Глубокое обучение для поисковых систем

Глубокое обучение поисковых систем решает самые сложные задачи, в частности позволяет получать релевантные результаты при неточных условиях поиска и плохо проиндексированных данных, извлекать изображения с минимальными метаданными. С помощью таких современных инструментов, как DL4J и TensorFlow, вы сможете применять мощные методы глубокого обучения, не обладая специальными знаниями в области науки о данных или обработки естественного языка. Книга покажет вам, как это сделать.
Вы узнаете, как глубокое обучение связано с основами поиска, такими как индексация и ранжирование, и изучите подробные примеры, позволяющие улучшить поиск, используя библиотеки Apache Lucene и Deeplearning4j. В ходе чтения вы освоите сложные темы: поиск по изображениям, перевод пользовательских запросов, проектирование поисковых систем, совершенствуемых по мере обучения.

В книге рассматриваются:

- генерация синонимов;
- точное и релевантное ранжирование;
- поиск по языкам;
- поиск изображений на базе содержимого;
- поиск с использованием рекомендательных систем.

Томмазо Теофили – инженер-программист, работающий с открытым исходным кодом и искусственным интеллектом. Он состоит в организации Apache Software Foundation и участвует в проектах по поиску информации, обработке естественного языка и распределенным вычислениям.

Издание предназначено для разработчиков, имеющих опыт работы с Java или похожим языком программирования и получивших представление об основах поиска. Специальные знания в области глубокого обучения или обработки естественного языка не требуются.

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК "Галактика"
books@alians-kniga.ru

www.дмк.рф

ISBN 978-5-97060-776-3

«Практический подход, показывающий современное состояние использования нейронных сетей, искусственного интеллекта и глубокого обучения при разработке поисковых систем».
Крис Мэттман, NASA JPL

«Глубокий синтез принципов традиционного поиска и последних достижений в области глубокого обучения».
Грег Занотти, Marquette Partners

«Пристальное погружение в новейшие технологии, которые выведут вашу поисковую систему на новый уровень».
Эндрю Уилли, Thynk Health

«Практические упражнения научат вас применять глубокое обучение для создания поисковых продуктов».
Антонио Маньаги, System1

Глубокое обучение для поисковых систем

Глубокое обучение для поисковых систем

Томмазо Теофили

MANNING

издательство дмк

Deep Learning for Search

TOMMASO TEOFILI
Foreword by CHRIS MATTMANN

MANNING
Shelter Island

Глубокое обучение для поисковых систем

ТОММАЗО ТЕОФИЛИ
Предисловие КРИСА МЭТТМАННА

ДМК Пресс
издательство
Москва, 2020

УДК 004.891
ББК 32.972.13
Т339

Теофили Т.
Т339 Глубокое обучение для поисковых систем / пер. с анг. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 318 с.: ил.

ISBN 978-5-97060-776-3

В книге рассказывается о том, как использовать глубокие нейронные сети для создания эффективных поисковых систем. Рассматривается несколько компонентов поисковой системы, дается представление о том, как они работают, и приводятся рекомендации по использованию нейронных сетей в разных контекстах поиска. Особое внимание уделено практическому объяснению методов поиска и глубокого машинного обучения на базе примеров, большинство которых включает фрагменты кода.

Автор освещает основные проблемы, связанные с поисковыми системами, и указывает пути решения этих проблем. Он раскрывает принципы тестирования эффективности нейронных сетей, а также измерения их затрат и выгод.

Издание предназначено для читателей, владеющих программированием на среднем уровне и отлаживающих поисковые системы с целью повышения их эффективности, то есть выдачи наиболее релевантных результатов.

УДК 004.891
ББК 32.972.13

ISBN 978-1-617-29479-2 (анг.)
ISBN 978-5-97060-776-3 (рус.)

Посвящается Маттие, Джакомо и Микеле

Содержание

Предисловие

Не просто дать количественную оценку того, насколько обыденными стали такие термины, как *нейронные сети* и *глубокое обучение*, и, более конкретно, как эти технологии влияют на нашу жизнь. От автоматизации рутинных заданий до тиражирования трудных решений, помощи в управлении автомобилем (и людьми) до места назначения – мощь нейронных сетей и глубокого обучения в качестве методов, которые произведут революцию в области вычислений, находится лишь в стадии зарождения.

Вот почему эта книга так важна. Нейронные сети, искусственный интеллект (ИИ) и глубокое обучение не только автоматизируют рутинные задания и решения, облегчая их. Они также облегчают и поиск. Прежде состояние дел в области поиска информации использовало сложную линейную алгебру, включая в себя матричное умножение для обозначения сопоставления пользовательских запросов с документами. Сегодня вместо использования алгебраических и линейных моделей применяются, например, нейронные сети для распознавания сходства слов между документами после изучения способов суммирования документов в слова с использованием обособленных сетей. И это только одна область в процессе поиска, где используются ИИ и глубокое обучение.

В своей книге Томмазо Теофили использует практический подход, чтобы показать вам современное состояние использования нейронных сетей, искусственного интеллекта и глубокого обучения в разработке поисковых систем. Книга изобилует примерами и знакомит читателя с архитектурой современных поисковых систем, а также дает вам достаточно информации, чтобы понять, как и где подходит глубокое обучение и как это улучшает поиск. Автор показывает вам, где искусственный интеллект и глубокое обучение могут перегрузить ваш код и возможность поиска, начиная от создания вашей первой сети для поиска похожих слов в расширении запроса и заканчивая изучением векторного представления слов для поискового ранжирования, а также мультиязычного поиска и поиска по изображениям.

Эта книга написана настоящим первопроходцем в области открытого исходного кода. Томмазо – бывший председатель проекта Apache Lucene – механизма индексации поиска де-факто, который поддерживает Elasticsearch и Apache Solr. Он также внес большой вклад в понимание языков и перевод при разработке библиотеки Apache OpenNLP. Совсем недавно его кандидатура была предложена на пост председателя (инкубационного) проекта Apache Joshua для статистического машинного перевода.

Я знаю, что вы многому научитесь из этой книги, и я рекомендую ее, чтобы вы могли найти золотую середину между здравым смыслом, толкованиями теории сложности и реальным кодом, с которым вы можете экспериментировать, используя новейшие технологии глубокого обучения и поиска.

Наслаждайтесь. Я знаю, о чем говорю, потому что именно это я и делал!

Крис Мэттманн,
заместитель директора по технологиям и инновациям,
Лаборатория реактивного движения НАСА

От автора

Обработка естественного языка околдовала меня, как только я узнал о ней, почти 10 лет назад, когда учился в магистратуре. Обещание, что компьютеры могут помочь нам понять (уже даже тогда) огромное количество существующих текстовых документов, было похоже на волшебство. Я до сих пор помню, как было здорово видеть, как мои первые программы для ОЕЯ извлекают пусть даже смутно правильную и полезную информацию из нескольких текстовых документов.

Примерно в то же время на работе меня попросили проконсультировать клиента по поводу новой архитектуры поиска с открытым исходным кодом. Мой коллега, который был экспертом в этой области, был занят другим проектом, поэтому мне дали копию книги *Lucene в действии*[1], которую я изучал на протяжении пары недель. Затем меня отправили на работу консультантом. Спустя пару лет после того, как я проработал над проектом на базе Lucene/Solr, заработала новая поисковая система (и, насколько я знаю, она используется до сих пор). Не могу сказать, сколько раз нужно было настраивать алгоритмы поисковой системы из-за того или иного запроса или того или иного фрагмента проиндексированного текста, но мы заставили ее работать. Я мог видеть запросы пользователей и мог видеть данные, которые должны были быть извлечены, но минимальная разница в написании или пропуске определенного слова могла привести к тому, что очень релевантная информация не была бы отображена в результатах поиска. Поэтому, хотя я очень гордился своей работой, я продолжал задаваться вопросом, как сделать все возможное, чтобы избежать множества ручных вмешательств, которые менеджеры по программному продукту просили меня выполнить, чтобы обеспечить наилучший опыт взаимодействия.

Сразу же после этого я совершенно случайно оказался вовлеченным в машинное обучение благодаря первому онлайн-занятию по машинному обучению от Эндрю Ына (которое было основано на серии Coursera MOOC). Я был настолько очарован концепциями нейронных сетей, показанными на уроке, что решил самостоятельно попробовать реализовать небольшую библиотеку для нейронных сетей на Java, просто ради удовольствия (**http://svn.apache.org/repos/asf/labs/yay/**). Я начал искать другие онлайн-курсы, такие как курс Андрея Карпати по сверточным нейронным сетям для визуального распознавания и курс Ричарда Сохера по глубоким нейронным сетям для обработки естественного языка. С тех пор я продолжаю работать над поисковыми системами, обработкой естественного языка и глубоким обучением, в основном в открытом исходном коде.

Пару лет назад (!) издательство Manning обратилось ко мне с рецензией на книгу об ОЕЯ, и я был достаточно наивен, чтобы написать в нижней части своего обзора, что мне было бы интересно написать книгу о поисковых системах и нейронных сетях. Когда издательство снова обратилось ко мне, проявив интерес, я был немного удивлен и спросил себя, *действительно* ли я хочу написать книгу об этом? Я понял, что да, мне это было интересно.

---

[1] **http://www.manning.com/books/lucene-in-action-second-edition**.

Несмотря на то что глубокое обучение произвело революцию в компьютерном зрении и обработке естественного языка, многое еще предстоит обнаружить, когда речь идет о приложениях, использующихся в поиске. Я уверен, что мы не можем (пока еще?) полагаться на глубокое обучение, чтобы автоматически настраивать поисковые системы от своего имени, но это может помочь сделать работу пользователя поисковой системы более гладкой. Благодаря глубокому обучению мы можем делать в поисковых системах то, чего пока не можем делать с помощью других существующих методов, и можем использовать глубокое обучение, чтобы улучшить техники, которые мы уже используем в поисковых системах. Путь к тому, чтобы сделать поисковые машины более эффективными с помощью глубоких нейронных сетей, только начался. Надеюсь, вам понравится.

Благодарности

Прежде всего я хотел бы поблагодарить свою любимую жену Микелу за помощь и поддержку на протяжении всего этого долгого путешествия: спасибо за любовь, энергию и преданность делу в течение долгих дней, ночей и выходных, в ходе которых я писал эту книгу!

Спасибо Джакомо и Маттиа за то, что они помогли мне выбрать самую классную иллюстрацию для обложки, а также за те игры и смех, которые сопровождали меня, пока я пытался писать.

Я хотел бы поблагодарить своего отца за то, что он гордится мной, и его веру в меня.

Большое спасибо моему другу Федерико за его неустанные усилия по просмотру всех материалов (книга, код, изображения и т. д.) и за приятные обсуждения и обмен идеями. Огромное спасибо моим друзьям и коллегам Антонио, Франческо и Симоне за их поддержку, смех и советы. Также адресую благодарность своим товарищам по проекту Apache OpenNLP (**http://opennlp.apache.org**), Сунилу, Йорну и Кодзи, за то, что предоставили отзывы, советы и идеи, которые помогли придать очертания этой книге.

Я благодарю Криса Мэтманна за написание такого вдохновляющего пролога.

