения среднего значения по оси пакета:

```
# Сохранение состояний.
with tf.control_dependencies([saved_output.assign(output),
                              saved_state.assign(state)]):
    # При вычислении потерь мы должны взять сумму по всем временны́м шагам
    # и усреднить по оси пакета
    loss = 0
    split_logits = tf.split(logits,num_or_size_splits=num_unrollings)
    for lgt,lbl in zip(split_logits, train_labels):
        loss += tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits_v2(logits=lgt,
            labels=lbl)
        )
```

Наконец, мы определим оптимизатор для оптимизации весов LSTM и слоя softmax, исходя из потерь:

```
optimizer = tf.train.AdamOptimizer(learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimizer = optimizer.apply_gradients(
    zip(gradients, v))
```

Итак, подведем промежуточный итог. Мы сгенерировали векторы признаков изображения, подготовили данные для ввода в LSTM и определили вычисления, необходимые для обучения LSTM. Далее обсудим метрики, которые можно использовать для оценки подписей, созданных на этапе валидации.

Количественная оценка результатов

Существует много различных методов оценки качества и актуальности созданных подписей. Мы кратко обсудим несколько метрик, которые можем использовать для оценки подписей, а именно метрики BLEU, ROGUE, METEOR и CIDEr. Все

эти показатели имеют общую цель: измерить адекватность (смысл) и гладкость (грамматическую правильность) сгенерированного текста. Для вычисления этих измерений мы будем использовать *предложение-кандидат* (candidate sentence) и *истинное предложение* (reference sentence), где первое – это предложение, сгенерированное нашим алгоритмом, а второе – это «эталонное» предложение, с которым мы хотим выполнить сравнение.

BLEU

Метрика BLEU (Bilingual Evaluation Understudy, оценка двуязычного взаимосоответствия) была предложена Папинени и др. в работе[1] «BLEU: метод автоматической оценки машинного перевода». Она измеряет сходство *n*-граммы между истинными и кандидатскими предложениями независимо от позиции в предложении. Иными словами, если некая *n*-грамма кандидата присутствует в любом месте истинного предложения, то это считается совпадением. BLEU вычисляет сходство *n*-грамм с точки зрения точности совпадения. BLEU существует в нескольких вариантах (BLEU-1, BLEU-2, BLEU-3 и т. д.), в зависимости от значения *n* в *n*-грамме.

$$BLEU(candidate, ref) = \frac{\sum_{\forall n-gram\, in\, candidate} Count_{clip}(n-gram)}{\sum_{n-gram\, in\, candidate} Count(n-gram)} \times BP.$$

Здесь $Count(n - gram)$ – это общее количество вхождений данной *n*-граммы в предложении-кандидате. $Count_{clip}(n - gram)$ – это мера, которая вычисляет $Count(n - gram)$ для данной *n*-граммы и обрезает это значение по максимальному значению. Максимальное значение для *n*-граммы рассчитывается как число вхождений этой *n*-граммы в истинное предложение. Например, рассмотрим эти два предложения:

Предложение-кандидат: *the the the the the the the*
Истинное предложение: *the cat sat on the mat*

$Count(«the») = 7$
$Count_{clip}(«the»)=2$

Обратите внимание, что выражение $\frac{\sum_{\forall n-gram\, in\, candidate} Count_{clip}(n-gram)}{\sum_{n-gram\, in\, candidate} Count(n-gram)}$ – это форма точности, именуемая *модифицированной точностью n*-граммы. Когда присутствует несколько истинных предложений, берут максимальное значение:

$BLEU = \max(BLEU(candidate, ref_i))$.

Однако модифицированная точность *n*-грамм имеет тенденцию быть выше для более коротких предложений-кандидатов, поскольку в расчетах присутствует

[1] BLEU: A Method for Automatic Evaluation of Machine Translation. Papineni and others; Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, July (2002): 311–318.

деление на количество n-грамм в кандидате. Это означает, что данная метрика будет склонять модель к получению более коротких фраз. Чтобы избежать этого, в вычисления вводят штраф за короткие предложения-кандидаты. BLEU имеет несколько ограничений, например игнорирует синонимы при расчете оценки и не учитывает отклик модели (recall), что также является важным показателем для измерения точности. Кроме того, BLEU плохо работает с некоторыми языками. Тем не менее это простая метрика, которая, как было установлено, хорошо коррелирует с человеческим суждением в большинстве ситуаций. Мы обсудим BLEU более подробно в следующей главе.

ROUGE

В качестве альтернативы BLEU можно предложить метрику ROUGE (Recall-Oriented Understudy for Gisting Evaluations, основанное на отклике взаимосоответствия в реферативных оценках), предложенную Чин-Ю Линем в работе[1] «ROUGE: пакет для автоматической оценки резюме». В данном подходе основной метрикой качества является отклик модели. Метрика ROGUE выглядит следующим образом:

$$ROUGE - N = \frac{Count_{match}}{Count_{ref}}.$$

Здесь $Count_{match}$ – это количество n-грамм от кандидатов, присутствующих в истинном предложении, а $Count_{ref}$ – общее количество n-грамм, присутствующих в истинном предложении. Если существует несколько истинных предложений, *ROUGE-N* рассчитывается следующим образом:

$$ROUGE - N = \max(ROUGE - N(ref_i, candidate)).$$

Здесь ref_i является единственным истинным предложением из пула доступных истинных предложений. Существует множество вариантов измерения ROGUE, которые вносят различные улучшения в стандартную метрику. ROUGE–L вычисляет оценку на основе самой длинной общей подпоследовательности, найденной между парами кандидатов и истинных предложений. Обратите внимание, что в этом случае самая длинная общая подпоследовательность не должна быть непрерывной. ROUGE–W рассчитывает оценку на основе самой длинной общей подпоследовательности, которая штрафуется за уровень фрагментации, присутствующей в этой подпоследовательности. ROUGE, в свою очередь, страдает от ограничений, связанных с игнорированием точности при расчете оценки.

METEOR

Метрика METEOR (Metric for Evaluation of Translation with Explicit ORdering, метрика для оценки перевода с помощью явного выравнивания) предложена Майк-

[1] ROUGE: A Package for Automatic Evaluation of Summaries. Chin-Yew Lin; Proceedings of the Workshop on Text Summarization Branches Out (2004).

лом Денковски и Алоном Лави в работе[1] «Meteor Universal: оценка качества перевода для любого целевого языка». Это более продвинутая метрика, выполняющая выравнивания для кандидата и истинного предложения. METEOR отличается от BLEU и ROUGE тем, что учитывает положение слов. При вычислении сходства между предложением-кандидатом и истинным предложением рассматриваются следующие варианты совпадения:

- **точное**: слово из предложения-кандидата точно соответствует слову из истинного предложения;
- **коренное**: слово с общим корнем соответствует слову из истинного предложения (например, *ходьба* и *походка*);
- **синонимическое**: слово из предложения-кандидата является синонимом слова из истинного предложения.

Для вычисления показателя METEOR соответствия между истинным предложением и предложением-кандидатом могут быть представлены с помощью таблицы, как показано на рис. 9.8. Затем значения точности (P) и отзыва (R) рассчитываются на основе количества совпадений, присутствующих в кандидате и истинном предложении. Наконец, среднее гармоническое P и R используется для вычисления оценки METEOR:

$$F_{mean} = \frac{P.R}{\alpha P + (1-\alpha)R}(1 - \gamma \times frag^{\beta}).$$

Здесь α, β и γ являются настраиваемыми параметрами, а *frag* штрафует за фрагментированные совпадения, чтобы предпочесть кандидатские предложения, которые имеют меньше пропусков в совпадениях, а также более точно следуют порядку слов истинного предложения. Значение *frag* рассчитывается по количеству пересечений в окончательном сопоставлении униграмм (рис. 9.8).

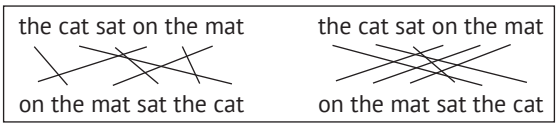


Рис. 9.8 ❖ Различные варианты сопоставления двух строк

Примеры различных совпадений слов между кандидатом и истинным предложением показаны в табл. 9.1. Точные совпадения обозначены сплошным черным кружком, синонимы – крупным пунктиром, а однокоренные слова – мелким пунктиром.

[1] Meteor Universal: Language Specific Translation Evaluation for Any Target Language. Michael Denkowski and Alon Lavie; Proceedings of the Ninth Workshop on Statistical Machine Translation (2014): 376–380.

Таблица 9.1. Примеры совпадений между кандидатом и истинным предложением

Предложение-кандидат	Истинное предложение											
	Боб	сбегает	прочь	когда	он	видит	Тома	ведь	Боб	должен	ему	деньги
Боб	●											
хотел												
бы												
исчезнуть		◌										
когда				●								
он					●							
увидел						◌						
Тома							●					
ведь								●				
Боб									◌			
задолжал										●		
много												

Метод METEOR в вычислительном отношении более сложен, но часто оказывается, что он коррелирует с человеческими суждениями больше, чем BLEU.

CIDEr

Метрика CIDEr (Consensus-based Image Description Evaluation, оценка описания изображения на основе консенсуса), предложенная Рамакришной Ведантамом и др. в статье[1] «CIDEr: оценка описания изображения на основе консенсуса», является еще одной мерой, которая оценивает соответствие предложения-кандидата заданному набору истинных высказываний. Метрика CIDEr предназначена для измерения грамматичности, значимости и точности предложения-кандидата.

Во-первых, CIDEr взвешивает каждую *n*-грамму, найденную как в кандидате, так и в истинном предложении, с помощью TF-IDF, чтобы более распространенные *n*-граммы (например, артикли *a* и *the*) имели меньший вес, и наоборот, редкие слова имели бы больший вес. Наконец, CIDEr рассчитывается как геометрическое расстояние между векторами, образованными взвешенными *n*-граммами TF-IDF, найденными в предложении-кандидате и в истинном предложении:

$$CIDEr(cand, ref) = \frac{1}{m}\sum_j \frac{TF-IDF_{vec}(cand).TF-IDF_{vec}(ref_i)}{TF-IDF_{vec}(cand)TF-IDF_{vec}(ref_i)}.$$

Здесь *cand* – предложение-кандидат, *ref* – набор истинных предложений, ref_j – *j*-е предложение *ref*, а *m* – количество истинных предложений для данного кандидата. Наиболее важно, что $TF\text{-}IDF_{vec}(cand)$ – это значения TF-IDF, рассчитанные для всех *n*-грамм в предложении-кандидате и сформированные как вектор. $TF\text{-}IDF_{vec}(ref_j)$ – это тот же вектор для истинного предложения, ref_i. $||TF - IDF_{vec}(.)||$ обозначает величину вектора.

В целом следует отметить, что не существует явного победителя, который мог бы хорошо справляться с любыми задачами, возникающими при обработке

[1] CIDEr: Consensus-based Image Description Evaluation. Ramakrishna Vedantam and others; IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

естественного языка. Метрики в значительной степени зависят от задачи и должны тщательно выбираться в зависимости от конкретной ситуации.

Изменение оценки BLEU-4 для нашей модели

На рис. 9.9 проиллюстрирована эволюция значения BLEU-4 для нашего упражнения. Мы видим, что с течением времени показатель увеличивается и приближается к 0,3. Обратите внимание, что наилучший результат на момент написания книги для набора данных MS-COCO составлял около 0,369[1], что достижимо при использовании гораздо более сложных моделей, а также более продвинутой регуляризации. Кроме того, фактический полный обучающий набор MS-COCO почти в три раза превышает размер используемой нами учебной выборки. Поэтому оценка BLEU-4 0,3 с учетом ограниченных обучающих данных, единственной ячейки LSTM и без специальной регуляризации является довольно хорошим результатом.

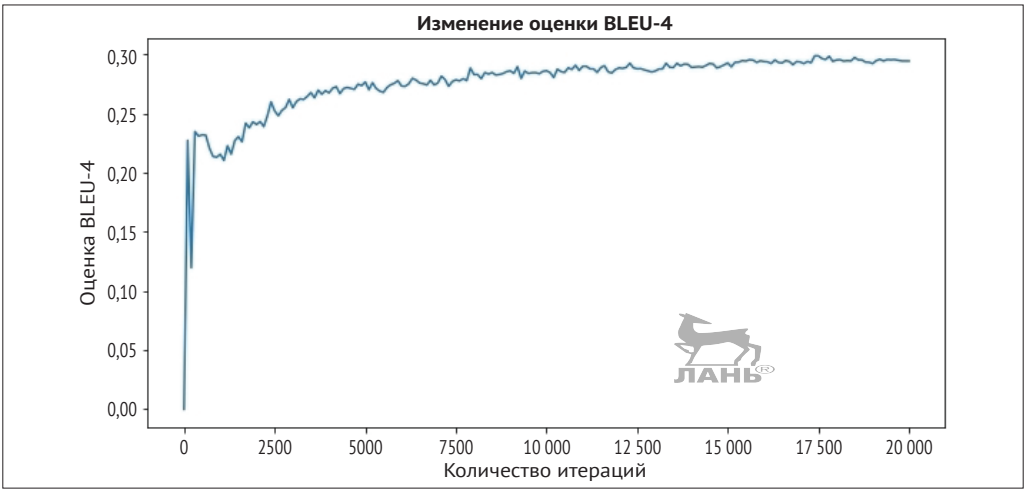


Рис. 9.9 ❖ График изменения BLEU-4 при генерации подписей к рисункам

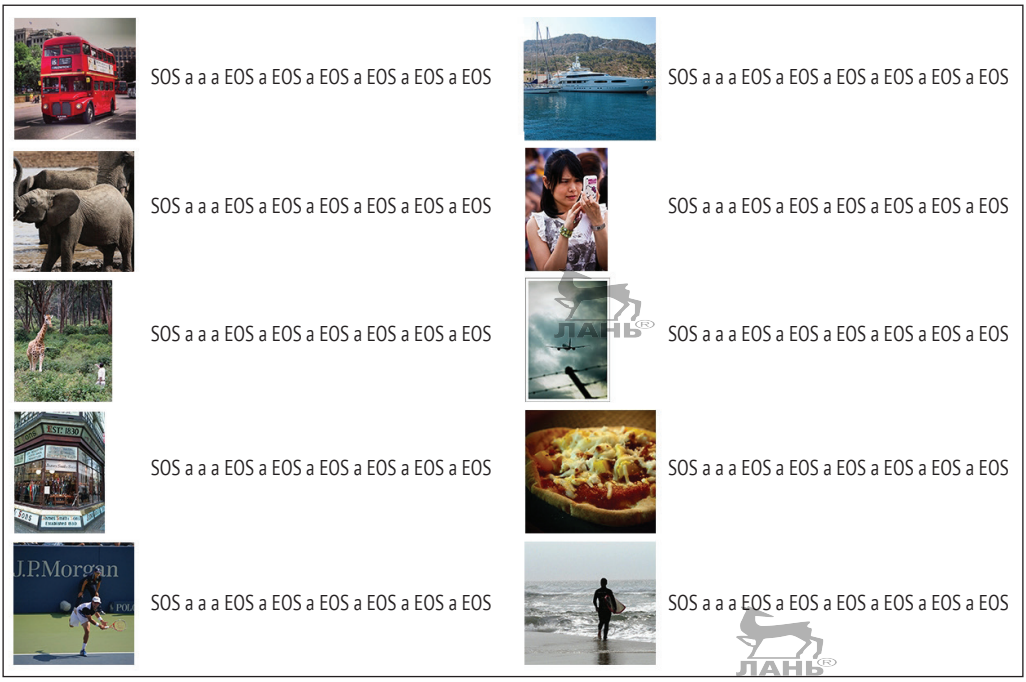
Подписи, созданные для тестовых изображений

Давайте посмотрим, какие подписи получаются для тестовых изображений при разной продолжительности обучения.

После 100 циклов обучения единственное, что узнала наша модель, – что подпись начинается с токена SOS и есть несколько слов, за которыми следует набор токенов EOS (рис. 9.10).

После 1000 циклов наша модель усвоила, что нужно генерировать более семантически сложные фразы, и правильно распознает объекты на некоторых изображениях (например, мужчина с теннисной ракеткой на рис. 9.11). Тем не менее текст является чересчур коротким и обобщенным, и, кроме того, некоторые изображения описаны неправильно.

[1] Bottom-Up and Top-Down Attention for Image Captioning and Visual Question Answering. Anderson and others, 2017.



SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

SOS a a a EOS a EOS a EOS a EOS a EOS a EOS

Рис. 9.10 ❖ Подписи к рисункам после 100 циклов обучения

SOS часовой транспорт на улице EOS

SOS часовой транспорт на улице EOS

SOS человек стоит на поле EOS

SOS человек сидит на столе EOS

SOS человек в поле EOS

SOS человек на здании EOS

SOS транспорт с часами на улице EOS

SOS тарелка тарелок на столе со столом EOS

SOS человек в теннисе с теннисом на теннисе EOS

SOS человек находится на теннисе EOS

Рис. 9.11 ❖ Подписи к рисункам после 1000 циклов обучения

После 2000 циклов наша модель стала заметно лучше генерировать выразительные подписи, составленные с учетом правильной грамматики (рис. 9.12). Уже нет слишком коротких и обобщенных предложений.

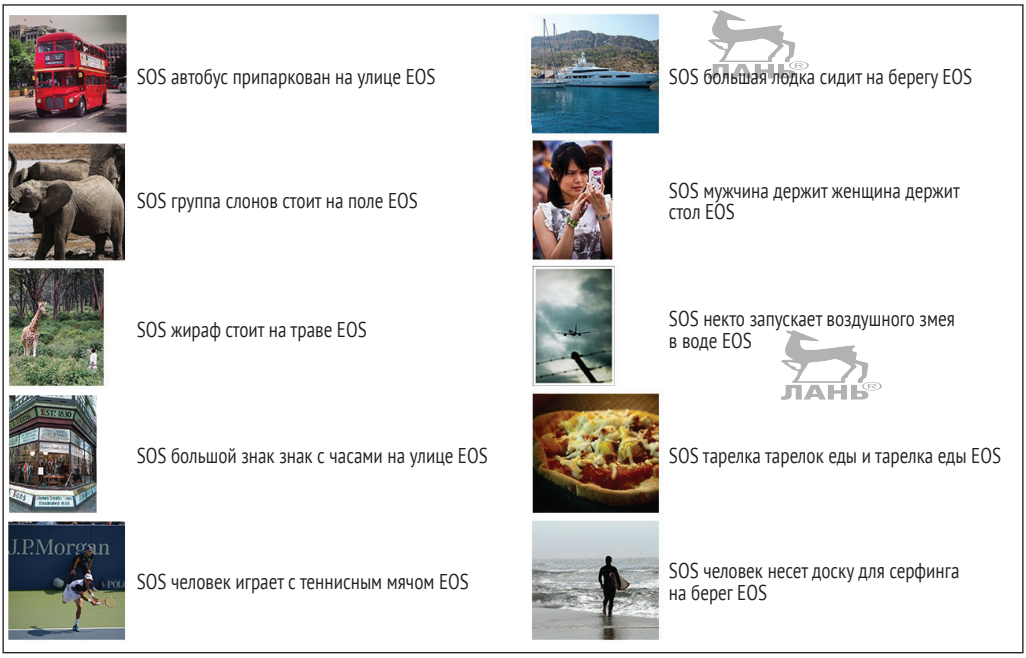


Рис. 9.12 ❖ Подписи к рисункам после 2000 циклов обучения

После 5000 циклов обучения наша модель правильно распознает большинство изображений (рис. 9.13). Также она может генерировать очень релевантные и грамматически правильные фразы, объясняющие, что происходит на изображении, однако не всегда работает идеально. Например, наш алгоритм классифицирует четвертое изображение совершенно неправильно. На нем изображено здание, и наш алгоритм справедливо предполагает, что это нечто урбанистическое, но делает вывод, что это дорожный указатель с часами. Восьмое изображение также распознается неправильно. Там изображен самолет в небе, но алгоритм принимает его за человека, управляющего воздушным змеем.

После 10 000 циклов обучения наш алгоритм достаточно хорошо описывает изображения, хотя и ошибается с девятым образцом. На рисунке показана пицца, но алгоритм думает, что это сэндвич (рис. 9.14). Другая неточность состоит в том, что седьмое изображение на самом деле – женщина с мобильным телефоном, но алгоритм думает, что это мужчина. Однако на заднем плане есть другие люди, поэтому алгоритм может ошибочно принять человека на переднем плане за человека на заднем плане. С этого момента алгоритм генерирует различные варианты того, что происходит на изображении, так как каждое изображение имеет несколько обучающих подписей.

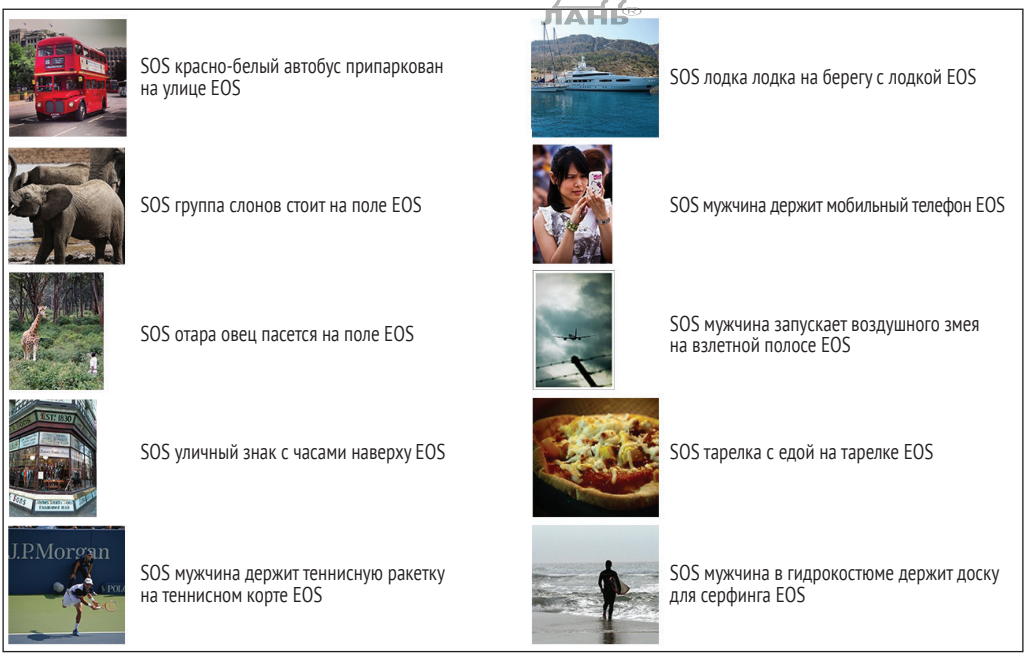


Рис. 9.13 ❖ Подписи к рисункам после 5000 циклов обучения

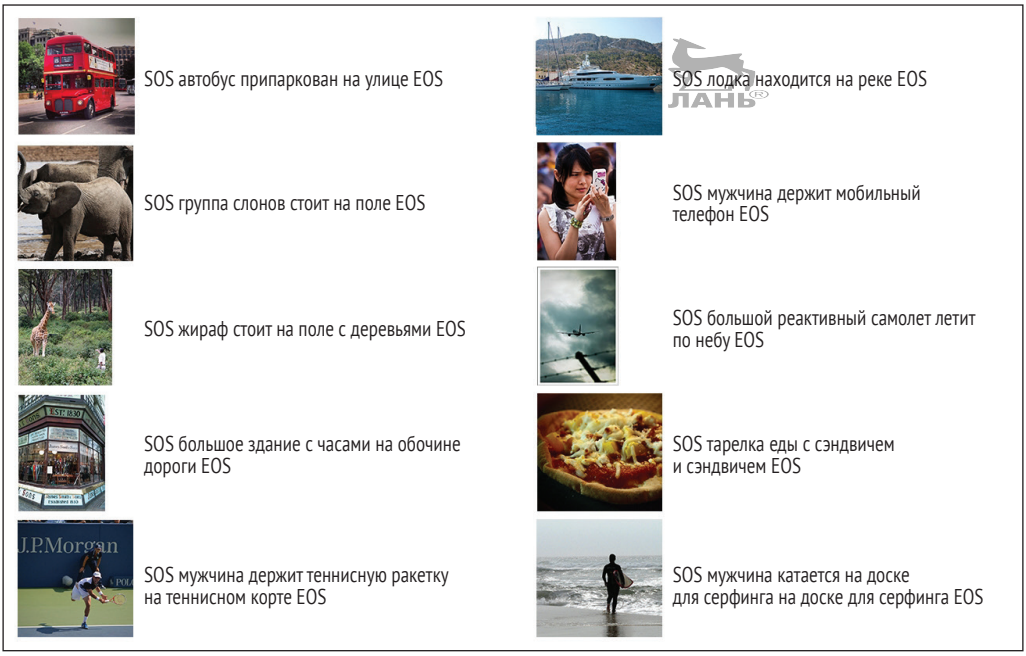


Рис. 9.14 ❖ Подписи к рисункам после 10 000 циклов обучения

Напоминаю, что эти результаты получены с использованием только приблизительно одной трети всех доступных обучающих данных. Кроме того, мы используем простую LSTM-сеть с одной ячейкой. Рекомендую вам попробовать максимизировать качество, используя полный набор обучающих данных, а также применить многослойные LSTM-сети или GRU с лучшей регуляризацией.

Использование TensorFlow RNN API с предварительно обученными векторами слов GloVe

Вы только что изучили реализацию нейросети с нуля, чтобы понять механизмы, лежащие в основе системы для генерации подписей к рисункам. Но это не оптимальный подход с точки зрения трудоемкости. Чтобы уменьшить объем кода и сократить обучение модели, можно использовать готовые решения RNN API вместе с предварительно обученными векторами слов GloVe. Полный код упражнения хранится в файле `lstm_image_caption_pretrained_wordvecs_rnn_api.ipynb` в папке `ch9`.

Сначала мы обсудим, как загружать векторы слов, а затем уточним, как выбрать из загруженного файла только нужные векторы, так как размер словаря предварительно обученных векторов GloVe составляет около 400 тыс. слов, а мы используем всего 18 тыс. Далее мы выполним простейшую корректировку правописания слов, так как в них присутствует много орфографических ошибок. Затем обработаем очищенные данные с помощью модуля `tf.nn.rnn_cell.LSTMCell`, представленного в RNN API.

Загрузка векторов слов GloVe

Начнем с загрузки файла представлений GloVe. Скачаем его по адресу **https://nlp.stanford.edu/projects/glove/** и поместим в папку `ch9`. Далее определим массив NumPy для хранения загруженных релевантных векторов слов из GloVe:

```
pret_embeddings = np.empty(shape=(vocabulary_size,50), dtype=np.float32)
```

Потом откроем ZIP-архив, содержащий загруженные векторы слов GloVe, и прочитаем его построчно. Данный архив содержит несколько различных вариантов GloVe, имеющих разные размеры представлений (например, 50, 100). Мы возьмем из этого архива файл `glove.6B.50d.txt`, так как он является наименьшим и соответствует проблеме, которую мы пытаемся решить. Каждая строка в файле имеет следующий формат (значения в строке разделены пробелами):

```
dog 0.11008 -0.38781 -0.57615 -0.27714 0.70521 ...
```

Теперь займемся извлечением представлений слов из файла. Сначала мы откроем ZIP-файл и прочитаем текстовый файл, который нам нужен (`glove.6B.50d.txt`):

```
with zipfile.ZipFile('glove.6B.zip') as glovezip:
    with glovezip.open('glove.6B.50d.txt') as glovefile:
```

Далее будем перебирать строки текстового файла и читать слово, которое является первым элементом строки, а также считывать соответствующий вектор представления для этого слова:

```
for li, line in enumerate(glovefile):
    # Декодируем строку и очищаем ее от любых
    # нераспознаваемых символов
    line_tokens = line.decode('utf-8').split(' ')

    # Получаем слово
    word = line_tokens[0]

    # Получаем вектор
    vector = [float(v) for v in line_tokens[1:]]
```

Затем, если текущее слово присутствует в нашем наборе данных, мы сохраняем его вектор в ранее определенном массиве NumPy. Вектор сохраняется в строке, заданной переменной `dictionary` и содержащей сопоставление слов с уникальным идентификатором. Нам понадобятся словоформы, обозначающие принадлежность. В английском языке принадлежность обозначается апострофом и буквой «*s*» в конце слова (например, *cat* → *cat's*). Но файл GloVe содержит слова только в начальной форме, поэтому мы обрабатываем текущее слово `word`, добавляя в конце апостроф и букву «*s*». Мы также сохраняем все слова из подписей, совпадающие со словами из набора GloVe, в список `words_in_glove`. Он пригодится на следующем шаге:

```
if word in dictionary.keys():
    words_in_glove.append(word)
    pret_embeddings[dictionary[word],:] = vector
    words_found += 1
    found_word_ids.append(dictionary[word])

    word_with_s = word + '\'s'
    if word_with_s in dictionary.keys():
        pret_embeddings[dictionary[word_with_s],:] =
            vector
        words_found += 1
        found_word_ids.append(dictionary[word_with_s])
```

Очистка данных

Теперь нам нужно разобраться с проблемой, которую мы игнорировали при изучении представлений слов с нуля. В наборе данных MS-COCO есть много орфографических ошибок. Чтобы максимально эффективно использовать предварительно изученные представления слов, нам необходимо исправить орфографические ошибки, иначе не получится сопоставить ошибочным словам правильные векторы признаков. Для исправления орфографии мы используем следующую процедуру.

В первую очередь вычислим идентификаторы слов MS-COCO, которые не были найдены в файле GloVe – возможно, из-за неправильного написания:

```
notfound_word_ids = list(set(list(range(0,vocabulary_size))) -
                         set(found_word_ids))
```

Затем, если какое-либо из этих слов окажется в подписи, мы постараемся исправить его написание, используя простейшую логику.

Сначала вычислим сходство между неправильным словом (cw) и всеми словами в списке words_in_glove (обозначены как gw), используя сопоставление строк:

```
# Для каждого слова, которое не найдено в наборе GloVe,
# мы ищем наиболее похожее написание
for gw in words_in_glove:
    cor, found_sim = correct_spellings.correct_wrong_word(cw,gw,cap)
```

Если это сходство больше 0,9 (порог найден опытным путем), мы заменим неправильное слово при помощи приведенного ниже кода. Мне пришлось исправить некоторые слова вручную, поскольку иногда ошибки допускают неоднозначное толкование. Например, слово «*stting*» одинаково похоже и на «*setting*» (настройка), и на «*sitting*» (сидение):

```
def correct_wrong_word(cw,gw,cap):
    '''
    Данный код реализует исправление ошибок правописания.
    Это очень простая логика, основанная на замене
    неправильного слова наиболее близким по написанию словом.
    Некоторые слова пришлось исправлять вручную по причине
    неоднозначного семантического толкования.
    '''

    correct_word = None
    found_similar_word = False
    sim = string_similarity(gw,cw)
    if sim>0.9:
        if cw != 'stting' and cw != 'sittign' and \
            cw != 'smilling' and \
            cw!='skiies' and cw!='childi' and cw!='sittion' and \
            cw!='peacefuly' and cw!='stainding' and \
            cw != 'staning' and cw!='lating' and cw!='sking' and \
            cw!='trolly' and cw!='umping' and cw!='earing' and \
            cw !='baters' and cw !='talkes' and cw !='trowing' and \
            cw !='convered' and cw !='onsie' and cw !='slying':
            print(gw,' ',cw,' ',sim,' (',cap,')')
            correct_word = gw
            found_similar_word = True
        elif cw == 'stting' or cw == 'sittign' or cw == 'sittion':
            correct_word = 'sitting'
            found_similar_word = True
        elif cw == 'smilling':
            correct_word = 'smiling'
            found_similar_word = True
        elif cw == 'skiies':
            correct_word = 'skis'
            found_similar_word = True
        elif cw == 'childi':
            correct_word = 'child'
            found_similar_word = True
    .
```

```
    .
    .
        elif cw == 'onsie':
            correct_word = cw
            found_similar_word = True
        elif cw =='slying':
            correct_word = 'flying'
            found_similar_word = True
        else:
        raise NotImplementedError
    else:
        correct_word = cw
        found_similar_word = False
return correct_word, found_similar_word
```

Приведенный выше код не способен исправить все найденные ошибки, но успешно справляется с большинством из них. Этого вполне достаточно для нашего упражнения.

Использование предварительно изученных представлений с RNN API

Завершив предварительную обработку данных, мы переходим к использованию RNN API с готовыми векторными представлениями слов GloVe. Сначала вы узнаете, как привести во взаимное соответствие размер векторов GloVe (50) и размер векторов изображения (1000). Затем вы научитесь использовать готовые модули LSTM из RNN API для изучения данных. Наконец, вы узнаете, как ввести в модель данные с различными модальностями – изображения и текст, которые должны обрабатываться по-разному. Ниже мы обсудим пошаговую реализацию упомянутых этапов. Общая структура обучаемой модели представлена в виде диаграммы на рис. 9.15.

Объявление слоя представлений и слоя подгонки

Сначала мы объявим переменную TensorFlow, содержащую предварительно изученные представления, и оставим для нее возможность изменений при обучении, поскольку для некоторых слов у нас есть только грубая инициализация. Например, мы использовали один и тот же вектор как для начальной формы слова, так и для формы, обозначающей принадлежность. Таким образом, векторы слов будут улучшаться по мере продолжения обучения:

```
embeddings = tf.get_variable(
        'glove_embeddings',shape=[vocabulary_size, 50],
        initializer=tf.constant_initializer(pret_embeddings,
        dtype=tf.float32)
)
```

Затем мы определим веса и смещения для адаптирующего слоя. Этот слой принимает входные данные размером `[batch_size, 50]`, представляющие собой пакет векторов слов GloVe, и преобразует их в пакет векторов с размером `[batch_size, 1000]`. Он действует как линейный слой, преобразующий векторы слов GloVe к правильному входному размеру, совпадающему с размерностью векторов изображений:

```
with tf.variable_scope('embeddings'):
    # Мы должны всегда приводить размерность входных данных LSTM
    # к размерности input_size.
    # Для этого мы используем разреживающий вектор,
    # который получает входные данные с размером 50
    # и выдает вектор с размером 1000 (размер входа)
embedding_dense = tf.get_variable('embedding_dense',
                  shape=[50,1000],
                  dtype=tf.float32,
              initializer=tf.contrib.layers.xavier_initializer())
    embedding_bias = tf.get_variable('embedding_bias',
                     dtype=tf.float32,
                     initializer=tf.random_uniform(
                         shape=[1000],
                         minval=-0.1,
                         maxval=0.1))
```

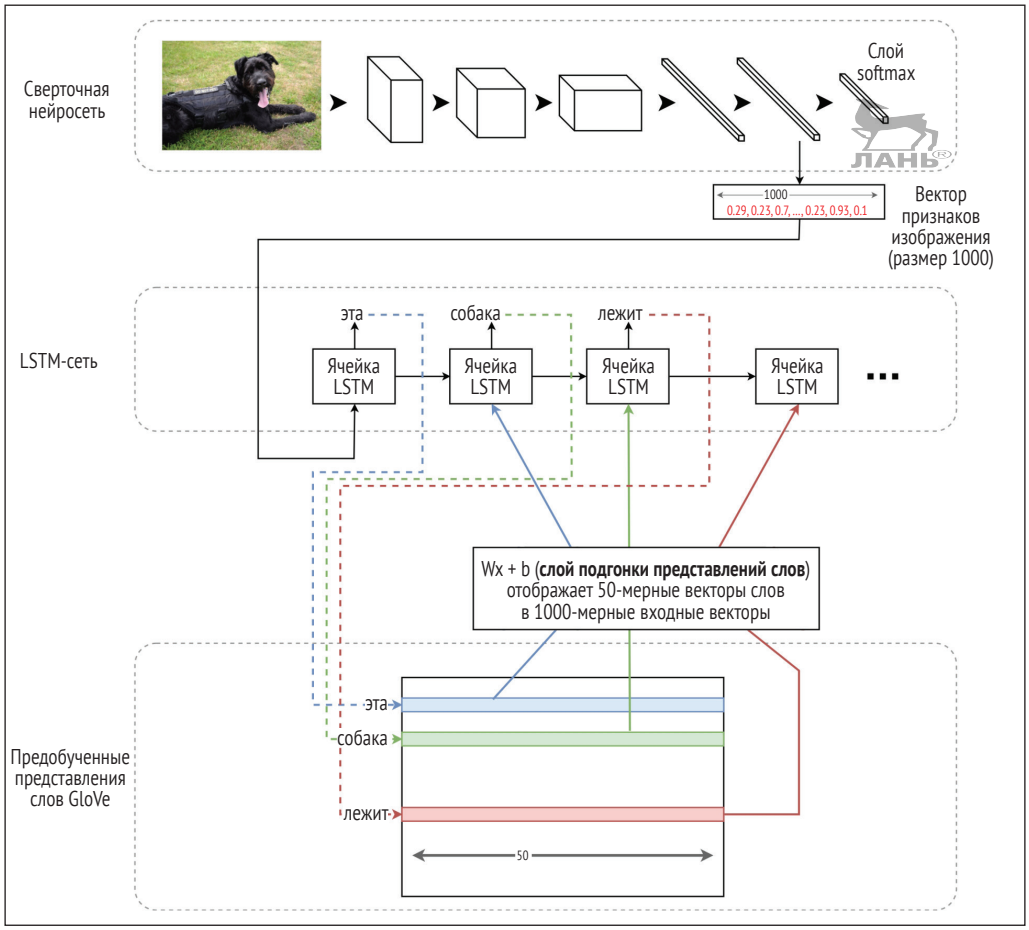


Рис. 9.15 ❖ Использование предварительно изученных представлений GloVe с RNN API

Объявление ячейки LSTM и слоя softmax

Далее мы определим ячейку LSTM, которая учится сопровождать изображение последовательностью слов, и слой softmax, преобразующий выходные данные ячейки LSTM в вероятностный прогноз. Для повышения точности и борьбы с переобучением будем использовать `DropoutWrapper` аналогично тому, как это описано в главе 8:

```
# Ячейка LSTM и дропаут
with tf.variable_scope('rnn'):
    lstm = tf.nn.rnn_cell.LSTMCell(num_nodes)
    # Используем дропаут для повышения точности
    dropout_lstm = rnn.DropoutWrapper(
        cell=lstm, input_keep_prob=0.8,
        output_keep_prob=0.8, state_keep_prob=1.0,
        dtype=tf.float32
    )
```

Задаем веса и смещения слоя softmax:

```
# Веса и смещения слоя softmax
with tf.variable_scope('rnn'):
    w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size],
                    stddev=0.01),
                    name='softmax_weights',
                    trainable=True)
    b = tf.Variable(tf.random_uniform([vocabulary_size],0.0,0.01),
                    name='softmax_bias',trainable=True)
```

Входные и выходные заполнители

Теперь мы определим входные и выходные заполнители, которые будут содержать входы и выходы, необходимые для обучения нашей модели. У нас будут три важных заполнителя:

- `is_train_text` – это список заполнителей длиной `num_unrollings`, в котором каждый заполнитель содержит логическое значение, указывающее, что именно мы вводим в данный момент – вектор признаков изображения или текст. Это важно, так как мы позже определим операцию условной обработки ввода, т. е. если логическое значение равно `false`, вернуть признак изображения как есть; если логическое значение равно `true`, выполнить обход `tf.nn.embedding_lookup` для входных данных;
- `train_inputs` – это список заполнителей в количестве `num_unrollings`, где каждый заполнитель содержит входные данные размера `[batch_size, 1000]` (1000 – это параметр `input_size`). Для изображений мы будем вводить вектор признаков изображения, а для текста – набор идентификаторов слов, составляющих заголовок. Мы будем добавлять к каждому идентификатору 999 нулей, чтобы привести вход к размерности 1000, где 999 нулей отбрасываются при последующей обработке;
- `train_labels` – это список заполнителей в количестве `num_unrollings`, которые будут содержать соответствующий выход для данного входа, т. е. SOS, если вход является вектором признаков изображения, или следующее слово в заголовке, если входом является слово в подписи.

Код выглядит следующим образом:

```
is_train_text, train_inputs, train_labels = [],[],[]

for ui in range(num_unrollings):
    is_train_text.append(tf.placeholder(tf.bool,
        shape=None, name='is_train_text_data_ %d' %ui))
    train_inputs.append(tf.placeholder(tf.float32,
        shape=[batch_size,input_size],name='train_inputs_ %d' %ui))
    train_labels.append(tf.placeholder(tf.int32,
        shape=[batch_size], name = 'train_labels_ %d' %ui))
```

Раздельная обработка текста и изображений

Обратите внимание на одно из самых важных отличий, возникающих при использовании предварительно изученных представлений, по сравнению с изучением представлений с нуля. Когда мы изучали представления с нуля, то могли по своему усмотрению задать размер вектора представления слова, чтобы он совпадал с размером вектора признаков изображения. Однако, поскольку сейчас мы используем предварительно подготовленные представления, и они не соответствуют размерности входа, нам нужно применять специальный адаптирующий слой, отображающий 50-мерные входные данные на 1000-мерный вход. Кроме того, мы должны сообщить TensorFlow, что не следует выполнять предварительное преобразование для векторов изображений. Давайте посмотрим, как это реализовать.

Во-первых, воспользуемся операцией `tf.cond`, чтобы различать два разных механизма обработки. Операция `tf.cond(pred, true_fn, false_fn)` может переключаться между различными операциями (т. е. `True_fn` и `false_fn`) в зависимости от того, является логическое значение `pred` истинным или ложным. Нам необходимо добиться следующего:

- если данные представляют собой вектор признаков изображения (т. е. `is_train_text` имеет значение `false`), нам не требуется дополнительная обработка. Мы просто передадим данные, используя операцию `tf.identity`;
- если данные являются идентификаторами слов, т. е. `is_train_text` имеет значение `true`, нам сначала нужно выполнить операцию `tf.nn.embedding_lookup` для набора идентификаторов слов, расположенных в нулевом столбце. Далее мы пропускаем возвращенные векторы слов размером `[batch_size, 50]` через адаптирующий слой, чтобы получить векторы с размером `[batch_size, 1000]`, используя `embedding_dense` и `embedding_bias`. Адаптирующий слой работает как типичный слой полносвязной нейронной сети без нелинейной активации.

Записываем обработанные входы в `train_inputs_processed`:

```
train_inputs_processed = []
for ui in range(num_unrollings):

    train_inputs_processed.append(
        tf.cond(is_train_text[ui],
            lambda: tf.add(
                tf.matmul(tf.nn.embedding_lookup(
                    embeddings, tf.reduce_sum(tf.cast(
                                    train_inputs[ui],tf.int32),
                    axis=1)
```

```
            ),embedding_dense),embedding_bias),
        lambda: tf.identity(train_inputs[ui]))
    )
```

Нам также нужно установить форму каждого тензора, найденного в списке train_inputs_processed, потому что после выполнения операции tf.cond информация о форме теряется. Кроме того, знание формы требуется для вычислений ячейки LSTM:

```
[t_in.set_shape([batch_size,input_size]) for t_in in train_inputs_
processed]
```

Вычисление выхода LSTM

Объявим начальное состояние ячейки LSTM:

```
initial_state = lstm.zero_state(batch_size, dtype=tf.float32)
```

Затем, используя функцию tf.nn.dynamic_rnn, вычислим выходные данные для всех временных шагов в окне num_unrollings, которые мы рассчитаем для LSTM за один шаг:

```
# Получаем выход с размерностью [num_unrolling, batch_size, num_nodes]
train_outputs, initial_state = tf.nn.dynamic_rnn(
    dropout_lstm, tf.concat([tf.expand_dims(t_in,axis=0) for t_in in
train_inputs_processed],axis=0),
    time_major=True, initial_state=initial_state
)
```

Определение логитов и прогнозов

Ранее рассчитанный результат train_output будет иметь размер [num_unrollings, batch_size, vocabulary_size]. Это размерность, основанная на оси времени (time-major format). Затем, чтобы рассчитать логиты и прогнозы из выходных данных LSTM за один раз для всех временных шагов num_unrollings, мы изменим окончательный результат следующим образом:

```
final_output = tf.reshape(train_outputs,[-1,num_nodes])
logits = tf.matmul(final_output, w) + b
train_prediction = tf.nn.softmax(logits)
```

Вычисление потери

Теперь переведем логиты и метки обратно в размерность по оси времени, поскольку это требуется функцией потерь, которую мы используем:

```
time_major_train_logits = tf.reshape(logits,[
    num_unrollings,batch_size,vocabulary_size])

time_major_train_labels = tf.reshape(tf.concat(
    train_labels,axis=0),[num_unrollings,batch_size])
```

Затем вычислим потерю, используя функцию tf.contrib.seq2seq.sequence_loss. Нам понадобится усреднение потерь по пакету, но суммирование по временным шагам:

```
loss = tf.contrib.seq2seq.sequence_loss(
    logits = tf.transpose(time_major_train_logits,[1,0,2]),
    targets = tf.transpose(time_major_train_labels),
    weights= tf.ones([batch_size, num_unrollings],
                     dtype=tf.float32),
    average_across_timesteps=False,
    average_across_batch=True
)
loss = tf.reduce_sum(loss)
```

Оптимизация

Наконец, мы объявим оптимизатор, который будет оптимизировать предварительно изученные представления, слой подгонки, ячейку LSTM и веса softmax, исходя из вычисленной выше потери. Для достижения высокой точности применим оптимизатор AdamOptimizer и снижение скорости обучения, как уже делали в главе 8:

```
# Эта переменная и операция нужны для снижения скорости обучения,
# как было показано в главе 8
global_step = tf.Variable(0, trainable=False)
inc_gstep = tf.assign(global_step,global_step + 1)

# Назначаем коэффициент снижения скорости обучения.
learning_rate = tf.train.exponential_decay(
    0.001, global_step, decay_steps=1, decay_rate=0.75,
    staircase=True)
# Назначаем оптимизатор Adam Optimizer.
optimizer = tf.train.AdamOptimizer(learning_rate)

# Отсечение градиента.
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimizer = optimizer.apply_gradients(
    zip(gradients, v))
```

Определив все необходимые операции TensorFlow, вы можете запустить процесс оптимизации для заранее определенного числа циклов обучения путем вычисления оценки BLEU на тестовых данных, а также прогнозов для нескольких тестовых изображений. Точный процесс можно найти в файле упражнения.

Заключение

В этой главе мы сфокусировались на очень интересной задаче – генерации подписей к изображениям. Наша модель представляет собой сложный конвейер машинного обучения, включающий следующие этапы:

- определение векторов признаков для данного изображения с использованием CNN;
- изучение представлений слов, найденных в подписи;
- обучение LSTM-сети с помощью векторов признаков изображения и соответствующих им подписей.

Мы подробно обсудили каждый этап. Во-первых, мы поговорили о том, как использовать предварительно обученную модель CNN на большом наборе классификационных данных ImageNet для извлечения хороших векторов признаков без обучения модели с нуля. Для этого использовали VGG с 16 слоями. Далее мы пошагово изучили создание переменных TensorFlow, загрузку весов и создание сети. Наконец, проверили модель на нескольких тестовых изображениях, чтобы убедиться, что она действительно способна распознавать визуальные объекты.

Затем мы воспользовались алгоритмом CBOW, чтобы изучить правильные представления слов, найденных в заголовках обучающих данных. Мы убедились, что сопоставили размерность векторных представлений слов с векторами признаков изображения, поскольку стандартные LSTM не могут обрабатывать входы с динамически меняющейся размерностью.

Наконец, мы взяли простую LSTM-сеть и вводили в нее последовательность данных, где первый элемент – это вектор признаков изображения, за которым следуют представления слов, составляющих подпись изображения. Мы предварительно обработали заголовки, введя два токена для обозначения начала и конца каждого заголовка, а затем нормировали заголовки, чтобы все они имели одинаковую длину.

После этого обсудили несколько различных метрик для количественной оценки сгенерированных подписей – BLEU, ROUGE, METEOR и CIDEr – и увидели, что при выполнении нашего алгоритма показатель BLEU-4 увеличивается по мере обучения. Кроме того, мы визуально убедились, что лексическое качество подписей растет по мере обучения модели.

Наконец, мы обсудили использование предварительно изученных представлений слов GloVe и TensorFlow RNN API для выполнения той же задачи с меньшим количеством кода и большей точностью.

В следующей главе вы узнаете, как реализовать систему машинного перевода, которая принимает предложение на немецком языке в качестве входных данных и выводит перевод этого предложения на английский язык.

Глава 10

Преобразование последовательностей и машинный перевод

Обучение преобразованию последовательности в последовательность – это термин, обозначающий задачи, которые требуют отображения последовательности произвольной длины в другую последовательность произвольной длины. Это одна из самых сложных задач, включающих в себя изучение сопоставлений «многие-ко-многим». Примеры этой задачи включают *нейронный машинный перевод* (neural machine translation, NMT) и создание чат-ботов. NMT – это инструмент для перевода предложения с исходного языка на целевой язык. Наиболее известным примером системы NMT является переводчик Google Translate. *Чат-боты*, т. е. программы, способные общаться с человеком или отвечать ему, поддерживают диалог с людьми в реалистичной манере. Это особенно полезно для различных поставщиков услуг, поскольку чат-боты можно использовать для поиска ответов на легко решаемые вопросы, возникающие у клиентов, вместо того чтобы адресовать их операторам-людям.

В этой главе вы узнаете, как создать собственную систему NMT. Однако прежде чем перейти непосредственно к таким передовым достижениям, мы сначала кратко рассмотрим некоторые методы *статистического машинного перевода* (statistical machine translation, SMT), которые предшествовали NMT и были на коне, пока их не обогнал нейронный перевод. Далее мы шаг за шагом реализуем настоящую систему нейронного машинного перевода с немецкого языка на английский.

Машинный перевод

Люди намного чаще общаются друг с другом с помощью живого языка по сравнению с другими способами общения, например жестами. В настоящее время во всем мире говорят на более чем 5 тыс. языков. Естественные языки необходимы для обмена знаниями, общения и расширения социальных связей. К сожалению, изучение языка до уровня, легко понятного носителю этого языка, является весьма сложной задачей. Поэтому естественные языки – это не только средство, но и барьер в общении с разными людьми. Вот тут и приходит на помощь *машинный*

перевод (machine translation, MT). Системы MT позволяют пользователю вводить предложение на родном языке (известном как исходный язык) и получать предложение на желаемом целевом языке.

В общем виде задачу MT можно сформулировать следующим образом. Скажем, нам дано предложение (последовательность слов), принадлежащее исходному языку S и определяемое следующим образом:

$$W_S = \{w_1, w_2, w_3, \ldots, w_L\}.$$

Здесь $W_S \in S$.

Исходный язык будет переведен в предложение W_T, принадлежащее целевому языку T и определяемое следующим образом:

$$W_T = \{w'_1, w'_2, w'_3, \ldots, w'_M\}.$$

Здесь $W_T \in T$.

Последовательность W_T генерируется системой машинного перевода, которая работает по следующему принципу:

$$p(W_T|W_S) \forall W_T \in W_T^*.$$

Здесь W_T^* – пул возможных *кандидатов* на перевод, найденных алгоритмом для исходного предложения. Кроме того, лучший кандидат из пула кандидатов определяется следующим уравнением:

$$W_T^{best} = \text{argmax}_{W_T \in W_T^*}(p(W_T|W_S); \theta).$$

Здесь θ – параметры модели. Во время обучения мы оптимизируем модель так, чтобы максимизировать вероятность некоторых известных целевых переводов для набора соответствующих исходных предложений, т. е. обучающих данных.

Это была формальная постановка проблемы языкового перевода, решение которой нам интересно. Далее мы бегло пройдемся по истории MT, чтобы понять, как люди пытались решить проблему машинного перевода до появления нейросетей.

Краткая историческая экскурсия по машинному переводу

Первые системы машинного перевода были основаны на правилах. Затем появились более точные и гибкие статистические системы перевода. Они при создании переводов опирались на различные показатели статистики языков. Потом наступила эра NMT. В настоящее время NMT обладает самыми передовыми характеристиками по сравнению с другими методами.

Перевод на основе правил

Статистический машинный перевод существует уже более полувека. История SMT-методов ведет отсчет с 1950–1960 гг., когда во время одного из первых заре-

гистрированных проектов, эксперимента Georgetown-IBM, более 60 русских предложений были переведены на английский язык.

Но мы немного забежали вперед. Одним из первых методов МТ был *прямой словарный перевод*. Эта система выполняла переводы «слово в слово» с использованием двуязычных словарей. Однако, как вы можете себе представить, такой метод имеет серьезные ограничения. Дело в том, что перевод слов не является взаимно однозначным отображением между разными языками. Кроме того, дословный перевод может привести к неверным результатам, так как он не учитывает контекст предложения. Перевод данного слова на исходном языке может радикально меняться в зависимости от контекста, в котором оно используется. Для наглядного представления давайте рассмотрим пример перевода с русского языка на французский, проиллюстрированный на рис. 10.1. В двух русских предложениях изменяется лишь одно слово. Однако это порождает радикальные изменения в переводе.

Рис. 10.1 ❖ Перевод с русского на французский очень далек от простых связей «один-к-одному»

В 60-х годах Консультативный комитет по автоматической обработке языков (automatic language processing advisory committee, ALPAC) опубликовал доклад[1] «Языки и машины: компьютеры в переводе и лингвистике» о перспективах машинного перевода. Вывод был просто убийственным:

> *Отсутствует очевидная или прогнозируемая перспектива извлечения пользы из машинного перевода.*

В то время машинный перевод был более дорогим, медленным и заметно менее качественным по сравнению с переводом, выполненным людьми. Доклад ALPAC нанес огромный удар по репутации МТ, и почти на десятилетие эта отрасль науки замерла в молчании.

Затем появились методы машинного перевода, основанные на текстовых корпусах (corpora-based MT, *корпусный перевод*), где алгоритм обучается с использованием кортежей исходного предложения, и соответствующее целевое предложение получается через параллельный корпус, имеющий формат ([(<*source_sentence_1*>, <*target_sentence_1*>), (<*source_sentence_2*>, <*target_sentence_2*>), ...]). *Параллельный корпус* – это большой текстовый корпус, сформированный в виде кортежей и состоящий из текста на исходном языке и соответствующего перевода этого текста. Следует отметить, что построение параллельного корпуса гораздо проще, чем

[1] Languages and machines: computers in translation and linguistics. National Academy of the Sciences (1966).

создание двуязычных словарей, и дает более качественный результат, поскольку данные для обучения богаче, чем при словарном методе. Вместо того чтобы напрямую полагаться на созданные вручную двуязычные словари, можно построить так называемую *модель перехода* (transition model) с использованием параллельного корпуса. Модель перехода показывает вероятность того, что целевое слово/фраза будет правильным переводом исходного слова/фразы. В дополнение к изучению модели перехода система корпусного перевода также изучает модели выравнивания слов. *Модель выравнивания слов* (word alignment model) изучает взаимное соответствие между словами во фразе из исходного языка и переводом этой фразы. Пример выравнивания слов изображен на рис. 10.2. Иллюстрация примера параллельных корпусов показана в табл. 10.1.

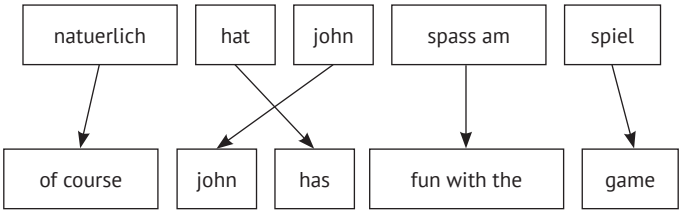


Рис. 10.2 ❖ Выравнивание слов между двумя разными языками

Таблица 10.1. Пример параллельных корпусов – русский и французский языки

Исходное предложение на русском	Целевое предложение на французском
Я ушел домой	Je suis rentré chez moi
Иван любит играть на гитаре	John aime jouer de la guitare
Он родом из России	Il est d'Russia
...	....

Другим популярным подходом был промежуточный машинный перевод, который включал перевод исходного предложения в нейтральный *промежуточный язык* (т. е. метаязык), а затем генерирование переведенного предложения на целевом языке. Более конкретно, система промежуточного машинного перевода состоит из двух важных компонентов: *анализатора* и *синтезатора*. Анализатор получает исходное предложение и определяет агенты (например, существительные), действия (например, глагол) и т. д., а также форму их взаимодействия друг с другом. Далее эти идентифицированные элементы представляют посредством *межъязыкового лексикона* (interlingual lexicon). Примером межъязыкового лексикона могут служить наборы (т. е. группы синонимов, имеющих общее значение), доступные в WordNet. Затем из этого межъязыкового представления синтезатор генерирует перевод. Поскольку синтезатор получает существительные, глаголы и т. д. через нейтральное межъязыковое представление, он может генерировать перевод с добавлением правил грамматики целевого языка.

Статистический машинный перевод (SMT)

Затем начали появляться статистически обоснованные системы. Одной из новаторских моделей этой эпохи были IBM Models 1–5, которые выполняли перевод

на основе слов. Однако, как мы обсуждали ранее, прямое сопоставление слов непродуктивно и в ряде случаев просто невозможно. В конце концов, исследователи начали экспериментировать с системами перевода на основе фраз, которые добились заметных успехов в машинном переводе.

Перевод по фразам работает аналогично словарному переводу, за исключением того, что в качестве атомных единиц перевода он использует фразы языка вместо отдельных слов. Это более разумный подход, поскольку он облегчает моделирование отношений «один-ко-многим», «многие-к-одному» или «многие-ко-многим». Основной целью перевода на основе фраз является изучение *модели фразового перевода* (phrase-translation model), которая содержит распределение вероятностей различных целевых фраз-кандидатов для данной исходной фразы. Как вы можете догадаться, этот метод предполагает ведение огромных баз данных различных фраз на двух языках. Приходится также выполнять операцию перестановки слов, поскольку между строением фраз на разных языках нет прямого монотонного соответствия. Пример перестроения фраз показан на рис. 10.2. Если бы слова были монотонно упорядочены между языками, то между отображениями не возникали бы перекрещивания.

Одним из ограничений этого подхода является то, что процесс декодирования (поиск лучшей целевой фразы для данной исходной фразы) является дорогостоящим. Это связано с размером базы данных фраз, а также с исходной фразой, которая часто сопоставляется нескольким фразам на целевом языке. Для облегчения поиска начали применять синтаксические переводы.

В системе *синтаксического перевода* исходное предложение представлено *синтаксическим деревом* (syntax tree). На рис. 10.3 NP представляет собой вещественную часть, VP – глагольную часть, а S – предложение. Затем выполняется *фаза перестановки*, когда узлы дерева переставляются для изменения порядка субъекта, глагола и объекта в зависимости от целевого языка. Это связано с тем, что структура предложения может меняться в зависимости от языка (например, в английском языке это субъект – глагол – объект, тогда как в японском это субъект – объект – глагол). Перестановку выполняют в соответствии с так называемой *r-таблицей*. Она содержит вероятности правдоподобия для перестановки узлов дерева в каком-либо другом порядке.

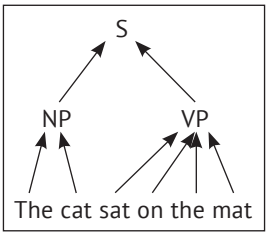


Рис. 10.3 ❖ Синтаксическое дерево предложения

Далее выполняется *фаза вставки*. На этом этапе мы стохастически вставляем слово в каждый узел дерева, исходя из допущения, что существует невидимое слово NULL, и оно генерирует целевые слова в случайных позициях дерева. Кроме

того, вероятность вставки слова определяется так называемой *n-таблицей*, содержащей вероятности вставки конкретного слова в дерево.

Затем происходит *фаза перевода*, где каждый конечный узел переводится в целевое слово и переведенное предложение считывается из синтаксического дерева.

Нейронный машинный перевод

Наконец, примерно в 2014 году появились системы нейронного машинного перевода. По сути, NMT – это сквозная система, которая принимает полное предложение в качестве входных данных, выполняет определенные преобразования, а затем выводит переведенное предложение. Таким образом, нейронный перевод устраняет необходимость в конструировании признаков, необходимых для машинного перевода, таких как построение моделей перевода фраз и построение синтаксических деревьев, что является большой победой сообщества NLP. Кроме того, нейронный перевод буквально за два-три года превзошел все другие популярные методы машинного перевода. На рис. 10.4 показаны достижения различных систем машинного перевода, отмеченные в научной литературе. Например, результаты 2016 года опубликованы Сеннрихом и др. в статье[1] «Эдинбургские системы нейронного машинного перевода для WMT16» и Уильямсом и др. в статье[2] «Эдинбургские системы статистического машинного перевода для WMT16». Все системы машинного перевода оценивались по метрике BLEU. Как мы поясняли в главе 9, показатель BLEU обозначает количество *n*-грамм (например, униграмм и биграмм) перевода кандидата, которые совпадают с эталонным переводом. Таким образом, чем выше показатель BLEU, тем лучше система машинного перевода. Мы подробно обсудим метрику BLEU ниже в этой главе. Очевидно, что NMT является победителем.

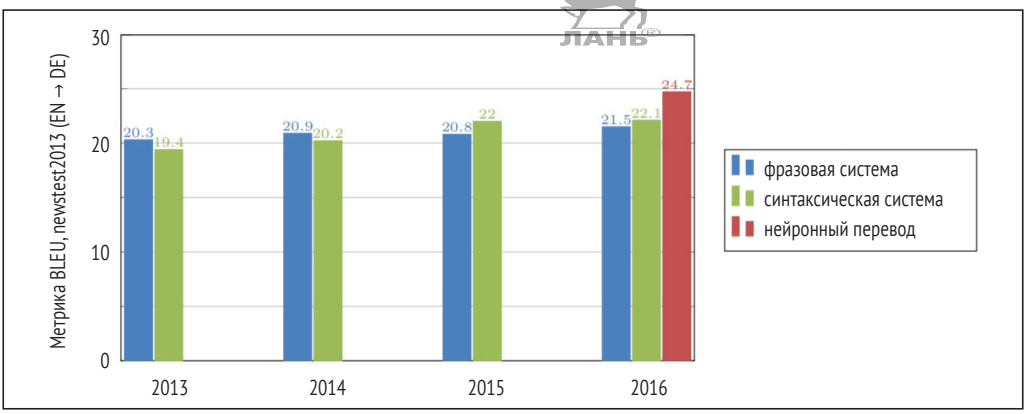


Рис. 10.4 ❖ Сравнение качества систем статистического машинного перевода и NMT. *Источник*: Rico Sennrich

[1] Edinburgh Neural Machine Translation Systems for WMT 16. Association for Computational Linguistics, Proceedings of the First Conference on Machine Translation, August 2016: 371–376.

[2] Edinburgh's Statistical Machine Translation Systems for WMT16. Association for Computational Linguistics, Proceedings of the First Conference on Machine Translation, August 2016: 399–410.

Относительно недавно опубликовано обзорное исследование[1] «Готовы ли системы нейронного машинного перевода к использованию?», оценивающее потенциал систем NMT. В исследовании проведен анализ качества работы различных систем по переводу между различными языками (английский, арабский, французский, русский и китайский). Результаты также подтверждают, что системы NMT (NMT 1.2M и NMT 2.4M) работают лучше, чем системы SMT (PB-SMT и Hiero).

На рис. 10.5 приведены некоторые статистические данные, характеризующие состояние машинного перевода в 2017 году. Это иллюстрация из презентации[2] «Состояние машинного перевода», подготовленной Константином Савенковым, соучредителем и генеральным директором Intento. Мы видим, что качество системы машинного перевода, созданной DeepL (**https://www.deepl.com**), похоже, тесно конкурирует с другими гигантами IT-индустрии, включая Google. Сравнение включает системы MT, такие как DeepL (NMT), Google (NMT), Яндекс (гибрид NMT-SMT), Microsoft (имеет SMT и NMT), IBM (SMT), Prompt (на основе правил) и SYSTRAN (гибрид на основе правил и SMT). Диаграмма ясно показывает, что системы NMT занимают лидирующие позиции. Для оценки различных систем применялась метрика LEPOR – более продвинутый показатель, чем BLEU, решающий проблему *неровной оценки*. Упомянутая проблема относится к явлению, когда некоторые метрики (например, BLEU) хорошо работают для одних языков, но плохо для некоторых других.

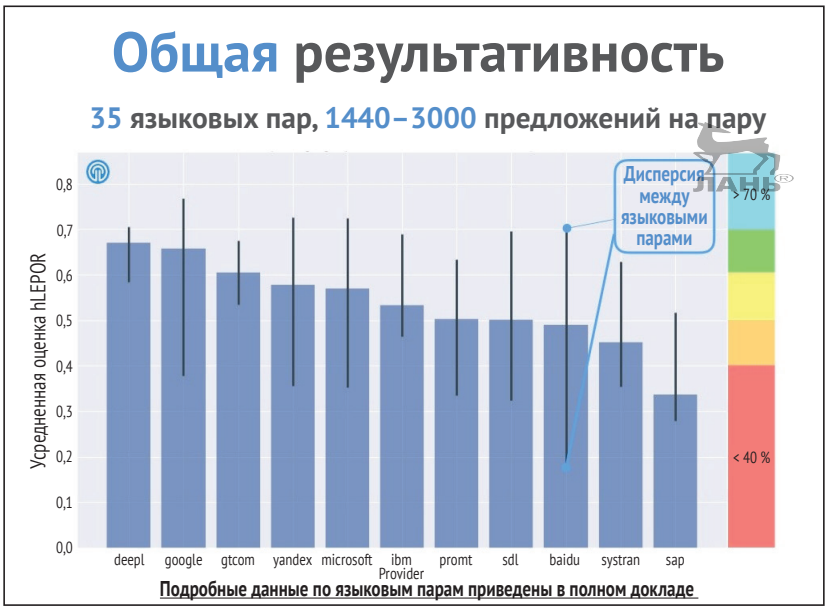


Рис. 10.5 ❖ Качество различных систем машинного перевода.
Источник: Intento Inc.

[1] Is Neural Machine Translation Ready for Deployment? A Case Study on 30 Translation Directions. Junczys-Dowmunt, Hoang and Dwojak, Proceedings of the Ninth International Workshop on Spoken Language Translation, Seattle (2016).

[2] State of the Machine Translation, Intento, Inc., 2017.

Тем не менее следует также отметить, что результатам присуща некоторая предвзятость из-за механизма усреднения, используемого в этом сравнении. Например, оценку системы Google Translator усреднили по большому набору языков, включая сложные задачи перевода, тогда как результаты DeepL усреднили по меньшему и относительно простому подмножеству языков. Поэтому не следует делать вывод, что система DeepL лучше, чем система Google Translator. Тем не менее результаты дают общее представление о качестве современных систем NMT и SMT.

Хорошо видно, что буквально за несколько лет системы нейронного машинного перевода превзошли SMT-системы. Теперь мы перейдем к обсуждению деталей и архитектуры NMT, а затем займемся созданием системы нейронного машинного перевода с нуля.

Общие принципы нейронного машинного перевода

Теперь, когда вы бегло познакомились с историей машинного перевода, давайте разберемся, как работают современные системы NMT. Сначала мы рассмотрим архитектуру модели нейронного машинного перевода, а затем перейдем к изучению фактического алгоритма обучения.

Устройство NMT

Во-первых, давайте разберемся с идеей, лежащей в основе системы NMT. Скажем, вы свободно говорите по-немецки и по-русски, и вас попросили перевести следующее немецкое предложение на русский язык:

Ich ging nach Hause

Это предложение означает:

Я ушел домой

Хотя подготовленному человеку для перевода предложения потребуется не более двух секунд, на самом деле это сложный процесс. Сначала вы читаете предложение на немецком языке, а затем формируете мысль или концепцию о том, что представляет или подразумевает это предложение. И наконец, переводите предложение на русский язык. Эта же идея лежит в основе систем NMT (рис. 10.6). Кодер считывает исходное предложение аналогично тому, как вы читаете немецкое предложение. Затем он выводит *вектор контекста* (context vector), который соот-

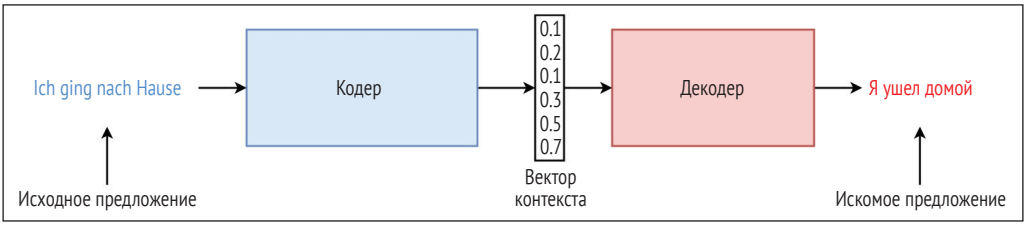


Рис. 10.6 ❖ Обобщенная архитектура системы NMT

ветствует мысли или концепции, возникающей у вас в голове. Наконец, декодер принимает вектор контекста и генерирует предложение на русском языке.

Архитектура NMT

Теперь рассмотрим архитектуру NMT более детально. Подход «последовательность–последовательность», обсуждаемый здесь, был предложен Суцкевером, Виньялсом и Ли в их статье[1] «Обучение нейросетей преобразованию последовательность–последовательность». Из диаграммы на рис. 10.6 видно, что в архитектуре NMT есть два основных компонента. Они называются кодером и декодером. Другими словами, NMT можно рассматривать как архитектуру кодер–декодер. *Кодер* преобразует предложение из данного исходного языка в *мысль*, а *декодер* переводит мысль на целевой язык. Как видите, здесь имеются некоторые общие черты с методом промежуточного метаязыка, о котором мы кратко упоминали выше. Этот подход проиллюстрирован на рис. 10.7. Слева от вектора контекста расположен кодер, принимающий исходное предложение слово за словом для обучения модели временных рядов. Справа расположен декодер, который выводит слово за словом, используя предыдущее слово в качестве текущего ввода. Таким образом, получается перевод исходного предложения. Мы также будем использовать слои представлений как для исходного, так и для целевого языка, чтобы передавать векторы слов в качестве входных данных.

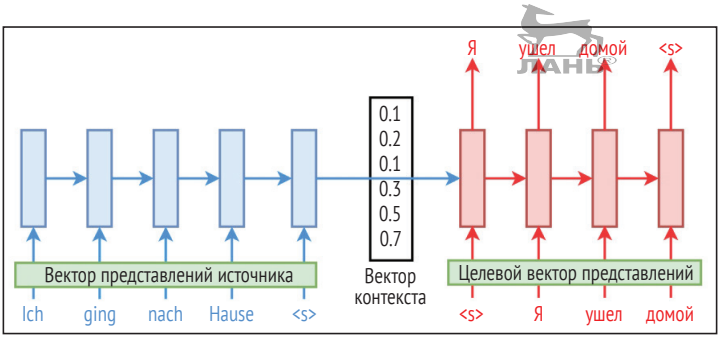


Рис. 10.7 ❖ Развертывание во времени исходной и целевой последовательности

Разобравшись в основной идее NMT, давайте формально определим задачу с математической точки зрения. Конечная цель системы NMT – максимизировать логарифмическое правдоподобие при исходном предложении x_S и соответствующем ему y_T, то есть максимизировать следующее выражение:

$$\frac{1}{N}\sum_{i=1}^{N}\log P(y_T|x_S).$$

[1] Sequence to Sequence Learning with Neural Networks. Sutskever, Vinyals, and Le, Proceedings of the 27th International Conference on Neural Information Processing Systems. Volume 2: 3104–3112.

N обозначает число исходных и целевых кортежей предложений, которые мы имеем в качестве обучающих данных.

Затем для данного исходного предложения x_S^{infer} путем логического вывода мы найдем лучший перевод y_T^{best}, используя условие:

$$y_T^{best} = \text{argmax}_{y \in Y_T} P(y_T | x_S^{infer}) = \text{argmax}_{y \in Y_T} \prod_{i=1}^{M} P(y_T^i | x_S^{infer}).$$

Здесь Y_T – набор возможных последовательностей-кандидатов.

Прежде чем детально рассмотреть архитектуру NMT, определим математические обозначения, относящиеся к системе машинного перевода.