Я также благодарю Фрэнсис Лефковиц, моего редактора по разработке, за ее терпение и руководство на протяжении всего процесса написания книги, включая наши дискуссии о Стефе, К.Д. и Воинах. И я благодарю других людей из издательства Manning, благодаря которым стало возможным появление этой книги, включая издателя Марьян Бэйс и лиц из редакционной и производственной команд, которые работали за кадром. Кроме того, я благодарю технических рецензентов во главе с Иваном Мартиновичем – Абхинава Упадхая, Эля Кринкера, Альберто Симоеса, Альваро Фалькину, Эндрю Уилли, Антонио Магнаги, Криса Моргана, Джулиано Бертоти, Грега Занотти, Йеруна Бенкхуйзена, Крифа Дэвида, Люка Мартина Бира, Майкла Уолла, Михала Пашкевича, Мирко Кемпфа, Паули Сутелайнен, Симона Русо, Срдана Дукича и Урсина Стаусса – и участников форума. Что касается технической стороны, выражаю благодарность Михилу Тримпе, который был техническим редактором книги, и Карстену Стрёбеку, который работал техническим корректором книги.

Наконец, я хотел бы поблагодарить сообщества Apache Lucene и Deeplearning4j за то, что предоставили такие превосходные инструменты, и за дружескую поддержку пользователей.

Об этой книге

Создание поисковых систем с использованием методов глубокого обучения – это практическая книга о том, как использовать (глубокие) нейронные сети для создания эффективных поисковых систем. В ней рассматривается несколько компонентов поисковой системы, дается представление о том, как они работают, и рекомендации о том, как можно использовать нейронные сети в каждом контексте. Основное внимание уделяется практическому объяснению методов поиска и глубокого обучения на базе примеров. Большинство из них сопровождается кодом. В то же время, когда это необходимо, приводятся ссылки на соответствующие исследовательские работы, чтобы побудить вас читать больше и углублять свои знания по конкретным темам. Нейронные сети и темы, связанные с поиском, объясняются на протяжении всей книги.

Прочитав ее, вы получите четкое представление об основных проблемах, связанных с поисковыми системами, о том, как они обычно решаются, и о том, как в этом может помочь глубокое изучение. Вы получите четкое представление о ряде различных методов глубокого обучения и о том, где они вписываются в контекст поиска. Вы также познакомитесь с библиотеками Lucene и Deeplearning4j. Кроме того, вы выработаете практическое отношение к тестированию эффективности нейронных сетей (вместо того чтобы рассматривать их как волшебство) и измерению их затрат и выгод.

Для кого предназначена эта книга

Данная книга предназначена для читателей, владеющих программированием на среднем уровне. Еще лучше, если вы хорошо разбираетесь в программировании на языке Java, интересуясь или активно участвуя в разработке поисковых систем. Вам следует прочитать эту книгу, если вы хотите сделать свою поисковую систему более эффективной, чтобы предоставлять релевантные результаты и, следовательно, сделать ее более полезной для конечных пользователей.

Даже если у вас нет такого опыта, вы будете знакомиться с основными понятиями, касающимися поисковых систем, на протяжении всей книги, когда будет затрагиваться каждый конкретный аспект поиска. Аналогично вам не обязательно иметь познания в области машинного или глубокого обучения. В этой книге будут представлены все необходимые основы машинного и глубокого обучения, а также практические советы, касающиеся применения глубокого обучения в поисковых системах в случаях реальной эксплуатации.

Вы должны быть готовы взять в руки код и расширить существующие библиотеки с открытым исходным кодом для реализации алгоритмов глубокого обучения для решения задач поиска.

Дорожная карта

Эта книга состоит из трех частей:

- первая часть знакомит вас с основными понятиями поиска, машинного и глубокого обучения. В главе 1 рассказывается об обосновании примене-

ния методов глубокого обучения для поиска проблем, затрагивая проблемы в отношении наиболее распространенных подходов к поиску информации. В главе 2 приводится первый пример того, как использовать модель нейронной сети для повышения эффективности поисковой системы путем генерации синонимов из данных;

- вторая часть посвящена общим задачам поисковых систем, которые можно лучше решать с помощью глубоких нейронных сетей. Глава 3 знакомит вас с использованием рекуррентных нейронных сетей для генерации запросов, альтернативных тем, которые вводят пользователи. Глава 4 посвящена задаче предоставления других предложений, в то время как пользователь набирает запрос, с помощью глубоких нейронных сетей. Глава 5 рассказывает о моделях ранжирования, в частности о том, как предоставить более релевантные результаты поиска, используя векторные представления слов. Глава 6 посвящена использованию векторных представлений документов как в функциях ранжирования, так и в контексте метода фильтрации на основе содержания, используемого при построении рекомендательных систем;
- в третьей части рассматриваются более сложные сценарии, такие как машинный перевод с использованием глубокого обучения и поиск изображений. В главе 7 рассказывается о мультиязычных возможностях поисковой системы с помощью подходов на базе нейронных сетей. Глава 8 посвящена поиску коллекции изображений на основе их содержимого, основанной на моделях глубокого обучения. В главе 9 обсуждаются темы, связанные с реальной эксплуатацией, такие как точная настройка моделей глубокого обучения и работа с постоянно поступающими потоками данных.

Сложность рассматриваемых тем и концепций возрастает в ходе прочтения книги. Если вы новичок в области глубокого обучения, поиска или того и другого, я настоятельно рекомендую сначала прочитать главы 1 и 2. В противном случае не стесняйтесь перепрыгивать и выбирать главы, основываясь на своих потребностях и интересах.

О КОДЕ

В этой книге предпочтение отдается фрагментам кода, а не подробным листингам, чтобы читатель мог быстро и легко понять, что и как делает код. Полный исходный код можно найти на странице книги на сайте издательства Manning: **www.manning.com/books/deep-learning-for-search**. Программное обеспечение также будет обновляться на официальной странице книги на GitHub (**https://github.com/dl4s**), включая исходный код на Java в книге (с использованием Apache Lucene и Deeplearning4j: **https://github.com/dl4s/dl4s**) и версию на Python для тех же алгоритмов (**https://github.com/dl4s/pydl4s**).

В примерах кода используется язык программирования Java и две библиотеки с открытым исходным кодом (по лицензии Apache): Apache Lucene (**http://lucene.apache.org**) и Deeplearning4j (**http://deeplearning4j.org**). Lucene является одной из наиболее широко используемых библиотек для создания поисковых систем, а Deeplearning4j на момент написания этих строк является лучшим выбором для нативной библиотеки Java для глубокого изучения. Вместе они позволят вам лег-

ко, быстро и без проблем проводить тестирование и экспериментировать с поиском и глубоким обучением.

Кроме того, многие специалисты, работающие над проектами, связанными с глубоким обучением, в настоящее время используют Python (с такими фреймворками, как TensorFlow, Keras, PyTorch и т. д.). Поэтому также предоставляется репозиторий Python, на котором размещены версии алгоритмов, подробно описанных в книге, на TensorFlow (**https://tensorflow.org**).

Исходный код в книге отформатирован `шрифтом фиксированной ширины, как этот`, чтобы отделить его от обычного текста. Во многих случаях первоначальный исходный код был переформатирован; мы добавили разрывы строк и переработали отступы, чтобы разместить доступное пространство страницы в книге. В редких случаях даже этого было недостаточно, и списки содержат маркеры продолжения строки (➥). Кроме того, комментарии в исходном коде часто удалялись из списков, когда описание кода приводится в тексте. Аннотации к коду сопровождают множество листингов, выделяя важные понятия.

Форум liveBook

Приобретение этой книги включает в себя бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где вы можете оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, перейдите по ссылке **https://livebook.manning.com/#!/book/deep-learning-for-search/discussion**. Вы можете узнать больше о форумах издательства и правилах поведения на странице **https://livebook.manning.com/#!/discussion**.

Обязательство Manning по отношению к нашим читателям состоит в том, чтобы обеспечить место, где может иметь место содержательный диалог между отдельными читателями и между читателями и автором. Это не обязательство какого-либо конкретного количества участия со стороны автора, чей вклад в форум остается добровольным (и неоплачиваемым). Мы предлагаем вам задавать автору сложные вопросы, чтобы его интерес не угас! Форум и архив предыдущих обсуждений будут доступны на сайте издателя, пока книга находится в печати.

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** или **www.дмк.рф** 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 этой 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.

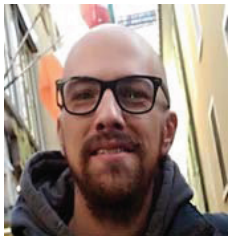
Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning 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.

Об авторе



Томмазо Теофили – инженер-программист со страстью к открытому исходному коду и машинному обучению. Будучи участником Apache Software Foundation, он участвует в ряде проектов с открытым исходным кодом, начиная от таких тем, как поиск информации (например, Lucene и Solr), и заканчивая обработкой естественного языка и машинным переводом (включая OpenNLP, Joshua и UIMA).

В настоящее время он работает в компании Adobe, разрабатывая компоненты инфраструктуры поиска и индексации, а также исследует области обработки естественного языка, поиска информации и глубокого изучения. Он выступал с докладами о поиске и машинном обучении на конференциях, включая BerlinBuzzwords, международную конференцию International Conference on Computational Science, ApacheCon, EclipseCon и др. Вы можете найти его в Twitter (**@tteofili**).

Об иллюстрации на обложке

Рисунок на обложке книги носит название «Одеяние китайской дамы». Иллюстрация взята из книги «Коллекция платьев разных народов, древних и современных» Томаса Джефферис (четыре тома), опубликованной в Лондоне между 1757 и 1772 годом. Титульный лист гласит, что это медные гравюры ручной работы, украшенные гуммиарабиком.

Томаса Джефферис (1719–1771) называли «географом короля Георга III». Он был английским картографом, который был ведущим создателем карт своего времени. Он гравировал и печатал карты для правительственных и других государственных учреждений и выпускал обширный спектр коммерческих карт и атласов, особенно касающихся Северной Америки. Его работа в качестве картографа пробудила интерес к местному дресс-коду, блистательно представленному в этой коллекции. Он был принят в тех землях, которые он исследовал и наносил на карту. Увлечение далекими землями и путешествия ради удовольствия были относительно новым явлением в конце XVIII века, и такие коллекции, как эта, были популярны, знакомя как туристов, так и путешественников, сидящих в креслах, с жителями других стран.

Разнообразие рисунков в этом издании Джефферис ярко свидетельствует об уникальности и индивидуальности народов мира около 200 лет назад. С тех пор дресс-код изменился, а богатое в ту пору разнообразие, в зависимости от региона и страны, исчезло. Сейчас часто трудно отличить жителей одного континента от другого. Возможно, пытаясь взглянуть на это с оптимизмом, мы обменяли культурное и визуальное разнообразие на более разнообразную личную жизнь – или на более разнообразную и интересную интеллектуальную и техническую жизнь.

В то время когда трудно отличить одну компьютерную книгу от другой, издательство Manning празднует изобретательность и инициативу компьютерного бизнеса с помощью обложек книг, основанных на богатом разнообразии жизни регионов двухвековой давности, которое ожило благодаря рисункам Джефферис.

Часть I

ПОИСК ВСТРЕЧАЕТСЯ С ГЛУБОКИМ ОБУЧЕНИЕМ

Настройка поисковых систем для эффективного реагирования на потребности пользователей – непростая задача. Традиционно многие внутренние настройки и корректировки, сделанные вручную, приходилось вносить в поисковую систему, чтобы она прилично работала при реальном сборе данных. С другой стороны, глубокие нейронные сети очень хорошо подходят для изучения полезной информации об огромных объемах данных. В этой первой части книги мы начнем изучать, как можно использовать поисковую систему в сочетании с нейронной сетью, чтобы обойти некоторые общие ограничения и предоставить пользователям более совершенные возможности поиска.