Давайте определим кодер LSTM как $LSTM_{enc}$ и декодер LSTM как $LSTM_{dec}$. На шаге времени t определим состояние ячейки LSTM как c_t, а внешнее скрытое состояние – как h_t. Следовательно, подача входа x_t в LSTM производит c_t и h_t:

$$c_t, h_t = LSTM(x_t | x_1, x_2, \ldots, x_{t-1}).$$

Теперь поговорим о слое представлений, кодере, векторе контекста и, наконец, о декодере.

Слой представлений

Как в главе 8, так и в главе 9 мы подробно обсудили преимущества векторного представления слов вместо использования унитарного кода, особенно с большими словарями. Мы также будем применять слой представлений из двух слов Emb_S для исходного языка и Emb_T для целевого языка. Таким образом, непосредственно в LSTM мы подаем $Emb(x_t)$, а не x_t. Однако, чтобы избежать ненужного усложнения записи, мы будем подразумевать, что $x_t = Emb(x_t)$.

Кодер

Как упоминалось ранее, кодер отвечает за генерацию «вектора мысли» или вектора контекста, представляющего смысл, подразумеваемый в исходном предложении. Для этого мы воспользуемся LSTM-сетью (рис. 10.8).

Кодер инициализирован нулевыми векторами c_0 и h_0. Кодер получает последовательность слов $x_s = \{x_s^1, x_s^2, \ldots, x_s^L\}$ на вход и вычисляет вектор контекста $v = \{v_c, v_h\}$, где v_c – окончательное состояние ячейки и v_h – окончательное внешнее скрытое состояние, достигнутое после обработки последнего элемента x_T^L последовательности x_T. Это можно представить следующим образом:

$$c_L, h_L = LSTM_{enc}(x_s^L | x_s^1, x_s^2, \ldots, x_s^{L-1});$$
$$v_c = c_L;$$
$$v_h = h_L.$$

Вектор контекста

Вектор контекста v представляет предложение на исходном языке в сжатой форме. Кроме того, если начальное состояние кодера инициализируется нулями, то вектор контекста становится начальным состоянием для декодера LSTM. Другими словами, декодер LSTM запускается не с начального нулевого состояния, а с век-

тором контекста в качестве исходного состояния. Мы поговорим об этом более подробно ниже.

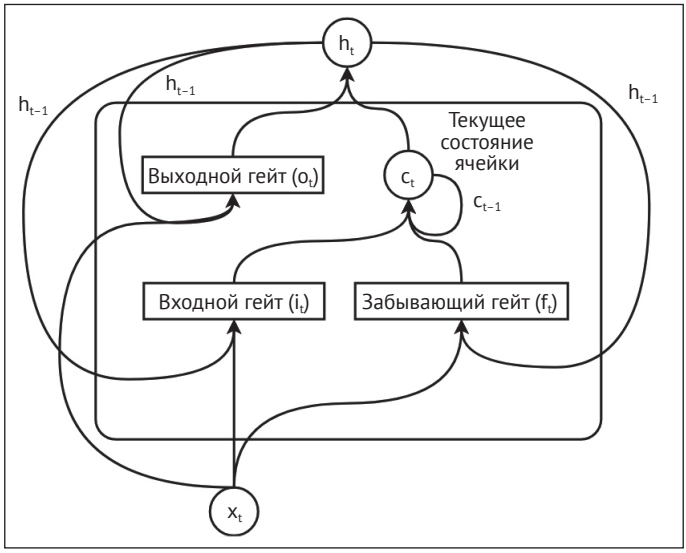


Рис. 10.8 ❖ Ячейка LSTM

Декодер

Декодер отвечает за декодирование вектора контекста в желаемый перевод и тоже является LSTM-сетью. Хотя кодер и декодер могут совместно использовать один и тот же набор весов, обычно лучше использовать две разные сети для кодера и декодера. Это увеличивает количество параметров в нашей модели, что позволяет нам более эффективно изучать переводы. Во-первых, состояния декодера инициализируются контекстным вектором $v = \{v_c, v_h\}$, как показано ниже:

$$c_0 = v_c;$$
$$h_0 = v_h.$$

Здесь $c_0, h_0 \in LSTM_{dec}$.

Вектор v является критически важным звеном, соединяющим кодер с декодером для формирования сквозной вычислительной цепочки. На рис. 10.6 вектор контекста – это единственное, что совместно используется кодером и декодером. Кроме того, это единственная часть информации об исходном предложении, которая доступна декодеру.

Далее мы вычисляем m-й прогноз переведенного предложения следующим образом:

$$c_m, h_m = LSTM_{dec}(y_T^{m-1}|v, y_T^1, y_T^2, \ldots, y_T^{m-2});$$
$$y_T^m = softmax(w_{softmax} \times h_m + b_{softmax}).$$

Полная система NMT с расшифровкой того, как ячейка LSTM в кодере подключается к ячейке LSTM в декодере и как уровень softmax используется для вывода прогнозов, показана на рис. 10.9.

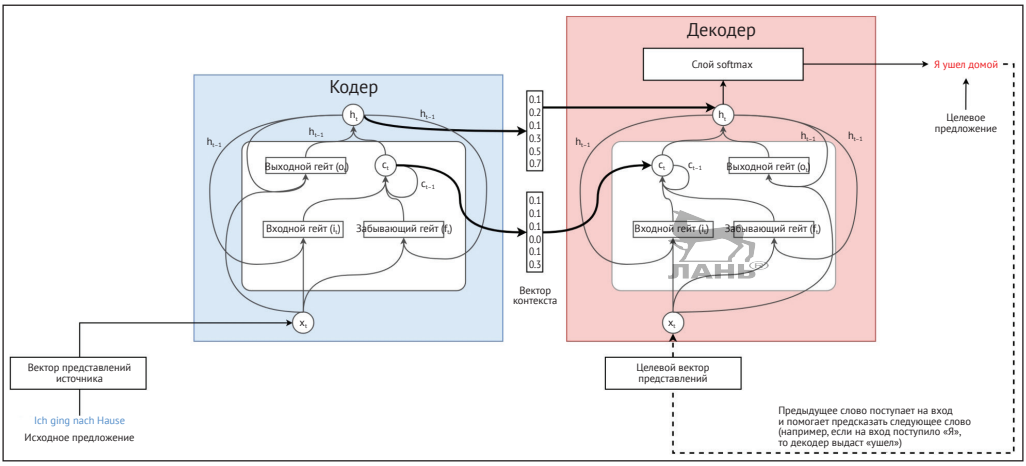


Рис. 10.9 ❖ Архитектура кодер–декодер

Подготовка данных для системы NMT

В этом разделе мы подробно рассмотрим процесс подготовки данных для обучения и прогнозирования в системе NMT. Начнем с подготовки обучающих пар, состоящих из исходного и целевого предложений.

Этап обучения

Обучающие данные состоят из сопоставлений исходных предложений и соответствующих переводов на целевой язык. Пример может выглядеть так:

- (*I went home*, *я пошел домой*)
- (*She was waiting at school*, *она ждала в школе*)

У нас есть *N* таких пар в нашем наборе данных. Если мы хотим реализовать довольно хороший переводчик, значение *N* должно быть в масштабе миллионов. Увеличение объема обучающих данных подразумевает увеличение времени обучения.

Далее мы введем два специальных токена: <*s*> и </*s*>. Токен <*s*> обозначает начало предложения, в свою очередь, </*s*> указывает на конец предложения. Теперь данные будут выглядеть так:

- (<*s*> *I went home* </*s*>, <*s*> *я пошел домой* </*s*>)
- (<*s*> *She was waiting at school* </*s*>, <*s*> *она ждала в школе* </*s*>)

После этого дополним предложения токенами </*s*> так, чтобы исходные предложения имели фиксированную длину *L*, а целевые предложения – фиксированную длину *M*. Следует отметить, что *L* и *M* не обязательно должны быть равными. Этот шаг приводит к следующей форме обучающих данных:

- (<*s*> *I went home* </*s*>, <*s*> *я пошел домой* </*s*>)
- (<*s*> *She was waiting at school* </*s*>, <*s*> *она ждала в школе* **</*s*>**</*s*>)

Если предложение имеет длину больше, чем *L* или *M*, оно усекается, чтобы соответствовать заданной длине. Затем предложения пропускаются через токенизатор для вывода токенизированных слов. Здесь я покажу только первый обучающий кортеж, так как все кортежи обрабатываются одинаково:

(['<s>', '*I*', '*went*', '*home*', '</s>'], ['<s>', '*я*', '*пошел*', '*домой*', '</s>'])

Следует отметить, что приведение предложений к фиксированной длине не является обязательным, поскольку LSTM-сети способны обрабатывать последовательности переменной длины по одной за раз. Однако приведение последовательностей к фиксированной длине помогает нам обрабатывать предложения пакетами, а не поштучно.

Переворачивание исходного предложения

Далее мы выполним специальный трюк с исходными предложениями. Скажем, у нас есть предложение *ABC* на исходном языке, которое мы хотим перевести в предложение αβγφ на целевом языке. Сначала перевернем исходные предложения, чтобы предложение *ABC* читалось как *CBA*. Это означает, что для перевода *ABC* в αβγφ нам придется подать в модель последовательность *CBA*. Подобный трюк значительно повышает точность модели, особенно когда исходный и целевой языки имеют одинаковую структуру предложений, например субъект – глагол – объект.

Давайте попробуем понять, почему это помогает. Главным образом этот прием помогает построить хорошую связь между кодером и декодером. Давайте начнем с предыдущего примера. Мы объединяем исходное и целевое предложения:

*ABC*αβγφ

Если вы вычислите расстояния (то есть количество слов, разделяющих два слова) от *A* до α или от *B* до β, они будут одинаковыми. Однако смотрите, что получится, когда вы перевернете исходное предложение, как показано здесь:

*CBA*αβγφ

Теперь *A* окажется ближе всего к α и т. д. Для создания качественных переводов важно с самого начала создавать хорошие связи. Этот простой прием помогает системам NMT улучшить точность.

Теперь наш набор данных приобретает следующий вид:

(['</s>', '*home*', '*went*', '*I*', '<s>'], ['<s>', '*я*', '*пошел*', '*домой*', '</s>'])

Далее, используя изученные представления Emb_s и Emb_T, мы заменяем каждое слово соответствующим вектором представления.

Другая хорошая новость заключается в том, что наше исходное предложение заканчивается токеном <s>, а целевое предложение начинается с токена <s>, поэтому во время обучения нам не нужно выполнять какую-либо специальную обработку для создания связи между окончанием исходного и началом целевого предложения.

Обратите внимание, что переворачивание исходного предложения не требуется в некоторых переводческих задачах. Например, если нужен перевод с японского языка (субъект – объект – глагол) на филиппинский (глагол – субъект – объект), то переворачивание может скорее навредить, чем помочь. Это связано с тем, что, переворачивая текст на японском языке, вы увеличиваете расстояние между начальным элементом исходного предложения (глаголом японским) и соответствующей сущностью целевого языка (глаголом филиппинским).

Этап тестирования

Во время тестирования у нас есть только исходное предложение, но нет целевого. Тем не менее мы подготавливаем исходные данные аналогично этапу обучения. Затем получаем перевод слово за словом, подавая последнее предсказанное декодером слово на вход рекуррентной сети. Новый цикл прогнозирования запускается подачей токена <*s*> в декодер.

Далее мы рассмотрим детали процедур обучения и прогнозирования для данного исходного предложения.

Обучение NMT

Теперь, когда мы выстроили архитектуру NMT и приготовили предварительно обработанные обучающие данные, обучение модели не составит труда. Давайте пошагово распишем и наглядно проиллюстрируем (рис. 10.10) полный процесс обучения.

1. Выполняем предварительную обработку (x_S, y_T), как говорилось выше.
2. Подаем x_S в $LSTM_{enc}$ и вычисляем вектор v, обусловленный x_S.
3. Инициализируем $LSTM_{dec}$ вектором v.
4. Прогнозируем в $LSTM_{dec}$ целевое предложение $\hat{y}_T = \{\hat{y}_T^1, \hat{y}_T^2, \ldots, \hat{y}_T^M\}$, соответствующее входному предложению x_S, где m-й прогноз в целевом словаре V вычисляется следующим образом:

   $$\hat{y}_T^m = softmax(w_{softmax} h^m + b_{softmax});$$
   $$w_T^m = \text{argmax}_{w^m \in V} P(\hat{y}_T^{(m,w^m)} | v, \hat{y}_T^1, \ldots, \hat{y}_T^{m-1}).$$

   Здесь w_T^m обозначает лучшее целевое слово для m-й позиции.
5. Вычисляем ошибку – категориальную перекрестную энтропию между предсказанным словом $\hat{y}_T^m$ и истинным словом y_T^m в m-й позиции.
6. Оптимизируем $LSTM_{enc}$, $LSTM_{dec}$ и слой softmax, исходя из ошибки.

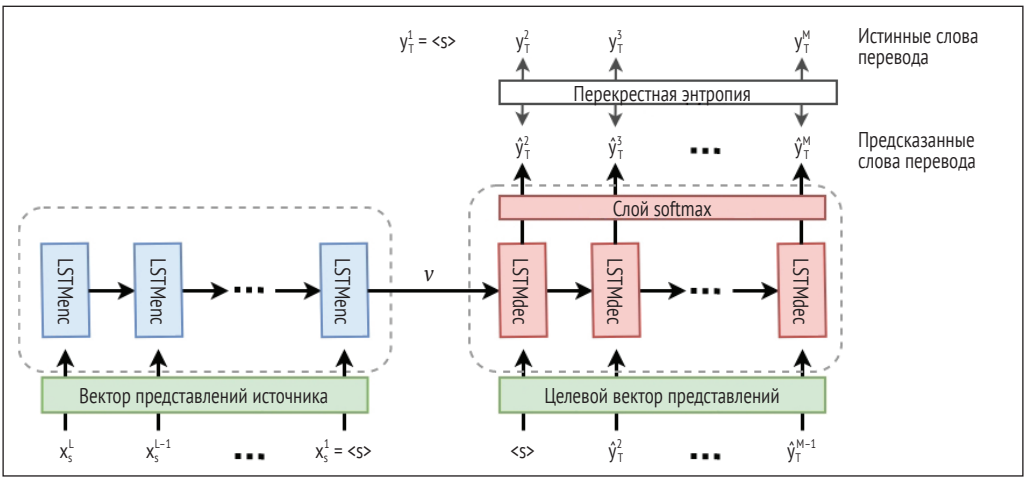


Рис. 10.10 ❖ Процесс обучения NMT

Вывод перевода из NMT

Процесс вывода искомого перевода немного отличается от процесса обучения (рис. 10.11). Поскольку у нас нет целевого предложения во время вывода, нам нужен способ запустить декодер в конце фазы кодирования. В этом текущая задача похожа на упражнение по созданию подписей к рисункам, которое мы выполнили в главе 9. Тогда мы использовали токен <SOS>, обозначающий начало подписи, и токен <EOS>, обозначающий ее конец.

Сейчас мы можем просто передать токен <s> в качестве первого входа для декодера, а затем получить прогноз на выходе, рекурсивно вводя последний прогноз в качестве следующего входа в NMT.

1. Выполняем предварительную обработку x_S, как обсуждалось выше.
2. Подаем x_S в $LSTM_{enc}$ и вычисляем v, обусловленный x_S.
3. Инициализируем $LSTM_{dec}$ вектором v.
4. На начальном шаге прогнозирования предсказываем $\hat{y}_T^2$, исходя из $\hat{y}_T^1$ = <s> и v.
5. На последующих шагах, пока $\hat{y}_T^i \neq$ </s>, предсказываем $\hat{y}_T^{m+1}$, исходя из $\{\hat{y}_T^m, \hat{y}_T^{m-1}, \ldots,$ <s>$\}$ и v.

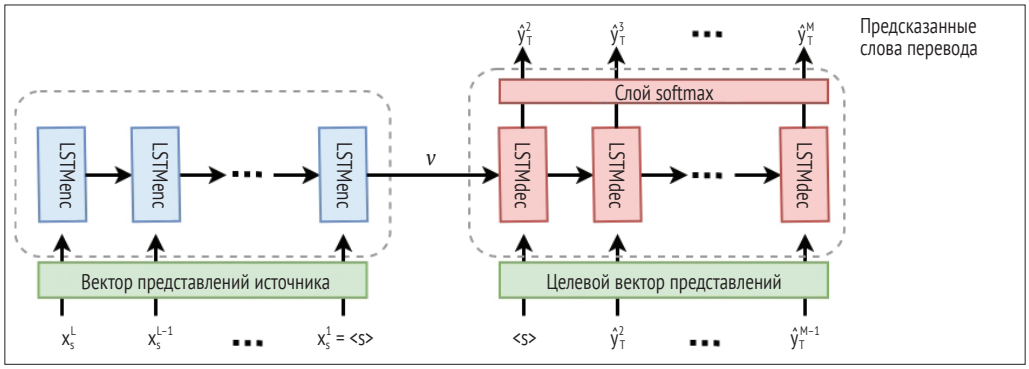


Рис. 10.11 ❖ Вывод перевода из NMT

Метрика BLEU – оценка систем машинного перевода

В главе 9 вы уже кратко познакомились с BLEU, способом автоматической оценки систем машинного перевода. Полный код алгоритма оценки находится в файле `bleu_score_example.ipynb`. Давайте разберемся, как он работает.

Рассмотрим простой пример. Допустим, у нас есть два предложения-кандидата, предсказанных системой машинного перевода и истинный (т. е. эталонный) перевод для некоторого исходного предложения:

- эталон 1: *Этот кот сидит на мягком коврике*;
- кандидат 1: *Этот кот лежит на мягком коврике*.

Чтобы оценить, насколько хорош перевод, мы можем использовать одну метрику – точность. *Точность* – это мера того, сколько слов из предложения-канди-

дата фактически присутствует в ссылке. В общем случае, если вы рассматриваете проблему классификации с двумя классами, обозначаемыми как отрицательные и положительные, точность определяется по следующей формуле:

$$\text{Точность} = \frac{\text{выборки, правильно определенные как положительные}}{\text{все выборки, определенные как положительные}}.$$

Вычислим точность для *кандидата 1* как отношение количества слов кандидата, встречающихся в истинном предложении, к общему количеству слов кандидата.

Математически это можно выразить следующей формулой:

$$Precision = \frac{\sum_{unigram \in Candidate} IsFoundInRef(unigram)}{|Candidate|};$$

Точность кандидата 1 = 5/6.

Данная точность также известна как точность 1-граммы уровня слов, так как мы рассматриваем по одному слову за раз.

Теперь давайте рассмотрим второе предложение-кандидат.

Кандидат 2: *Этот этот этот кот кот кот*

Человек с первого взгляда увидит, что *кандидат 1* намного лучше, чем *кандидат 2*. Но давайте вычислим математическую точность:

Точность кандидата 2 = 6/6 = 1.

Как видите, математическая оценка точности не совпадает с нашим суждением. Следовательно, точность сама по себе не может служить исключительным критерием оценки качества перевода.

Модифицированная точность

Чтобы устранить недостатки вычисления точности, мы можем использовать модифицированную 1-граммную точность. *Модифицированная точность* обрезает количество вхождений каждого уникального слова в кандидате до количества раз, когда это слово появилось в истинном предложении:

$$p_1 = \frac{\sum_{unigram \in \{Candidate\}} \mathrm{Min}(Occurences(unigram), unigram_{\max})}{|Candidate|}.$$

Для рассмотренных выше *кандидата 1* и *кандидата 2* модифицированная точность будет следующей:

Мод. точность кандидата 1 = (1+1+1+1+1)/6 = 5/6.
Мод. точность кандидата 2 = (2+1)/6 = 3/6.

Как видите, это полезная модификация, поскольку точность кандидата 2 снижается. Модификация может быть расширена до любой *n*-граммы, рассматривая *n* слов за раз вместо одного слова.

Штраф за краткость

Точность, естественно, побуждает систему использовать небольшие предложения. Это поднимает вопрос объективности оценки, поскольку нейросеть может генерировать короткие неправильные переводы длинных исходных предложений и при этом иметь более высокую точность. По этой причине в цикл обучения вводят специальный *штраф за краткость* (brevity penalty, BP). Он вычисляется следующим образом:

$$BP = \begin{cases} 1, & \text{если } c > r \\ e^{(1-r/c)}, & \text{если } c \le r \end{cases}.$$

Здесь c – это длина предложения-кандидата, r – длина истинного предложения. В нашем примере штраф за краткость вычисляется следующим образом:

$$BP_{candidate\ 1} = e^{(1-(6/6))} = e^0 = 1;$$
$$BP_{candidate\ 2} = e^{(1-(6/6))} = e^0 = 1.$$

Окончательная оценка BLEU

Чтобы вычислить окончательную оценку BLEU, мы сначала вычислим несколько различных модифицированных точностей n-грамм для различных значений n = 1, 2, ..., N. Затем вычислим средневзвешенное геометрическое точностей n-грамм:

$$BLEU = BP \times \exp\left(\sum_{i=1}^{N} w_n p_n\right).$$

Здесь w_n – вес модифицированной n-граммной точности p_n. По умолчанию для всех значений n-грамм используются равные веса. Наконец, BLEU вычисляет модифицированную точность n-граммы и штрафует ее за краткость. Модифицированная точность позволяет избежать фиктивных значений высокой точности, которые получаются из бессмысленных предложений, таких как *кандидат 2*.

Собственная система NMT с нуля – переводчик с немецкого на английский

Теперь вы готовы создать настоящий нейронный машинный переводчик. Мы будем разрабатывать NMT с использованием базовых операций TensorFlow. Упражнение доступно в файле `ch10/neural_machine_translation.ipynb`. Однако в TensorFlow есть мощная библиотека под названием seq2seq. Вы можете получить больше информации о seq2seq, а также научиться программировать NMT с seq2seq в приложении к этой книге.

Причина, по которой мы используем «чистые» базовые операции TensorFlow, заключается в том, что как только вы научитесь реализовывать машинный переводчик с нуля без использования готовых функций, вы сможете быстро научиться

использовать библиотеку seq2seq и другие библиотеки. Кроме того, пока еще доступно слишком мало онлайн-ресурсов для обучения реализации последовательных моделей с использованием «чистого» TensorFlow, зато встречается множество ресурсов и учебных пособий о том, как использовать библиотеку seq2seq для машинного перевода.

TensorFlow предоставляет очень информативное обучающее пособие по системам «последовательность-в-последовательность», предназначенным для машинного перевода, по адресу **https://github.com/tensorflow/nmt**.

Знакомство с данными

Мы воспользуемся англо-немецкими парами предложений, расположенными по адресу **https://nlp.stanford.edu/projects/nmt/**. Для скачивания доступно около 4,5 млн пар предложений. Однако мы будем использовать только 250 тыс. пар из-за высокой вычислительной нагрузки. Словарь состоит из 50 тыс. наиболее распространенных английских слов и 50 тыс. наиболее распространенных немецких слов. Слова, не найденные в словаре, заменяются специальным маркером <unk>. Взгляните на примеры предложений, представленных в наборе данных:

```
DE: Das Großunternehmen sieht sich einfach die Produkte des kleinen
Unternehmens an und unterstellt so viele Patentverletzungen , wie es
nur geht.

EN: The large corporation will look at the products of the small
company and bring up as many patent infringement assertions as
possible.

DE: In der ordentlichen Sitzung am 22. September 2008 befasste
sich der Aufsichtsrat mit strategischen Themen aus den einzelnen
Geschäftsbereichen wie der Positionierung des Kassamarktes im
Wettbewerb mit außerbörslichen Handelsplattformen , den Innovationen
im Derivatesegment und verschiedenen Aktivitäten im Nachhandelsbereich.

EN: At the regular meeting on 22 September 2008 , the Supervisory
Board dealt with strategic issues from the various business areas ,
such as the positioning of the cash market in competition with OTC
trading platforms , innovation in the derivatives segment and various
post ##AT##-##AT## trading activities.
```

Предварительная обработка данных

Загрузив обучающие данные `train.en` и `train.de`, как указано в файле упражнения, посмотрите, что находится в этих файлах. Файл `train.en` содержит английские предложения, тогда как `train.de` содержит соответствующие немецкие предложения. Далее мы выберем 250 тыс. пар предложений из большого исходного корпуса данных. Также мы отберем из исходных данных 100 предложений, которые будем использовать в качестве теста. Словари для этих двух языков находятся в файлах `vocab.50K.en.txt` и `vocab.50K.de.txt`.

Затем мы предварительно обработаем эти данные, как было показано выше в данной главе. Переворачивание предложений не является обязательным при

изучении представлений слов (если выполняется отдельно), так как перестановка слов не влияет на контекст данного слова. Мы будем использовать следующий простой алгоритм токенизации для разбиения предложений на слова. По сути, мы всего лишь вводим пробелы перед различными знаками препинания, чтобы их можно было разбить на отдельные элементы. Затем заменим специальным токеном `<unk>` каждое слово, не найденное в словаре. Параметр `is_source` указывает, обрабатываем мы исходные предложения (`is_source=True`) или целевые предложения (`is_source=False`):

```
def split_to_tokens(sent,is_source):
    '''
    Эта функция берет предложение (исходное или целевое)
    и выполняет предварительную обработку,
    включая удаление знаков пунктуации.
    '''

    global src_unk_count, tgt_unk_count

    # Удаление знаков препинания и символа переноса строки
    sent = sent.replace(',',' ,')
    sent = sent.replace('.',' .')
    sent = sent.replace('\n',' ')

    sent_toks = sent.split(' ')
    for t_i, tok in enumerate(sent_toks):
        if is_source:
            # src_dictionary содержит сопоставление слова
            # с его индексом в словаре исходного текста
            if tok not in src_dictionary.keys():
                if not len(tok.strip())==0:
                    sent_toks[t_i] = '<unk>'
                    src_unk_count += 1
        else:
            # tgt_dictionary содержит сопоставление слова
            # с его индексом в словаре целевого текста
            if tok not in tgt_dictionary.keys():
                if not len(tok.strip())==0:
                    sent_toks[t_i] = '<unk>'
                    # print(tok)
                    tgt_unk_count += 1
    return sent_toks
```

Изучение представлений слов

Далее мы перейдем к изучению представлений слов, для чего воспользуемся моделью CBOW. Тем не менее вы можете поэкспериментировать с другими методами, такими как GloVe. Мы не будем здесь разбирать код, расположенный в файле `word2vec.py`, а поделимся некоторыми из готовых представлений слов, чтобы вы имели представление, как они выглядят:

Представления немецких слов:

```
Nearest to In: in, Aus, An, Neben, Bei, Mit, Trotz, Auf,
Nearest to war: ist, hat, scheint, wäre, hatte, bin, waren, kam,
```

```
Nearest to so: verbreitet, eigentlich, ausserdem, ziemlich, Rad-,
zweierlei, wollten, ebenso,
Nearest to Schritte: Meter, Minuten, Gehminuten, Autominuten, km,
Kilometer, Fahrminuten, Steinwurf,
Nearest to Sicht: Aussicht, Ausblick, Blick, Kombination, Milde,
Erscheinung, Terroranschläge, Ebenen,
```

Представления английских слов:

```
Nearest to more: cheaper, less, easier, better, further, greater,
bigger, More,
Nearest to States: Kingdom, Nations, accross, attrition, Efex,
Republic, authoritative, Sorbonne,
Nearest to Italy: Spain, Poland, France, Switzerland, Madrid,
Portugal, Fuengirola, 51,
Nearest to island: shores, Principality, outskirts, islands, skyline,
ear, continuation, capital,
Nearest to 2004: 2005, 2001, 2003, 2007, 1996, 2006, 1999, 1995,
```

Изучать представления можно одновременно с обучением системы машинного перевода. Альтернативой является использование предварительно изученных представлений слов. О том, как это сделать, мы поговорим позже в этой главе.

Кодер и декодер

Мы будем использовать две отдельные LSTM-сети в качестве кодера и декодера. Сначала определим гиперпараметры:

- `batch_size` – вам нужно быть очень осторожным при настройке размера пакета. Система NMT может занять довольно много памяти при работе;
- `num_nodes` – это количество скрытых блоков в LSTM. Увеличение гиперпараметра `num_nodes` приведет к повышению качества, но и к более высокой вычислительной стоимости;
- `enc_num_unrollings` – количество слов в исходном предложении. Мы развернем LSTM на всю длину предложения за одно вычисление. Чем выше значение `enc_num_unrollings`, тем лучше будет работать ваша модель. Однако это замедлит работу;
- `dec_num_unrollings` – количество слов в целевом предложении. Более высокое значение `dec_num_unrollings` также приведет к лучшему качеству, но и к большим вычислительным затратам;
- `embedding_size` – это размерность векторов, которые мы изучаем. Представления размером 100–300 будет достаточно для большинства реальных задач, в которых используются векторы слов.

Итак, определим гиперпараметры:

```
# Определяем размер входа, читая сохраненные представления слов
# и извлекая размер столбца
tgt_emb_mat = np.load('en-embeddings.npy')
input_size = tgt_emb_mat.shape[1]

num_nodes = 128
batch_size = 10
```

```
# Развертываем последовательность во времени на всю длину за раз
# как для исходного, так и для целевого предложения
enc_num_unrollings = 40
dec_num_unrollings = 60
```

Если у вас слишком большой размер пакета (более 20 на стандартном ноутбуке), вы можете столкнуться с таким сообщением о проблеме:

```
Resource exhausted: OOM when allocating tensor with ...
(Ресурс исчерпан: недостаточно памяти при размещении тензора с ...)
```

В этом случае вам следует уменьшить размер пакета и повторно запустить код.

Далее мы объявим веса и смещения для LSTM-сети и слоя softmax. Мы будем использовать область переменных кодера и декодера, чтобы сделать именование переменных более интуитивным. Это уже знакомая вам стандартная ячейка LSTM, и мы не будем повторять здесь объявление веса.

Затем мы объявим заполнители TensorFlow для этапа обучения:

- `enc_train_inputs` – это список заполнителей в количестве, равном `enc_num_unrollings`, где каждый заполнитель имеет размер `[batch_size, input_size]`. Он используется для подачи пакета предложений исходного языка в кодировщик;
- `dec_train_inputs` – это список заполнителей в количестве, равном `dec_num_unrollings`, где каждый заполнитель имеет размер `[batch_size, input_size]`. Он используется для подачи пакета предложений целевого языка;
- `dec_train_labels` – это список заполнителей в количестве, равном `dec_num_unrollings`, где каждый заполнитель имеет размер `[batch_size, vocabulary_size]`. Он содержит слова `dec_train_inputs` со смещением на 1. Таким образом, два заполнителя из `dec_train_inputs` и `dec_train_labels` с одинаковым индексом в списке будут содержать i-е слово и $(i + 1)$-е слово;
- `dec_train_masks` – он имеет тот же размер, что и `dec_train_inputs`, и убирает любой элемент, имеющий метку `</s>`, из расчета потерь. Это важно, поскольку существует много точек данных с токеном `</s>`, так как он используется для заполнения предложений до фиксированной длины:

```
for ui in range(dec_num_unrollings):
    dec_train_inputs.append(tf.placeholder(tf.float32,
        shape=[batch_size,input_size],
        name='dec_train_inputs_ %d' %ui))
    dec_train_labels.append(tf.placeholder(tf.float32,
        shape=[batch_size,vocabulary_size],
        name = 'dec_train_labels_ %d' %ui))
    dec_train_masks.append(tf.placeholder(tf.float32,
        shape=[batch_size,1],
        name='dec_train_masks_ %d' %ui))

for ui in range(enc_num_unrollings):
    enc_train_inputs.append(tf.placeholder(tf.float32,
        shape=[batch_size,input_size],
        name='train_inputs_ %d' %ui))
```

Чтобы инициализировать веса как ячеек LSTM, так и слоев softmax, мы будем использовать инициализацию Ксавье, предложенную Ксавье Глоро и Йошуа Бенжи в их статье[1] «Проблемы обучения глубоких нейронных сетей с прямым распространением». Это научно обоснованная методика инициализации, предназначенная для смягчения проблемы исчезающего градиента в очень глубоких сетях и реализованная через инициализатор переменной `tf.contrib.layers.xavier_initializer()`, предоставленный в TensorFlow. В частности, при инициализации Ксавье веса j-го слоя нейронной сети инициализируются в соответствии с равномерным распределением $U[a,b]$, где a – минимальное, а b – максимальное значение:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right].$$

Здесь n_j – размер j-го слоя.

Сквозные вычисления

Закончив с объявлением переменных и заполнителей ввода/вывода, мы перейдем к объявлению последовательных вычислений от кодера к декодеру и функции потерь.

Сначала мы вычислим состояние ячейки LSTM и скрытое состояние для всех слов в данном пакете предложений. Это достигается запуском цикла `for`, где в i-й итерации мы вводим i-й заполнитель `enc_train_inputs`, а также состояние ячейки и скрытое состояние вывода из i-й итерации. Функция `enc_lstm_cell` работает аналогично функции `lstm_cell`, которую мы видели в главах 8 и 9:

```
# Итеративно обновляем выход и состояние кодера
for i in enc_train_inputs:
    output, state = enc_lstm_cell(i, output,state)
```

Далее мы аналогичным образом вычисляем выход декодера для всего целевого предложения. Однако, чтобы сделать это, мы должны дождаться окончания вычислений, показанных в предыдущем фрагменте кода, для получения вектора v для инициализации состояний декодера. Ожидание достигается с помощью оператора `tf.control_dependencies(...)`. Таким образом, вложенные команды в операторе `with` будут выполняться только после полного вычисления выходных данных кодера:

```
# Как только вычисление enc_lstm_cell будет завершено,
# вычисляем выход и состояние декодера.
with tf.control_dependencies([saved_output.assign(output),
                              saved_state.assign(state)]):
    # Итеративно обновляем выход и состояние кодера
    for i in dec_train_inputs:
        output, state = dec_lstm_cell(i, output, state)
        outputs.append(output)
```

[1] Understanding the difficulty of training deep feedforward neural networks. Glorot and Bengio, Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, 2010.

Затем, после вычисления выходных данных декодера, мы рассчитаем логиты слоя softmax, используя скрытое состояние LSTM в качестве входных данных:

```
# Вычисляем логиты декодера для всех временных шагов
logits = tf.matmul(tf.concat(axis=0, values=outputs), w) + b
```

Теперь, рассчитав логиты, мы можем вычислить потери. Обратите внимание, что мы используем маску для отбрасывания элементов, которые не должны вносить свой вклад в ошибку. В данном случае это токены </s>, которые были добавлены при обработке, чтобы получить предложение фиксированной длины:

```
loss_batch = tf.concat(axis=0,values=dec_train_masks)*
             tf.nn.softmax_cross_entropy_with_logits_v2(
                 logits=logits, labels=tf.concat(axis=0,
                 values=dec_train_labels))
loss = tf.reduce_mean(loss_batch)
```

Далее, в отличие от предыдущих глав, мы будем использовать *два* оптимизатора – Adam и стандартный стохастический градиентный спуск. Это связано с тем, что использование оптимизатора Adam в долгосрочной перспективе дало нежелательные результаты, например внезапные большие колебания показателя BLEU. Мы также применим отсечение градиента, чтобы избежать градиентного взрыва.

```
# Используем два оптимизатора: Adam и стандартный градиентный спуск (SGD)
# использование оптимизатора Adam в долгосрочной перспективе дает
# нежелательные результаты, такие как большие колебания оценки BLEU.
# Сначала мы используем Adam как хорошее начало оптимизации,
# а затем переключаемся на SGD.
with tf.variable_scope('Adam'):
    optimizer = tf.train.AdamOptimizer(learning_rate)
with tf.variable_scope('SGD'):
    sgd_optimizer = tf.train.GradientDescentOptimizer(sgd_learning_
rate)

# Вычисляем градиенты с отсечением для Adam
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimize = optimizer.apply_gradients(zip(gradients, v))

# Вычисляем градиенты с отсечением для SGD
sgd_gradients, v = zip(*sgd_optimizer.compute_gradients(loss))
sgd_gradients, _ = tf.clip_by_global_norm(sgd_gradients, 5.0)
sgd_optimize = optimizer.apply_gradients(zip(sgd_gradients, v))
```

Мы будем использовать следующую инструкцию, чтобы гарантировать, что градиент правильно течет от декодера к кодеру, следя за тем, чтобы градиент существовал для всех обучаемых переменных:

```
for (g_i,v_i) in zip(gradients,v):
    assert g_i is not None, 'Gradient none for  %s' %(v_i.name)
```

Обратите внимание, что выполнение кода NMT будет происходить намного медленнее по сравнению с предыдущими упражнениями, и на одном графическом процессоре может потребоваться более 12 часов для завершения работы.