Глава 1

Поиск на основе нейронных сетей

О чем идет речь в этой главе:

- деликатное введение в основы поиска;
- важные проблемы в поиске;
- почему нейронные сети могут помочь поисковым системам быть более эффективными.

Предположим, вам нужно узнать что-то о последних научных открытиях в области искусственного интеллекта. Что вы будете делать, чтобы найти информацию? Сколько времени и сил требуется, чтобы получить факты, которые вы ищете? Если вы находитесь в (огромной) библиотеке, можно спросить библиотекаря, какие книги есть по этой теме, и он, вероятно, укажет на несколько книг, о которых он знает. В идеале библиотекарь подскажет определенные главы, которые нужно искать.

Звучит довольно просто. Но библиотекарь обычно происходит из другого контекста, в отличие от вас, то есть у вас и у библиотекаря могут быть разные мнения относительно того, что является важным. В библиотеке могут быть книги на разных языках, или библиотекарь может говорить на другом языке. Информация по этой теме может быть устаревшей, учитывая, что *последняя* является довольно относительным моментом во времени, и вы не знаете, когда библиотекарь в последний раз читал что-либо об искусственном интеллекте, или регулярно ли библиотека получает публикации в этой области. Кроме того, библиотекарь может не понять ваш запрос должным образом и подумать, что вы говорите об интеллекте с точки зрения психологии[1]. Пройдет несколько повторных циклов, прежде чем вы поймете друг друга и получите нужные вам фрагменты информации.

Затем, после всего этого вы можете обнаружить, что в библиотеке нет нужной вам книги; или информация может содержаться в нескольких книгах, и вы должны прочитать их все. Как это утомительно!

Только если вы сами не библиотекарь, именно это часто происходит в наши дни, когда вы что-то ищете в интернете. Хотя мы можем рассматривать интернет как единую огромную библиотеку, существует множество разных библиотекарей,

[1] Такое случилось со мной на самом деле.

которые помогут вам найти необходимую информацию: поисковые системы. Некоторые из них являются экспертами в определенных темах; другие знают только подмножество библиотеки или лишь одну книгу.

А теперь представьте, что некто, назовем его Робби, который уже знает о библиотеке и ее посетителях, может помочь вам обмениваться данными с библиотекарем, чтобы найти то, что вы ищете. Это поможет вам быстрее получить ответы. Робби может помочь библиотекарю понять запрос посетителя, например предоставив дополнительный контекст. Робби знает, о чем обычно читает посетитель, поэтому он пропускает все книги по психологии. Также, прочитав множество книг в библиотеке, Робби лучше понимает, что является важным в области искусственного интеллекта. Было бы чрезвычайно полезно иметь таких советников, как Робби, чтобы помочь поисковым системам работать лучше и быстрее, а также помогать пользователям получать больше полезной информации.

Эта книга посвящена использованию методов из области машинного обучения, называемой *глубоким обучением*, чтобы создавать модели и алгоритмы, которые могут влиять на поведение поисковых систем, чтобы сделать их более эффективными. Алгоритмы глубокого обучения будут играть роль Робби, помогая поисковой системе обеспечить лучший опыт поиска и предоставлять более точные ответы конечным пользователям.

Важно отметить, что глубокое обучение и *искусственный интеллект* – не одно и то же. Как видно по рис. 1.1, искусственный интеллект представляет собой огромную область для исследований. Машинное обучение является лишь частью этого, а глубокое обучение, в свою очередь, является подразделом машинного обучения. В основном глубокое обучение изучает, как заставить машины «учиться», используя модель вычислений глубоких нейронных сетей.

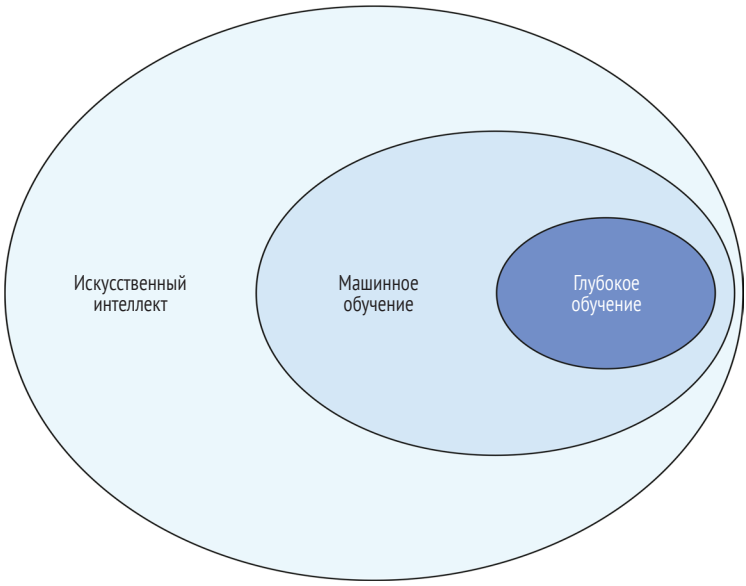


Рис. 1.1 ❖ Искусственный интеллект, машинное обучение и глубокое обучение

1.1. Нейронные сети и глубокое обучение

Цель этой книги – дать вам возможность использовать глубокое обучение в контексте поисковых систем, чтобы улучшить процесс поиска и его результаты. Даже если вы не собираетесь создавать еще одну поисковую систему в Google, вы должны научиться пользоваться методами глубокого обучения в небольших или средних поисковых системах, чтобы помочь пользователям в процессе поиска. Поиск на основе нейронных сетей должен помочь вам автоматизировать работу, которую в противном случае вам пришлось бы выполнять вручную. Например, вы узнаете, как автоматизировать извлечение синонимов из данных поисковой системы, избегая ручного редактирования файлов синонимов (глава 2). Это экономит время, повышая эффективность поиска, независимо от конкретного случая использования или области. То же самое относится и к подходящим предложениям по сопутствующему контенту (глава 6). Во многих случаях пользователи удовлетворены сочетанием простого поиска с возможностью навигации по сопутствующему контенту. Мы также рассмотрим некоторые более конкретные случаи использования, такие как поиск контента на нескольких языках (глава 7) и поиск изображений (глава 8).

Единственное требование к методам, которые мы будем обсуждать, заключается в том, что у них достаточно данных для подачи в нейронные сети. Но в общем виде трудно определить границы «достаточного количества данных». Вместо этого давайте суммируем минимальное количество документов (текст, изображения и т. д.), которые обычно необходимы для каждой проблемы, рассматриваемой в книге: *см*. табл. 1.1.

Таблица 1.1. Требования к задачам для методов поиска на базе нейронных сетей

Задача	**Минимальное количество документов (диапазон)**	**Глава**
Изучение представлений слов	1000–10 000	2, 5
Генерирование текста	10 000–100 000	3, 4
Изучение представлений документов	1000–10 000	6
Машинный перевод	10 000–100 000	7
Изучение представлений изображений	10 000–100 000	8

Обратите внимание, что не нужно строго следовать этой таблице; цифры взяты из опыта. Например, даже если в поисковой системе насчитывается менее 10 000 документов, вы все равно можете попытаться реализовать методы нейронного машинного перевода, описанные в главе 7; но вы должны принять во внимание тот факт, что получить качественные результаты может быть труднее (например, совершенные переводы).

Читая книгу, вы многое узнаете о глубоком обучении, а также обо всех необходимых основах поиска для реализации этих принципов ГО в поисковой системе. Поэтому, если вы инженер, связанный с разработкой поисковых систем, или программист, который хочет изучать поиск на базе нейронных сетей, эта книга для вас.

На данный момент вам не обязательно знать, что такое глубокое обучение или как оно работает. Вы подробнее узнаете об этом, когда мы будем рассматривать

конкретные алгоритмы один за другим, когда они станут полезными для решения определенных типов поисковых задач. А пока я начну с основных определений. Глубокое обучение – это область машинного обучения, где компьютеры способны учиться представлять и распознавать вещи постепенно, используя глубокие нейронные сети. Глубокие *искусственные нейронные сети* – это вычислительная парадигма, изначально вдохновленная тем, как мозг организован в графы нейронов (хотя мозг гораздо сложнее, чем искусственная нейронная сеть). Обычно информация поступает в нейроны во *входном слое*, затем через сеть скрытых нейронов (образуя один или несколько *скрытых слоев*), а потом выходит через нейроны в *выходном слое*. Нейронные сети также можно рассматривать как черные ящики: интеллектуальные функции, которые могут преобразовывать входные данные в результаты на основе того, чему была обучена каждая сеть. Обычная нейронная сеть имеет, по крайней мере, один входной слой, один скрытый слой и один выходной слой. Когда у сети есть более одного скрытого слоя, мы называем сеть *глубокой*. На рис. 1.2 вы видите глубокую нейронную сеть с двумя скрытыми слоями.

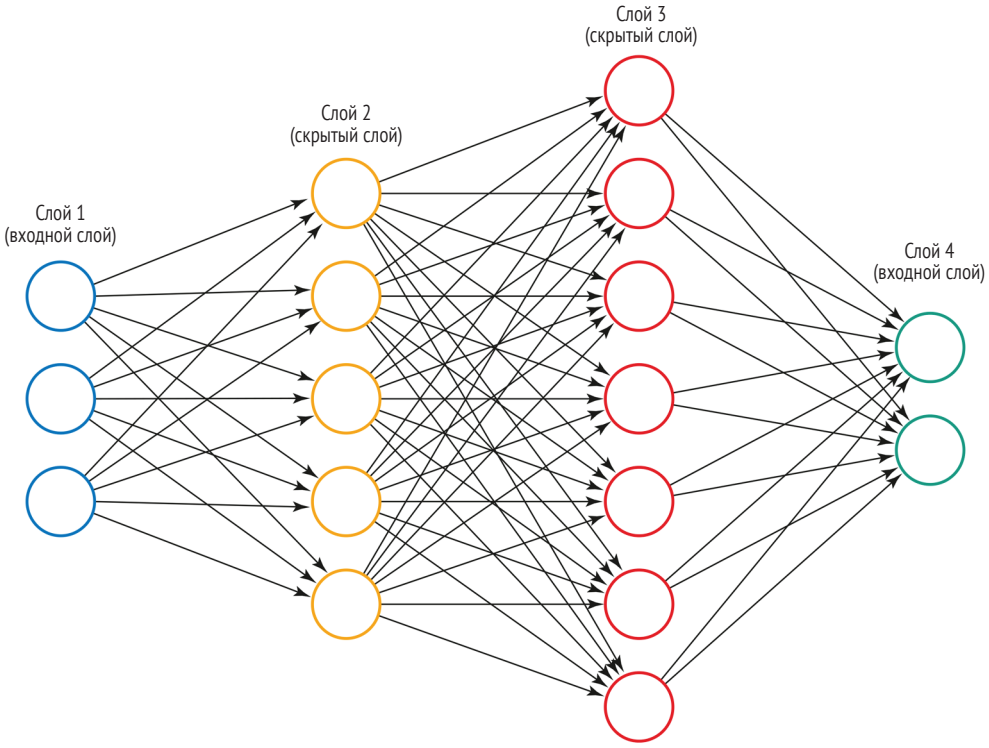


Рис. 1.2 ❖ Глубокая нейронная сеть с двумя скрытыми слоями

Прежде чем углубляться в подробности, касающиеся нейронных сетей, давайте сделаем шаг назад. Я сказал, что глубокое обучение – это подобласть машинного обучения, которое является частью более широкой области искусственного интеллекта. Но что такое машинное обучение?

1.2. Что такое машинное обучение?