Примеры результатов перевода

После 10 тыс. циклов обучения были получены следующие результаты:

```
DE: &#124; Ferienwohnungen 1 Zi &#124; Ferienhäuser &#124; Landhäuser
&#124; Autovermietung &#124; Last Minute Angebote ! !

EN (TRUE):&#124; 1 Bedroom Apts &#124; Holiday houses &#124; Rural
Homes &#124; Car Rental &#124; Last Minute Offers !

EN (Predicted): Casino Tropez &#124; Club &#124; Club &#124;
Aparthotels Hotels &#124; Club &#124; Last Minute Offers &#124; Last
Minute Offers &#124; Last Minute Offers &#124; Last Minute Offers
&#124; Last Minute Offers ! </s>

DE: Wie hilfreich finden Sie die Demo ##AT##-##AT## CD ?

EN (TRUE): How helpful do you find the demo CD ##AT##-##AT## ROM ?

EN (Predicted): How to install the new version of XLSTAT ? </s>

DE: Das „ Ladino di Fassa " ist jedoch mehr als ein Dialekt – es ist
eine richtige Sprache .

EN (TRUE):This is Ladin from Fassa which is more than a dialect : it
is a language in its own right .

EN (Predicted): The <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk>
<unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk>
<unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk>
<unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk>
<unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk>
<unk> <unk> <unk> <unk> <unk> <unk> <unk>

DE: In der Hotelbeschreibung im Internet müßte die Zufahrt beschrieben
werden .

EN (TRUE): There are no adverse comments about this hotel at all .

EN (Predicted): The <unk> <unk> is a bit of the <unk> <unk> . </s>
```

Меткой `TRUE` обозначен эталонный перевод, а меткой `Predicted` – перевод, сгенерированный нашей моделью. Мы видим, что первое предложение распознается довольно хорошо[1]. Однако второе предложение очень плохо переведено.

А вот результаты, полученные после 100 тыс. шагов:

```
DE: Das Hotel Opera befindet sich in der Nähe des Royal Theatre ,
Kongens Nytorv , &apos; Stroget &apos; und Nyhavn .

EN (TRUE): Hotel Opera is situated near The Royal Theatre , Kongens
Nytorv , &quot; Strøget &quot; and fascinating Nyhavn .
```

---

[1] Строго говоря, в этом блоке примеров даже лучший перевод почти не имеет общего с оригиналом. Слишком маленький набор обучающих данных, слишком мало циклов обучения. Впрочем, ситуация улучшается с увеличением длительности обучения. – *Прим. перев.*

```
EN (Predicted): Best Western Hotel <unk> <unk> , <unk> , <unk> ,
<unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk>
, <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> ,
<unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> ,
```

DE: Alle älteren Kinder oder Erwachsene zahlen EUR 32,00 pro Übernachtung und Person für Zustellbetten .

```
EN (TRUE):All older children or adults are charged EUR 32.00 per night
and person for extra beds .

EN (Predicted): All older children or adults are charged EUR 15 <unk>
per night and person for extra beds . </s>
```

DE: Im Allgemeinen basieren sie auf Datenbanken , Templates und Skripts .

```
EN (TRUE):In general they are based on databases , template and
scripts .

EN (Predicted): The user is the most important software of the
software . </s>
```

DE: Tux Racer wird Ihnen helfen , die Zeit totzuschlagen und sie können OpenOffice zum Arbeiten verwenden .

```
EN (TRUE): Tux Racer will help you pass the time while you wait ,
and you can use OpenOffice for work .

EN (Predicted): <unk> .com we have a very friendly and helpful
staff . </s>
```

Мы видим, что хотя переводы не идеальны, в большинстве случаев они уже лучше отражают контекст исходного предложения, и наша система NMT достаточно хорошо создает грамматически правильные предложения.

На рис. 10.12 показано изменение оценки BLEU. С течением времени наблюдается явное увеличение показателя BLEU как для обучающих, так и для тестовых наборов данных.

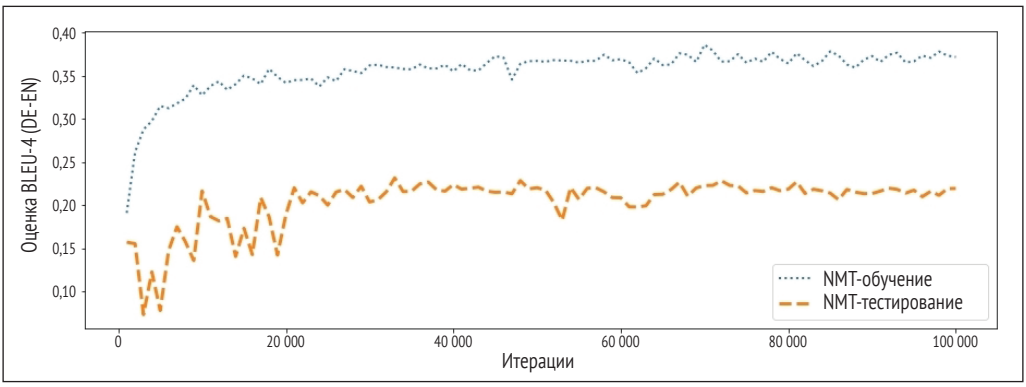


Рис. 10.12 ❖ Изменение оценки BLEU со временем

Обучение NMT одновременно с изучением представлений слов

Здесь мы обсудим, как можем обучить NMT одновременно с изучением представлений слов. В этом разделе мы рассмотрим две концепции:

- обучение NMT одновременно с изучением представлений слов;
- использование предварительно изученных представлений вместо случайной инициализации слоя представлений.

Нам доступно несколько многоязычных репозиториев для представлений слов, например:

- fastText Facebook: **https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md**;
- многоязычные представления CMU: **http://www.cs.cmu.edu/~afm/projects/multilingual_embeddings.html**.

Мы будем использовать представления CMU (~200 МБ), так как они намного меньше по сравнению с fastText (~5 ГБ). Сначала нам нужно загрузить немецкие (`multilingual_embeddings.de`) и английские (`multilingual_embeddings.en`) представления. Полный код доступен в качестве упражнения в файле `nmt_with_pretrained_wordvecs.ipynb` в папке `ch10`.

Максимизация совпадений между словарем набора данных и предварительно подготовленными представлениями

Перед нами стоит задача получить подмножество предварительно подготовленных представлений слов, которые имеют отношение к проблеме, которую мы хотим решить. Это важно, так как словарь представлений может быть большим и содержать много «лишних» слов, которых нет в словаре набора данных. Файл предварительно изученных представлений слов содержит набор строк, где начало строки – это слово и за ним через пробел следует векторное представление. Пример строки может выглядеть так:

```
дверь 0.283259492301 0.198089365764 0.335635845187 -0.385702777914 0.491404970211 …
```

Один очевидный и примитивный способ отбора слов – построчно пробежаться по файлу предварительно подготовленных представлений, и если слово в текущей строке совпадает с каким-либо словом в словаре обучающего набора данных, мы сохраним это представление слова для использования в будущем. Тем не менее это будет крайне неэффективно, так как обычно формат словарного запаса зависит от решений, принимаемых создателем словаря. Например, некоторые могут считать слова «кошкин», «кошка» и «Кошка» одним и тем же словом, тогда как другие могут рассматривать их как отдельные слова. Если мы буквально сопоставим словарь предварительно подготовленных представлений и словарь набора данных, то можем пропустить много слов. Поэтому для извлечения максимума пользы из предварительно подготовленных векторов слов мы будем опираться на специальную логику.

Сначала определим два массива NumPy для хранения соответствующих представлений слов как для исходного, так и для целевого языка:

```
de_embeddings = np.random.uniform(size=(vocabulary_size, embeddings_
size),low=-1.0, high=1.0)
en_embeddings = np.random.uniform(size=(vocabulary_size, embeddings_
size),low=-1.0, high=1.0)
```

Затем откроем текстовый файл, содержащий векторы слов, как показано здесь. Параметр `filename` – `multilingual_embeddings.de` для немецкого языка и `miltilingual_embeddings.en` для английского:

```
with open(filename,'r',encoding='utf-8') as f:
```

Далее мы разделим слово и вектор слова по первому пробелу:

```
line_tokens = line.split(' ')
lword = line_tokens[0]
vector = [float(v) for v in line_tokens[1:]]
```

Проигнорируем пустые слова, т. е. состоящие только из пробелов, символов табуляции или новой строки:

```
if len(lword.strip())==0:
continue
```

Удалим все акценты[1], присутствующие в словах (особенно в немецких словах), чтобы обеспечить максимальные шансы на совпадение слов:

```
lword = unidecode.unidecode(lword)
```

После этого воспользуемся следующей логикой для проверки совпадений. Напишем набор каскадных условий для проверки совпадений как для исходного, так и для целевого языка:

1) сначала проверяем, находится ли слово из предварительно подготовленных представлений (`lword`) в словаре набора данных, как оно есть;
2) если нет, проверяем, не является ли первая буква заглавной;
3) если слово по-прежнему не обнаружено в словаре, пробуем удалить специальные символы (апострофы, умляуты, значки ударений).

Обнаружив слово из списка готовых представлений в словаре обучающих данных, мы берем вектор представления этого слова и связываем его со строкой, помеченной индексом этого слова (word → ID). Мы сделаем это для обоих языков:

```
# Обновляем случайно инициализированную
# матрицу представлений.
# Обновляем количество слов,
# совпадающих с подготовленными представлениями.
try:
    dword = dictionary[lword]
    words_found_ids.append(dictionary[lword])
    embeddings[dictionary[lword],:] = vector
    words_found += 1

# Если данное слово не найдено в нашем словаре
except KeyError:
    try:
```

[1] Символы со значком правого и левого ударений.

```
        # Сначала пробуем найти это же слово,
        # но с заглавной буквы.
        if len(lword)>0:
            firt_letter_cap = lword[0].upper()+lword[1:]
        else:
            continue

        # Обновляем матрицу представлений слов
        dword = dictionary[firt_letter_cap]
        words_found_ids.append(dictionary[
                                firt_letter_cap])
        embeddings[dictionary[firt_letter_cap],:] = vector
        words_found += 1

    except KeyError:
        # Если совпадение не найдено, пробуем найти слово
        # без специальных символов акцентирования
        try:
            dword = unaccented_dict[lword]
            words_found_ids.append(dictionary[lword])
            embeddings[dictionary[lword],:] = vector
            words_found += 1
        except KeyError:

            continue
```

Объявление слоя представлений как переменной TensorFlow

Объявим две обучаемые переменные TensorFlow для слоев представлений – `tgt_word_embeddings` и `src_word_embeddings` – следующим образом:

```
tgt_word_embeddings = tf.get_variable(
    'target_embeddings',shape=[vocabulary_size,
        embeddings_size],
    dtype=tf.float32, initializer = tf.constant_initializer(
        en_embeddings)
)
src_word_embeddings = tf.get_variable(
    'source_embeddings',shape=[vocabulary_size,
        embeddings_size],
    dtype=tf.float32, initializer = tf.constant_initializer(
        de_embeddings)
)
```

Затем изменим размерность заполнителей в `dec_train_inputs` и `enc_train_inputs` на `[batch_size]`, а тип данных на `tf.int32`. Это делается для того, чтобы мы могли использовать их для выполнения поиска представлений `tf.nn.embedding_lookup(...)` для каждого развернутого ввода:

```
# Объявляем развернутые обучающие входы, а также перебор представлений(кодер).
for ui in range(enc_num_unrollings):
    enc_train_inputs.append(tf.placeholder(tf.int32,
```

```
                            shape=[batch_size],
                            name='train_inputs_ %d' %ui))
    enc_train_input_embeds.append(tf.nn.embedding_lookup(
                                  src_word_embeddings,
                                  enc_train_inputs[ui]))

# Объявляем развернутые обучающие входы, представления,
# выходы и маски (декодер).
for ui in range(dec_num_unrollings): dec_train_inputs.append(tf.placeholder(tf.int32,
                            shape=[batch_size],
                            name='dec_train_inputs_ %d' %ui))
    dec_train_input_embeds.append(tf.nn.embedding_lookup(
                                  tgt_word_embeddings,
                                  dec_train_inputs[ui]))
    dec_train_labels.append(tf.placeholder(tf.float32,
                            shape=[batch_size,vocabulary_size],
                            name = 'dec_train_labels_ %d' %ui))
    dec_train_masks.append(tf.placeholder(tf.float32,
                           shape=[batch_size,1],
                           name='dec_train_masks_ %d' %ui))
```

Затем вычисления ячеек LSTM для кодера и декодера изменяются, как показано далее. В этой части кода мы сначала вычисляем выход кодирующей ячейки LSTM с входными данными исходного предложения. Потом, используя информацию о конечном состоянии от кодера в качестве инициализации для декодера (т. е. используя `tf.control_dependencies(...)`), мы вычисляем выход декодера, а также логиты и прогнозы softmax:

```
# Итеративно обновляем выход и состояние кодера
for i in enc_train_inputs:
    output, state = enc_lstm_cell(i, output,state)

print('Calculating Decoder Output')
# Как только вычисление enc_lstm_cell завершено,
# вычисляем выход и состояние декодера
with tf.control_dependencies([saved_output.assign(output),
                              saved_state.assign(state)]):
    # Итеративно вычисляем состояние и выход декодера
    for i in dec_train_inputs:
        output, state = dec_lstm_cell(i, output, state)
        outputs.append(output)
```

Обратите внимание, что файл упражнения содержит немного иной расчет выходных данных, чем показано здесь. Вместо того чтобы вводить предыдущее предсказание в качестве входных данных, мы вводим истинное слово. Этот подход обеспечивает лучшую точность, чем ввод предыдущего прогноза, и будет подробно рассмотрен ниже. Однако общая идея остается прежней.

Заключительные шаги включают вычисление ошибки декодера и объявление оптимизатора для оптимизации параметров модели, как мы уже делали выше.

Вы можете ознакомиться с упрощенной визуализацией графа вычислений нашей системы NMT, представленной на рис. 10.13.

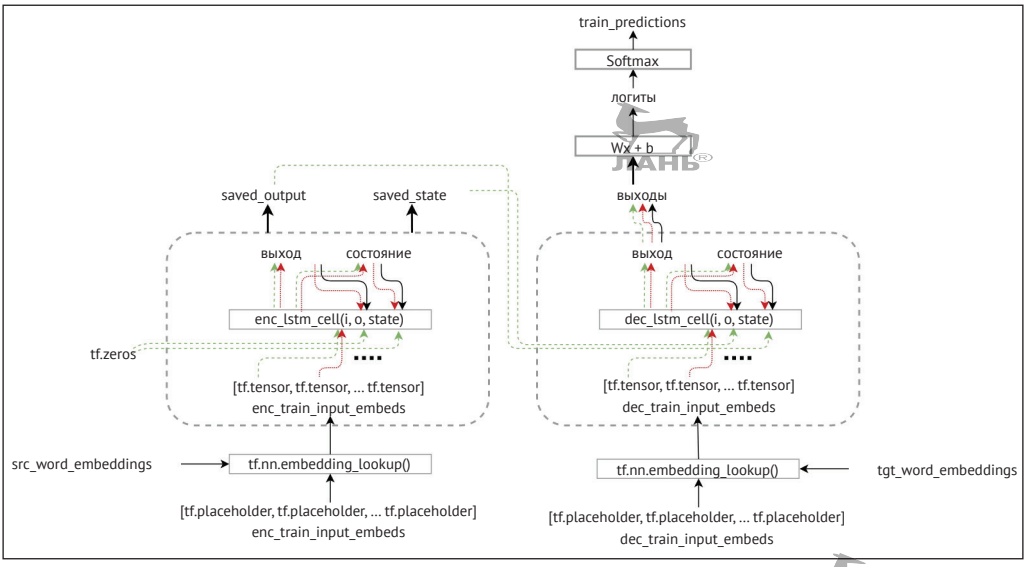


Рис. 10.13 ❖ Граф вычислений системы машинного перевода с заранее подготовленными представлениями слов

Совершенствование NMT

Как вы можете судить по предыдущим результатам, наша модель перевода не ведет себя идеально. Эти результаты были получены при выполнении оптимизации более 12 часов на одном графическом процессоре NVIDIA 1080 Ti. Также обратите внимание, что это даже не полный набор данных, мы использовали только 250 тыс. пар предложений для обучения. Однако если вы что-то вводите в Google Translate, который использует систему Google Neural Machine Translation (GNMT), перевод почти всегда выглядит очень реалистично и содержит лишь небольшие ошибки. Поэтому важно знать, как мы можем улучшить качество модели. В этом разделе обсудим несколько способов улучшения NMT, таких как помощь наставника, глубокие LSTM и механизм внимания.

Помощь наставника

Как мы обсуждали выше, обучение NMT происходит следующим образом:

- сначала мы подаем полное предложение на кодер, чтобы получить его конечное выходное состояние;
- затем мы устанавливаем начальное состояние декодера равным конечному состоянию кодера;
- наконец, мы просим декодер предсказать полное целевое предложение без какой-либо дополнительной информации, кроме последнего выходного состояния кодера.

Это может оказаться слишком сложной задачей для модели. Мы можем проиллюстрировать это на примере. Допустим, учитель просит первоклассника закончить следующее предложение, учитывая только первое слово:

Я ___ ___ ___ ___ ___ ___

Это означает, что ребенок должен выбрать субъект, глагол и объект, знать синтаксис языка, понимать грамматические правила и т. д.

Следовательно, высока вероятность, что ребенок составит неправильное предложение. Тем не менее если мы попросим ребенка написать это предложение слово за словом, результат может оказаться намного лучше. Другими словами, мы просим ребенка написать только первое слово:

Я ___

Затем мы просим его заполнить прочерк с учетом предыдущего слова:

Я люблю ___

И продолжаем в том же духе:

Я люблю ___
Я люблю играть ___
Я люблю играть в футбол ___

Таким образом, ребенок может пошагово составить правильное и осмысленное предложение. Этот прием известен как *помощь наставника* (teacher forcing). Мы можем применить подобный подход, чтобы уменьшить сложность задачи перевода, как показано на рис. 10.14.

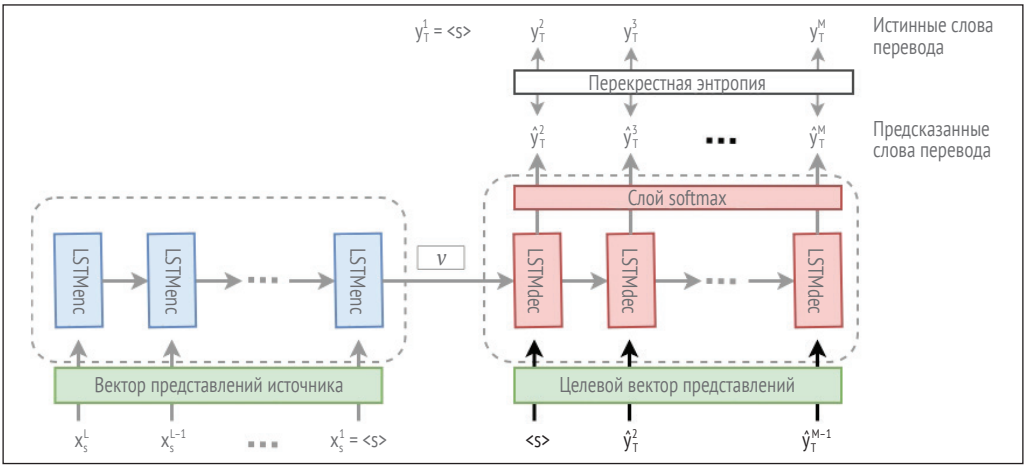


Рис. 10.14 ❖ Помощь наставника.
Более темные стрелки на входах изображают новые входные соединения с декодером.
На рисунке справа показано, как изменяется ячейка LSTM-декодера

Как показано черными стрелками на рис. 10.14, входные данные для декодера были заменены фактическими целевыми словами из обучающих данных. Поэтому декодер больше не должен нести бремя прогнозирования целевого предложения с учетом исходного предложения. Скорее, декодер должен только правильно

предсказать текущее слово, учитывая предыдущее слово. Стоит отметить, что в предыдущих разделах мы обсуждали процедуру обучения без упоминания помощи наставника. Тем не менее мы фактически используем этот прием во всех упражнениях текущей главы.

Глубокие LSTM

Одним из очевидных улучшений, которое мы можем сделать, является увеличение количества слоев путем размещения LSTM друг на друге, создавая тем самым глубокую LSTM-сеть (рис. 10.15). Например, система машинного перевода Google использует восемь слоев LSTM, наложенных друг на друга[1]. Хотя многослойность снижает вычислительную эффективность, наличие большего количества слоев значительно улучшает способность нейронной сети изучать синтаксис и другие лингвистические характеристики двух языков.

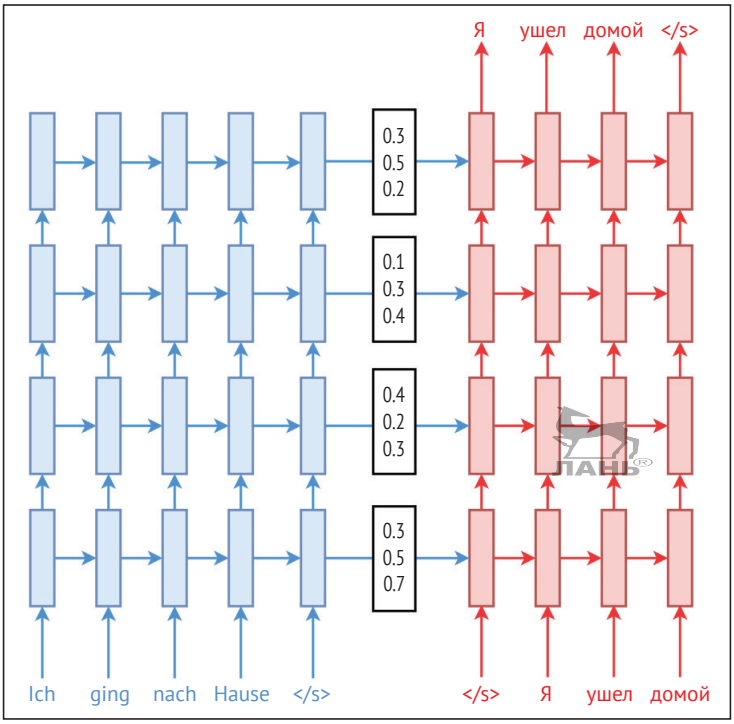


Рис. 10.15 ❖ Иллюстрация глубокой LSTM-сети

Механизм внимания

Механизм внимания (attention mechanism) является одним из ключевых достижений в машинном переводе. Внимание позволяет декодеру получать доступ

[1] Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. Wu and others, Technical Report (2016).

к полной истории состояний кодера, что помогает более качественно и развернуто воссоздать смысл исходного предложения в переводе. Прежде чем углубляться в детали механизма внимания, давайте разберемся с одним из критических узких мест в нашей нынешней системе NMT и преимуществами, которые дает внимание.

Узкое место: вектор контекста

Как вы, наверное, уже догадались, узким местом является вектор контекста, или, как его еще называют, *мыслительный вектор*, который находится между кодером и декодером (рис. 10.16).

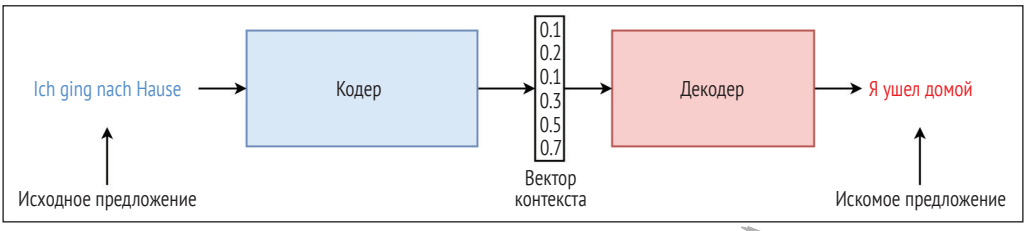


Рис. 10.16 ❖ Архитектура кодер–декодер

Чтобы понять, почему это узкое место, давайте представим перевод следующего английского предложения:

I went to the flower market to buy some flowers
(*Я пошел на цветочный рынок, чтобы купить цветы*)

На немецком языке это предложение выглядит так:

Ich ging zum Blumenmarkt, um Blumen zu kaufen

Если мы хотим сжать исходное предложение в вектор фиксированной длины, он должен содержать следующую информацию:

- информация о субъекте (*Я*);
- информация о глаголах (*пошел* и *купил*);
- информация об объектах (*цветы* и *цветочный рынок*);
- взаимодействие субъекта, глаголов и объектов друг с другом в предложении.

Обычно вектор контекста имеет размер 128 или 256 элементов. Втиснуть столько информации в ограниченный объем – это очень непрактичное и чрезвычайно сложное требование для системы. Поэтому в большинстве случаев контекстный вектор не может предоставить полную информацию, необходимую для хорошего перевода. В свою очередь, декодер не может сформировать однозначный и правильный перевод.

Кроме того, во время декодирования вектор контекста наблюдается только в начале. После этого декодер LSTM должен запомнить контекстный вектор до конца процесса перевода. Хотя LSTM-сети хороши для долговременного запоми-

нания, они не могут преодолеть ограничения, налагаемые вектором контекста. Это сильно повлияет на результаты, особенно в случае длинных предложений.

Благодаря механизму внимания декодер будет иметь доступ к полной истории состояний кодера для каждого временного шага декодирования. Это позволяет декодеру получить доступ к очень богатому представлению исходного предложения. Кроме того, механизм внимания вводит слой softmax, позволяющий декодеру вычислять средневзвешенное значение последних наблюдаемых состояний кодера и затем использовать его в качестве вектора контекста для декодера. Иными словами, декодер может уделять разное количество внимания различным словам на разных этапах декодирования.

Механизм внимания в деталях

Теперь давайте подробно рассмотрим фактическую реализацию механизма внимания. Мы будем использовать алгоритм, подробно описанный в статье[1] «Нейронный машинный перевод путем обучения совместному взаимодействию». Для согласованности со статьей будем использовать следующие обозначения:

- скрытое состояние кодировщика: h_i;
- слова целевого предложения: y_i;
- скрытое состояние декодера: s_i;
- контекстный вектор: c_i.

До сих пор наш декодер LSTM состоял из ввода y_i и скрытого состояния s_{i-1}. Мы будем игнорировать состояние ячейки, так как это внутренняя часть LSTM. Декодер можно представить следующим образом:

$$LSTM_{dec} = f(y_i, s_{i-1}).$$

Здесь f представляет действительные правила обновления, используемые для вычисления $y_i + 1$ и s_i. С помощью механизма внимания мы вводим новый зависящий от времени вектор контекста c_i для i-го этапа декодирования. Вектор c_i представляет собой средневзвешенное значение скрытых состояний всех развернутых шагов кодера. Более высокий вес будет придан j-му скрытому состоянию кодера, если j-е слово более важно для перевода i-го слова на целевой язык. Теперь декодер LSTM приобретает форму:

$$LSTM_{dec} = f(y_i, s_{i-1}, c_i).$$

Концептуально механизм внимания можно представить в виде отдельного слоя и проиллюстрировать его, как показано на рис. 10.17. Слой внимания отвечает за создание c_i для i-го временного шага процесса декодирования.

[1] Neural Machine Translation by Learning to Jointly Align and Translate. Bahdanau, Cho and Bengio, arXiv:1409.0473 (2014).

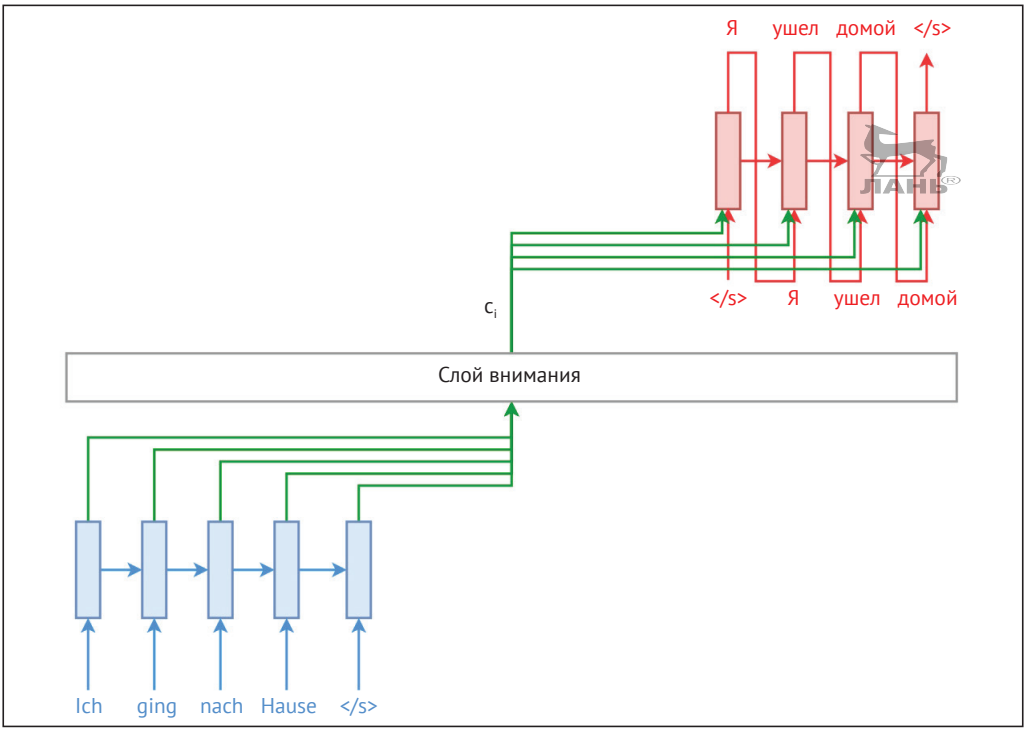


Рис. 10.17 ❖ Механизм внимания NMT в общем виде

Вектор c_i вычисляется следующим образом:

$$c_i = \sum_{j=1}^{L} \alpha_{ij} h_j .$$

Здесь L – количество слов в исходном предложении, а α_{ij} – нормализованный вес, представляющий важность скрытого состояния j-го кодера для вычисления предсказания i-го декодера. Он рассчитывается с использованием слоя softmax. L – длина предложения кодера:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{L} \exp(e_{ik})} .$$

Здесь e_{ij} – это *энергия* или *важность*, означающая, насколько j-е скрытое состояние кодера и предыдущее состояние декодера s_{i-1} способствуют вычислению s_i:

$$e_{ij} = v_a^T tanh(W_a s_{i-1} + U_a h_j).$$

По сути, это означает, что e_{ij} рассчитывается с помощью многослойного персептрона, веса которого равны v_a, W_a и U_a, а s_{i-1} и h_j – входы в сеть. Механизм внимания показан на рис. 10.18.

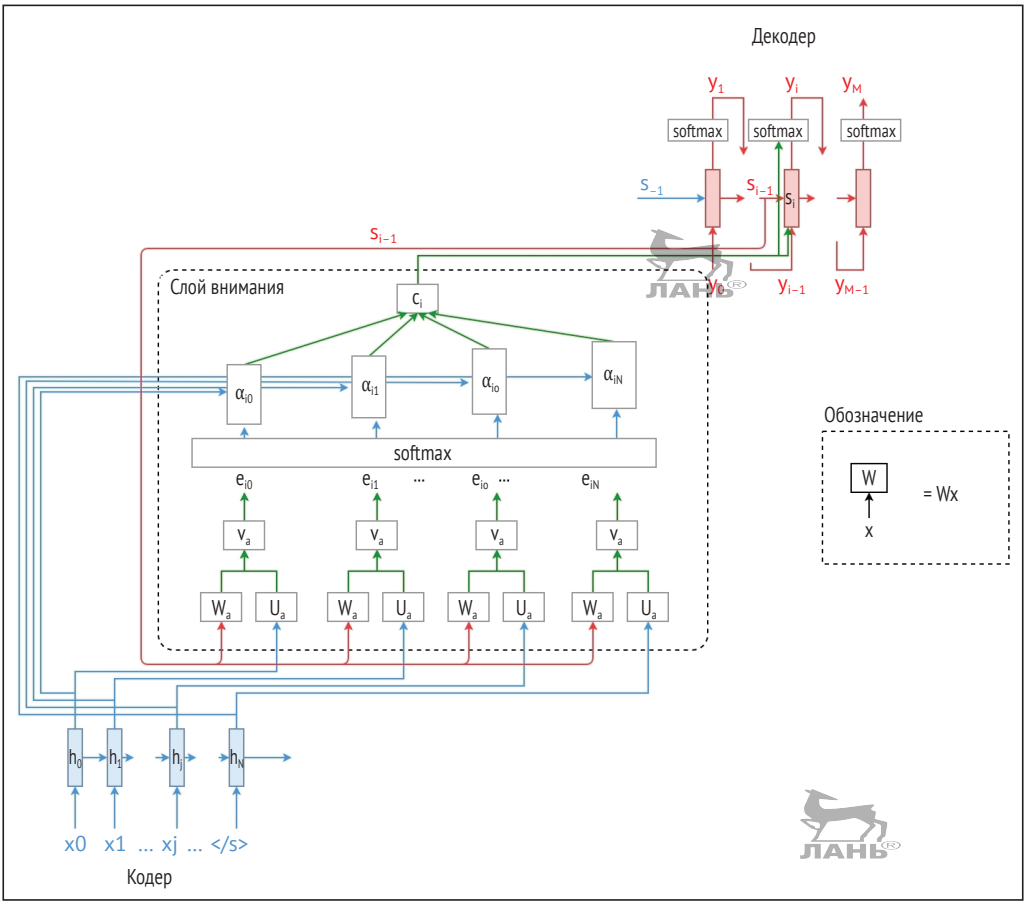


Рис. 10.18 ❖ Механизм внимания

Реализация механизма внимания

Обрисуем реализацию механизма внимания в общих чертах. Система претерпит три основных изменения:

- введено больше параметров (т. е. весов) для расчета вектора внимания и использования его в качестве входных данных для ячейки LSTM-декодера;
- представлена новая функция для вычислений, связанных со слоем внимания `attn_layer`;
- внесены изменения в вычисления ячейки LSTM-декодера для получения *взвешенной по вниманию* (attention-weighted) суммы всех выходов ячейки LSTM-кодера в качестве входа.

Далее мы будем обсуждать только дополнения к стандартной модели NMT. Вы можете найти полное упражнение для NMT со вниманием в файле `neural_machine_translation_attention.ipynb`.

Добавление весов

Для реализации механизма внимания вводятся три новых набора весов. Все эти веса используются для вычисления *энергетического компонента* (energy term) e_{ij}, который мы упоминали выше:

```
W_a = tf.Variable(tf.truncated_normal([num_nodes,num_nodes],
    stddev=0.05),name='W_a')
U_a = tf.Variable(tf.truncated_normal([num_nodes,num_nodes],
    stddev=0.05),name='U_a')
v_a = tf.Variable(tf.truncated_normal([num_nodes,1],
    stddev=0.05),name='v_a')
```

Кроме того, мы объявим новый набор весов, который будет использован для получения c_i в качестве входных данных для i-го этапа развертывания декодера:

```
dec_ic = tf.get_variable('ic',shape=[num_nodes, num_nodes],
    initializer = tf.contrib.layers.xavier_initializer())
dec_fc = tf.get_variable('fc',shape=[num_nodes, num_nodes],
    initializer = tf.contrib.layers.xavier_initializer())
dec_cc = tf.get_variable('cc',shape=[num_nodes, num_nodes],
    initializer = tf.contrib.layers.xavier_initializer())
dec_oc = tf.get_variable('oc',shape=[num_nodes, num_nodes],
    initializer = tf.contrib.layers.xavier_initializer())
```

Вычисление внимания

Для вычисления вектора внимания для каждой позиции кодера и декодера мы определим метод `attn_layer(...)`. Он вычисляет внимание для всех позиций (т. е. в количестве `num_enc_unrollings`) кодера для одного этапа развертывания декодера. Метод `attn_layer(...)` получает два аргумента:

```
attn_layer(h_j_unrolled, s_i_minus_1)
```

Метод нуждается в следующих аргументах:

- `h_i_unrolled` – выходы ячейки LSTM-кодера в количестве `num_enc_unrolling`, рассчитанные при подаче исходного предложения в кодер. Это список тензоров `num_enc_unrolling`, где каждый элемент имеет размер `[batch_size, num_nodes]`;
- `s_i_minus_1` – предыдущий выход ячейки LSTM-декодера. Это тензор размером `[batch_size, num_nodes]`.