Обзор основных концепций машинного обучения здесь будет полезен, прежде чем подробно рассматривать глубокое обучение и особенности поиска. Многие из концепций, которые применяются к обучению с использованием искусственных нейронных сетей, такие как *контролируемое* и *неконтролируемое* обучение, *обучение* и *прогнозирование*, берут свое начало из машинного обучения. Давайте быстро рассмотрим некоторые основные концепции машинного обучения, которые мы будем использовать в глубоком обучении (применительно к поиску) в этой книге.

Машинное обучение – это автоматизированный подход к решению проблем, основанный на алгоритмах, которые могут научиться оптимальным решениям из предыдущего опыта. Во многих случаях этот опыт проявляется в форме пар, составленных из того, что ранее наблюдалось, вместе с тем, что вы хотите, чтобы алгоритм выводил из этого. Например, алгоритм машинного обучения может быть подан текстовыми парами, где ввод – это некий текст, а вывод – категория, которая может быть использована для классификации похожих текстов. Представьте, что вы вернулись в библиотеку, но на этот раз в качестве библиотекаря. Вы купили тысячи книг и хотите разместить их на полках, чтобы люди могли с легкостью их найти. Для этого вам нужно классифицировать их так, чтобы книги, принадлежащие к одной и той же категории, помещались близко друг к другу на одной и той же книжной полке (которая, возможно, имеет небольшую метку, обозначающую категорию). Если вы можете потратить несколько часов на классификацию книг вручную, то получите опыт, необходимый вашему алгоритму. После этого вы можете обучить алгоритм на основе вашего мудрого суждения, и он будет выполнять классификацию оставшихся книг от вашего имени.

Этот тип обучения, при котором вы указываете желаемый результат, соответствующий каждому вводу, называется *контролируемым обучением*. Каждая пара, состоящая из входных данных и соответствующих им целевых результатов, называется *обучающим образцом*. В табл. 1.2 показаны некоторые из категорий, которые библиотекарь может составить вручную, чтобы помочь создать алгоритм контролируемого обучения.

Таблица 1.2. Пример данных для классификации книг

Название книги	Текст	Категории
Обработка неструктурированных текстов	Если вы читаете эту книгу, шансы, что вы программист…	OEЯ, поиск
Релевантный поиск	Заставить поисковик вести себя может быть невыносимо…	Поисковые системы, релевантность
OAuth2 в действии	Если вы разработчик программного обеспечения в интернете сегодня…	Безопасность, OAuth
Властелин колец	…	Фэнтези, романы
Lucene в действии	Lucene – это мощная поисковая библиотека Java, которая позволяет…	Lucene, поиск

Алгоритм контролируемого обучения подает данные, подобные тем, которые показаны в таблице, во время так называемой *фазы обучения*. Во время фазы обучения алгоритм машинного обучения разбивает тренировочный набор (набор

обучающих образцов) и изучает, как отобразить, например, вводимый текст в выходные категории. То, *что* изучает алгоритм машинного обучения, зависит от задачи, для которой он используется; в этом примере он используется для *классификации документов*. То, *как* алгоритм учится, зависит от того, как он сам построен. Не существует только одного алгоритма для выполнения машинного обучения. Есть различные подобласти машинного обучения, и для каждого из них существует множество различных алгоритмов.

> **ПРИМЕЧАНИЕ** Глубокое обучение – это всего лишь один из способов машинного обучения с использованием нейронных сетей. Но существует множество альтернатив, когда речь заходит о том, какой тип нейронной сети лучше всего подходит для определенной задачи. В этой книге мы будем в основном освещать темы, касающиеся машинного обучения, посредством глубокого обучения. Тем не менее мы быстро рассмотрим другие типы алгоритмов, в основном для сравнения и обоснования реальных сценариев.

После завершения фазы обучения вы обычно будете получать *модель машинного обучения*. Можно рассматривать это как артефакт, который фиксирует то, чему алгоритм научился во время обучения. Этот артефакт затем используется для выполнения *прогнозов*. Прогноз выполняется, когда модели предоставляется новый ввод без какого-либо прикрепленного желаемого вывода, и вы просите модель сообщить вам правильный вывод, основываясь на том, чему она научилась на этапе обучения. Обратите внимание, что нужно предоставить большое количество данных для обучения (не сотни, а как минимум десятки тысяч обучающих образцов), если вы хотите получить хорошие результаты при прогнозировании итогов.

В примере с классификацией книг, когда дается приведенный ниже текст, модель извлечет такие категории, как «поиск» и «Lucene»:

```
Lucene - это мощная библиотека для поиска, написанная на языке Java, которая позволяет легко
добавлять поиск в любое приложение ...
```

Это вступительные слова из книги *Lucene в действии*, *2-е изд*.

Как я уже упоминал, извлеченные категории можно использовать для размещения книг, принадлежащих к одной и той же категории, на одной и той же полке в библиотеке. Существуют ли другие способы для достижения этой цели, без предварительного предоставления учебного набора с текстами книг, помеченными по категориям? Было бы полезно, если бы вы могли найти способ измерить сходство между книгами, чтобы иметь возможность разместить похожие книги рядом друг с другом, не слишком заботясь о точном названии каждой категории. Чтобы сделать это без категорий, можно использовать методы *неконтролируемого обучения* для объединения похожих документов. В этом случае, в отличие от контролируемого обучения, алгоритм машинного обучения просматривает данные без какой-либо информации о каком-либо ожидаемом результате и извлекает шаблоны и представления данных в ходе *фазы обучения*. Во время *кластеризации* каждый фрагмент входных данных, в данном случае текст книги, преобразуется в точку, которая размещается на графике. Во время фазы обучения алгоритм кластеризации размещает точки в кластерах, предполагая, что близлежащие точки семантически похожи. После завершения обучения книги, относящиеся к одним и тем же кластерам, можно соответствующим образом подбирать и размещать на книжных полках.

В этом случае результатом неконтролируемого обучения является набор кластеров с назначенными им точками. Как и раньше, такие модели можно использовать для прогнозов, например «К какому кластеру относится эта новая книга/точка?».

Машинное обучение может помочь решить множество различных проблем, включая классификацию книг и группирование похожих текстов. До начала 2000-х годов при решении таких задач для достижения достойных результатов использовалось несколько разных методов. Затем глубокое обучение стало мейнстримом не только в исследовательских лабораториях университетов, но и отрасли. Многие проблемы машинного обучения лучше было решать с помощью глубокого обучения, поэтому оно стало более известным и более часто используемым. Успех и широкое использование глубокого обучения привели к извлечению более точных категорий книг, более точной кластеризации и множеству других улучшений.

1.3. Что глубокое обучение может сделать для поиска

Когда для решения проблем, связанных с поиском, используются глубокие искусственные нейронные сети, это поле называется поиском на базе нейронных сетей. Из этой книги вы узнаете, как составлять нейронные сети, как они работают и как их можно использовать на практике, и все это в контексте поисковых систем.

> **Поиск на базе нейронных сетей**
>
> Термин «поиск на базе нейронных сетей» является менее академической формой термина «поиск информации на базе нейронных сетей», который впервые появился во время исследовательского семинара на конференции SIGIR 2016 (**www.microsoft.com/en-us/research/event/neuir2016**), посвященной применению глубоких нейронных сетей в области поиска информации.

Вам, наверное, интересно, зачем нужен поиск на базе нейронных сетей: в конце концов, у нас уже есть хорошие поисковые системы в интернете, и нам часто удается найти то, что нам нужно. Так в чем же состоит ценность этого поиска?

Глубокие нейронные сети хорошо подходят для того, чтобы:

- обеспечить представление текстовых данных, которое фиксирует семантику слов и документов, позволяя машине определить, какие слова и документы семантически похожи;
- генерировать текст, который имеет смысл в определенном контексте: например, полезен для создания чат-ботов;
- обеспечить представления изображений, которые относятся не к пикселям, а скорее к их составным объектам. Это позволяет нам создавать эффективные системы распознавания лиц/объектов;
- эффективно выполнять машинный перевод;
- при определенных предположениях аппроксимировать любую функцию[1]. Теоретически для видов задач, которые могут выполнять глубокие нейронные сети, не существует никаких ограничений.

[1] *См.*: *Kurt Hornik, Maxwell Stinchcombe, and Halbert White*. Multilayer Feedforward Networks Are Universal Approximators // Neural Networks 2. 1989. № 5: 359–366 (**http://mng.bz/Mxg8**).

Возможно, это звучит несколько абстрактно, поэтому давайте посмотрим, какую пользу эти возможности могут принести вам как инженеру, работающему с поисковыми системами, и/или пользователю. Подумайте об основных моментах при использовании поисковых систем. Скорее всего, вы столкнетесь с такими проблемами:

- я получил плохие результаты: я нашел несколько связанных документов, но не тот, который искал;
- мне потребовалось слишком много времени, чтобы найти информацию, которую я искал (а потом я сдался);
- я должен был прочитать некоторые из предоставленных результатов, прежде чем получить представление о теме, о которой я хотел узнать;
- я искал контент на своем родном языке, но смог найти подходящие результаты только на английском;
- я искал определенное изображение, которое когда-то видел на сайте, но не смог найти его снова.

Это распространенные проблемы, и чтобы сгладить их, существуют различные решения. Но самое интересное состоит в том, что глубокие нейронные сети, если их правильно настроить, могут помочь во всех этих случаях.

С помощью алгоритмов глубокого обучения поисковая система может:

- предоставить более релевантные результаты своим конечным пользователям, повышая степень их удовлетворенности;
- выполнить поиск по двоичному содержимому, например изображениям, так же как мы ищем текст. Рассматривайте это как возможность поиска изображения с фразой «изображение леопарда, который охотится на импалу» (а вы не Google);
- предоставить контент пользователям, говорящим на разных языках, что позволяет бóльшему количеству пользователей получать доступ к данным в поисковой системе;
- как правило, становиться более чувствительной к данным, которые она обслуживает, что означает меньше шансов для запросов, которые не дают результатов.

Если вы когда-либо работали над проектированием, реализацией или настройкой поисковой системы, то наверняка сталкивались с проблемой получения решения, которое адаптируется к вашим данным. Глубокое обучение очень помогает в решении этих проблем, которые строго основаны на ваших данных, а не на фиксированных правилах или алгоритмах.

Качество результатов поиска имеет решающее значение для конечных пользователей. Есть одна вещь, которую поисковая система должна делать хорошо: выяснить, какой из возможных подходящих результатов поиска будет наиболее полезным для информационных потребностей конкретного пользователя. Хорошо ранжированные результаты поиска позволяют пользователям находить важные результаты проще и быстрее; вот почему мы уделяем большое внимание теме *релевантных результатов*. В реальной жизни это может иметь огромное значение. Согласно статье, опубликованной в *Forbes*: «Предоставляя лучшие результаты поиска, компания Netflix считает, что избегает отмененных подписок, из-за чего ее доходы будут снижаться на 1 млрд долларов в год»[1].

---

[1] *Louis Columbus*. McKinsey's State of Machine Learning and AI, 2017. 2017. July 9 (**http://mng.bz/a7KX**).

Глубокие нейронные сети могут помочь, автоматически настраивая запрос конечного пользователя за кулисами на основе прошлых запросов пользователя или на основе содержимого поисковой системы.

Сегодня люди привыкли работать с поисковыми системами для поиска изображений. Например, если вы будете искать «изображения злых львов» в Google, то получите сильно релевантные изображения. До появления глубокого обучения такие изображения должны были быть украшены *метаданными* (данными о данных), описывающими их содержание, прежде чем их помещали в поисковую систему. И эти метаданные обычно должны были набираться человеком. Глубокие нейронные сети могут абстрагировать представление изображения, которое захватывает то, что там находится, поэтому не требуется никакого вмешательства со стороны человека, чтобы поместить описание изображения в поисковую систему.