Сначала создаем одиночный тензор со списком развернутых выходов кодера размером `[num_enc_unrollings * batch_size, num_nodes]`:

```
enc_logits = tf.concat(axis=0,values=h_j_unrolled)
```

Затем вычисляем $W_a s_{i-1}$ с помощью следующей операции:

```
# размерность [enc_num_unroll x batch_size, num_nodes]
w_a_mul_s_i_minus_1 = tf.matmul(enc_outputs,W_a)
```

Далее вычисляем $U_a h_j$:

```
# размерность [enc_num_unroll x batch_size, num_nodes]
u_a_mul_h_j = tf.matmul(tf.tile(s_i_minus_1,[enc_num_
unrollings,1]), U_a)
```

Теперь рассчитаем энергию как $e_{ij} = v_a^T tanh(W_a s_{i-1} + U_a h_j)$. Это тензор размера [enc_num_unroll * batch_size, 1]:

```
e_j = tf.matmul(tf.nn.tanh(w_a_mul_s_i_minus_1 +
    u_a_mul_h_j),v_a)
```

Далее можем сначала разбить большой e_j на длинный список тензоров enc_num_unrolling с помощью tf.split(...), где каждый тензор имеет размер [batch_size, 1]. После этого мы объединяем этот список вдоль оси 1, чтобы получить тензор размера [batch_size, enc_num_unrollings] (т. е. reshaped_e_j). Поэтому одна строка reshaped_e_j будет соответствовать значениям внимания для всех позиций развернутых во времени тактов кодера:

```
# список элементов enc_num_unroll,
# каждый элемент имеет размер [batch_size, 1].
batched_e_j = tf.split(axis=0,
    num_or_size_splits=enc_num_unrollings,value=e_j)
# размерность [batch_size, enc_num_unroll]
reshaped_e_j = tf.concat(axis=1,values=batched_e_j)
```

Теперь мы можем легко вычислить нормализованные значения внимания для reshaped_e_j. Значения будут нормализованы по развернутым во времени тактам (ось 1 reshaped_e_j):

```
# размерность [batch_size, enc_num_unroll]
alpha_i = tf.nn.softmax(reshaped_e_j)
```

Затем следует разбить alpha_i на список тензоров в количестве enc_num_unroll, каждый из которых имеет размер [batch_size, 1]:

```
alpha_i_list = tf.unstack(alpha_i,axis=1)
```

После этого мы вычислим взвешенную сумму каждого из выходов кодера (то есть h_j_unrolled) и назначим ее для c_i, который будет использоваться в качестве входных данных для *i*-го временного шага развертывания ячейки LSTM-декодера:

```
c_i_list = [tf.reshape(alpha_i_list[e_i],
           [-1,1])*h_j_unrolled[e_i] for e_i in range(enc_num_
            unrollings)]
c_i = tf.add_n(c_i_list) # of size [batch_size, num_nodes]
```

Затем, чтобы получить c_i в качестве входных данных для *i*-го шага развертывания ячейки LSTM-декодера, вычисление ячейки LSTM изменяется следующим образом:

```
# Объявление ячейки (декодер)
def dec_lstm_cell(i, o, state, c):
"""Create a LSTM cell"""
input_gate = tf.sigmoid(tf.matmul(i, dec_ix) + tf.matmul(o, dec_
im) + tf.matmul(c, dec_ic) + dec_ib)

forget_gate = tf.sigmoid(tf.matmul(i, dec_fx) + tf.matmul(o, dec_
fm) + tf.matmul(c, dec_fc) + dec_fb)

update = tf.matmul(i, dec_cx) + tf.matmul(o, dec_cm) +
tf.matmul(c, dec_cc) +dec_cb
```

```
state = forget_gate * state + input_gate * tf.tanh(update)

output_gate = tf.sigmoid(tf.matmul(i, dec_ox) + tf.matmul(o, dec_
om) + tf.matmul(o, dec_oc) + dec_ob)

return output_gate * tf.tanh(state), state
```

Результаты работы NMT со вниманием

Так выглядят результаты, полученные после 10 тыс. шагов обучения:

```
DE: &#124; Ferienwohnungen 1 Zi &#124; Ferienhäuser &#124; Landhäuser
&#124; Autovermietung &#124; Last Minute Angebote ! !

EN (TRUE):&#124; 1 Bedroom Apts &#124; Holiday houses &#124; Rural
Homes &#124; Car Rental &#124; Last Minute Offers !

EN (Predicted): &#124; Apartments &#124; Hostels &#124; Hostels &#124;
Last Minute Offers ! </s>

DE: Wie hilfreich finden Sie die Demo ##AT##-##AT## CD ?

EN (TRUE): How helpful do you find the demo CD ##AT##-##AT## ROM ?

EN (Predicted): How can you find the XLSTAT ##AT##-##AT## MX ? </s>

DE: Das „ Ladino di Fassa " ist jedoch mehr als ein Dialekt – es ist
eine richtige Sprache .

EN (TRUE):This is Ladin from Fassa which is more than a dialect : it
is a language in its own right .

EN (Predicted): The <unk> &quot; is a very important role in the world
. </s>

DE: In der Hotelbeschreibung im Internet müßte die Zufahrt
beschrieben werden .

EN (TRUE): There are no adverse comments about this hotel at all .

EN (Predicted): The <unk> <unk> is the <unk> of the Internet . </s>
```

Подобно тому, что мы наблюдали ранее, NMT со вниманием относительно удачно переводит некоторые предложения, но плохо переводит другие.

Посмотрим теперь, как выглядит перевод после 100 тыс. шагов обучения:

```
DE: Das Hotel Opera befindet sich in der Nähe des Royal Theatre ,
Kongens Nytorv , &apos; Stroget &apos; und Nyhavn .

EN (TRUE): Hotel Opera is situated near The Royal Theatre , Kongens
Nytorv , &quot; Strøget &quot; and fascinating Nyhavn .

EN (Predicted): Best Western Hotel <unk> <unk> , <unk> , <unk> ,
<unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk>
, <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> ,
<unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> , <unk> ,

DE: Alle älteren Kinder oder Erwachsene zahlen EUR 32,00 pro
Übernachtung und Person für Zustellbetten .
```

```
EN (TRUE):All older children or adults are charged EUR 32.00 per night
and person for extra beds .

EN (Predicted): All older children or adults are charged EUR 15 <unk>
per night and person for extra beds . </s>

DE: Im Allgemeinen basieren sie auf Datenbanken , Templates und
Skripts .

EN (TRUE):In general they are based on databases , template and
scripts .

EN (Predicted): The user is the most important software of the
software . </s>

DE: Tux Racer wird Ihnen helfen , die Zeit totzuschlagen und sie
können OpenOffice zum Arbeiten verwenden .

EN (TRUE): Tux Racer will help you pass the time while you wait ,
and you can use OpenOffice for work .

EN (Predicted): <unk> .com we have a very friendly and helpful
staff . </s>
```

Для удобства сравнения мы использовали тот же набор тестовых предложений, который ранее применяли для оценки стандартной системы NMT. Мы можем видеть, что модель NMT со вниманием обеспечивает ощутимо лучшее качество перевода. Но все же встречаются ошибочные переводы, поскольку мы используем ограниченный объем данных.

Рисунок 10.19 иллюстрирует изменение оценки BLEU в сравнении между стандартной NMT и системой NMT со вниманием. Очевидно, что усовершенствованная система демонстрирует лучший результат BLEU как при обучении, так и при тестировании.

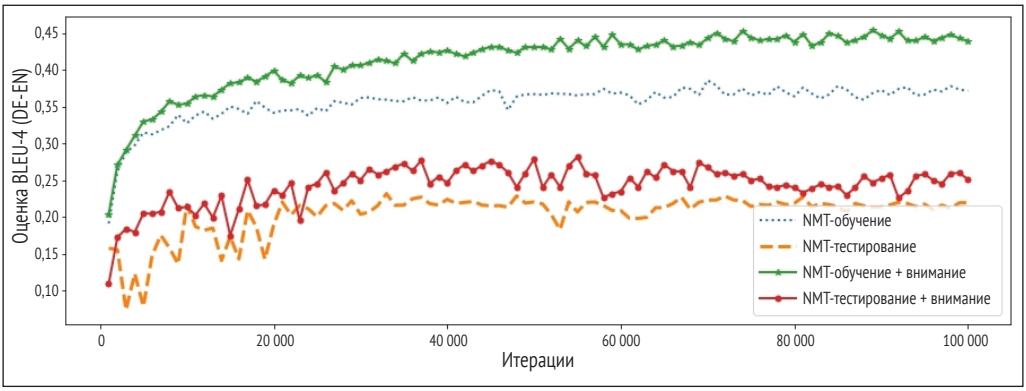


Рис. 10.19 ❖ График изменения оценки BLEU для систем NMT и NMT со вниманием

По результатам 2017 года оценка BLEU для немецко-английского перевода составила 35,1 (*The University of Edinburgh's Neural MT Systems for WMT17 by Rico Sennrich and others, arXiv preprint arXiv:1708.00726 (2017)*).

Визуализация внимания к исходным и целевым предложениям

Мы можем визуализировать уровень внимания, проявляемый системой к разным исходным словам для заданного целевого слова, причем сделать это для множества пар исходного и целевого слов в переводе (рис. 10.20). Как вы помните, при вычислении внимания нам известно значение внимания `enc_num_unrollings` для заданной позиции декодера. Следовательно, если вы объединяете все векторы внимания для всех позиций в декодере, то можете создать *матрицу внимания* (attention matrix).

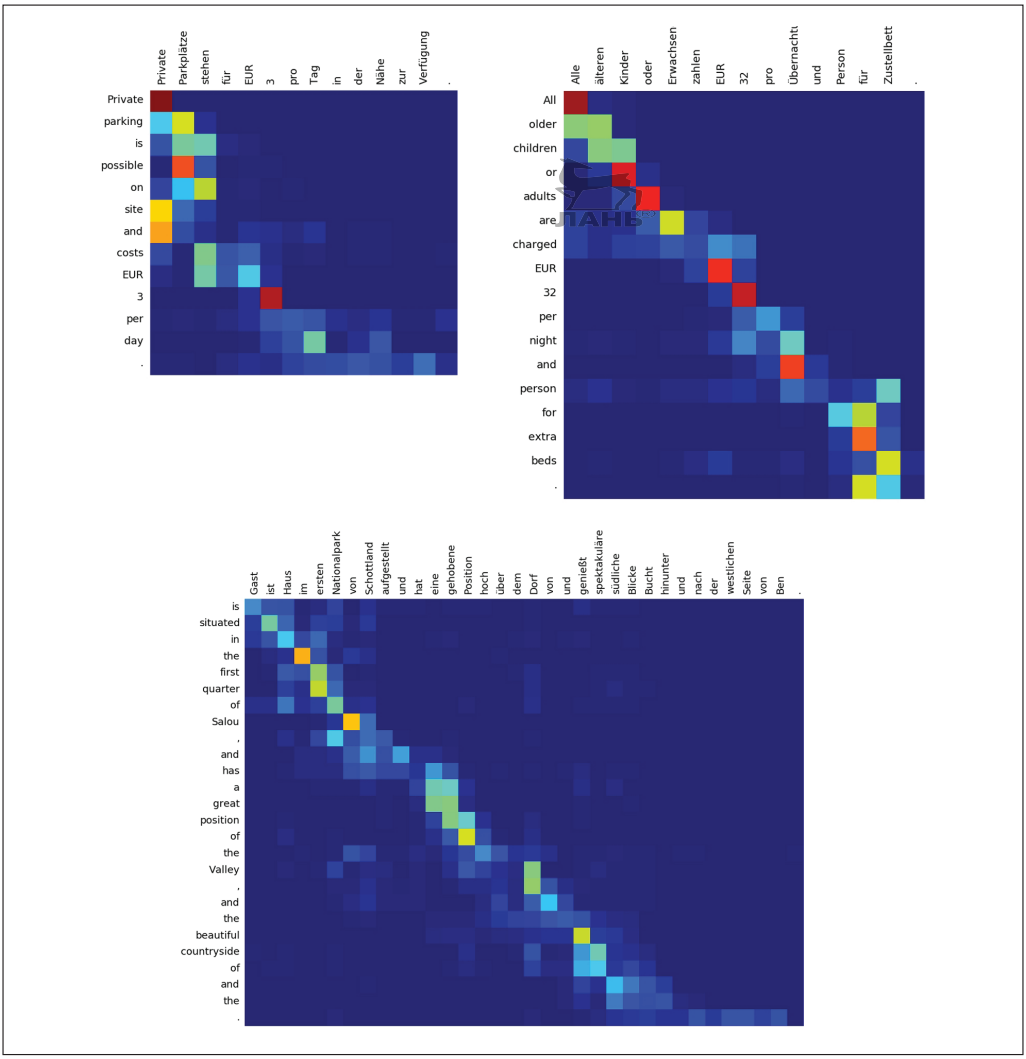


Рис. 10.20 ❖ Матрицы внимания для нескольких различных пар перевода источник–цель

В матрице внимания целевые слова расположены в виде строк, а исходные – в виде столбцов. Более высокое (более светлое) значение на пересечении некоторых строк и столбцов означает, что при прогнозировании целевого слова, найденного в этой строке, декодер в основном обращал внимание на исходное слово, заданное соответствующим столбцом. Например, вы можете видеть, что предлог «*for*» в целевом предложении сильно коррелирует с предлогом «*für*» в исходном предложении.

На этом мы завершаем обсуждение систем машинного перевода. Мы рассмотрели применяемую в NMT базовую архитектуру кодер–декодер, а также обсудили, как оценивают качество перевода. Затем вы узнали про несколько способов улучшения систем NMT, таких как помощь наставника, использование глубоких LSTM и механизм внимания.

Важно понимать, что системы NMT чрезвычайно широко применяются в реальном мире. Одним из очевидных вариантов использования является международный бизнес, имеющий филиалы во многих странах. В таких компаниях сотрудники из разных стран должны иметь быстрые способы общения, не зависящие от языкового барьера. Поэтому автоматический перевод электронной почты с одного языка на другой может быть очень полезным для бизнеса. Далее, на производстве NMT можно использовать для создания многоязычных описаний продуктов или руководств пользователя. Затем эксперты могут выполнить легкую финальную обработку, чтобы убедиться в точности переводов. Наконец, NMT может пригодиться для повседневных задач, таких как многоязычные переводы. Скажем, пользователь не является носителем английского языка и не может полностью описать на английском языке предмет поискового запроса. В этом случае пользователь может написать многоязычный поисковый запрос. Затем система NMT может перевести запрос на разные языки и найти интернет-ресурсы, которые соответствуют поисковому запросу пользователя.

Применение моделей Seq2Seq в чат-ботах

Еще одно популярное применение моделей, основанных на обработке языковых последовательностей, – создание чат-ботов, т. е. компьютерных программ, способных вести реалистичный разговор с человеком. Такие приложения очень полезны для компаний с огромной клиентской базой. Ответы на вопросы клиентов, задающих банальные вопросы, – это значительная часть работы службы поддержки. Чат-бот вполне может отвечать на стандартные вопросы пользователей. Если чат-бот не может ответить на вопрос, обращение перенаправляется оператору-человеку. Чат-боты экономят много времени и позволяют операторам службы поддержки выполнять более сложные задачи.

Обучение чат-бота

Итак, можем ли мы использовать модель типа «последовательность-в-последовательность» для обучения чат-бота? Ответ очевиден, поскольку вы уже знаете о модели машинного перевода. Единственная разница заключается лишь в том, как формируются пары исходных и целевых предложений.

В системе NMT пары предложений состоят из исходного предложения и эталонного перевода на целевой язык. Однако при обучении чат-бота данные извлекаются из диалога между двумя людьми. Исходными предложениями будут фразы, произнесенные человеком A, а целевыми предложениями будут ответы, сделанные человеком B. Иногда в качестве обучающих данных используют диалоги из фильмов. Один из наборов доступен по адресу **https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html**. Взгляните на пример такого диалога[1]:

```
BIANCA: They do not!
CAMERON: They do to!
BIANCA: I hope so.
CAMERON: She okay?
BIANCA: Let's go.
CAMERON: Wow
BIANCA: Okay - you're gonna need to learn how to lie.
CAMERON: No
BIANCA: I'm kidding. You know how sometimes you just become this "persona"?
And you don't know how to quit?
BIANCA: Like my fear of wearing pastels?
CAMERON: The "real you".
```

Вот ссылки на несколько других наборов данных для обучения разговорных чат-ботов:

- набор данных комментариев Reddit: **https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/**;
- набор диалогов Maluuba: **https://datasets.maluuba.com/Frames**;
- корпус диалогов Ubuntu: **http://dataset.cs.mcgill.ca/ubuntu- corpus-1.0/**;
- конкурсный набор NIPS: **http://convai.io/**;
- корпус текстов, извлеченный компанией Microsoft из социальных сетей: **https://tinyurl.com/ y7ha9rc5**.

На рис. 10.21 проиллюстрировано сходство между системами чат-ботов и NMT. Например, мы тренируем чат-бот с набором данных, состоящим из диалогов между двумя людьми. Кодер принимает предложения/фразы, произносимые одним человеком, а декодер обучается предсказывать ответ другого человека. После обучения мы можем использовать чат-бота, чтобы дать ответ на заданный вопрос.

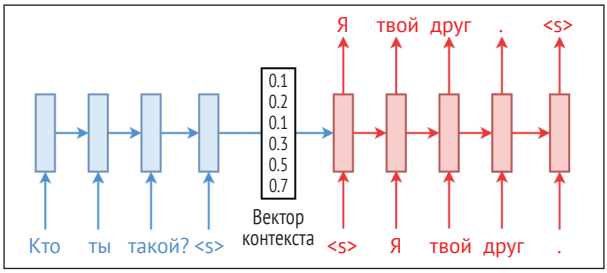


Рис. 10.21 ❖ Иллюстрация работы чат-бота

[1] Диалог из молодежной комедии 1999 г. «10 Things I Hate About You» («10 причин моей ненависти»). – *Прим. перев.*

Оценка чат-ботов – тест Тьюринга

Тест Тьюринга был изобретен Аланом Тьюрингом в 1950-х гг. как способ измерения интеллекта машины. Идея эксперимента хорошо подходит для оценки чат-ботов. Эксперимент устроен следующим образом.

Участвуют три стороны: оценщик (т. е. человек) A, другой человек B и машина C. Все трое находятся в трех разных комнатах, так что никто из них не может видеть другого. Единственным средством связи является текст, который стороны вводят в компьютер, а получатель видит текст на экране компьютера у себя в комнате. Оценщик общается как с человеком, так и с машиной. В конце разговора оценщик должен отличить машину от человека. Если оценщик не может достоверно отличить машину от человека, говорят, что машина прошла тест Тьюринга. Схема эксперимента изображена на рис. 10.22.

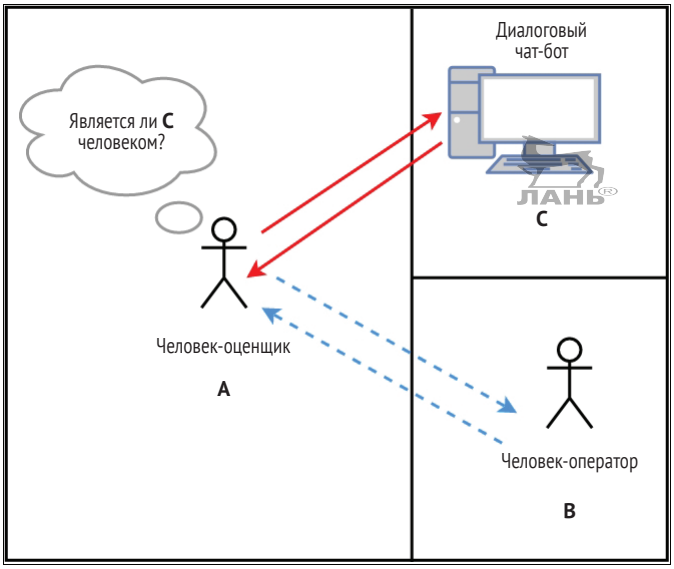


Рис. 10.22 ❖ Тест Тьюринга

Заключение

В этой главе мы подробно поговорили о системах NMT. Машинный перевод – это задача перевода определенного текстового корпуса с исходного языка на целевой. Сначала мы кратко поговорили об истории машинного перевода, чтобы у читателя возникло чувство благодарности к первым системам, из которых вырос современный нейронный машинный перевод. Мы увидели, что сегодня NMT – это самые эффективные системы перевода. Далее поговорили о фундаментальной концепции этих систем и разложили модель на слой представлений, кодер, вектор контекста и декодер. Сначала мы выявили пользу от наличия слоя представлений, поскольку он дает семантическое представление слов по сравнению с унитарным кодированием. Затем вы узнали назначение кодера, которое заключается в том,

чтобы сформировать качественный вектор фиксированной размерности, представляющий исходное предложение – вектор контекста. Потом мы использовали его для инициализации декодера. Декодер отвечает за фактический перевод исходного предложения. Затем мы обсудили, как в системах NMT работают обучение и вывод.

Далее мы разобрали фактическую реализацию системы NMT, которая переводит предложения с немецкого языка на английский, и постарались понять внутренние механизмы системы NMT. Мы реализовали систему NMT на основе базовых операций TensorFlow, поскольку это дает нам более глубокое понимание принципов работы алгоритма перевода, по сравнению с использованием готовых библиотек TensorFlow, таких как `seq2seq`. Затем вы узнали, что контекстный вектор является узким местом системы, поскольку она вынуждена втискивать все знания об исходном предложении в сравнительно небольшой вектор. Понимая сложность задачи, которую система не может полноценно решить, мы перешли к изучению механизма внимания, способного устранить узкое место модели. Вместо того чтобы зависеть исключительно от контекстного вектора фиксированной размерности, механизм внимания позволяет декодеру наблюдать полную историю состояний кодера на каждом шаге и помогает формировать вектор с расширенным контекстом. Вы убедились, что этот метод помогает системам NMT работать намного лучше.

Наконец, мы поговорили о чат-ботах, еще одном популярнейшем направлении машинной обработки текстовых последовательностей. Чат-боты – это приложения машинного обучения, способные вести реалистичную беседу с человеком и даже отвечать на вопросы. Мы отметили, что системы NMT и чат-боты устроены и работают по очень похожему принципу, разница заключается только в обучающих данных. В заключение рассмотрели тест Тьюринга, который можно использовать для оценки качества чат-ботов.

В следующей главе мы обсудим различные современные тенденции в области машинной обработки естественного языка.

Глава 11

Современные тенденции и будущее обработки естественного языка

В этой главе мы обсудим, что происходит в NLP сегодня, и попытаемся предсказать, что нас ждет в будущем. Глава начинается с обсуждения сегодняшнего положения дел в NLP. Если кратко, то сегодня основное внимание сосредоточено на улучшении существующих моделей, в первую очередь – повышении качества и производительности методов представления слов и систем машинного перевода.

Остальная часть главы посвящена новым направлениям NLP, появившимся в последнее время. У нас имеется пять увлекательных тем для обсуждения, подкрепленных уникальными и познавательными статьями. Сначала вы узнаете, как NLP проникает в другие области исследований, такие как компьютерное зрение и обучение с подкреплением. Далее мы обсудим несколько новых попыток достичь искусственного общего интеллекта за счет обучения одной NLP-модели выполнению нескольких задач. Мы также рассмотрим новые задачи, возникающие в области NLP, такие как обнаружение сарказма и смысловое основание. Затем вы увидите, как NLP применяется в социальных сетях, особенно при сборе и анализе информации. Наконец, узнаете о некоторых новых моделях на основе временных рядов, таких как фазовые LSTM. Подобные сети намного лучше идентифицируют конкретные события, происходящие непредсказуемо в течение очень длительных периодов времени.

Подводя итог, мы будем говорить о последних тенденциях NLP, а затем о наиболее важных инновациях:

- современные тенденции в NLP;
- проникновение NLP в другие области;
- достижения в AGI с точки зрения NLP;
- новейшие задачи NLP;
- NLP для социальных сетей;
- улучшенные модели временных рядов.

Большая часть материала этой главы, относящаяся к текущему состоянию и новым направлениям, основана на научных работах в области NLP. Я предусмотрел ссылки на первоисточники, чтобы упомянуть авторов и предоставить вам ресурсы для дальнейшего чтения. В текстовые ссылки входит число в скобках, которое соответствует нумерации в разделе «Источники» в конце главы.

Современные тенденции в NLP

В этом разделе мы поговорим о текущей ситуации в NLP. Обзор основан на исследованиях NLP, проведенных между 2012 и началом 2018 года. Начнем с обсуждения текущего состояния технологии представления слов. Это очень важная тема, поскольку мы уже видели много интересных приложений, которые не могут обойтись без правильного представления слов. Затем мы рассмотрим важные усовершенствования NMT.

Представления слов

Со временем появилось много вариантов представления слов. Благодаря высококачественным представлениям слов (см. «Распределенные представления слов и фраз и их композиционность» [1], Миколов (Mikolov) и др.) можно сказать, что NLP переживает второе рождение в самых разных задачах, таких как анализ настроений, машинный перевод и ответы на вопросы. Кроме того, исследователи не оставляют попытки улучшить методики, что привело к созданию еще лучших представлений. Четыре модели, о которых я вам расскажу, относятся к представлению областей, вероятностному представлению слов, мета-представлению и представлению тем.

Представление областей

Модель *tv-представлений* (сокращение от two-view embedding – представление в двух проекциях) была введена в статье Ри Джонсона (Rie Johnson) и Тон Чжана (Tong Zhang) «Сверточные нейронные сети со слабым подкреплением для категоризации текста с помощью представлений областей» [2]. Этот подход отличается от представления слов, поскольку задействует представление областей, когда вектор фиксированной размерности представляет целую область текста. Например, в отличие от векторного представления слова «*кошка*», мы имеем векторное представление фразы «*кошка сидела на коврике*». Подобное представление называется представлением в двух проекциях, если оно сохраняет информацию, необходимую для прогнозирования представления одного вида из другого – например, предсказать слово, исходя из представления фразы, или наоборот.

Входное представление

Давайте теперь посмотрим на детали этого подхода. На рис. 11.1 изображена *tv*-система в общем виде. Сначала находят числовое представление области слов, т. е. *вектор области* (region vector). Например, рассмотрим следующую фразу:

очень хороший фильм

Она может быть представлена в следующем виде:

очень хороший фильм	*очень хороший фильм*	*очень хороший фильм*
1 0 0	*0 1 0*	*0 0 1*

Эта форма называется вектором с унитарным кодированием последовательности. Другим способом эту фразу можно представить так:

очень хороший фильм
1 1 1

Это представление Bag of of Words (BOW). Мы можем видеть, что представление BOW более компактно и не увеличивается с размером фразы. Однако обратите внимание, что это представление теряет контекстную информацию. В данном случае BOW – это представление признаков, которое мы используем для представления слов или текстовых фраз. Он не относится к алгоритму обучения представлений слов CBOW, который мы обсуждали в главе 3.

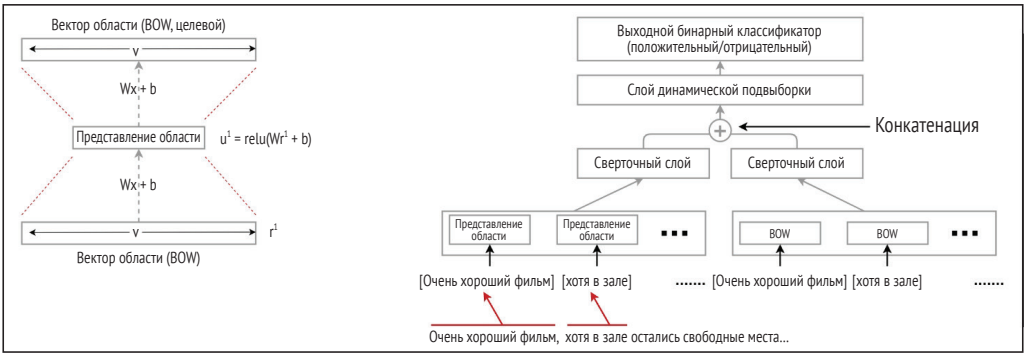


Рис. 11.1 ❖ Изучение представления областей и использование *tv*-представлений для анализа настроений

Изучение представлений областей

Мы изучаем представления областей так же, как изучали представления слов. Мы вводим данные, содержащие текстовую область, и просим модель предсказать целевую область в данном контексте. Например, мы используем область размером 3 для фразы:

очень хороший фильм, мне он понравился

Затем подаем на вход фразу:

очень хороший фильм

Результат (целевая фраза) будет следующим:

мне он понравился

В качестве упражнения мы попробуем проверить, помогают ли представления областей улучшить результат анализа настроений. Для этого мы воспользуемся набором данных, доступных по адресу **http://ai.stanford.edu/~amaas/data/sentiment/**. Это текстовый корпус, содержащий обзоры фильмов IMDB. Сначала мы изучим полезные представления областей, обучив слой представлений правильному прогнозированию области контекста для данной области входного текста. Затем будем использовать эти представления в качестве дополнительного источника данных в модели анализа настроений. Исходный код упражнения хранится в файле `tv_embeddings.ipynb` в папке `ch11`.

Реализация – представления областей

Для этого примера в качестве обучающих данных мы будем использовать 400 положительных и 400 отрицательных выборок из упомянутого выше набора. Мы также отложим в сторону тестовый набор, состоящий из 150 положительных и 150 отрицательных образцов. Здесь я лишь бегло расскажу о реализации и не буду обсуждать конкретные детали. Всю подробную информацию вы найдете в файле упражнения.

Во-первых, для изучения представлений области мы объявим полностью связанный набор весов и смещение:

```
w1 = tf.get_variable('w1', shape=[vocabulary_size,500],
    initializer = tf.contrib.layers.xavier_initializer_conv2d())
b1 = tf.get_variable('b1',shape=[500],
    initializer = tf.random_normal_initializer(stddev=0.05))
```

Далее, используя веса и смещения, мы вычислим скрытое значение, используя нелинейность типа ReLU, что является стандартным типом нелинейности в нейронных сетях:

```
h = tf.nn.relu(
    tf.matmul(train_dataset,w1) + b1
)
```

Затем определим другой набор весов и смещений, который действует как верхний регрессионный слой. Верхний слой прогнозирует представление BOW контекстной области для данной текстовой области:

```
w = tf.get_variable('linear_w', shape=[500, vocabulary_size],
    initializer= tf.contrib.layers.xavier_initializer())
b = tf.get_variable('linear_b', shape=[vocabulary_size],
    initializer= tf.random_normal_initializer(stddev=0.05))
```

И наконец, вычисляем окончательный выход:

```
out = tf.matmul(h,w)+b
```

Теперь мы определим среднеквадратическую ошибку как разницу между прогнозируемой контекстной областью BOW и истинным контекстом BOW. Мы будем использовать `train_mask` для маскировки некоторых несуществующих слов (0 в истинном представлении BOW), аналогично методу отрицательной выборки, который обсуждали в главе 3.

```
loss = tf.reduce_mean(tf.reduce_sum(train_mask*(
       out - train_labels)**2,axis=1))
```

Наконец, объявим оптимизатор вычисленной выше ошибки:

```
optimizer = tf.train.AdamOptimizer(
learning_rate = 0.0005).minimize(loss)
```

Затем воспользуемся изученными представлениями областей в качестве дополнительных данных для классификации текста, как показано на рис. 11.1. Для этого мы последовательно объединяем (конкатенируем) представления всех текстовых областей, найденных в данном обзоре. Мы сделаем то же самое для входов BOW. Затем мы параллельно свертываем конкатенированные векторы (то есть

представления областей и векторы BOW) и конкатенируем результаты свертки. Далее передаем объединенный вывод свертки в верхний слой классификации, который выводит прогноз, был обзор фильма положительным или отрицательным.

Точность классификации

При измерении точности на наборе проверенных данных модель с *tv*-представлениями, кажется, немного превосходит модель, лишенную такого представления (рис. 11.2). Эта разница может быть улучшена за счет использования методов регуляризации, таких как отсев и обучение в течение более длительного времени. Таким образом, можно сделать вывод, что *tv*-представления действительно способствуют повышению точности при выполнении задач классификации текста по сравнению с использованием стандартного представления BOW.

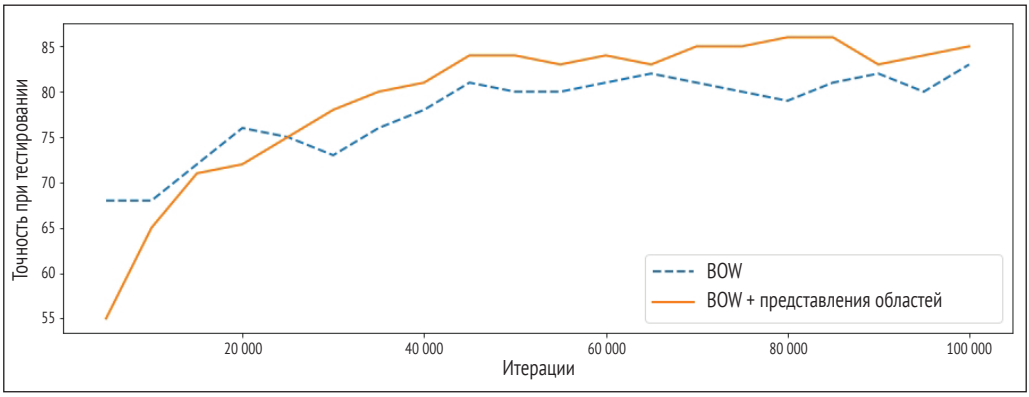


Рис. 11.2 ❖ Точность классификации настроений для модели, использующей BOW, и модели, использующей BOW и представление области

Вероятностное представление слов

Вероятностные модели представления слов – это еще одна новая разработка в области представлений слов. В статье Шаохуа Ли (Shaohua Li) и др. «Генеративная модель представления слов и ее низкоранговая реализация» [3] описан метод представления слов PSDVec, позволяющий создавать представления, которые отличаются от детерминированных векторов слов, таких как skip-gram, CBOW и GloVe, и более информативны по сравнению с ними. PSDVec предоставляет распределение вероятностей представления слова вместо точного числового вектора. Например, если мы предположим, что вектор слова имеет размерность 1, а GloVe говорит, что вектор слова «собака» равен 0,5, то PSDVec обеспечит распределение по *всем* возможным значениям, которые могут выглядеть, как показано на рис. 11.3. PSDVec вполне может утверждать, что значение вектора для слова «собака» равно 0,5 с более высокой вероятностью (например, 0,3), и это же значение равно 0,1 с меньшей вероятностью (например, 0,05).

Вероятностные модели имеют более богатую интерпретацию, чем детерминированные модели, такие как Word2vec. Чтобы изучить вероятностные распределения векторов слов, они используют технику, известную как *вариационный вывод* (variational inference). В своей работе они обучают не только слой представления,

но также и *остаточный слой* (residual layer), который фиксирует шумовые и нелинейные отношения между словами. Авторы показывают, что PSDVec обеспечивает конкурентоспособную точность по сравнению со стандартными Word2vec и GloVe.

Рис. 11.3 ❖ Как PSDVec работает с одномерным представлением слова

Ансамблевое представление

В своей статье «Изучение мета-представлений слов» [4] Вэнпен Инь (Wenpeng Yin) и Хинрих Шютце (Hinrich Schütze) предлагают подход к изучению *мета-представлений* (meta-embeddings) – представлений, состоящих из ансамбля нескольких общедоступных наборов представлений. Два ключевых преимущества этого подхода: (1) улучшенная точность, поскольку они используют множественные наборы представлений слов, и (2) более высокий словарный запас из-за использования нескольких наборов представлений.