В случае с такими сценариями, как поиск в интернете (поиск по всем сайтам в интернете), пользователи могут приходить со всего мира, поэтому лучше всего, если они могут осуществлять поиск на своих родных языках. Кроме того, поисковая система может выбирать профили пользователей и возвращать результаты на их языке, даже если они используют для поиска английский язык; это распространенный сценарий для технических запросов, потому что большое количество контента создается на английском языке. Интересное применение глубоких нейронных сетей носит название *нейронного машинного перевода*. Это набор техник, которые используют глубокие нейронные сети для перевода фрагмента текста с исходного языка на другой язык.

Также интересна возможность использования глубоких нейронных сетей для изменения способа, которым поисковые системы возвращают релевантную информацию конечным пользователям. Чаще всего поисковая система будет выдавать список результатов поиска в ответ на поисковый запрос. Можно использовать методы глубокого обучения, чтобы позволить поисковой системе вернуть один фрагмент текста, который должен дать всю необходимую пользователю информацию[1]. Это избавит пользователей от просмотра каждого результата, чтобы получить все то, что им необходимо. Мы могли бы даже объединить все эти идеи и создать поисковую систему, органично предоставляющую пользователям со всего мира и текст, и изображения, которая, вместо того чтобы возвращать результаты поиска, возвращает один фрагмент текста или изображения, который нужен пользователю.

Эти приложения представляют собой примеры *поиска на базе нейронных сетей*. Как вы можете себе представить, они могут революционизировать то, как мы работаем и используем поисковые системы сегодня.

Есть много возможностей относительно того, как компьютеры могут помочь людям получить необходимую им информацию. Обсуждение нейронных сетей ведется на протяжении последних нескольких лет, но только недавно они стали так популярны; связано это с тем, что исследователи обнаружили, как сделать их намного эффективнее, чем раньше. Например, в начале 2000-х годов добавление помощи со стороны более мощных компьютеров стало ключевым шагом вперед. Чтобы воспользоваться всем потенциалом глубоких нейронных сетей, люди, ин-

[1] *Christina Lioma et al.* Deep Learning Relevance: Creating Relevant Information (As Opposed to Retrieving It). 2016. June 27 (**https://arxiv.org/pdf/1606.07660.pdf**).

тересующиеся компьютерными науками, особенно в области обработки естественного языка, компьютерного зрения и поиска информации, должны знать, как такие искусственные нейронные сети работают на практике.

Эта книга предназначена для тех, кто интересуется созданием интеллектуальных поисковых систем с помощью глубокого обучения. Это не обязательно означает, что вы будете создавать еще один поисковик Google. Это может означать, что то, что вы узнали здесь, будет использовано для проектирования и реализации эффективной поисковой системы для вашей компании или расширения вашей базы знаний, чтобы применять методы глубокого обучения в более крупных проектах, которые могут включать в себя поисковые системы.

Цель здесь состоит в том, чтобы обогатить ваши навыки, связанные с поисковыми системами и ГО, поскольку эти навыки могут быть полезны в различных контекстах. Например:

- обучение глубокой нейронной сети, чтобы научиться распознавать объекты на изображениях и использовать то, чему научилась нейронная сеть при поиске изображений;
- использование нейронных сетей для заполнения панели «связанный контент» в списке результатов поиска поисковой системы;
- обучение нейронных сетей, чтобы научиться делать пользовательский запрос более конкретным (меньше результатов поиска, но они лучше) или шире (больше результатов поиска, даже если некоторые из них могут быть менее актуальными).

1.4. Глубокое обучение: дорожная карта

Мы будем запускать наши примеры поверх программного обеспечения с открытым исходным кодом, написанного на Java, с помощью Apache Lucene (**http://lucene.apache.org**), библиотеки для поиска информации, и Deeplearning4j (**http://deeplearning4j.org**), библиотеки, используемой как фреймворк для глубокого обучения. Но мы по возможности сосредоточимся на принципах, а не на реализации, чтобы убедиться, что методы, описанные в этой книге, могут использоваться с различными технологиями и/или сценариями. На момент написания этих строк Deeplearning4j был широко используемым фреймворком для глубокого обучения в корпоративных сообществах; является частью организации Eclipse Foundation. Он также хорошо зарекомендовал себя благодаря интеграции с популярными фреймворками для реализации распределенной обработки данных, такими как Apache Spark. Полный исходный код, содержащийся в этой книге, можно найти на сайте **www.manning.com/books/deep-learning-for-search** и на GitHub по адресу **https://github.com/dl4s/dl4s**. Однако существуют и другие фреймворки для глубокого обучения. Например, TensorFlow (от компании Google) популярен среди сообществ Python и исследовательских сообществ. Почти каждый день появляются новые инструменты, поэтому я решил сосредоточиться на относительно простом в использовании фреймворке, который может быть легко интегрирован с Lucene, одной из наиболее распространенных поисковых библиотек для виртуальной машины Java. Если вы работаете с Python, то можете найти реализации большей части кода для TensorFlow, использованного в этой книге, а также некоторые инструкции по GitHub по адресу **https://github.com/dl4s/pydl4s**.

Планируя эту книгу, я решил представить главы в порядке возрастания уровня сложности, поэтому каждая глава научит определенному применению нейронных сетей для конкретной задачи поиска, поддерживаемой известными алгоритмами. Мы будем следить за современными алгоритмами глубокого обучения, но мы также осознаем, что нельзя охватить все. Цель состоит в том, чтобы обеспечить подходящие исходные данные, которые можно легко расширить, если на следующей неделе появится новый и более совершенный алгоритм на основе нейронной сети. Ключевые вещи, которые мы улучшим с помощью глубоких нейронных сетей, – это релевантность, понимание запросов, поиск изображений, машинный перевод и рекомендательные системы, работающие с документами. Не беспокойтесь, если ни один из этих терминов вам не знаком: я представлю эти задачи как есть, без какой-либо техники глубокого обучения, а затем покажу, когда и как ГО может помочь.

В первой части книги я дам обзор того, как нейронные сети могут помочь улучшить поисковые системы в целом. Сначала я сделаю это, используя приложение, в котором нейронные сети помогают поисковой системе создавать несколько версий одного и того же запроса, генерируя синонимы. Во второй части книги мы рассмотрим методы на базе глубокого обучения, чтобы сделать поисковые запросы более выразительными. Эта улучшенная выразительность сделает запросы более подходящими для целей пользователя и, таким образом, заставит поисковую систему возвращать более точные (более релевантные) результаты. Наконец, в третьей части книги мы будем работать над более сложными вещами, такими как поиск на нескольких языках и поиск изображений, и, наконец, рассмотрим аспекты производительности поисковых систем на базе нейронных сетей.

Попутно мы также сделаем паузу, чтобы рассмотреть верность и то, как измерить окончательные результаты, когда мы применяем поиск на базе нейронных сетей. Без цифр, постоянно демонстрирующих то, что мы считаем подходящим, мы далеко не уйдем. Нам нужно измерить, насколько хороши наши системы с использованием и без использования причудливых нейронных сетей.

В этой главе мы начнем с рассмотрения проблем, которые пытаются решить поисковые системы, и наиболее распространенных методов, используемых для их решения. Это исследование познакомит вас с основами анализа, загрузки и извлечения текста в поисковой системе, чтобы вы могли узнать, как запросы попадают в результаты поиска, а также с некоторыми основами решения проблемы, связанной с возвратом релевантных результатов в первую очередь. Мы также раскроем некоторые слабые стороны, присущие обычным методам поиска, что приводит к обсуждению того, для чего можно использовать глубокое обучение в контексте поиска. Затем мы посмотрим, какие задачи может помочь решить ГО и каковы практические последствия его применения в области поиска. Это поможет нарисовать реалистичную картину того, что вы можете и чего не можете ожидать от нейронного поиска в реальных ситуациях.

1.5. Получение полезной информации

Давайте начнем с изучения того, как получать результаты поиска, которые соответствуют нуждам пользователей. Это даст вам основы, необходимые для понимания того, как глубокие нейронные сети могут помочь в создании инновационных поисковых платформ.

Первый вопрос: что такое поисковая система? Это система, программа, работающая на компьютере, которую люди могут использовать для получения информации. Основная ценность поисковой системы состоит в том, что хотя она и принимает «данные», она должна предоставлять «информацию». Эта цель означает, что поисковая система должна делать все возможное, чтобы разобраться в данных, которые она получает, чтобы предоставить нечто, что может быть легко потреблено ее пользователями. Будучи пользователями, мы редко нуждаемся в большом количестве данных по определенной теме; мы часто ищем конкретную информацию, и нас бы удовлетворил только *один* ответ, а не сотни или тысячи результатов, которые нужно проверять.

Когда дело доходит до поисковых систем, большинство людей обычно думает о Google, Bing, Baidu и других крупных популярных поисковых системах, которые предоставляют доступ к огромным объемам информации, поступающей из множества разнообразных источников. Но есть также много небольших поисковых систем, которые фокусируются на контенте из определенной области или темы. Их часто называют *вертикальными поисковыми системами*, потому что они работают с ограниченным набором типов документов или тем, а не со всем контентом, который в настоящее время находится в сети. Вертикальные поисковые системы также играют важную роль, потому что часто они могут предоставить более точные результаты о «своих» данных – поскольку были адаптированы к этому конкретному контенту. Они нередко позволяют нам получать более точные результаты с более высокой верностью (например, сравните поиск академической статьи в Google и Google Scholar). (Пока мы не будем вдаваться в подробности того, что означает понятие *верность*; здесь я говорю об общей концепции верности ответа на запрос. Но верность – это также название четко определенной меры, используемой для того, чтобы оценить, насколько хороши и точны результаты информационно-поисковой системы.) На данном этапе мы не будем разграничивать размер данных и базы пользователей, потому что все концепции, которые идут далее, применимы к большинству существующих поисковых систем, независимо от их размера.

Основные обязанности поисковой системы обычно включают в себя:

- *индексирование* – эффективная загрузка и хранение данных, что позволяет быстро их извлекать;
- *запросы* – обеспечение функциональности поиска, чтобы конечный пользователь мог выполнять поиск;
- *ранжирование* – представление и ранжирование результатов в соответствии с определенными метриками для наилучшего удовлетворения информационных потребностей пользователей.

Ключевым моментом на практике также является *эффективность*. Если для получения искомой информации требуется слишком много времени, скорее всего, в следующий раз вы переключитесь на другую поисковую систему.

Но как поисковая система работает со страницами, книгами и подобными текстами? В следующих разделах вы узнаете:

- как большие куски текста разделяются на более мелкие части, чтобы поисковая система могла принять заданный запрос и быстро получить документ;
- основы того, как зафиксировать важность и релевантность результатов поиска для конкретного запроса.

Давайте начнем с основ поиска информации (индексация, запросы и ранжирование). Прежде чем заняться этим, вы должны понять, как большие потоки текстов попадают в поисковую систему; это важно, потому что это влияет на возможности быстрого поиска поисковой системы и предоставления важных результатов.

1.5.1. Текст, токены, термы и основы поиска

Поставьте себя на место библиотекаря, который только что получил запрос на книги, связанные с определенной темой. Как узнать, что одна из книг содержит информацию по определенной теме? Как узнать, что книга даже содержит определенное слово?