Тематическое представление

Тематическое представление (topic embedding) также вызывает интерес в сообществе NLP. Этот подход позволяет любому документу быть представленным набором тем (например, информационные технологии, медицина и развлечения), где для данного документа вычисляют весовые коэффициенты каждой темы, отражая тем самым, насколько документ имеет отношение к этой теме. Например, документ об использовании машинного обучения в здравоохранении будет иметь больший вес для таких тем, как «Информационные технологии» и «Медицина», но меньший вес для темы «Закон и право».

В статье «Тематическое представление слов» [5] Ян Лю (Yang Liu) и др. этот подход используется для изучения представлений слов. Тематические представления слов основаны на представлении нескольких прототипов. Многопрототипные представления отличаются от стандартных представлений слов, поскольку дают разные значения вектора представления в зависимости от контекста, в котором используется слово. Например, вполне разумно, если в контексте информационных технологий слово *Windows* (окна) будет представлено совершенно иным вектором по сравнению с контекстом жилого дома. Изучение тем происходит с помощью процесса, известного как *скрытое распределение Дирихле* (latent Dirichlet allocation, LDA), популярного метода, используемого для моделирования тем. Авторы оценивают свой метод по качеству классификации новостных текстов

на разные темы, такие как информационные технологии, медицина и политика. Методика TWE превосходит другие методы моделирования, такие как BOW и LDA, используемые по отдельности.

Нейронный машинный перевод

Нейронный машинный перевод уже доказал свою универсальность, и многие компании и исследователи вкладывают силы и средства в улучшение систем NMT. На сегодняшний день системы NMT достигли уровня, позволяющего работать автономно и в режиме реального времени. Однако эти системы все еще не сравнялись с человеческим переводом. Поэтому работа по улучшению систем NMT далека от завершения. Как мы обсуждали в главе 10, машинный перевод обладает огромным потенциалом в различных областях, таких как производство и бизнес. Еще один пример использования машинного перевода в реальном времени можно найти в области туризма, когда туристы получают переводы на родной язык вывесок, текстов, речи и т. д. во время посещения какой-либо другой страны.

Улучшение механизма внимания

Мы уже говорили о механизме внимания, устраняющем узкое место классической архитектуры в стиле кодер–декодер. Благодаря механизму внимания декодер получает возможность просматривать полное исходное предложение на каждом этапе декодирования. Тем не менее это не последнее улучшение. В статье «Эффективные подходы к основанному на внимании нейронному машинному переводу» [6] Минь-Тан Лон (Minh-Thang Luong) и др. предлагают метод *входной подводки* (input feeding). В этом методе передают предыдущий вектор внимания в качестве входных данных для текущего временного такта декодера. Это помогает декодеру обладать информацией о предыдущем выравнивании слов и повышает качество системы машинного перевода.

В статье Таики Ватанабе (Taiki Watanabe) и др. «Сверточный механизм внимания на основе CKY для нейронного машинного перевода» [7] представлен подход, в котором для изучения того, на какое место обращать внимание в исходном предложении, используется сложная сверточная нейронная сеть. В отличие от многослойных персептронов, задействованных в первоначальном механизме внимания, сверточные нейросети способны собирать пространственную информацию. Это преимущество идет на пользу качеству машинного перевода.

Гибридные модели машинного перевода

Как показали результаты работы системы машинного перевода в главе 10, выходной текст часто содержит маркер <unk>, заменяющий редкие и незнакомые системе слова. Однако нам не нравится такое поведение. Хотелось бы найти способ заменить эти редкие слова в исходных и целевых предложениях какими-то значащими словами.

К сожалению, мы не можем поддерживать словари всех существующих слов на исходном и целевом языке, так как это будет гигантская база данных. В настоящее время Оксфордский словарь английского языка содержит более 150 тыс. различных слов. Но если добавить в него различные формы глаголов, имена и названия объектов по всему миру, то словарь моментально раздуется до гигантского размера.

Здесь как раз будут уместны *гибридные модели* (рис. 11.4). В таких моделях мы не заменяем редкие слова маркером <unk>, а делегируем задачу обработки редких слов кодировщику на уровне символов. Подобный подход вполне осуществим, поскольку существует очень маленький набор возможных символов. Затем последнее (выходное) состояние кодировщика уровня символов возвращается текстовому машинному переводчику и выполняется обычный перевод. Кроме того, тот же самый процесс используется в декодере, если он вынужден выдать маркер <unk>. Технология гибридной модели предложена в диссертации Минь-Тан Лона «Нейронный машинный перевод» [8]. Вы можете найти реализацию гибридной модели NMT по адресу **https://github.com/lmthang/nmt.hybrid**.

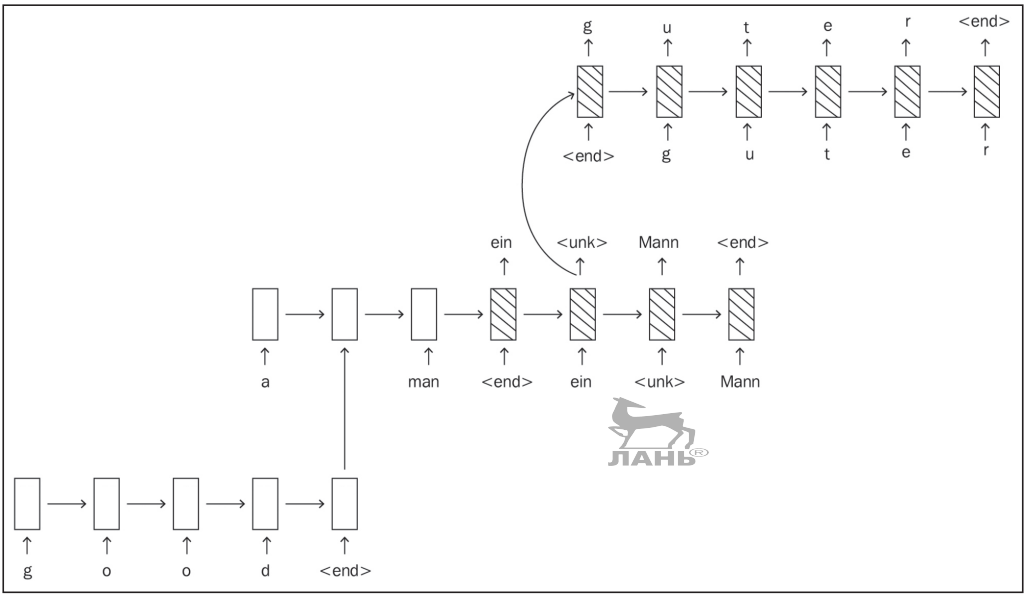


Рис. 11.4 ❖ Гибридная модель машинного перевода

Здесь для ясности мы покажем метод прогнозирования, используемый в гибридных NMT, представив его в виде *псевдокода*[1].

Для каждого слова в исходном предложении псевдокод выглядит следующим образом:

```
If word != <unk>
    кодируем слово представлением на уровне слова
Else
    Для каждого символа в исходном редком слове
        формируем посимвольный код.
    Возвращаем последнее скрытое состояние символьного кодера
на вход кодера уровня слов вместо токена <unk>.
```

[1] Псевдокод – это описание алгоритма на смеси компьютерного и «человеческого» языков. – *Прим. перев.*

Для каждого слова, предсказанного декодером, предсказание выглядит следующим образом:

```
If word != <unk>
    Используем декодер уровня слов.
If word == <end>
    Завершаем предсказание.
Else
    Инициализируем декодер символьного уровня скрытым состоянием декодера уровня слов.
    Выводим последовательность символов при помощи декодера символьного уровня,
    пока не встретим <end>.
```

Теперь давайте окинем взглядом некоторые из перспективных направлений развития NLP. Как правило, новые направления подразумевают слияние NLP с другими признанными областями исследований, такими как обучение с подкреплением и генеративные состязательные сети.

Применение NLP в смежных прикладных областях

Далее мы обсудим три различные области, которые интенсивно взаимодействуют с NLP при решении некоторых интересных задач машинного обучения. Мы рассмотрим три конкретных направления:

- NLP и компьютерное зрение;
- NLP и обучение с подкреплением;
- NLP и генеративные состязательные сети.

Сочетание NLP с компьютерным зрением

Мы начнем с двух примеров, где NLP объединяется с различными приложениями компьютерного зрения для обработки мультимодальных данных (в данном случае изображений и текста).

Визуально-описательные системы

Визуально-описательные системы (visual question answering, VQA) – это новая область исследований и перспективная технология, способная давать ответы на текстовые вопросы об изображении. Например, рассмотрим вопросы о рис. 11.5:

Вопрос 1: *какого цвета диван?*
Вопрос 2: *сколько там темных кресел?*

Исходя из имеющегося рисунка, система должна вывести ответы наподобие таких:

Ответ 1: *цвет дивана белый*
Ответ 2: *в комнате два темных кресла*

Рис. 11.5 ❖ Изображение, о котором мы задаем вопросы

Модель обучения для этого типа задач будет очень похожа на архитектуру, которую мы использовали для создания подписей к изображениям в главе 9. Набор обучающих данных будет состоять из изображений, вопросов и ответов, соответствующих изображению.

Обучение модели происходит следующим образом.

1. Пропускаем изображения через сверточную нейросеть, например предварительно обученную на ImageNet, чтобы получить вектор контекста, представляющий изображение.
2. Создаем последовательность данных, из которой состоит обучающая последовательность (`вектор изображения, <s>, вопрос, </s>, <s>, ответ, </s>`), где `<s>` обозначает начало, а `</s>` – конец вопроса.
3. Используем эту и другие подобные последовательности, чтобы обучить LSTM ответам на соответствующие вопросы.

Во время прогнозирования процесс выглядит следующим образом.

1. Пропускаем изображения через сверточную нейросеть, например предварительно обученную на ImageNet, чтобы получить вектор контекста, представляющий изображение.
2. Создаем последовательность данных, включающую контекст изображения и вопрос (`вектор изображения, <s>, вопрос, </s>, <s>`).
3. Подаем последовательность в LSTM, и после ввода последнего токена `<s>` нейросеть итеративно предсказывает слова, подавая последнее предсказанное слово в качестве входных данных для следующего шага, пока LSTM не выведет `</s>`. Вновь предсказанные слова составят ответ.

Одна из ранних моделей на основе CNN и LSTM, успешно дающая ответы на вопросы об изображениях, описана в работе Менжи Рена (Mengye Ren) и др. «Изучение моделей и данных для ответов на вопросы об изображениях» [8]. Другой более продвинутый метод предложен в работе Цзисен Лю (Jiasen Lu) и др. «Иерархическое совместное внимание в визуальных системах вопрос–ответ» [9].

Код для системы VQA, основанный на TensorFlow, доступен по адресу **https://github.com/tensorflow/models/tree/master/research/qa_kg**. Этот код содержит метод, описанный в статье Ронхан Ху (Ronghang Hu) и др. «Формирование осмысленности: сквозные модульные сети для визуальных систем вопрос–ответ» [10].

Хороший набор данных для обучения и тестирования моделей VQA (набор данных с изображениями, а также вопросы и ответы, соответствующие каждому изображению) находится по адресу **http://www.visualqa.org/vqa_v1_download.html**, который был представлен в статье «VQA: визуальные обоснованные ответы на вопросы» [11] Станислава Антола (Stanislaw Antol) и др.

Генерация подписей для изображений на основе внимания

Кельвин Сюй (Kelvin Xu) и др. в статье под названием «Показать, понять и рассказать: нейронная генерация подписей к изображениям с визуальным вниманием» [12] описывают интересное исследование, в котором основное внимание уделялось изучению того, где искать изображение для создания заголовка. Основная идея авторов заключается в том, что, в отличие от стандартных моделей генерации подписей, которые используют полностью связанный слой CNN для извлечения векторов признаков, этот метод для представления признаков изображения использует нижний слой свертки. Затем поверх данного слоя свертки

накладывают двумерный слой внимания – аналогично одномерному слою внимания, который мы использовали в главе 10, – представляющий часть изображения, на которой модель должна фокусироваться при генерации слова. Например, учитывая изображение собаки, сидящей на ковре, при создании слова «*собака*» генератор подписей к изображению может уделять больше внимания той части изображения, где находится собака, чем остальной части изображения.

Обучение с подкреплением

Еще одной областью исследований, связанной с NLP, является *обучение с подкреплением* (reinforcement learning, RL). Технологии NLP и RL не взаимодействовали друг с другом в течение десятилетий, и довольно любопытно наблюдать, как проблемы NLP формулируются и решаются с точки зрения обучения с подкреплением. Давайте бегло усвоим, что такое RL. В обучении с подкреплением агент взаимодействует со средой. Агент может наблюдать за окружающей средой (полностью или частично), которая подается агенту как состояние. Затем, в зависимости от состояния, агент выполнит действие, выбранное из некоторого пространства действий. Наконец, после выполнения действия агенту будет предоставлено вознаграждение. Цель агента – максимизировать долгосрочное вознаграждение, которое он накапливает.

Далее мы обсудим, как RL используется для решения различных задач, связанных с естественным языком. Сначала вы узнаете, как RL применяется для обучения нескольких агентов «языку» совместного общения. За этим последует обучение агентов наилучшему выполнению запроса пользователя при помощи уточняющих вопросов.

Обучение агентов общению на собственном языке

В исследовании «Многоагентное сотрудничество и появление (естественного) языка» [13] Анжелика Лазариду (Angeliki Lazaridou) и др. учат нескольких агентов изобретать уникальный язык для общения. Это делается путем выбора двух агентов из группы – отправителя и получателя. Отправителю предоставляется пара изображений (где одно изображение является целью), и он должен отправить небольшое сообщение получателю. Сообщение состоит из символов, выбранных из фиксированного словаря. Изначально словарь не содержит смысловой связи между символами. Получатель видит изображения, но не знает, какое из них целевое, и должен идентифицировать цель по полученному сообщению. Задача обучения состоит в том, чтобы агент активировал один и тот же символ для похожих изображений. Если получатель правильно предсказывает целевое изображение, оба агента получат вознаграждение 1. В случае неудачи оба получают вознаграждение, равное 0. Этот процесс изображен на рис. 11.6.

Диалоговые агенты с подкреплением

В следующих двух статьях методика RL используется для обучения сквозных диалоговых систем, основанных на глубоком обучении: «Обучение с подкреплением в создании полных диалоговых агентов для доступа к информации» [14] Бхуван Дхингра (Bhuwan Dhingra) и др. и «Сетевые обучаемые диалоговые системы для

прикладных задач» [15] Цзун-Сянь Вэнь (Tsung-Hsien Wen) и др. Диалоговая система общается с человеком на естественном языке и пытается выполнить задачу, вытекающую из фразы, произнесенной человеком. Например, человек задает вопрос:

Где расположены рестораны французской кухни в Сиднее?

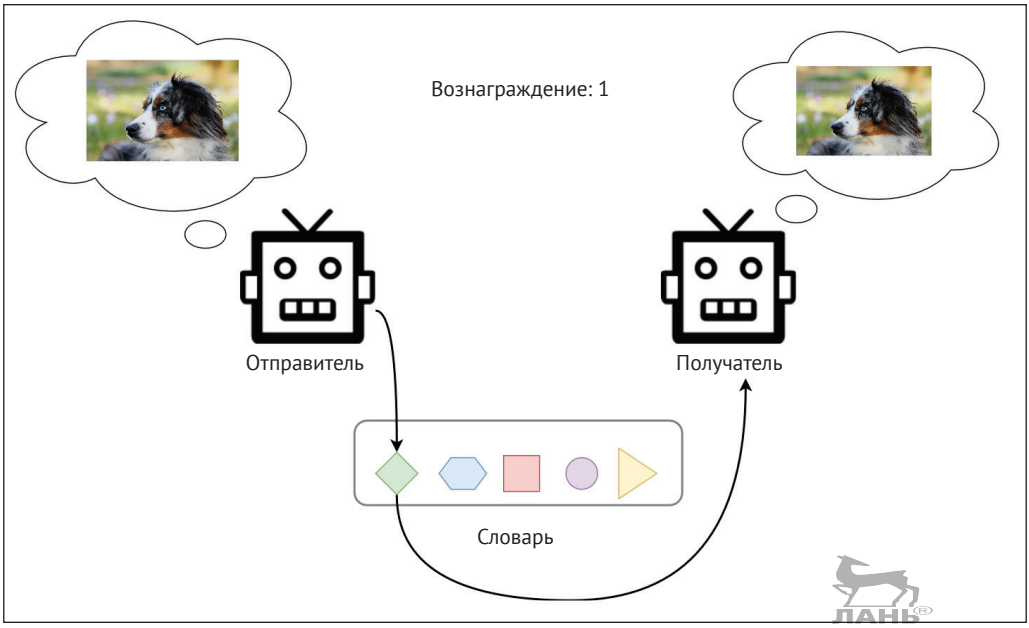


Рис. 11.6 ❖ Агенты учатся использовать словарный запас для общения об изображениях. Если получатель правильно идентифицирует изображение, отправитель и получатель получат положительное вознаграждение

Затем агент должен преобразовать вопрос в вектор требуемых признаков, что достигается с помощью системы, называемой *трекером побуждений* (belief tracker). Трекер побуждений отображает запрос произвольной формы на естественном языке на фиксированный вектор признаков. Его также можно рассматривать как семантический анализатор. Далее вектор признаков используется для запроса к структурированной базе знаний, чтобы найти ответ.

Однако могут быть сложные ситуации, когда человек задает лишь частичный вопрос, например:

Каковы лучшие рестораны в городе?

Тогда система может уточнить:

Какой город?

На это человек отвечает:

Сидней.

Тогда система вновь уточнит:

Какая кухня?

На это человек отвечает:

Французская.

Теперь система получила необходимую информацию о запросе и может обратиться к базе данных. Обучение с подкреплением здесь заключается в том, что можно разработать функцию, выдающую системе вознаграждение за каждый правильно найденный ответ. Это заставит агента научиться задавать правильные уточняющие вопросы, необходимые для восполнения недостающей информации в запросе пользователя.

Генеративные состязательные сети и NLP

Генеративные модели – это семейство моделей, которые способны генерировать новые выборки из некоторого наблюдаемого распределения выборок. Мы уже видели пример генеративной модели, когда использовали LSTM для генерации текста. Другим примером применения генеративной модели является создание изображений. Модель обучается на рукописных цифрах и приобретает навык генерировать новые изображения цифр. Для создания изображений можно использовать популярный современный метод – *генеративные состязательные сети* (generative adversarial networks, GAN). Их устройство в общем виде представлено на рис. 11.7.

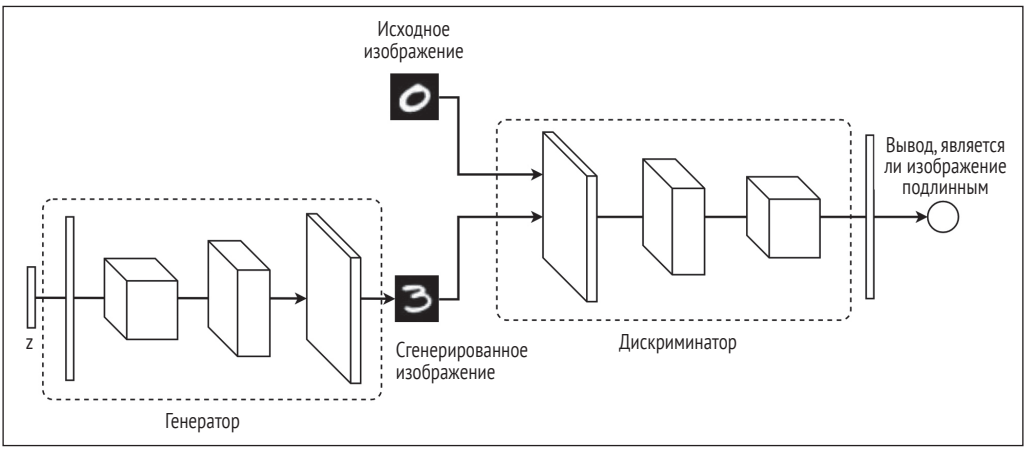


Рис. 11.7 ❖ Генеративная состязательная сеть

В системе есть два разных компонента: *генератор* и *дискриминатор*. Целью генератора является создание изображений, которые выглядят как реальные изображения. Дискриминатор пытается правильно отличить реальные изображения от созданных генератором «поддельных» объектов. Мы подаем на генератор некоторый шум, т. е. выборки, полученные из нормального распределения,

и он создает изображение. Генератор представляет собой обратную сверточную нейросеть, которая принимает вектор в качестве входа и выводит изображение. Этим генератор отличается от стандартной CNN, принимающей изображение в качестве входных данных и выдающей вектор признаков. Дискриминатор пытается различить истинные и искусственные изображения. Таким образом, в начале обучения дискриминатор с легкостью различает изображения. Но генератор постепенно учится создавать все более реалистичные изображения, и дискриминатору становится все труднее их отличать от подлинных картинок.

GAN изначально были предназначены для создания реалистичных изображений. Однако было несколько попыток адаптировать GAN для генерирования предложений. Рисунок 11.8 иллюстрирует общий подход к использованию GAN в этой задаче. Далее давайте рассмотрим подробности этого подхода.

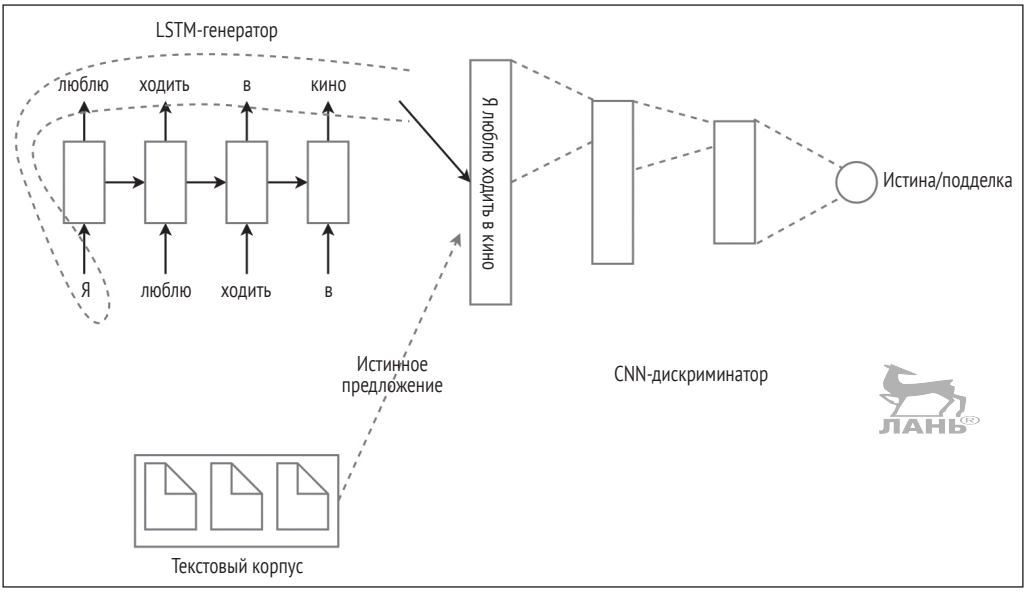


Рис. 11.8 ❖ Общая концепция использования генератора LSTM и дискриминатора CNN для генерации предложений

В исследовании «Генерация текста с помощью состязательного обучения» [16] Ичжэ Чжан (Yizhe Zhang) и др. используют модифицированную GAN для генерации текста. В их работе есть существенные отличия от сверточной GAN, которую мы обсуждали выше. Во-первых, они используют генератор на основе LSTM-сети, который берет некоторый случайный элемент из словаря в качестве входных данных и генерирует предложение произвольной длины. Далее, дискриминатором является CNN, которая обучена для классификации данного предложения в один из двух классов – поддельное или реальное. Процесс обучения CNN схож с обучением классификации предложений, алгоритм которого мы обсуждали в главе 5. Сначала CNN будет очень хорошо различать реальные и поддельные предложения. Со временем LSTM научится создавать более реалистичные предложения, чтобы обмануть классификатор.

В статье «SeqGAN: Последовательные генеративные состязательные сети с градиентным спуском по стратегиям» [17] Лантао Юй (Lantao Yu) и др. демонстрируют другой подход к синтезу текста с использованием генеративной модели. В этом случае также генератор является сетью LSTM, а дискриминатор – сетью CNN по аналогии с упомянутой выше статьей [16]. Однако в данном случае применяется обучение с подкреплением.

Состояние системы – это текущая текстовая строка генератора, а пространство действия – словарь для выбора слов. Процесс продолжается до тех пор, пока для данного шага не будет сгенерирован полный текст. Награда выдается только в конце полной последовательности, а в качестве приза выступает выход дискриминатора. Следовательно, вознаграждение будет высоким, если выходной сигнал дискриминатора близок к 1 (т. е. дискриминатор считает данные реальными), и низким, если выходной сигнал близок к 0. Затем, определив вознаграждение, авторы используют *градиентный спуск по стратегиям* (policy gradient) для обучения генератора через обратное распространение ошибки. В частности, алгоритм градиентного спуска вычисляет градиенты для весов генератора, исходя из вознаграждения, выданного дискриминатором. Реализация SeqGAN в TensorFlow доступна по адресу **https://github.com/LantaoYu/SeqGAN**.

На пути к искусственному общему интеллекту

Искусственный общий интеллект (artificial general intelligence, AGI) позволяет машинам выполнять когнитивные или интеллектуальные задачи, с которыми обычно хорошо справляется только человек. Это иная и более сложная концепция, чем AI, поскольку AGI подразумевает навык решения неявных задач, помимо стандартного запроса к машине для выполнения задачи с учетом необходимых данных. Например, мы помещаем робота в новую среду (скажем, дом, в котором робот никогда не бывал) и просим его приготовить кофе. Если робот действительно сможет перемещаться по дому, найти кофе-машину, научиться управлять ею, выполнить правильную последовательность действий, необходимых для приготовления кофе, и принести кофе человеку, то мы можем сказать, что робот достиг AGI. Пока мы еще далеки от достижения AGI, но в этом направлении ведется непрерывная работа, шаг за шагом. Кроме того, NLP будет играть большую роль в будущем использовании AGI, так как наиболее естественный способ взаимодействия для людей – это голосовое общение.

Статьи, которые мы будем обсуждать далее, описывают отдельные модели, пытающиеся научиться выполнять множество задач. Другими словами, одна сквозная модель сможет классифицировать изображения, обнаруживать объекты, распознавать речь, переводить тексты на другой язык и т. д. Мы можем воспринимать модели машинного обучения, способные выполнять множество задач, как один из шагов к AGI.

Обучил одну модель – обучил их все

В статье «Обучил одну модель – обучил их все» [18] Лукаш Кайзер (Lukasz Kaiser) и др. представляют единую модель глубокого обучения, которая способна выучить

множество задач – например, классификацию изображений, создание подписей к изображениям, языковой перевод и распознавание речи. В частности, эта модель со вполне логичным названием MultiModel состоит из нескольких модулей: подсетей, кодера, микшера ввода/вывода и декодера.

Во-первых, MultiModel состоит из нескольких подсетей, или *модальных сетей*, преобразующих входные данные определенной модальности, например изображения, в единое представление. Таким образом, все входные данные, имеющие разные модальности, могут обрабатываться одной глубокой сетью. Обратите внимание, что модальные сети не являются специфичными для задач; они ориентированы только на модальности. Это означает, что несколько разных задач, имеющих одинаковую модальность ввода, будут совместно использовать одну модальную сеть. Теперь давайте посмотрим, какую роль играют кодер, микшер ввода-вывода и декодер.

Кодер обрабатывает входные данные, созданные модальной сетью, используя вычислительные элементы, такие как блоки свертки, блоки внимания и набор экспертных блоков. Мы подробнее рассмотрим каждый из этих элементов немного позже.

Микшер ввода/вывода объединяет (или смешивает) кодированный вход с ранее наблюдаемыми выходами для получения кодированных выходов. Этот модуль обрабатывает входные данные и ранее наблюдаемые выходные данные как *авторегрессивная модель* (autoregressive model). Чтобы понять, что такое авторегрессивная модель, рассмотрим временной ряд $y = \{y_0, y_1, y_2, \ldots, y_{t-1}\}$. В своей простейшей форме модель предсказывает y_t как функцию y_{t-1} (т. е. $y_t = \beta_1 y_{t-1} + \beta_0 + \epsilon$, где β_0 и β_1 – обучаемые коэффициенты, а ϵ захватывает шум, присутствующий в y). Однако это уравнение может быть обобщено на произвольное число предыдущих значений y, например $y_t = \beta_2 y_{t-2} + \beta_1 y_{t-1} + \beta_0 + \epsilon$. Это полезно, поскольку MultiModel обрабатывает в том числе различные типы временных рядов, таких как речь и текст.

Декодер принимает как кодированные выходы, так и кодированные входы и формирует декодированный выход, используя блоки свертки и внимания и набор экспертных блоков. Опишем эти блоки подробнее:

- сверточный блок – обнаруживает локальные и пространственные структуры и преобразует их в карты признаков;
- блок внимания – решает, на что обратить внимание во входных данных при кодировании/декодировании;
- набор экспертных блоков – это способ увеличить мощность модели при незначительном увеличении вычислительных затрат. Набор экспертов – это совокупность нескольких сетей прямого распространения (т. е. экспертов) с обучаемым переключающим механизмом, который выбирает разные сети в зависимости от входных данных.

Хотя детали реализации сильно различаются, вы должны увидеть сходство с системой NMT, которую мы изучали в главе 10. MultiModel сначала кодирует входные данные, так же, как мы кодировали исходное предложение при помощи кодера NMT. Наконец, MultiModel декодирует внутреннее представление и производит удобочитаемый вывод, так же как NMT-декодер создает целевое предложение.

MultiModel обучается выполнению различных задач со следующими наборами данных, которые перечислены в упомянутой выше статье [18].

1. Речевой корпус Wall Street Journal (WSJ) – представляет собой большой набор данных, содержащий высказывания (~73 часа речи) различных людей,

в том числе журналистов с различным опытом. Этот набор данных находится по адресу **https://catalog.ldc.upenn.edu/LDC93S6A**.

2. Набор данных ImageNet – это набор изображений, который мы обсуждали в главе 9. Он содержит более миллиона изображений, принадлежащих 1000 различных классов. Набор данных находится по адресу **http://image-net.org/download**.
3. Набор подписей к изображениям MS-COCO – эти данные также использовались в главе 9. Набор содержит изображения и соответствующие описания, созданные людьми. Этот набор данных можно найти по адресу **http://coco-dataset.org/#download**.
4. Набор данных синтаксического разбора WSJ. *Синтаксический разбор* – это процесс идентификации существительных, местоимений, глаголов, прилагательных и т. д. в предложении и построения дерева разбора для этого предложения. Набор данных создан с помощью разбора текстового корпуса WSJ и находится по адресу **https://catalog.ldc.upenn.edu/LDC99S42**.
5. Англо-немецкий переводческий корпус WMT – это двуязычный текстовый корпус, содержащий английские предложения и соответствующие немецкие переводы, аналогично набору данных, который мы использовали в главе 10. Набор доступен по адресу **http://www.statmt.org/wmt14/translation-task.html**.
6. Немецко-английский переводческий корпус – реверс данных п. 5.
7. Англо-французский переводческий корпус WMT – это двуязычный текстовый корпус, содержащий предложения на английском языке и соответствующий перевод на французский язык, аналогичный набору данных, который мы использовали в главе 10. Наборы данных находятся по адресу **http://www.statmt.org/wmt14/ translation-task.html**.
8. Французско-английский переводческий корпус – реверс данных п. 7. В оригинальной статье авторы называют его немецко-французским, что мы считаем случайной ошибкой, поскольку предыдущий корпус – английский с переводом на французский.

Ожидается, что после обучения на этих наборах данных модель с высокой точностью выполнит следующие задачи:

- преобразование речи в текст;
- создание подписи для заданного изображения;
- идентификация объектов на заданном изображении;
- перевод с английского на немецкий или французский;
- создание деревьев синтаксического разбора английских фраз.

Реализация модели MultiModel на платформе TensorFlow находится по адресу **https://github.com/tensorflow/tensor2tensor**.

Совместная многозадачная модель – развитие нейронной сети для множества задач NLP

В работе «Совместная многозадачная модель – развитие нейронной сети для множества задач NLP» [19] Казума Хасимото (Kazuma Hashimoto) и др. обучают сквозную модель различным задачам NLP. Однако данный подход отличается от рассмотренного выше. В этом случае нижние уровни модели изучают простые за-

дачи, а более высоким уровням достаются более сложные задачи. С точки зрения обучения необходимые метки (например, метки частей речи) распределяются по отдельным уровням сети. Эти задачи подразделяются на три различных типа от низшего к высшему: задачи на уровне слов, синтаксические задачи и семантические задачи. Таким образом, более высокие уровни могут использовать информацию о выполнении простых задач. Например, для определения зависимостей предложения могут быть полезны теги частей речи. Эта концепция показана на рис. 11.9.

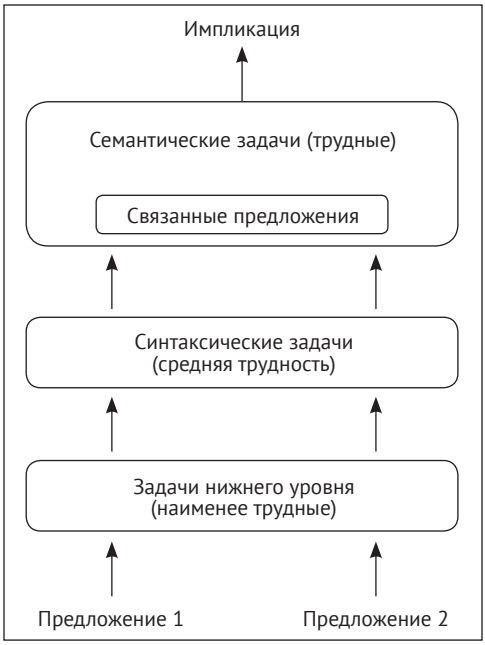


Рис. 11.9 ❖ Решение задач с нарастающей сложностью

Первый уровень – задачи на уровне слов

Первые два слоя выполняют задачи на уровне слов. Первый уровень выполняет маркировку частей речи для каждого слова в предложении. Следующий слой выполняет так называемый *чанкинг* (chunking), процесс, в котором каждому слову снова присваиваются теги.

Второй уровень – синтаксические задачи

Следующий уровень выполняет разбор зависимостей предложения. Это задача анализа грамматической структуры предложения и выявления связей между словами.

Третий уровень – задачи семантического уровня

Следующий уровень кодирует информацию о *связанности* (relatedness) предложений. Однако связанность измеряется между двумя предложениями. Чтобы

обработать два предложения одновременно, у нас есть два параллельных стека, устроенных, как описано выше. Следовательно, у нас есть две разные сети, кодирующие два предложения относительно их связанности. Последний слой выполняет текстовую *импликацию*, т. е. аналитическое умозаключение о том, влечет ли предложение-предпосылка (второе предложение) предложение-гипотезу (первое предложение). Импликация может быть подтверждающей, исключающей или нейтральной. Здесь мы перечислим примеры всех трех упомянутых категорий.

- Подтверждающее
  Гипотеза: *облачное небо ведет к дождю*
  Предпосылка: *если будет облачно, пойдет дождь*
- Исключающее
  Гипотеза: *облачное небо не ведет к дождю*
  Предпосылка: *если будет облачно, пойдет дождь*
- Нейтральное
  Гипотеза: *облачное небо ведет к дождю*
  Предпосылка: *если облачно, ваша собака будет лаять*

NLP для социальных сетей

Далее мы поговорим о том, как технология NLP повлияла на сбор информации в социальных сетях. Здесь мы обсудим результаты, представленные в нескольких статьях. Эти результаты включают в себя обнаружение слухов, анализ настроений и выявление политических манипуляций словами, например для победы на выборах.

Обнаружение слухов в соцсетях

В статье «Обнаружение слухов с использованием информации временных рядов о социальном контексте в микроблогах» [20] Цзин Ма (Jing Ma) и др. предлагают способ обнаружения слухов в микроблогах. Слухи – это истории или утверждения, которые либо являются заведомо ложными, либо достоверность которых не подтверждена. Выявление слухов на ранних стадиях важно для предотвращения распространения ложной или неподтвержденной информации. В этой статье событие определяется по набору микроблогов, относящихся к этому событию. Для каждого микроблога создается контекстно-зависимый признак, и они объединяются во временны́е интервалы в зависимости от времени появления микроблога. После этого авторы используют *временную структуру динамического ряда* (dynamic-series time structure, DSTS), чтобы выучить «форму» временных рядов при эволюции контекстных признаков. Более конкретно, располагая временным рядом контекстных признаков, DSTS представляет форму временного ряда как комбинацию векторов признаков во времени (f_0, f_1, f_2, ..., f_t) и функций перепада между соседними контекстными признаками во времени (0, $f_1 - f_0$, $f_2 - f_1$, ...). Это помогает идентифицировать слухи, поскольку эти модели, как правило, ведут себя по-разному для слухов и достоверных событий. Например, количество вопросительных знаков в микроблогах, связанных с достоверным событием, уменьшается со временем, тогда как для слухов – нет.

Обнаружение эмоций в социальных сетях

В работе «EmoNet: детальное обнаружение эмоций с помощью управляемых рекуррентных нейронных сетей» [21] Мухаммад Абдул-Магид (Muhammad Abdul-Mageed) и Лайл Унгар (Lyle Ungar) демонстрируют подход к выявлению эмоций в постах социальных сетей, таких как Твиттер. Обнаружение эмоций в социальных сетях играет важную роль, поскольку эмоции помогают определить физическое и психическое здоровье. Способность распознавать эмоции также представляет ценность для бизнеса, поскольку помогает понять клиента. Следовательно, правильное извлечение эмоций из постов в социальных сетях раскрывает родителям физическое/психологическое состояние их детей или помогает бизнесу расти. Однако для методов автоматического распознавания эмоций существуют очевидные препятствия в силу противоречивой природы самих эмоций. Например, когда кто-то говорит «*я люблю понедельники*», это может быть саркастическое замечание, указывающее на ненависть человека к своей работе. И наоборот, кто-то может быть действительно счастлив из-за приятного еженедельного события, происходящего по понедельникам.

Для классификации эмоций авторы использовали *колесо эмоций Плутчика* (Plutchik's wheel of emotions, рис. 11.10), в результате чего получили 24 различные категории. Тем не менее твиты могут использовать различные синонимы для обозначения одного и того же понятия – например, «*счастье*» может быть выражено словами «*радостный*», «*блаженный*» и «*восторженный*». Поэтому авторы использовали синонимы Google и другие ресурсы и обнаружили 665 различных хештегов эмоций, относящихся к 24 основным категориям.

Для сбора данных авторы просканировали посты в Твиттере, начиная с 2009 года, и собрали около 0,5 млрд твитов. Затем они выполнили предварительную обработку исходных данных, главным образом для удаления дубликатов и твитов со множественными эмоциями, и в итоге получили около 1,5 млн твитов. Наконец, авторы воспользовались рекуррентной сетью типа GRU (см. главу 7) для классификации твитов и прогнозирования того, какой тип эмоции выражает данный твит.

Анализ политического наполнения в твитах

Социальные медиа широко используются в качестве платформы для политической деятельности. На недавних выборах в США кандидаты активно использовали Твиттер, чтобы рекламировать свою политическую программу, расширять лагерь сторонников, а также атаковать и контратаковать политических противников. Это подчеркивает важность анализа политических постов для извлечения оперативной информации. Распознавание политического наполнения является одной из таких актуальных и трудных задач. *Политическое наполнение* – это разновидность изощренного манипулирования словами для воздействия на общественное мнение.

В работе «Использование поведенческой и социальной информации для гибкой классификации политического дискурса в Твиттере» [22] Кристен Джонсон (Kristen Johnson) и др. сформировали размеченный набор данных, состоящий из выбранных случайным образом твитов 40 членов конгресса США. Сначала тви-

ты разметили с использованием специально разработанной кодовой таблицы политического наполнения для аннотирования твитов. Далее, из-за динамического характера проблемы, для изучения твитов были использованы *ограниченно обучаемые модели*[1] (weakly supervised model). Подобные модели предназначены для обучения на ограниченном объеме данных, в отличие от моделей глубокого обучения.

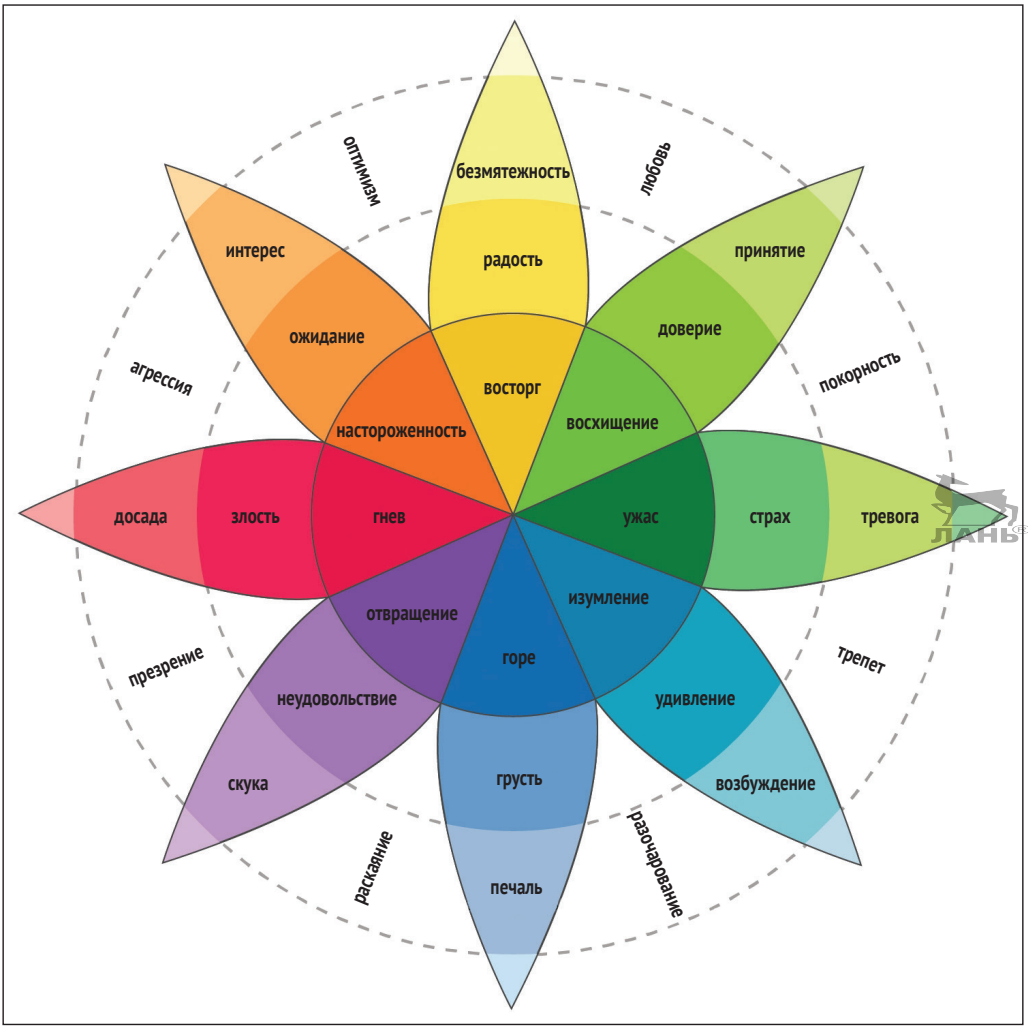


Рис. 11.10 ❖ Колесо эмоций Плутчика

[1] Иногда их именуют неуклюжим термином «*модели с незначительным привлечением учителя*», но в данном случае речь идет именно об ограниченном наборе обучающих данных. – *Прим. перев.*

Новые задачи и вызовы

Теперь мы рассмотрим несколько новых областей применения NLP, которые появились совсем недавно. Эти области включают в себя *обнаружение сарказма*, *языковое обоснование* – т. е. процесс извлечения здравого смысла из естественного языка, а также *поверхностный обзор*, или *скимминг* (skimming), текста.

Обнаружение сарказма

Сарказм – это когда человек произносит что-то, что на самом деле означает противоположность высказыванию (например, фраза «*Как я люблю понедельники!*»). Обнаружение сарказма иногда вызывает затруднение даже у людей, а решение этой проблемы с помощью NLP затруднительно вдвойне. В исследовании «Sarcasm SIGN: интерпретация сарказма с помощью одноязычного машинного перевода на основе настроений» [23] Лотем Пелед (Lotem Peled) и Рой Рейхарт (Roi Reichart) используют технологию NLP для обнаружения сарказма в постах Твиттера. Сначала они создали набор данных из 3000 пар твитов, где один твит – это саркастический твит, а другой – его несаркастическая расшифровка. Расшифровки созданы пятью экспертами-людьми, которые просматривали твиты и искали их реальное значение. Затем они использовали механизм одноязычного машинного перевода для изучения сарказма. Это уже знакомая нам из предыдущей главы модель «последовательность–последовательность». Вместо того чтобы брать пары предложений, принадлежащих двум разным языкам, авторы используют пары твитов с сарказмом и буквальным расшифрованным значением.

Смысловое основание языка

Смысловое основание языка (language grounding) является задачей извлечения здравого смысла из естественного языка. Например, в момент использования языка у нас часто возникает отчетливая мыслительная идея объектов и действий, которые мы хотим объяснить. Это позволяет нам делать различные выводы об объектах, даже если эти выводы непосредственно не присутствуют в предложении. Однако это не относится к машинам. Ведь машины не изучают естественный язык, связывая его с фактическими концептуальными объектами, которые он представляет. Тем не менее эту задачу надо решить, если мы хотим создать полноценный искусственный интеллект. Смысловое основание представляет собой решение этой проблемы. Например, когда мы говорим, что «*машина заехала в гараж*», то нам без слов понятно, что данный гараж больше, чем автомобиль. Однако этот скрытый смысл вряд ли будет изучен алгоритмом машинного обучения, если за него не предусмотрено вознаграждение. В работе «Физика глаголов: относительное физическое знание о действиях и объектах» [24] Максвелл Форбс (Maxwell Forbes) и Ецзинь Чой (Yejin Choi) предлагают оригинальный подход к изучению смысловых основ языка.

В этой статье авторы фокусируются на пяти различных свойствах, или измерениях, на которых основан смысл: размере, весе, прочности, жесткости и скорости. Для изучения различных свойств объектов, появляющихся в разговоре, используется модель *графа показателей* (factor graph). Такой граф содержит подграфы, состоящие из двух подграфов для каждого атрибута – *объектного подграфа* и *глагольного подграфа*.

Далее каждый подграф содержит узлы. Есть два типа узлов:

- **узлы пары объектов** (узлы, найденные в подграфе объекта): они фиксируют относительную силу атрибута для двух объектов, например обозначенных как $O^{размер}_{(человек,ягода)}$: *вероятность размер*(*человек*) > *размер*(*ягода*);
- **узлы рамок действия** (узлы, найденные в подграфе глагола): они фиксируют, как глаголы связаны с атрибутами и обозначаются, например обозначение $F^{размер}_{умещает}$ для предложения «*x умещает y*» показывает, какова вероятность того, что *размер*(*x*) > *размер*(*y*).

Затем можно создать связи (т. е. бинарные факторы) между двумя узлами пары объектов или двумя узлами рамок действия, в зависимости от того, насколько вероятно появление данной пары узлов в аналогичном контексте. Например, пара $O^{размер}_{(человек,мяч)}$ и $O^{размер}_{(человек,камень)}$ должна иметь высокий бинарный фактор, в то время как пара $O^{размер}_{(человек,мяч)}$ и $O^{размер}_{(человек,автомобиль)}$ – низкий бинарный фактор. Затем при помощи изучения неструктурированного естественного языка устанавливаются наиболее важные связи, т. е. соединения между узлами рамок действия и узлами пары объектов.

Наконец, если нам нужно знать взаимосвязь между *вес*(*человек*) и *вес*(*мяч*), мы можем вывести силу соединения, связывающую $F^{размер}_{умещает}$ с $O^{вес}_{(человек,мяч)}$. Это осуществляется с помощью приема, известного как *замкнутое распространение представления* (loopy belief propagation).

Скимминг текста с помощью LSTM

Скимминг, или *поверхностный обзор*, текста играет важную роль во многих областях применения NLP. Например, если LSTM-сеть предназначена для ответа на вопросы о книге, она, вероятно, должна не читать полный текст, а лишь бегло изучить информацию, помогающую ответить на вопросы. Другим применением скимминга может быть поиск документов в большой базе данных по условию наличия точного или приблизительно подходящего фрагмента текста. В работе «Обучение обзорному чтению текста» [25] Адамс Вей Ю (Adams Wei Yu) и др. предлагают модель под названием LSTM-Jump, которая выполняет именно эту работу.

У модели есть три важных гиперпараметра:

- N – общее количество разрешенных скачков (jump);
- R – количество токенов, которые нужно прочитать между двумя скачками;
- K – максимально допустимый размер скачка за раз.

Затем создается LSTM-сеть со слоем softmax с K узлами поверх LSTM. Этот слой softmax решает, сколько скачков сделать на данном такте модели. Подобное функционирование этого слоя softmax чем-то похоже на механизм внимания. Скачки или скимминг прекращаются, если встречается одно из следующих условий:

- softmax выдает ноль;
- LSTM-сеть достигла конца текста;
- количество скачков превышает N.

Новые модели машинного обучения

Теперь мы обсудим несколько более новых моделей машинного обучения, появившихся для разрешения различных ограничений существующих моделей (напри-

мер, стандартных LSTM). Одной из таких моделей являются *фазированные LSTM* (phased LSTM), которые позволяют нам уделять внимание очень специфическим событиям, происходящим в процессе обучения. Другая модель – это *расширенные RNN* (dilated RNN, DRNN), которые обеспечивают возможность моделирования сложных зависимостей, присутствующих во входных данных. DRNN также позволяют выполнять параллельные вычисления развернутых RNN, по сравнению со стандартным механизмом итераций.

Фазированные LSTM

Стандартные LSTM-сети показали прекрасную точность во многих последовательных задачах. Однако они не подходят для обработки данных с нерегулярной синхронизацией, таких как данные, поступающие от управляемых событиями датчиков. Дело в том, что состояние LSTM-ячейки постоянно обновляется вне зависимости от того, поступают ли данные из источника. Такое поведение может привести к пропуску особых событий – редких или нерегулярных, – но тем не менее важных для текущей задачи.

Поэтапные LSTM представлены в публикации «Фазированные LSTM: ускорение обучения рекуррентной сети на длинных или событийных последовательностях» [26] Дэниел Нил и др. Авторы пытаются решить эту проблему, вводя новые *временны́е гейты* (time gate). Обновления состояния ячейки и скрытого состояния разрешены только при открытых временны́х гейтах. Следовательно, если событие не происходит, временные гейты будут закрыты, в результате чего состояние ячейки и скрытое состояние модели останутся неизменными. Такое поведение помогает сохранить информацию в течение более длительного времени и обратить внимание на произошедшее событие. Рисунок 11.11 иллюстрирует общую концепцию временного гейта.

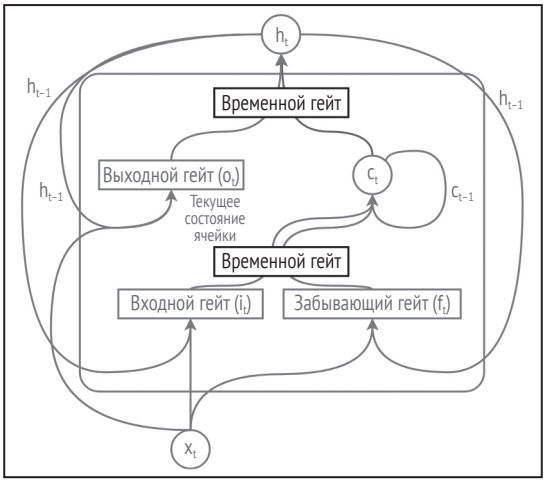


Рис. 11.11 ❖ Общая концепция временного гейта. Скрытое состояние и состояние ячейки разрешено обновлять только в том случае, если включены временные гейты

Операция временного гейта зависит от трех вновь введенных параметров:

- τ – определяет период колебаний в реальном времени;
- r_{on} – определяет полную продолжительность открытого состояния гейта;
- s – определяет сдвиг фазы колебаний гейта.

Эти переменные можно настроить вместе с остальными параметрами LSTM. В TensorFlow уже добавлена реализация фазированных LSTM, которую можно найти по адресу **https://www.tensorflow.org/versions/r1.14/api_docs/python/tf/contrib/rnn/PhasedLSTMCell**.

Расширенные рекуррентные нейронные сети (DRNN)

Текущие RNN имеют ряд ограничений в изучении долгосрочных зависимостей, а именно:

- необходимость обработки сложных зависимостей на одном входе;
- исчезающий градиент;
- невозможность эффективно распараллелить обучение.

В статье «Расширенные рекуррентные нейронные сети» [27] Ши-ю Чан (Shiyu Chang) и др. представили новую архитектуру DRNN, ориентированную на устранение всех упомянутых ограничений сразу.

DRNN решает проблему изучения сложных зависимостей, гарантируя, что текущее состояние связано со старыми скрытыми состояниями, а не только с непосредственным предыдущим скрытым состоянием. Это позволяет более точно изучить долгосрочные зависимости.

Предложенная архитектура решает проблему исчезающего градиента, поскольку текущее скрытое состояние видит прошлые шаги за пределами непосредственно предшествующего скрытого состояния, поэтому удается легко распространять градиент во времени на большие расстояния.

Если мысленно сжать архитектуру DRNN, ее можно представить в виде стандартной RNN, которая обрабатывает несколько входов одновременно. Поэтому, опять-таки, по замыслу авторов, сети DRNN допускают бóльшую параллелизацию, по сравнению со стандартными сетями RNN. На рис. 11.12 показано, как DRNN отличаются от стандартных RNN. Исходный код реализации DRNN можно скачать по адресу **https://github.com/code-terminator/DilatedRNN**.

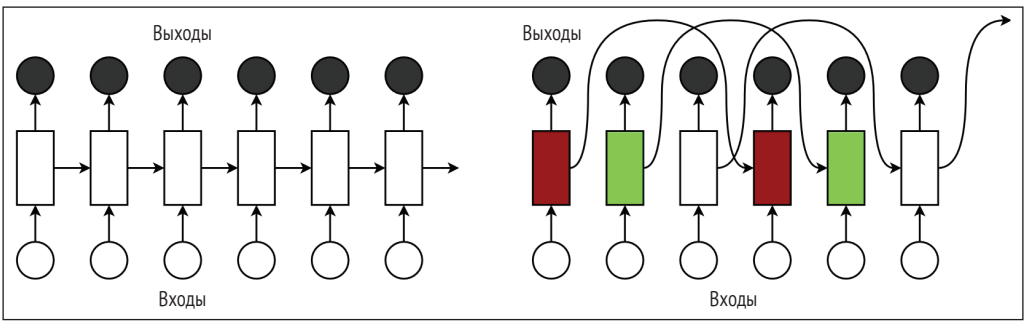


Рис. 11.12 ❖ Развернутые во времени стандартная RNN (слева) и DRNN (справа). Расширенные RNN, помеченные разными цветами, могут обрабатываться параллельно, поскольку они не имеют общих связей

Заключение

Эта глава была посвящена изучению текущих тенденций и перспективных направлений развития NLP. Это очень широкая тема, и мы ограничились обсуждением лишь некоторых из самых последних достижений в области обработки естественного языка. Сначала мы рассмотрели улучшения в области представления слов. Вы узнали, что существуют гораздо более точные представления с более богатыми интерпретациями, например вероятностные. Затем мы рассмотрели улучшения и достижения в машинном переводе, поскольку это одно из самых востребованных применений NLP. С каждым годом появляются более совершенные механизмы внимания и улучшенные модели, способные производить все более реалистичные переводы.

Затем мы рассмотрели некоторые новые исследования в области NLP (в основном за 2017 год). Сначала мы исследовали проникновение NLP в другие области: компьютерное зрение, обучение с подкреплением и состязательные генеративные сети. Мы наблюдали за последовательным улучшением систем NLP, которые шаг за шагом движутся в направлении искусственного общего интеллекта. Затем мы познакомились с достижениями NLP в социальных сетях, такими как обнаружение и опровержение слухов, выявление эмоций и анализ политических манипуляций.

Мы также рассмотрели некоторые новые и интересные задачи, набирающие все большую популярность среди сообщества NLP, такие как обнаружение сарказма с использованием модели кодер–декодер, смысловое основание, т. е. понимание неявного смысла высказывания, а также навык беглого обзора текста (скимминг). Мы обсудили некоторые новые модели машинного обучения. Фазированные LSTM – это улучшенная модификация LSTM, обладающая более глубоким контролем над обновлением состояния ячейки и скрытого состояния системы. Такое поведение позволяет LSTM изучать нерегулярные долгосрочные зависимости. Наконец, мы обсудили модели под названием DRNN, представляющие собой расширенный вариант развертывания стандартной RNN. Благодаря этой модификации сети DRNN могут моделировать сложные зависимости, решают проблему исчезающего градиента и обеспечивают распараллеливание обработки данных.

Литература

[1] Distributed representations of words and phrases and their compositionality, T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, Advances in Neural Information Processing Systems, p. 3111–3119, 2013.

[2] Semi-supervised convolutional neural networks for text categorization via region embedding, Johnson, Rie and Tong Zhang, Advances in Neural Information Processing Systems, pp. 919–927, 2015.

[3] A Generative Word Embedding Model and Its Low Rank Positive Semidefinite Solution, Li, Shaohua, Jun Zhu, and Chunyan Miao, Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, p. 1599–1609, 2015.

[4] Learning Word Meta-Embeddings, Wenpeng Yin and Hinrich Schütze, Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, vol. 1, p. 1351–1360, 2016.

[5] Topical Word Embeddings, Yang Liu, Zhiyuan Liu, Tat-Seng Chua, and Maosong Sun, AAAI, p. 2418–2424, 2015.

[6] Effective Approaches to Attention-based Neural Machine Translation, Thang Luong, Hieu Pham, and Christopher D. Manning, Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, p. 1412–1421, 2015.

[7] CKY-based Convolutional Attention for Neural Machine Translation, Watanabe, Taiki, Akihiro Tamura, and Takashi Ninomiya, Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers), vol. 2, p. 1–6, 2017.

[8] Neural Machine Translation, Minh-Thang Luong. Stanford University, 2016.

[9] Exploring Models and Data for Image Question Answering, Ren, Mengye, Ryan Kiros, and Richard Zemel, Advances in Neural Information Processing Systems, p. 2953–2961, 2015.

[10] Learning to Reason: End-to-End Module Networks for Visual Question Answering, Hu, Ronghang, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko, CoRR, abs/1704.05526 3, 2017.

[11] VQA: Visual Question Answering, Antol, Stanislaw, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh, Computer Vision (ICCV), 2015 IEEE International Conference on, p. 2425–2433, IEEE, 2015.

[12] Show, Attend and Tell: Neural Image Caption Generation with Visual Attention, Xu, Kelvin, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio, International Conference on Machine Learning, p. 2048–2057, 2015.

[13] Multi-agent cooperation and the emergence of (natural) language, Lazaridou, Angeliki, Alexander Peysakhovich, and Marco Baroni, International Conference on Learning Representations, 2016.

[14] Towards End-to-End Reinforcement Learning of Dialogue Agents for Information Access, Dhingra, Bhuwan, Lihong Li, Xiujun Li, Jianfeng Gao, Yun-Nung Chen, Faisal Ahmed, and Li Deng, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, p. 484–495, 2017.

[15] A Network-based End-to-End Trainable Task-oriented Dialogue System, Wen, Tsung-Hsien, David Vandyke, Nikola Mrksic, Milica Gasic, Lina M. Rojas-Barahona, Pei-Hao Su, Stefan Ultes, and Steve Young, arXiv:1604.04562v3, 2017.

[16] Generating Text via Adversarial Training, Zhang, Yizhe, Zhe Gan, and Lawrence Carin, NIPS workshop on Adversarial Training, vol. 21, 2016.

[17] SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient, Yu, Lantao, Weinan Zhang, Jun Wang, and Yong Yu, AAAI, p. 2852–2858, 2017.

[18] One Model To Learn Them All, Kaiser, Lukasz, Aidan N. Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit, arXiv:1706.05137v1, 2017.

[19] A Joint Many-Task Model: Growing a Neural Network for Multiple NLP Tasks, Hashimoto, Kazuma, Yoshimasa Tsuruoka, and Richard Socher, Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, p. 1923–1933, 2017.

[20] Detect Rumors Using Time Series of Social Context Information on Microblogging Websites, Ma, Jing, Wei Gao, Zhongyu Wei, Yueming Lu, and Kam-Fai Wong, Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, p. 1751–1754, ACM, 2015.

[21] Emonet: Fine-Grained Emotion Detection with Gated Recurrent Neural Networks, Abdul-Mageed, Muhammad, and Lyle Ungar, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, p. 718–728, 2017.

[22] Leveraging Behavioral and Social Information for Weakly Supervised Collective Classification of Political Discourse on Twitter, Johnson, Kristen, Di Jin, and Dan Goldwasser, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, p. 741–752, 2017.

[23] Sarcasm SIGN: Interpreting Sarcasm with Sentiment Based Monolingual Machine Translation, Peled, Lotem, and Roi Reichart, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, p. 1690–1700, 2017.

[24] Verb Physics: Relative Physical Knowledge of Actions and Objects, Forbes, Maxwell, and Yejin Choi, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, p. 266–276, 2017.

[25] Learning to Skim Text, Yu, Adams Wei, Hongrae Lee, and Quoc Le, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, p. 1880–1890, 2017.

[26] Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences, Neil, Daniel, Michael Pfeiffer, and Shih-Chii Liu, Advances in Neural Information Processing Systems, p. 3882–3890, 2016.

[27] Dilated recurrent neural networks, Chang, Shiyu, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark A. Hasegawa-Johnson, and Thomas S. Huang, Advances in Neural Information Processing Systems, p. 76–86, 2017

Приложение

Математические основы и углубленное изучение TensorFlow

В приложении мы подробнее рассмотрим некоторые концепции, которые будут полезны для понимания материала книги. Сначала обсудим несколько математических структур данных, упоминаемых на протяжении всей книги, а затем приведем описание различных операций, выполняемых с этими структурами. Далее обзорно рассмотрим вероятности, играющие жизненно важную роль в машинном обучении, поскольку они обычно дают представление о том, насколько неопределенным является прогноз модели. После этого вы познакомитесь с высокоуровневой библиотекой Keras и узнаете о том, как реализовать нейронный машинный переводчик с помощью библиотеки seq2seq. Заключительной темой приложения станет руководство по использованию TensorBoard в качестве инструмента визуализации представлений слов.

Основные структуры данных

Скаляр

Скаляр – это величина, которая, в отличие от матрицы или вектора, может быть выражена одиночным числом. Например, `1,3` – это скаляр. Скаляр n можно математически обозначить следующим образом:

$$n \in R,$$

где R – пространство вещественных чисел.

Векторы

Вектор – в машинном обучении это массив вещественных чисел. В отличие от набора, в котором нет порядка элементов, вектор имеет определенный порядок элементов. Примером вектора является `[1.0, 2.0, 1.4, 2.3]`. Математически это можно обозначить следующим образом:

$$a = (a_0, a_1, \ldots, a_{\{n-1\}}),$$
$$a \in R^n,$$

или можно записать иначе:

$$a \in R^{n \times 1}.$$

Здесь R – пространство вещественных чисел и n – количество элементов вектора.

Матрицы

Матрица может рассматриваться как упорядоченное двумерное размещение набора скаляров. Другими словами, матрица может рассматриваться как вектор векторов. Пример матрицы может выглядеть так:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 4 & 2 & 3 \\ 2 & 7 & 7 & 1 \\ 5 & 6 & 9 & 0 \end{pmatrix}.$$

В более общем виде матрица размера $m \times n$ может быть математически определена следующим образом:

$$\boldsymbol{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix},$$

а также:

$$\boldsymbol{A} = R^{m \times n}.$$

Здесь m – количество строк в матрице, n – количество столбцов в матрице, а R – пространство вещественных чисел.

Индексы матрицы

Мы будем использовать нотацию с нулевой индексацией, т. е. индексы элементов начинаются с 0. Чтобы индексировать отдельный элемент из матрицы в (i,j)-й позиции, мы используем следующие обозначения:

$$\boldsymbol{A}_{i,j} = a_{i,j}.$$

Обращаясь к предыдущему примеру матрицы

$$\boldsymbol{A} = \begin{pmatrix} 1 & 4 & 2 & 3 \\ 2 & 7 & 7 & 1 \\ 5 & 6 & 9 & 0 \end{pmatrix},$$

мы можем указать на определенный элемент матрицы так:

$$\boldsymbol{A}_{1,0} = 2.$$

Указать на определенную строку матрицы можно так:

$A_{1,:} = (2, 7, 1, 1)$.

Обозначим *срез*, начиная с (i,k)-го индекса до (j,l)-го индекса любой матрицы $\boldsymbol{A}$, как показано здесь:

$$\boldsymbol{A}_{i:j,k:l} = \begin{pmatrix} a_{i,k} & \cdots & a_{i,l} \\ \vdots & \ddots & \vdots \\ a_{j,k} & \cdots & a_{j,l} \end{pmatrix}.$$

В нашем примере матрицы мы можем обозначить срез от первого ряда третьего столбца до второго ряда четвертого столбца, как показано здесь:

$$\boldsymbol{A}_{0:1,2:3} = \begin{pmatrix} 2 & 3 \\ 7 & 1 \end{pmatrix}.$$

Специальные типы матриц

Тождественная матрица

Тождественная (*единичная*) матрица – это матрица, элементы которой равны 1 на диагонали матрицы и 0 в остальных местах. Математически это условие можно записать следующим образом:

$$I_{i,j} = \begin{pmatrix} 1, & \text{если } i = j \\ 0 & \text{остальные} \end{pmatrix}.$$

Иными словами, тождественная матрица в общем виде выглядит так:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Здесь $I \in R^{n \times n}$.

Тождественная матрица I имеет следующее полезное свойство при умножении на другую матрицу A:

$AI = A$.

Диагональная матрица

Диагональная матрица является более общим случаем единичной матрицы, где значения по диагонали могут принимать любое значение, а остальные значения равны нулю:

$$\boldsymbol{A} = \begin{pmatrix} a_{0,0} & 0 & \cdots & 0 \\ 0 & a_{1,1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} \end{pmatrix}.$$

Тензоры

N-мерная матрица называется *тензором*. Другими словами, мы можем назвать тензором матрицу с произвольным числом измерений. Например, четырехмерный тензор может быть записан, как показано здесь:

$$T \in R^{k \times l \times m \times n},$$

где R – пространство вещественных чисел.

Тензорные и матричные операции

Транспонирование

Транспонирование является важной операцией, определенной для матриц и тензоров. Для матрицы транспонирование определяется следующим образом:

$$(A_{i,j})^T = A_{j,i}.$$

Здесь $\boldsymbol{A}^T$ обозначает результат транспонирования матрицы $\boldsymbol{A}$.

Рассмотрим пример транспонирования матрицы:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 4 & 2 & 3 \\ 2 & 7 & 7 & 1 \\ 5 & 6 & 9 & 0 \end{pmatrix}.$$

После транспонирования получаем:

$$\boldsymbol{A}^T = \begin{pmatrix} 1 & 2 & 5 \\ 4 & 7 & 6 \\ 2 & 7 & 9 \\ 3 & 1 & 0 \end{pmatrix}.$$

Для тензора транспонирование можно рассматривать как перестановку порядка размерностей. Например, давайте определим тензор S, как показано здесь:

$$S \in R^{d_1, d_2, d_3, d_4}.$$

Теперь операция транспонирования (одна из многих возможных) может быть определена следующим образом:

$$S^T \in R^{d_4, d_3, d_2, d_1}.$$

Умножение

Умножение матриц – это еще одна важная операция, которая часто встречается в линейной алгебре.

Пусть у нас есть матрицы $A \in R^{m \times n}$ и $B \in R^{n \times p}$, тогда операцию перемножения можно записать как

$$C = AB,$$

где $C = R^{m \times p}$.

Рассмотрим пример:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 \\ 4 & 5 \\ 7 & 8 \end{pmatrix},$$

$$\boldsymbol{B} = \begin{pmatrix} 8 & 5 & 2 \\ 9 & 6 & 3 \end{pmatrix}.$$

Выполнив перемножение $C = AB$, получим следующую матрицу:

$$C = \begin{pmatrix} 26 & 17 & 8 \\ 77 & 50 & 23 \\ 128 & 83 & 38 \end{pmatrix}.$$

Поэлементное умножение

Поэлементное умножение матриц, или *произведение Адамара* (Hadamard product), вычисляется для матриц, имеющих одинаковую размерность. Пусть у нас есть матрицы $A \in R^{m \times n}$ и $B \in R^{m \times n}$. Операция поэлементного умножения матриц обозначается следующим образом:

$$C = A \circ B,$$

где $C \in R^{m \times n}$.

Рассмотрим пример:

$$A = \begin{pmatrix} 2 & 3 \\ 1 & 2 \\ 6 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 2 \\ 1 & 3 \\ 3 & 5 \end{pmatrix}.$$

В результате поэлементного умножения получим матрицу C:

$$C = \begin{pmatrix} 6 & 6 \\ 1 & 6 \\ 18 & 5 \end{pmatrix}.$$

Обратная матрица

Матрица, *обратная* по отношению к матрице A, обозначается как A^{-1} и удовлетворяет условию:

$$A^{-1}A = I.$$

Обратная матрица очень полезна при решении системы линейных уравнений. Рассмотрим следующий пример:

$$Ax = b.$$

Мы можем найти x при помощи обратной матрицы:

$$A^{-1}(Ax) = A^{-1}b.$$

Это выражение можно переписать как $(A^{-1}Ax) = A^{-1}b$ на основании закона ассоциативности $A(BC) = (AB)C$. Отсюда мы получаем $Ix = A^{-1}b$, потому что $A^{-1}A = I$, где I – единичная матрица.

Наконец, $x = A^{-1}b$, поскольку $Ix = x$.

Например, полиномиальная регрессия, один из методов регрессии, использует линейную систему уравнений для решения задачи регрессии. Регрессия аналогична классификации, но вместо вывода класса регрессионные модели выводят непрерывное значение. Давайте рассмотрим пример: учитывая количество спален в доме, необходимо рассчитать стоимость дома в каталоге недвижимости. Формально проблему полиномиальной регрессии можно записать следующим образом:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_m x_i^m + \varepsilon_i \quad (i = 1, 2, \ldots, n).$$

Здесь x_i – это i-й вход и y_i – метка этого входа, а ε_i – шум данных. В нашем примере x – это количество спален, а y – стоимость дома. Данные можно записать в виде системы линейных уравнений:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix}.$$

Однако A^{-1} существует не для всех A. Для существования обратной матрицы должны выполняться определенные условия. Например, матрица A должна быть квадратной, т. е. $R^{n \times n}$. И даже когда обратная матрица существует, мы не всегда можем найти ее в закрытом виде; иногда обратную матрицу можно только аппроксимировать на компьютерах с конечной точностью. Ниже мы рассмотрим несколько способов нахождения обратной матрицы.

Когда говорят, что A должна быть квадратной матрицей, как условие существования обратной матрицы, то имеют в виду стандартную инверсию. Существуют варианты операции обращения, например *инверсия Мура–Пенроуза*, также известная как псевдообратная матрица, которая может выполняться для матриц произвольного размера $m \times n$.