Извлечение категорий, к которым принадлежит определенная книга (темы высокого уровня, такие как «искусственный интеллект» и «глубокое обучение»), отличается от извлечения всех слов, содержащихся в книге. Например, категории облегчают поиск книги об искусственном интеллекте для новичка, поскольку предварительного знания специфических для искусственного интеллекта методов или авторов не требуется. Пользователь зайдет на сайт поисковой системы и начнет просматривать существующие категории и искать что-то, что достаточно близко к теме искусственного интеллекта. С другой стороны, для эксперта по искусственному интеллекту знание того, содержит ли книга слова *градиентный спуск* или *метод обратного распространения ошибки*, позволяет находить результаты, которые содержат более детальную информацию об определенных методах или проблемах в области ИИ.

Людям, как правило, трудно вспомнить все слова, содержащиеся в книге, хотя мы легко можем определить тему книги, прочитав несколько абзацев из нее или даже посмотрев на пролог или предисловие. Компьютеры, как правило, ведут себя иначе. Они могут легко хранить большие объемы текста и «запоминать» все слова, содержащиеся на миллионах страниц, чтобы их можно было использовать при поиске; с другой стороны, они не так хорошо извлекают информацию, которая скрыта, разбросана или не сформулирована непосредственно в данном фрагменте текста, например к какой категории относится книга. Например, в книге о нейронных сетях никогда не упоминается «искусственный интеллект» (хотя это, вероятно, будет относиться к машинному обучению). Но это все равно относится к широкой категории «книг об искусственном интеллекте».

Давайте сначала посмотрим на задачу, которую компьютеры уже могут выполнять: извлечение и хранение фрагментов текста (также известных как *термы*) из потоков текста. Можно рассматривать этот процесс, который носит название *анализ текста*, как разбиение текста книги на все составляющие его слова. Представьте себе ленту, на которой содержимое книги написано потоком, и машину (алгоритм анализа текста), в которую вы вставляете такую ленту в качестве ввода. Вы получаете множество фрагментов такой ленты в качестве результатов, и каждый из этих выходных фрагментов содержит слово, или предложение, или именную фразу (например, «искусственный интеллект»); вы можете понять, что некоторые слова, записанные на ленте ввода, были съедены машиной и не были выведены в какой-либо форме.

Поскольку последние единицы, которые должны быть созданы алгоритмом анализа текста, могут быть словами, но также могут быть группой слов, или предложений, или даже частями слов, мы называем эти фрагменты термами. Можно

рассматривать терм как основную единицу, которую поисковая система использует для хранения данных и, следовательно, их извлечения.

Это основа одной из самых фундаментальных форм поиска – *поиска по ключевым словам*: пользователь вводит набор слов и ожидает, что поисковая система выдаст все документы, которые содержат некоторые или все эти термы. Так начинался поиск в интернете десятилетия назад. Хотя сегодня многие поисковые системы намного умнее, немало пользователей продолжает составлять запросы на основе ключевых слов, которые, как они ожидают, будут содержаться в результатах поиска. Это то, что вы увидите сейчас: как текст, введенный пользователем в поле поиска, заставляет поисковую систему возвращать результаты. *Запрос* – это то, что мы называем текстом, который вводит пользователь, чтобы что-то найти. Хотя запрос – это просто текст, он передает и кодирует то, что нужно пользователю, и то, как пользователь выражает эту, возможно, общую или абстрактную потребность (например, «Я хочу узнать о последних и самых больших исследованиях в области искусственного интеллекта») в способ, который является кратким, но все же описательным (например, «последнее исследование в области искусственного интеллекта», как показано на рис. 1.3).

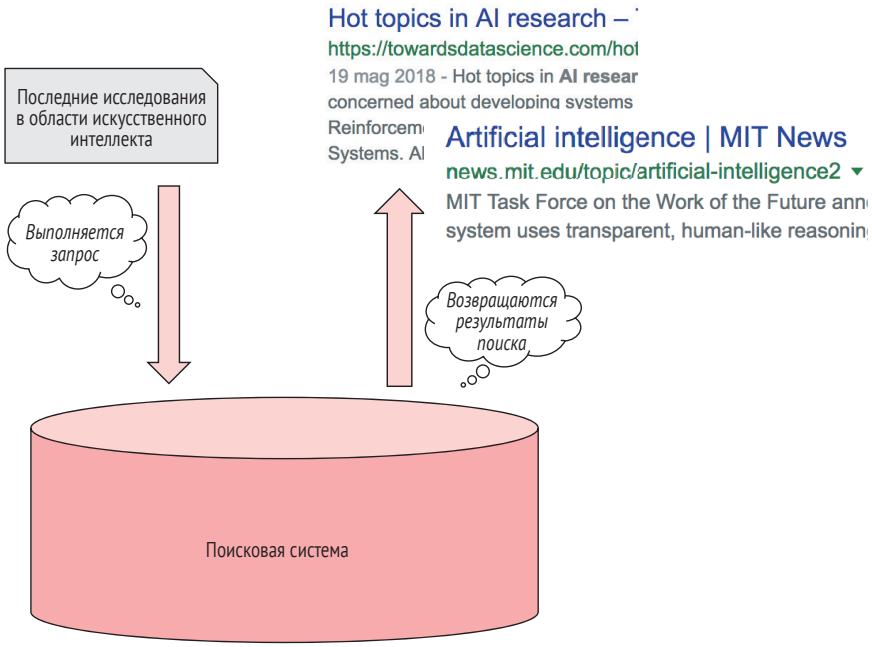


Рис. 1.3 ❖ Поиск и получение результатов

Если, будучи пользователем, вы хотите найти документы, содержащие слово «поиск», как поисковая система будет возвращать такие документы? Не слишком умный способ сделать это – просмотреть содержимое каждого документа с самого начала и сканировать его, пока поисковая система не найдет совпадение. Но выполнять такие проверки текста для каждого запроса очень затратно, особенно когда речь идет о большом количестве больших документов:

- многие документы могут не содержать слова «поиск»; поэтому их сканирование было бы напрасной тратой вычислительных ресурсов;
- даже если документ содержит слово «поиск», это слово может встречаться ближе к концу документа, что требует от поисковой системы «прочитать» все предыдущие слова, прежде чем найти совпадение для слова «поиск».

У вас есть *совпадение* или *попадание*, когда в результате поиска найден один или несколько *термов*, являющихся частью запроса.

Вам нужно найти способ быстро вычислить этот этап поиска. Одним из фундаментальных методов для достижения данной цели является разбиение предложений типа «мне нравятся поисковые системы» на более мелкие единицы: в данном случае [«мне», «нравятся», «поисковые», «системы»]. Это обязательное условие для эффективных механизмов хранения, называемых *инвертированными индексами*, о которых мы поговорим в следующем разделе. Программа анализа текста часто организована в виде конвейера: цепочки компонентов, каждый из которых принимает вывод предыдущего компонента в качестве ввода. Такие конвейеры обычно состоят из строительных блоков двух типов:

- *токенизаторы* – компоненты, которые разбивают поток текста на слова, фразы, символы или другие единицы, называемые *токенами*;
- *фильтры токенов* – компоненты, принимающие поток токенов (от токенизатора или другого фильтра) и могущие изменять, удалять или добавлять новые токены.

Вывод таких конвейеров анализа текста представляет собой последовательность следующих друг за другом термов, как показано на рис. 1.4.

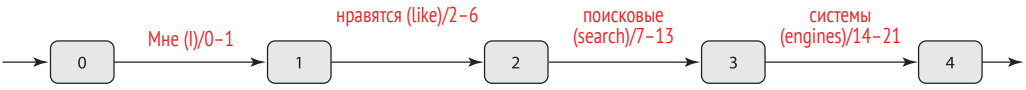


Рис. 1.4 ❖ Получение слов «мне нравятся поисковые системы» с использованием простого конвейера анализа текста

Теперь вы знаете, что анализ текста полезен по соображениям производительности для создания быстрых поисковых систем. Другим не менее важным аспектом является то, что он контролирует соответствие запросов и текста, которые будут помещены в индекс. Часто конвейеры анализа текста используются для фильтрации токенов, которые не считаются полезными или необходимыми для поисковой системы. Например, избегать хранения распространенных термов, таких как статьи или предлоги, в поисковой системе является обычной практикой, поскольку эти слова существуют в большинстве текстовых документов на таких языках, как английский, и обычно вам не нужно, чтобы запрос возвращал все в поисковой системе: это не принесло бы много пользы пользователю. В подобных случаях можно создать фильтр, отвечающий за удаление таких токенов, как «the», «a», «an», «of», «in» и т. д., позволяя всем остальным токенам вытекать по мере того, как токенизатор будет производить их. В этом упрощенном примере:

- токенизатор будет разделять токены каждый раз, когда будет встречать символ пробела;
- фильтр токенов удалит токены, которые соответствуют определенному черному списку (также известному как список *стоп-слов*).

В реальной жизни, особенно при первоначальной настройке поисковой системы, принято создавать несколько различных алгоритмов анализа текста и проверять их на данных, которые вы хотите поместить в поисковую систему. Это позволяет визуализировать, как контент будет обрабатываться такими алгоритмами, например какие токены генерируются, какие в конечном итоге отфильтровываются и т. д. Вы создали эту цепочку анализа текста (также называемую *анализатором*) и хотите убедиться, что она работает должным образом и фильтрует статьи, предлоги и т. д. Давайте попробуем передать первый фрагмент текста упрощенному анализатору и отправить предложение «the brown fox jumped over the lazy dog» (бурая лисица перепрыгнула через ленивого пса) в конвейер. Вы ожидаете, что статьи будут удалены. Сгенерированный выходной поток будет выглядеть так, как показано на рис. 1.5.

Рис. 1.5 ❖ Схема токенов

В полученном потоке токенов, как и ожидалось, «токены» удалены; это можно увидеть из пунктирных стрелок в начале графа и между узлами «over» и «lazy». Числа рядом с токенами обозначают начальную и конечную позиции (в количестве символов) каждого токена. Важным моментом этого примера является то, что запрос «the» не будет соответствовать, потому что анализатор удалил все подобные токены, и они не будут частью содержимого поисковой системы. В реальной жизни конвейеры анализа текста часто являются более сложными; некоторые из них вы увидите в следующих главах. Теперь, когда вы познакомились с анализом текста, давайте посмотрим, как поисковые системы хранят текст (и термы), чтобы они были запрошены конечными пользователями.

Индексация

Хотя поисковая система должна разбивать текст на термы для быстрого поиска, конечные пользователи ожидают, что результаты поиска будут представлены в виде единого блока: документа. Представьте себе результаты поиска в Google. Если вы ищете слово «книги», то получите список результатов, каждый из которых состоит из заголовка, ссылки, фрагмента текста и т. д. Каждый из этих результатов содержит слово «книги», но вы видите документ, который содержит гораздо больше информации и контекста, чем просто текстовый фрагмент, где это слово совпало. На практике токены, полученные в результате анализа текста, хранятся со ссылкой на оригинальный фрагмент текста, которому они принадлежат.

Эта связь между термом и документом позволяет:

- подобрать ключевое слово или поисковый терм из запроса;
- вернуть указанный исходный текст в качестве результата поиска.

Весь этот процесс анализа потоков текста и хранения результирующих термов (вместе со ссылочными документами) в поисковой системе обычно называют *индексацией*.

Причиной такой формулировки является то, что термы хранятся в *инвертированном индексе*: структуре данных, которая отображает терм в текст, в котором он

изначально содержался. Вероятно, самый простой способ взглянуть на это – это аналитический указатель реальной книги, где каждое слово указывает на страницы, на которых оно упомянуто; в случае с поисковой системой слова – это термы, а страницы – оригинальные части текста.

Отныне мы будем обозначать фрагменты текста для индексирования (страницы, книги) как *документы*. Чтобы наглядно представлять себе, что происходит с документами после индексации, давайте предположим, что у вас есть два очень похожих документа:

- «the brown fox jumped over the lazy dog» (бурая лисица перепрыгнула через ленивого пса) (документ 1);
- «a quick brown fox jumps over the lazy dog» (быстрая бурая лисица перепрыгивает через ленивого пса) (документ 2).

Предполагая, что вы используете алгоритм анализа текста, определенный ранее (токенизация пробелов со стоп-словами «a», «an» и «the»), в табл. 1.3 приводится подходящая аппроксимация инвертированного индекса, содержащего такие документы.

Таблица 1.3. Инвертированный индекс

Терм	Идентификаторы документа
brown	1, 2
fox	1, 2
jumped	1
over	1, 2
lazy	1, 2
dog	1, 2
quick	2
jumps	1

Как видно, здесь нет записи для терма «the», потому что фильтр токенов на основе стоп-слов удалил подобные токены. В таблице можно найти словарь термов в первом столбце и *постинг-лист* – набор идентификаторов документов, – связанный с каждым термом в каждой строке. При использовании инвертированных индексов поиск документов, содержащих данный терм, выполняется очень быстро: поисковая система выбирает инвертированный индекс, ищет запись для поискового терма и в конечном итоге извлекает документы, содержащиеся в постинг-листе. В примере индекса, если вы ищете терм «quick» (быстрая), инвертированный индекс вернет документ 2, просмотрев постинг-лист, соответствующий терму «quick». Мы только что рассмотрели быстрый пример индексации текста в поисковую систему.

Давайте подумаем о шагах по индексации книги. Книга состоит из страниц, основного содержимого, но у нее также есть название, автор, редактор, год публикации и т. д. Вы не можете использовать один и тот же конвейер анализа текста для всего, и вам бы не хотелось удалять слова «the» или «an» из названия книги. Пользователь, знающий название книги, должен найти ее с помощью точного сопоставления! Если цепочка анализа текста удаляет предлог «in» (в) из названия книги *Tika in Action* («Тика в действии»), запрос «Tika in Action» не позволит

вам найти ее. С другой стороны, вы можете избежать хранения таких токенов для содержания книги, поэтому у вас есть конвейер анализа текста, который более агрессивен при фильтрации нежелательных термов. Если цепочка анализа текста удаляет слова «in» и «the» из названия книги *Living in the Information Age* («Жизнь в век информации»), это не должно быть проблематичным: очень маловероятно, что пользователь будет искать «Living in the Information Age», но он может искать «information age». В этом случае потери информации практически нет, но вы получаете преимущество хранения небольших текстов и, что более важно, повышения релевантности (об этом мы поговорим в следующем разделе). Обычный подход в реальной жизни состоит в том, чтобы иметь несколько инвертированных индексов, которые обращаются к индексации различных частей документа, и все это происходит в одной поисковой системе.

Поиск

Теперь, когда у нас есть контент, проиндексированный в поисковой системе, мы рассмотрим поиск. Традиционно первые поисковые системы позволяли пользователям выполнять поиск с помощью определенных термов, также известных как *ключевые слова*, и в конечном итоге логических операторов, которые позволяют пользователям определять, какие термы *должны* совпадать, *не должны* совпадать или *могут* совпадать в результатах поиска. Наиболее часто терм в запросе *должен* совпадать, но это не обязательно. Если вам нужны результаты поиска, которые должны содержать такой терм, вы должны добавить соответствующий оператор: например, поставить знак + перед термом. Запрос типа «deep + learning for search» требует результатов, которые содержат и слово «deep», и слово «learning», а еще могут содержать слова «for» и «search». Также принято разрешать пользователям указывать, что им нужны целые фразы, а не отдельные термы. Это позволяет им искать точную последовательность слов вместо отдельных термов. Предыдущий запрос можно перефразировать как «“deep learning” for search», чтобы возвращать результаты поиска, которые должны содержать последовательность «deep learning» и, необязательно, термы «for» и «search».

Это может звучать удивительно, но анализ текста также важен на этапе поиска. Предположим, вы хотите найти книгу *Deep Learning for Search* поверх данных, которые вы только что проиндексировали; при условии что у вас есть веб-интерфейс, вы, вероятно, наберете запрос типа «deep learning for search». Задача на этом этапе поиска состоит в том, чтобы сделать возможным поиск нужной книги. Первое, что находится между пользователем и пользовательским интерфейсом классической поисковой системы, – это *синтаксический анализатор запросов*.

Анализатор запросов отвечает за преобразование текста поискового запроса, введенного пользователем, в набор операторов, которые указывают, какие термы должна искать поисковая система и как их использовать при поиске совпадения в инвертированных индексах. В предыдущих примерах запросов анализатор отвечал за понимание символов + и ”. Еще один широко распространенный синтаксис позволяет помещать логические операторы в термы запроса: «deep AND learning». В этом случае анализатор запросов придаст особое значение оператору «AND»: термы слева и справа от него являются обязательными. Синтаксический анализатор запросов можно рассматривать как функцию, которая берет некий текст и выводит набор ограничений для применения к базовому инвертированному индексу (индексам) для поиска результатов. Давайте снова возьмем при-

мер запроса типа «latest research in artificial intelligence» (последние исследования в области искусственного интеллекта). Умный анализатор запросов будет создавать операторы, отражающие семантику слов; например, вместо двух операторов для слов «artificial» и «intelligence» он должен создать только один оператор для «artificial intelligence». Кроме того, вероятно, терм «latest» не должен совпадать; вам не нужны результаты, содержащие слово «latest»; вместо этого вам нужно получить результаты, которые были «созданы» недавно. Таким образом, хороший анализатор запросов преобразует терм «latest» в оператор, который можно выразить, например, как «созданный между сегодняшним днем и двумя месяцами ранее» на естественном языке. Механизм запросов будет кодировать такого оператора таким образом, чтобы его было легче обработать с помощью компьютера, например `created < today() AND created > (today() - 60days)`; *см.* рис. 1.6.

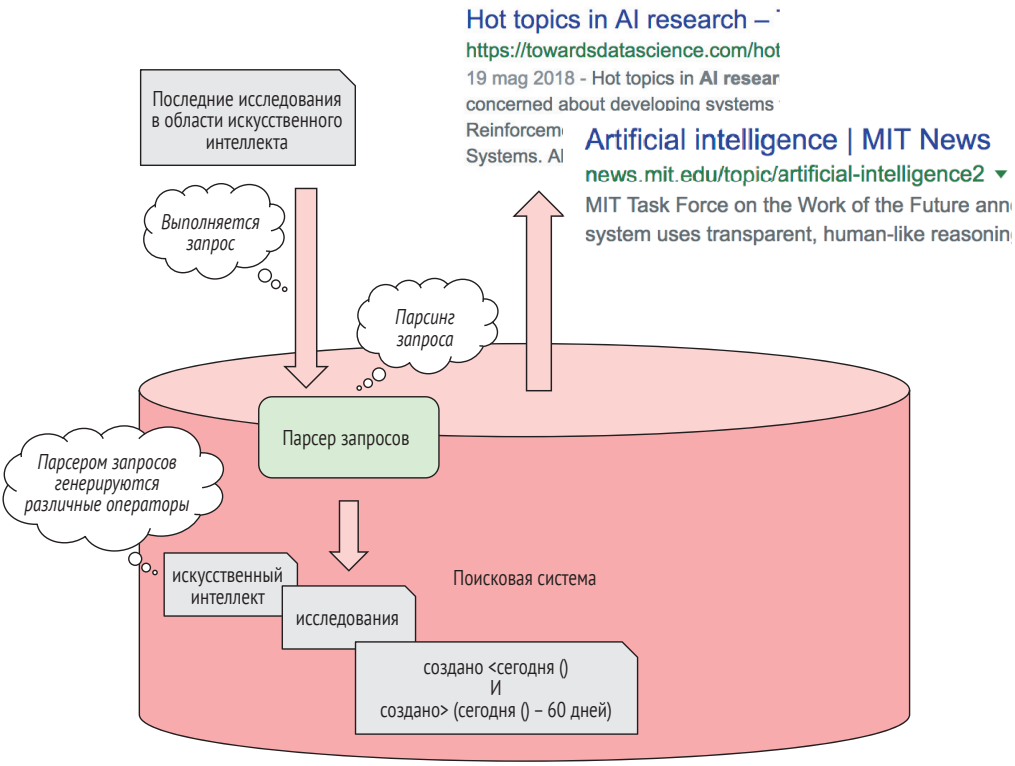


Рис. 1.6 ❖ Парсинг запросов

Во время индексации конвейер анализа текста используется для разделения входного текста на термы, которые должны храниться в индексе; это также называется *анализом текста во время индексации*. Точно так же анализ текста может применяться во время поиска по запросу, чтобы разбить строку запроса на термы, поэтому это называется *анализом текста во время поиска*. Документ извлекается поисковой системой, когда термы времени поиска совпадают с термом в инвертированном индексе, на который ссылается этот документ.

На рис. 1.7 показан анализ во время индексации, который используется для разделения текста документа на термы. Они попадают в индекс и все ссылаются на *документ 1*. Анализ во время индексации состоит из токенизатора пробелов и двух фильтров: первый используется для удаления нежелательных стоп-слов (например, «the»), а второй – для преобразования всех термов в нижний регистр (например, «Fox» преобразуется в «fox»). В правом верхнем углу запрос «lazy foxes» (ленивые лисицы) передается в анализ во время поиска, который разбивает токены с помощью токенизатора пробельных символов, но выполняет фильтрацию с использованием фильтра строчных букв и стемминг-фильтра. Стемминг-фильтр преобразует термы, приводя флективные или производные слова к их корневой форме; это означает удаление множественных суффиксов, -ing формы в глаголах и т. д. В этом случае «foxes» превращаются в «fox».