Нахождение обратной матрицы – сингулярное разложение (SVD)

Давайте теперь посмотрим, как мы можем использовать *сингулярное разложение* (singular value decomposition, SVD), чтобы найти обратную матрицу A. Операция SVD разлагает A на три разные матрицы следующим образом:

$$A = UDV^T.$$

Здесь столбцы U известны как *левые сингулярные векторы*, столбцы V – как *правые сингулярные векторы*, а диагональные значения D – как *сингулярные числа*, или *особые значения*. Левые сингулярные векторы являются собственными векторами матрицы AA^T, а правые сингулярные векторы – собственными векторами матрицы A^TA. Наконец, сингулярные числа являются квадратными корнями из собственных значений AA^T и A^TA. Собственный вектор и соответствующее ему собственное значение квадратной матрицы A удовлетворяют следующему условию:

$$Av = \lambda v.$$

Тогда, если SVD существует, обратная величина A определяется следующим образом:

$$A^{-1} = VD^{-1}U^T.$$

Поскольку D является диагональной матрицей, D^{-1} является просто поэлементной обратной величиной ненулевых элементов D. SVD является важной техникой матричной факторизации, которая часто встречается в машинном обучении. Например, SVD используется для расчета *анализа основных компонентов* (principal component analysis, PCA), который является популярным методом уменьшения размерности данных (цель, аналогичная цели t-SNE, которую мы видели в главе 4). Другое, более ориентированное на NLP приложение SVD – ранжирование документов. То есть когда вы хотите получить наиболее релевантные документы и ранжировать их по релевантности тому или иному термину, например футболу, для достижения этой цели можно использовать SVD.

Нормы

Норма используется как мера «значимости» матрицы (например, когда надо ответить на вопрос, какая из матриц «больше»). p-я норма рассчитывается и обозначается, как показано здесь:

$$\|A\|_p = \left(\sum_i |A_i|^p\right)^{1/p}.$$

Например, норма L2 будет такой:

$$\|A\|_2 = \sqrt{\sum_i |A_i|^2}.$$

Определитель

Определитель (determinant) квадратной матрицы, обозначаемый как $det(A)$, является произведением всех собственных значений матрицы. Определитель очень полезен во многих отношениях. Например, матрица A является обратимой тогда и только тогда, когда определитель отличен от нуля. Следующее уравнение показывает расчеты для определителя матрицы 3×3:

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix} =$$

$$= a(ei - fh) - b(di - fg) + c(dh - eg) = aei + bfg + cdh - ceg - bdi - afh.$$

Вероятность

Далее мы обсудим терминологию, связанную с вероятностями. Теория вероятностей является важной частью машинного обучения, так как моделирование данных с помощью вероятностных моделей позволяет нам сделать выводы о том, насколько неопределенной является модель в отношении некоторых предсказаний. Вспомним пример, когда мы выполнили анализ настроений в главе 11, где выходом была классификация «положительный/отрицательный» для заданного отзыва на фильм. Хотя модель выдает некоторое значение между 0 и 1 (0 для абсолютно отрицательного и 1 для абсолютно положительного), мы не знаем, насколько неопределенным является ее ответ.

Давайте разберемся, как неопределенность помогает нам делать лучшие прогнозы. Например, детерминистическая модель может неправильно сказать, что положительность обзора *Я никогда не терял интереса* составляет 0,25 (то есть, скорее всего, это будет отрицательный комментарий). Однако вероятностная модель даст среднее значение и стандартное отклонение для прогноза. Например, будет сказано, что этот прогноз имеет среднее значение 0,25 и стандартное отклонение 0,5. Со второй моделью мы знаем, что прогноз может быть неверным из-за высокого стандартного отклонения. Однако в детерминированной модели у нас нет этой роскоши. Это свойство особенно ценно для критических машинных систем (например, модель оценки риска терроризма).

Для разработки таких вероятностных моделей машинного обучения (например, байесовской логистической регрессии, байесовских нейронных сетей или гауссовых процессов) вы должны быть знакомы с базовой теорией вероятностей. Поэтому мы предоставляем некоторую базовую информацию о вероятности здесь.

Случайные величины

Случайная величина – это переменная, которая может принимать случайное значение. Также случайные величины представлены как x1, x2 и т. д. Случайные величины могут быть двух типов: дискретные и непрерывные.

Дискретные случайные величины

Дискретная случайная величина – это переменная, которая может принимать дискретные случайные значения. Например, попытки подбрасывания монеты можно смоделировать как случайную переменную, то есть сторона монеты, на которую она приземляется при подбрасывании монеты, является дискретной переменной, поскольку значения могут быть только головами или хвостами. В качестве альтернативы, значение, которое вы получаете при броске кубика, также является дискретным, поскольку значения могут быть получены только из набора {1,2,3,4,5,6}.

Непрерывные случайные величины

Непрерывная случайная величина – это переменная, которая может принимать любое действительное значение, то есть если x является непрерывной случайной величиной:

$$x \in R.$$

Здесь R – пространство вещественных чисел.

Например, рост человека – это непрерывная случайная величина, поскольку она может принимать любое вещественное значение.

Функция вероятности масса/плотность

Функция *распределения масс* (probability mass function, PMF), или функция *плотности распределения вероятности* (probability density function, PDF), – это способ показать распределение вероятности по различным значениям, которые может принимать случайная величина. Для дискретных переменных определен PMF, а для непрерывных переменных – PDF. На рис. П.1 показан пример PMF.

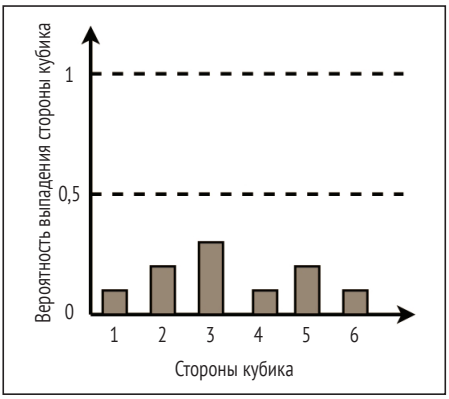


Рис. П.1 ❖ Функция распределения масс (дискретная)

Предыдущий PMF может быть достигнут с помощью смещенной матрицы. На этом графике мы можем видеть, что существует высокая вероятность получения 3 с этим кубиком. Такой график можно получить, запустив ряд испытаний (скажем, 100), а затем посчитав, сколько раз каждая грань упала на вершину. Наконец, разделите каждый счет на количество испытаний, чтобы получить

нормированные вероятности. Обратите внимание, что все вероятности должны составлять до 1, как показано здесь:

$P(X \in \{1, 2, 3, 4, 5, 6\}) = 1.$

Эта же концепция распространяется на непрерывную случайную величину для получения PDF. Скажем, мы пытаемся смоделировать вероятность определенной высоты для данной популяции. В отличие от дискретного случая, у нас есть не отдельные значения для вычисления вероятности, а скорее непрерывный спектр значений (в примере он простирается от 0 до 2,4 м). Если мы хотим нарисовать график для этого примера, подобный тому, что показан на рис. П.1, нам нужно думать об этом в терминах бесконечно малых ячеек. Например, мы обнаруживаем, что плотность вероятности роста человека составляет от 0,0 до 0,01 м, от 0,01 до 0,02 м, ..., от 1,8 до 1,81 м, ... и т. д. Плотность вероятности можно рассчитать по следующей формуле:

$$\text{Плотность вероятности для } bin_i = \frac{\text{вероятность нахождения роста в } bin_i}{\text{размер } bin_i}.$$

Затем мы построим эти столбцы близко друг к другу, чтобы получить непрерывную кривую, как показано на рис. П.2. Обратите внимание, что плотность вероятности для данного бина может быть больше 1 (поскольку это плотность), но площадь под кривой должна быть 1.

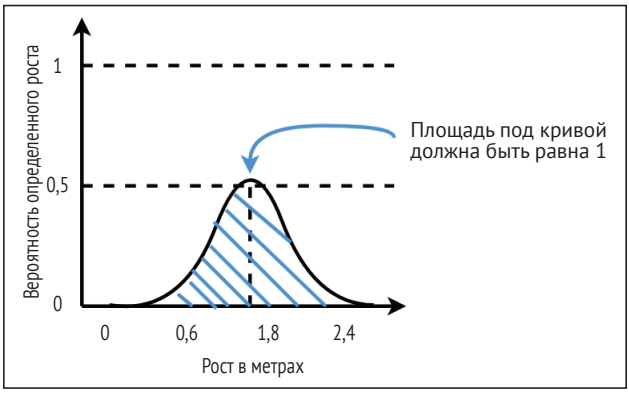


Рис. П.2 ❖ Функция распределения плотности вероятности (непрерывная)

Форма, показанная на рис. А.2, называется нормальным (или *гауссовым*) распределением. Это также называют кривой колокола. Ранее мы дали только интуитивное объяснение того, как думать о непрерывной функции плотности вероятности. Более формально, непрерывный PDF нормального распределения имеет уравнение и определяется следующим образом. Предположим, что непрерывная случайная величина X имеет нормальное распределение со средним значением μ и стандартным отклонением. Вероятность $X = x$ для любого значения x определяется по следующей формуле:

$$P(X = x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Вы должны получить площадь (которая должна быть 1 для действительного PDF), если интегрируете эту величину во все возможные бесконечно малые значения dx, как обозначено этой формулой:

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx.$$

Интеграл от нормали для произвольных значений a, b задается следующей формулой:

$$\int_{-\infty}^{\infty} e^{-a(x+b)^2} dx = \sqrt{\frac{\pi}{a}}.$$

(Вы можете найти развернутую информацию на сайте **http://mathworld.wolfram.com/GaussianIntegral.html**, или менее сложное изложение по адресу **https://en.wikipedia.org/wiki/Gaussian_integral**.)

Исходя из вышесказанного, мы можем взять интеграл нормального распределения, где $a = 1/2\sigma^2$ и $b = -\mu$:

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \frac{1}{\sqrt{2\pi\sigma^2}} \sqrt{\frac{\pi}{1/2\sigma^2}} = \frac{1}{\sqrt{2\pi\sigma^2}} \sqrt{\sqrt{2\pi\sigma^2}} = 1.$$

Это дает накопление всех значений вероятности для всех значений x и дает вам значение 1.

Условная вероятность

Условная вероятность представляет собой вероятность наступления события, учитывая возникновение другого события. Например, для двух случайных величин, X и Y, условная вероятность $X = x$, при условии что $Y = y$, обозначается следующей формулой:

$$P(X = x|Y = y).$$

Реальный пример такой вероятности может быть следующим:

$$P(\textit{Боб ходит в школу} = \textit{да} \mid \textit{идет дождь} = \textit{да}).$$

Совместная вероятность

Для двух случайных величин, X и Y, мы будем ссылаться на вероятность $X = x$ вместе с $Y = y$ как на *совместную вероятность* $X = x$ и $Y = y$. Это обозначается следующей формулой:

$$P(X = x, Y = y) = P(X = x)P(Y = y \mid X = x).$$

Если X и Y являются взаимоисключающими событиями, это выражение сводится к такому:

$$P(X = x, Y = y) = P(X = x)P(Y = y).$$

Реальный пример такой вероятности может быть следующим:

$$P(\textit{дождливо=да}, \textit{прогулка=да}) = P(\textit{дождливо=да})P(\textit{прогулка=да} \mid \textit{дождливо=да}).$$

Предельная вероятность

Предельное распределение вероятностей – это распределение вероятностей подмножества случайных величин, учитывая совместное распределение вероятностей всех переменных. Например, предположим, что существуют две случайные величины, X и Y, и мы уже знаем $P(X = x, Y = y)$, и мы хотим вычислить $P(x)$:

$$P(X = x) = \sum_{\forall y'} P(X = x, Y = y').$$

Интуитивно мы возьмем сумму по всем возможным значениям Y, эффективно делая вероятность $Y = 1$. Это дает нам $P(X = x, Y = 1) = P(X = x)$.

Правило Байеса

Правило Байеса позволяет найти $P(Y = y \mid X = x)$, если известны вероятности $P(X = x \mid Y = y)$, $P(X = x)$ и $P(Y = y)$. Мы можем с легкостью вывести правило Байеса следующим образом:

$$P(X = x, Y = y) = P(X = x)P(Y = y \mid X = x) = P(Y = y)P(X = x \mid Y = y).$$

Теперь давайте возьмем среднюю и правую части:

$$P(X = x)P(Y = y \mid X = x) = P(Y = y)P(X = x \mid Y = y);$$

$$P(Y = y \mid X = x) = \frac{P(X = x \mid Y = y)P(Y = y)}{P(X = x)}.$$

Это правило Байеса. Его можно записать проще:

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

Введение в Keras

Здесь мы дадим краткое введение в Keras – библиотеку TensorFlow, которая предоставляет функции более высокого уровня для реализации алгоритмов глубокого обучения. Keras использует базовые операции TensorFlow; тем не менее она обеспечивает API высокого уровня, более удобный для начинающих пользователей. В качестве ознакомительного примера использования Keras мы рассмотрим создание сверточной нейросети. Полное упражнение можно найти в файле `keras_cnn.ipynb`, расположенном в папке `appendix`.

Сначала мы решим, какой тип модели будем использовать. Keras имеет два разных API: последовательный и функциональный. Последовательный API проще и позволяет создавать модель слой за слоем. Однако последовательный API имеет ограниченную гибкость при проектировании структуры и связей между уровнями сети. С другой стороны, функциональный API обладает гораздо большей гибкостью и позволяет разрабатывать конкретные детали нейронной сети. В демонстрационных целях мы создадим CNN с использованием последовательного API Keras. Последовательная модель в этом случае представляет собой последовательность стеков слоев (например, входной слой, слой свертки и слой подвыборки):

```
model = Sequential()
```

Далее мы последовательно определяем слои нашей нейросети. Сначала создадим слой свертки с 32 фильтрами, размером ядра 3×3 и нелинейностью ReLU. Этот слой будет принимать входные данные размером 28×28×1 (т. е. размер изображения MNIST):

```
model.add(Conv2D(32, 3, activation='relu', input_shape=[28, 28, 1]))
```

Далее мы определим слой max-pooling. Если размер ядра и шаг не определены, по умолчанию они равны 2 (размер ядра) и 1 (шаг):

```
model.add(MaxPool2D())
```

Затем добавим слой пакетной нормализации:

```
model.add(BatchNormalization())
```

Слой пакетной нормализации[1] нормализует (т. е. приводит к нулевому среднему и единичной дисперсии) выходы предыдущего слоя. Это дополнительный шаг, применяемый для повышения качества CNN, особенно в приложениях компьютерного зрения. Обратите внимание, что мы не использовали пакетную нормализацию в упражнениях, так как пакетная нормализация не очень интенсивно применяется в задачах NLP, по сравнению с задачами компьютерного зрения.

Далее мы добавим еще два слоя свертки, затем слой max-pooling и слой пакетной нормализации:

```
model.add(Conv2D(64, 3, activation='relu'))
model.add(MaxPool2D())
model.add(BatchNormalization())
model.add(Conv2D(128, 3, activation='relu'))
model.add(MaxPool2D())
model.add(BatchNormalization())
```

Потом мы добавляем сглаживание данных, поскольку это необходимо для подачи выходных данных в полностью связанный слой:

```
model.add(Flatten())
```

Затем добавим полностью связанный слой с `256` скрытыми ячейками, активацию ReLU и окончательный выходной слой softmax с `10` ячейками softmax (т. е. для 10 различных классов MNIST):

---

[1] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Ioffe and Szegedy, International Conference on Machine Learning, 2015.

```
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Наконец, мы *скомпилируем* модель, при этом скажем Keras использовать Adam в качестве оптимизатора и категориальную кросс-энтропию в качестве метрики точности модели:

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

После того как модель, потери и оптимизатор определены, мы можем запустить модель Keras.

Для обучения модели используйте следующую команду:

```
model.fit(x_train, y_train, batch_size = batch_size)
```

Здесь `x_train` и `y_train` – данные обучения, а `batch_size` определяет размер пакета. После запуска в терминале будет отображаться ход обучения.

Чтобы оценить модель, используйте следующую команду:

```
test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
```

Эта строка снова выведет индикатор выполнения, а также ошибку тестирования и точность каждой эпохи.

Введение в библиотеку TensorFlow seq2seq

Мы использовали «сырой» API TensorFlow для всех предыдущих упражнений в этой книге для более глубокого понимания функциональности и устройства моделей. Однако TensorFlow имеет различные библиотеки высокого уровня, скрывающие все мелкие детали реализаций. Подобные библиотеки помогают пользователям создавать модели типа «последовательность–последовательность», такие как модель нейронного машинного перевода, которую мы обсуждали в главе 10, с меньшим количеством строк кода и не беспокоясь о технических деталях. Знание этих библиотек важно, так как они предоставляют гораздо более прозрачный способ использования этих моделей в прикладном коде или в исследованиях за пределами существующих методов. Поэтому мы кратко расскажем, как использовать библиотеку TensorFlow `seq2seq`. Обсуждаемый код доступен в качестве упражнения в файле `seq2seq_nmt.ipynb`.

Определение вложений для кодера и декодера

Сначала мы объявим заполнители входа кодера, входа декодера и выхода декодера:

```
enc_train_inputs = []
dec_train_inputs, dec_train_labels = [],[]
for ui in range(source_sequence_length):
    enc_train_inputs.append(tf.placeholder(tf.int32, shape=[batch_
size],name='train_inputs_%d'%ui))

for ui in range(target_sequence_length):
```

```
    dec_train_inputs.append(tf.placeholder(tf.int32, shape=[batch_
size],name='train_inputs_%d'%ui))
    dec_train_labels.append(tf.placeholder(tf.int32, shape=[batch_
size],name='train_outputs_%d'%ui))
```

Далее объявим функцию перебора представлений для всех входов кодера и декодера:

```
encoder_emb_inp = [tf.nn.embedding_lookup(encoder_emb_layer, src) for
src in enc_train_inputs]
encoder_emb_inp = tf.stack(encoder_emb_inp)

decoder_emb_inp = [tf.nn.embedding_lookup(decoder_emb_layer, src) for
src in dec_train_inputs]
decoder_emb_inp = tf.stack(decoder_emb_inp)
```

Объявление кодера

Кодер создается из ячеек LSTM в качестве основного строительного блока. Далее мы объявляем слой `dynamic_rnn`, который принимает определенную ячейку LSTM в качестве входных данных, а начальное состояние инициализируется нулями. Для параметра `time_major` установим значение `True`, поскольку наши данные имеют временную ось в качестве первой оси (т. е. оси 0). Другими словами, наши данные имеют форму `[sequence_length, batch_size, embeddings_size]`, где зависящая от времени `sequence_length` находится на первой оси. Преимущество `dynamic_rnn` заключается в способности обрабатывать входы динамического размера. Вы можете использовать необязательный аргумент `sequence_length`, чтобы определить длину каждого предложения в пакете. Например, представьте, что у вас есть партия размером `[3,30]` с тремя предложениями длиной `[10,20,30]` (обратите внимание, что мы дополняем короткие предложения до 30 специальным токеном). Передача тензора со значениями `[10,20,30]` в качестве `sequence_length` приведет к обнулению выходных данных LSTM, которые вычисляются за пределами длины каждого предложения. Что касается состояния ячейки, оно не обнуляется, а берет последнее состояние ячейки, вычисленное в пределах длины предложения, и копирует это значение за пределы длины предложения, пока не будет достигнуто значение 30:

```
encoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

initial_state = encoder_cell.zero_state(batch_size, dtype=tf.float32)

encoder_outputs, encoder_state = tf.nn.dynamic_rnn(
    encoder_cell, encoder_emb_inp, initial_state=initial_state,
    sequence_length=[source_sequence_length for _ in range(batch_
size)],
    time_major=True, swap_memory=True)
```

Опция `swap_memory` позволяет TensorFlow обмениваться тензорами, созданными в процессе логического вывода между GPU и CPU, в случае если модель слишком сложна, чтобы полностью поместиться в GPU.

Объявление декодера

Декодер объявлен аналогично кодеру, но имеет дополнительный уровень, называемый `projection_layer`, представляющий собой выходной уровень softmax для выборки прогнозов, сделанных декодером. Мы также определим функцию `TrainingHelper`, которая правильно подает входные данные в декодер. В этом примере мы еще объявляем два типа декодеров: декодеры `BasicDecoder` и `BahdanauAttention`. Механизм внимания обсуждался в главе 10. В библиотеке существует множество других декодеров, таких как `BeamSearchDecoder` и `BahdanauMonotonicAttention`:

```
decoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

projection_layer = Dense(units=vocab_size, use_bias=True)

helper = tf.contrib.seq2seq.TrainingHelper(
    decoder_emb_inp, [target_sequence_length for _ in range(batch_
size)], time_major=True)

if decoder_type == 'basic':
    decoder = tf.contrib.seq2seq.BasicDecoder(
        decoder_cell, helper, encoder_state,
        output_layer=projection_layer)

elif decoder_type == 'attention':
    decoder = tf.contrib.seq2seq.BahdanauAttention(
        decoder_cell, helper, encoder_state,
        output_layer=projection_layer)
```

Для получения выходных данных декодера мы будем использовать динамическое декодирование:

```
outputs, _, _ = tf.contrib.seq2seq.dynamic_decode(
    decoder, output_time_major=True,
    swap_memory=True
)
```

Далее объявляем операции логитов, кросс-энтропии и обучения:

```
logits = outputs.rnn_output

crossent = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=dec_train_labels, logits=logits)
loss = tf.reduce_mean(crossent)

train_prediction = outputs.sample_id
```

Затем определяем два оптимизатора – `AdamOptimizer` для первых 10 000 шагов и стохастический градиентный спуск `GradientDescentOptimizer` для остальной части процесса оптимизации. Это связано с тем, что использование оптимизатора Adam в течение длительного времени приводит к непредсказуемым результатам. Поэтому мы начнем с оптимизатора Adam, чтобы получить хорошую начальную позицию для оптимизатора SGD, а затем продолжим использовать SGD:

```
with tf.variable_scope('Adam'):
    optimizer = tf.train.AdamOptimizer(learning_rate)
with tf.variable_scope('SGD'):
```

```
    sgd_optimizer = tf.train.GradientDescentOptimizer(learning_rate)

gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 25.0)
optimize = optimizer.apply_gradients(zip(gradients, v))

sgd_gradients, v = zip(*sgd_optimizer.compute_gradients(loss))
sgd_gradients, _ = tf.clip_by_global_norm(sgd_gradients, 25.0)
sgd_optimize = optimizer.apply_gradients(zip(sgd_gradients, v))
```

Строгая оценка качества работы оптимизаторов при обучении NMT содержится в статье[1] Бахара (Bahar) и др. под названием «Эмпирическое исследование алгоритмов оптимизации в нейронном машинном переводе».

Визуализация представлений слов с помощью TensorBoard

Когда мы решили визуализировать представление слов в упражнении главы 3, мы вручную реализовали визуализацию с помощью алгоритма t-SNE. Тем не менее для визуализации представлений слов вы также можете использовать TensorBoard – специальный инструмент визуализации в составе TensorFlow. Этот инструмент можно применять для визуализации различных переменных TensorFlow в вашей программе. Благодаря ему вы сможете видеть, как различные переменные (например, ошибка/точность модели) ведут себя во время работы модели, так что вам будет легче обнаружить потенциальные проблемы.

TensorBoard позволяет визуализировать скалярные значения и векторы в виде гистограмм. Помимо этого, TensorBoard также визуализирует представления слов. Если вам нужно проанализировать, как выглядят представления слов, то достаточно лишь правильно использовать TensorBoard. Далее вы увидите пример практического использования TensorBoard для визуализации представлений слов. Код этого упражнения находится в файле `tenorboard_word_embeddings.ipynb` в папке `appendix`.

Первые шаги с TensorBoard

Начнем с перечисления необходимых шагов для запуска TensorBoard. Этот инструмент действует как служба и работает на определенном порту (по умолчанию порт 6006). Чтобы запустить TensorBoard, вам необходимо сделать следующее.

1. Откройте командную строку (Windows) или терминал (Ubuntu/macOS).
2. Зайдите в домашний каталог проекта.
3. Если вы используете `python virtualenv`, активируйте виртуальную среду, в которой вы установили TensorFlow.
4. Убедитесь, что библиотека TensorFlow доступна через Python. Для этого выполните следующие действия:
   1) наберите `python3`, и вы получите приглашение ввода `>>>`;

[1] Empirical Investigation of Optimization Algorithms in Neural Machine Translation. Bahar and others, The Prague Bulletin of Mathematical Linguistics, 2017.

   2) попробуйте ввести команду `import tensorflow as tf`;
   3) если команда успешно выполнена, у вас все хорошо;
   4) выйдите из Python, набрав команду `exit()` после `>>>`.
5. Наберите `tensorboard --logdir=models`:
   1) опция `--logdir` указывает на каталог, в котором вы будете создавать данные для визуализации;
   2) при желании вы можете использовать опцию `--port=<ваш_порт>`, чтобы изменить порт, на котором работает TensorBoard.
6. Теперь вы должны получить следующее сообщение:

```
TensorBoard 1.6.0 at <url>;:6006 (Press CTRL+C to quit)
```

7. Введите `<url>:6006` в веб-браузер. Вы должны увидеть в окне браузера оранжевую приборную панель. Вам пока нечего отображать, потому что вы не создали данные.

Сохранение представлений слов и визуализация в TensorBoard

Сначала мы скачаем и загрузим 50-мерные вложения GloVe, которые использовали в главе 9. Для этого скачайте файл представлений GloVe (`glove.6B.zip`) по адресу **https://nlp.stanford.edu/projects/glove/** и поместите его в папку приложения. Мы загрузим из файла первые 50 тыс. векторов слов и позже используем их для инициализации переменной TensorFlow. Мы также запишем строковое значение каждого слова, так как позже они послужат метками для каждой отображаемой точки на TensorBoard:

```
vocabulary_size = 50000
pret_embeddings = np.empty(shape=(vocabulary_size,50),dtype=np.
float32)

words = []

word_idx = 0
with zipfile.ZipFile('glove.6B.zip') as glovezip:
    with glovezip.open('glove.6B.50d.txt') as glovefile:
        for li, line in enumerate(glovefile):
            if (li+1)%10000==0: print('.',end='')
            line_tokens = line.decode('utf-8').split(' ')
            word = line_tokens[0]

            vector = [float(v) for v in line_tokens[1:]]
            assert len(vector)==50
            words.append(word)
            pret_embeddings[word_idx,:] = np.array(vector)
            word_idx += 1
            if word_idx == vocabulary_size:
                break
```

Теперь объявим переменные и операции, связанные с TensorFlow. Перед этим мы создадим каталог с именем `models`, который будет задействован для хранения переменных:

```
log_dir = 'models'

if not os.path.exists(log_dir):
    os.mkdir(log_dir)
```

Затем объявляем переменную, которая будет инициализирована представлениями слов, скопированными из текстового файла ранее:

```
embeddings = tf.get_variable('embeddings',shape=[vocabulary_size, 50],
                             initializer=tf.constant_initializer(pret_embeddings))
```

Теперь создадим сеанс и инициализируем переменную, которую определили ранее:

```
session = tf.InteractiveSession()
tf.global_variables_initializer().run()
```

После этого создадим объект `tf.train.Saver`. Данный объект `Saver` можно использовать для сохранения переменных TensorFlow в памяти, чтобы впоследствии восстановить их при необходимости. В следующем коде мы сохраним переменную представлений в каталоге `models` под именем `model.ckpt`:

```
saver = tf.train.Saver({'embeddings':embeddings})
saver.save(session, os.path.join(log_dir, "model.ckpt"), 0)
```

Нам также нужно сохранить файл метаданных. Он содержит метки, изображения или другую информацию, связанную с представлением слов, поэтому при наведении курсора на визуализацию представлений соответствующие точки будут показывать слово/метку, которые они представляют. Файл метаданных должен иметь формат `.tsv` (значения, разделенные табуляцией) и содержать `vocabulary_size + 1` строк, где первая строка содержит заголовки для информации, которую вы представляете. В следующем коде мы сохраним две части информации: строковые значения слов и уникальный идентификатор (то есть индекс строки) для каждого слова:

```
with open(os.path.join(log_dir,'metadata.tsv'), 'w',encoding='utf-8')
as csvfile:
    writer = csv.writer(csvfile, delimiter='\t',
                        quotechar='|', quoting=csv.QUOTE_MINIMAL)
    writer.writerow(['Word','Word ID'])
    for wi,w in enumerate(words):
        writer.writerow([w,wi])
```

Затем нам нужно сообщить TensorFlow, где он может найти метаданные, которые мы сохранили на диск. Для этого придется создать объект `ProjectorConfig`, содержащий различные сведения о конфигурации представлений, которые мы хотим отобразить. Данные, связанные с объектом `ProjectorConfig`, будут сохранены в файле с именем `projector_config.pbtxt` в каталоге `models`:

```
config = projector.ProjectorConfig()
```

В следующем фрагменте кода мы заполним обязательные поля созданного нами объекта `ProjectorConfig`. Сначала назовем имя переменной, которую мы хотим визуализировать. Далее мы скажем ему, где он может найти метаданные, соответствующие этой переменной:

```
embedding_config = config.embeddings.add()
embedding_config.tensor_name = embeddings.name
embedding_config.metadata_path = 'metadata.tsv'
```

Теперь используем `summary_writer`, чтобы записать результаты в файл `projector_config.pbtxt`. TensorBoard прочитает этот файл при запуске:

```
summary_writer = tf.summary.FileWriter(log_dir)
projector.visualize_embeddings(summary_writer, config)
```

Теперь, если вы запустите TensorBoard, вы увидите нечто похожее на рис. П.3.

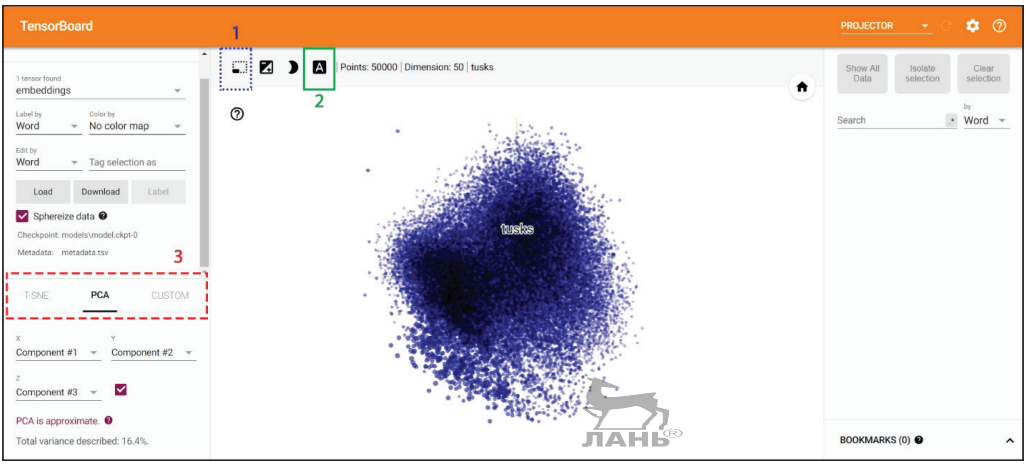


Рис. П.3 ❖ Визуализация представлений в панели TensorBoard

При наведении курсора на облако точек будет отображаться метка слова, над которым вы в данный момент находитесь, поскольку мы предоставили эту информацию в файле `metadata.tsv`. Кроме того, у вас есть несколько вариантов визуализации. Первый вариант (показанный пунктирной линией и помеченный как 1) позволит вам выбрать подмножество полного пространства представлений. Вы можете нарисовать ограничивающий прямоугольник над областью интересующего вас пространства, и он будет выглядеть так, как показано на рис. П.4. Я выбрал представления в правом нижнем углу.

Другой вариант – отображение слов вместо точек. Вы можете сделать это, выбрав второй вариант на рис. П.3 (показан внутри сплошного прямоугольника и обозначен как 2). Результат будет выглядеть так, как показано на рис. П.5. Кроме того, вы можете панорамировать, масштабировать и вращать вид по своему вкусу. Если вы нажмете кнопку справки (показанную внутри сплошного прямоугольника, помеченного цифрой 1 на рис. А.5), она покажет вам руководство по управлению видом.

Наконец, вы можете изменить алгоритм визуализации в панели слева, обведенной пунктирной линией и отмеченной цифрой 3 на рис. П.3.

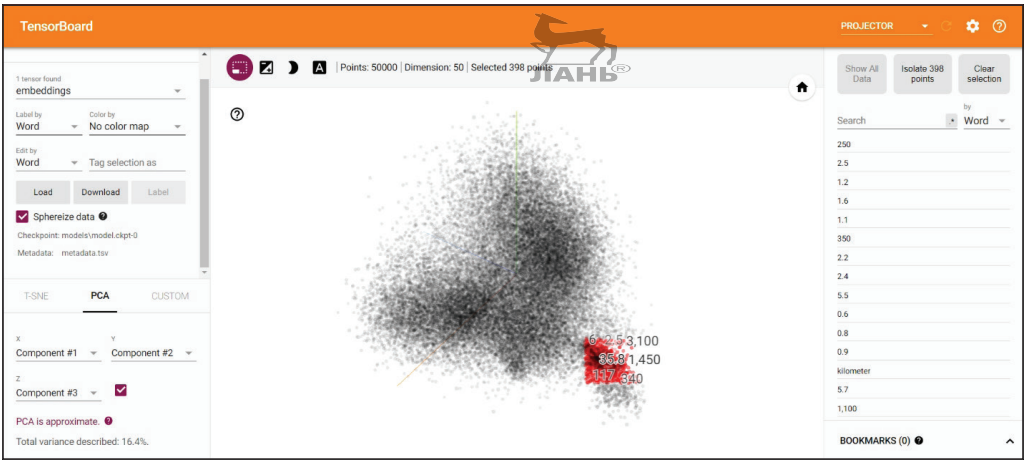


Рис. П.4 ❖ Выбор подмножества в пространстве представлений

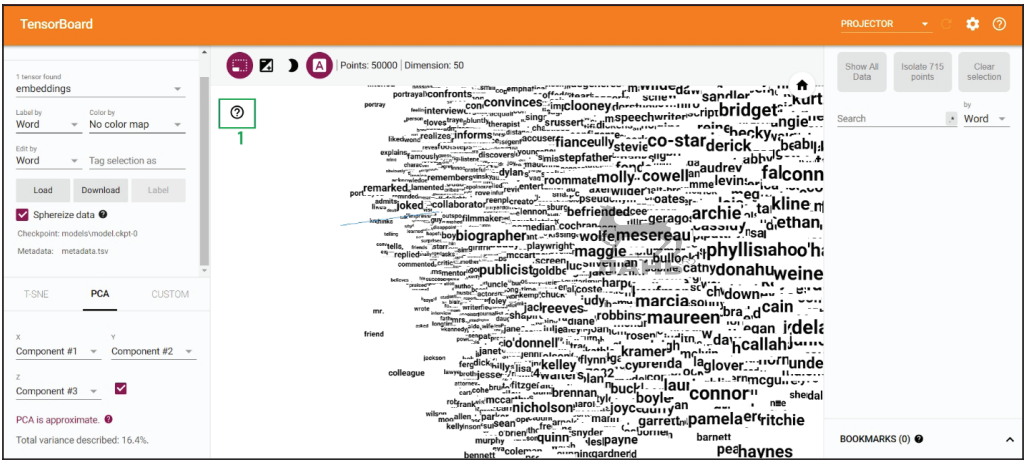


Рис. П.5 ❖ Векторы представлений, отображаемые в виде слов вместо точек

Заключение

В приложении мы обсудили некоторые математические основы и реализации, которые не рассмотрели в других главах. Сначала мы привели математические обозначения для скаляров, векторов, матриц и тензоров. Затем рассмотрели различные операции, выполняемые с этими структурами данных, такие как умножение матриц и инверсия. Далее вы обзорно познакомились с различными понятиями, которые полезны при работе с вероятностным машинным обучением, такими как функции плотности вероятности, совместная вероятность, предельная веро-

ятность и правило Байеса. После этого мы перешли к рассмотрению различных реализаций, не вошедших в другие главы. Мы узнали, как использовать Keras – высокоуровневую библиотеку TensorFlow для реализации CNN. Затем рассмотрели пример реализации машинного перевода с библиотекой TensorFlow seq2seq и сравнили его с реализацией на уровне базовых функций, которую мы обсуждали в главе 10. Наконец, мы завершили приложение руководством, которое научит вас визуализировать представление слов с помощью инструмента визуализации TensorBoard, поставляемого с TensorFlow.

Предметный указатель

Н

О

П

Р

С

Т

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.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Тушан Ганегедара

Обработка естественного языка с TensorFlow

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 31,04. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**