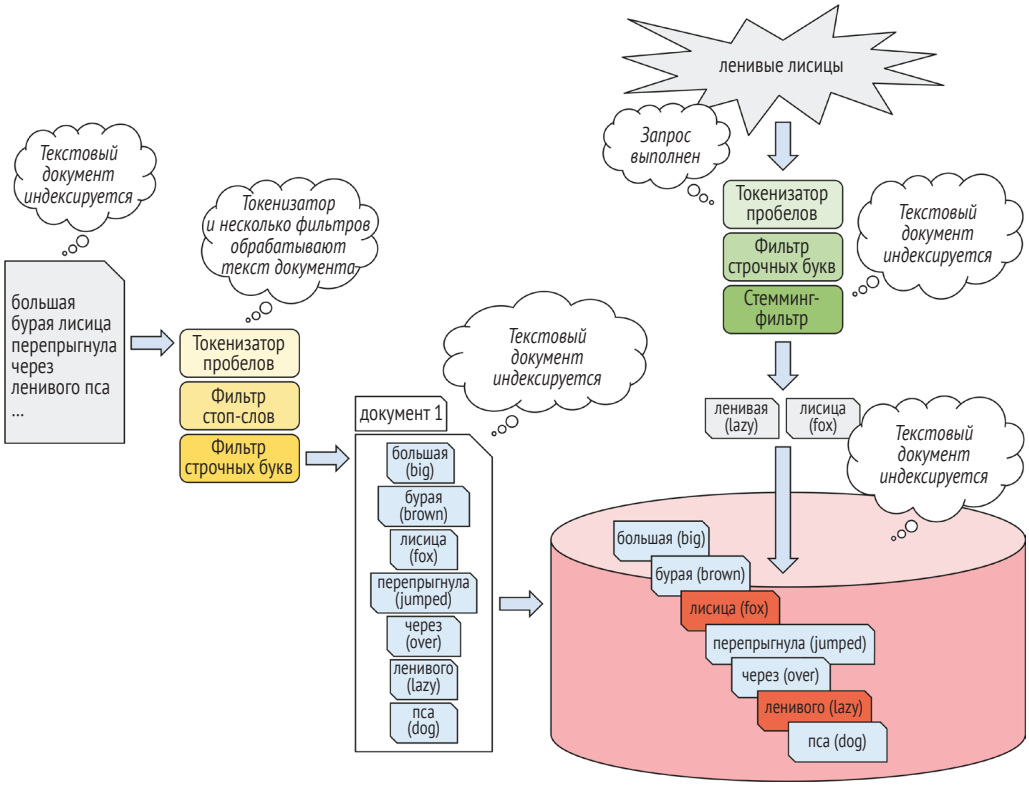


Рис. 1.7 ❖ Индекс, анализ времени поиска и сопоставление термов

Привычный способ убедиться, что конвейеры индексирования и анализа текста работают должным образом, состоит в следующем:

1) взять образец контента;
2) передать контент в цепочку анализа текста во время индексации;
3) взять пример запроса;
4) передать запрос в цепочку анализа текста во время поиска;
5) проверить полученные термы на предмет соответствия.

Например, во время индексации обычно используются фильтры стоп-слов, потому что выполнение фильтрации не повлияет на производительность на этапе поиска. Но возможно иметь и другие фильтры на этапах индексирования или поиска. Имея цепочки анализа текста во время индексации и поиска и синтаксический анализ запросов, мы можем увидеть, как работает процесс получения результатов поиска.

Вы изучили один из базовых методов, лежащих в основе каждой поисковой системы: анализ текста (токенизация и фильтрация) позволяет системе разбивать текст на термы, которые, как вы ожидаете, пользователи будут вводить во время запроса, и помещать их в структуру данных под названием инвертированный индекс, который обеспечивает эффективное хранение (по пространству) и поиск (по времени). Однако, будучи пользователями, мы не хотим просматривать все результаты поиска, поэтому нужна поисковая система, чтобы она сообщала нам, какие из них должны быть наиболее подходящими. Теперь вам, наверное, интересно, что значит *наиболее подходящие*? Можно ли измерить, насколько полезна информация, учитывая наши запросы? Ответ – да: мы называем это *релевантностью*. Точное ранжирование результатов поиска – одна из самых важных задач, которую должен выполнить поисковик. В следующем разделе мы кратко рассмотрим, как решить проблему релевантности.

1.5.2. Релевантность прежде всего

Теперь вы знаете, как поисковые системы получают документ по заданному запросу. В этом разделе вы узнаете, как поисковые системы ранжируют результаты поиска, чтобы самые важные результаты возвращались первыми. Это даст вам четкое представление о работе обычных поисковых систем.

Релевантность является ключевым понятием в поиске; это мера того, насколько важен документ для определенного поискового запроса. Будучи людьми, нам часто легко определить, почему некоторые документы более актуальны, чем другие, в отношении запроса. Таким образом, теоретически мы могли бы попытаться извлечь набор правил, отражающих наши знания о ранжировании важности документа. Но на практике такое упражнение, вероятно, потерпит неудачу:

- объем информации, которая у нас есть, не позволяет нам извлечь набор правил, применимых к большинству документов;
- документы в поисковой системе со временем сильно меняются, и очень важно постоянно корректировать правила соответствующим образом;
- документы в поисковой системе могут принадлежать разным доменам (например, в поиске по сети), и невозможно найти хороший набор правил, который работает для всех типов информации.

Одной из центральных тем в области поиска информации является определение модели, для которой не требуется специалист для извлечения таких правил. Подобная *модель поиска* должна как можно точнее охватывать понятие релевантности. Учитывая набор результатов поиска, поисковая модель будет ранжировать *каждый* из них: чем более релевантен результат, тем выше его оценка.

В большинстве случаев, будучи специалистом по разработке поисковых систем, вы не получите идеальных результатов, просто выбрав поисковую модель; релевантность – это капризная бестия! В реальных сценариях вам, возможно, придется постоянно корректировать свои конвейеры анализа текста, а также модель

поиска и, возможно, выполнять тонкую настройку внутренних компонентов поисковой системы. Но модели поиска очень помогают, предоставляя надежную основу для получения хорошей релевантности.

1.5.3. Классические модели поиска

Вероятно, одной из наиболее часто используемых моделей поиска информации является модель векторного пространства[1]. В этой модели каждый документ и запрос представлен в виде вектора. Можно рассматривать вектор как стрелку в координатной плоскости; каждая стрелка в векторной модели может обозначать запрос или документ. Чем ближе две стрелки, тем больше они похожи (*см.* рис. 1.8); направление каждой стрелки определяется термами, которые образуют запрос/документ.

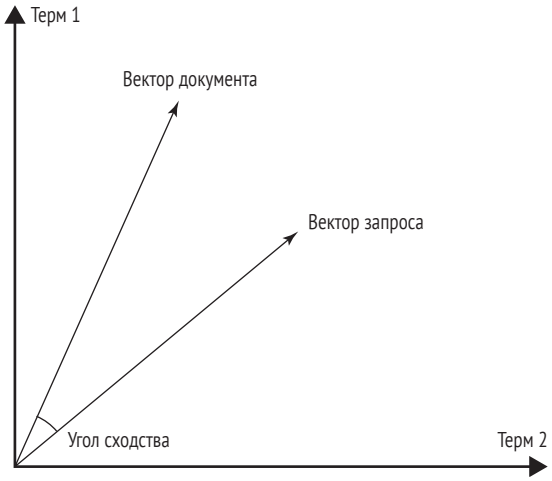


Рис. 1.8 ❖ Сходство между векторами документа и запроса согласно модели векторного пространства

В таком векторном представлении каждый терм связан с *весом*: действительным числом, которое указывает, насколько этот терм важен в этом документе/запросе по отношению к остальным документам в поисковой системе. Такой вес можно рассчитать различными способами. На этом этапе мы не будем вдаваться в подробности того, как это делается. Я упомяну, что наиболее распространенный алгоритм называется TF-IDF (*term frequency – inverse document frequency – частота терма – обратная частота документа*). Основная идея, лежащая в основе TF-IDF, заключается в том, что чем чаще терм появляется в одном документе (частота терма, или TF), тем он важнее. В то же время он устанавливает, что чем более распространен терм среди всех документов, тем менее он важен (обратная частота документа, или IDF). Таким образом, в векторной модели результаты поиска ранжируются по вектору запроса, поэтому документы отображаются выше в списке

[1] *См.*: *G. Salton, A. Wong, and C. S. Yang*. A vector space model for automatic indexing // Communications of the ACM 18. 1975. № 11: 613–620 (**http://mng.bz/gNxG**).

результатов (получают более высокий ранг/оценку), если они находятся ближе к такому вектору.

В то время как векторная модель – это информационно-поисковая модель на базе линейной алгебры, с годами появились альтернативные подходы, основанные на вероятностных моделях релевантности. Вместо того чтобы вычислять, насколько близко находятся документ и вектор запроса, вероятностная модель ранжирует результаты поиска на основе оценки вероятности того, что документ является релевантным к определенному запросу. Одна из самых распространенных функций ранжирования для таких моделей – это *Okapi BM25*. Мы не будем углубляться в детали, но она показала хорошие результаты, особенно в не очень длинных текстах.

1.5.4. Точность и полнота

Мы рассмотрим, как поиск на базе нейронных сетей может помочь в определении релевантности в следующих главах, но для начала нам нужно измерить релевантность! Стандартный способ измерения того, насколько хорошо работает информационно-поисковая система, – это расчет ее точности и повторного вызова. *Точность* – это доля найденных документов, которые являются релевантными. Если система имеет высокую точность, пользователи в основном находят результаты поиска в верхней части списка результатов. *Полнота* – это доля релевантных документов, которые были найдены. Если у системы хорошая полнота, пользователи найдут все релевантные для них результаты в результатах поиска, хотя не все они могут быть в числе топовых.

Как вы, возможно, заметили, для измерения точности и полноты требуется, чтобы кто-то судил о том, насколько релевантны результаты поиска. В небольших случаях это решаемая задача; но требуемое усилие делает это едва выполнимым для огромных коллекций документов. Возможность измерить эффективность поисковых систем заключается в использовании общедоступных наборов данных для поиска информации, например из серий конференций TREC[1] Национального института стандартов и технологий (NIST), в которых содержится множество ранжированных запросов, чтобы использовать их для проверки точности и полноты.

В этом разделе вы узнали некоторые основы классических моделей поиска информации, таких как векторная модель и вероятностные модели. Теперь мы рассмотрим распространенные проблемы, которые влияют на поисковые системы. В остальной части книги обсудим, как исправить их с помощью глубокого обучения.

1.6. Нерешенные проблемы

Мы более подробно рассмотрели, как работает поисковая система, в частности как она стремится извлекать информацию, соответствующую потребностям конечного пользователя. Но давайте сделаем шаг назад и попытаемся увидеть проблему с точки зрения того, как мы, пользователи, каждый день пользуемся поисковыми системами. Мы рассмотрим некоторые проблемы, которые остаются нерешенными во многих поисковых сценариях, чтобы лучше понять, какие задачи мы можем надеяться решить с помощью глубокого обучения.

---

[1] **http://trec.nist.gov/data.html**.

Заполнение пробела в знаниях, в отличие от поиска информации, – несколько более сложная тема. Давайте снова возьмем в качестве примера поход в библиотеку, потому что вы хотите узнать больше об интересных недавних исследованиях в области искусственного интеллекта. Как только вы встречаетесь с библиотекарем, у вас возникает проблема: как заставить библиотекаря четко понять, что вам нужно и что было бы полезно для вас?

Хотя это звучит просто, полезность информации вряд ли объективна. Она скорее субъективна и основана на контексте и мнении. Вы можете предположить, что у библиотекаря достаточно знаний и опыта, для того чтобы вы получали то, что вам нужно. В реальной жизни вы, вероятно, поговорите с библиотекарем, чтобы представиться и поделиться информацией о вашем прошлом и о том, зачем вам нужно что-либо; что позволило бы библиотекарю использовать такой контекст, чтобы:

- исключить некоторые книги, прежде чем пытаться искать;
- отказаться от некоторых книг, найдя их;
- выполнить явный поиск в одной или нескольких областях, которые имеют более близкое отношение к вашему контексту (например, в научных или отраслевых источниках).

Вы сможете дать отзыв о книгах, переданных вам библиотекарем, позже, хотя иногда вы можете выразить озабоченность, основываясь на прошлом опыте (например, вам не нравятся книги, написанные определенным автором, поэтому вы советуете библиотекарю явно не учитывать их). И контекст, и мнение могут значительно различаться и, следовательно, влияют на актуальность информации с течением времени среди разных людей. Как библиотекарь справляется с этим несоответствием?

Вы как пользователь можете не знать библиотекаря или, по крайней мере, недостаточно хорошо понимать его контекст. Подоплека и мнения библиотекаря важны, потому что они влияют на результаты, которые вы получаете. Поэтому чем лучше вы понимаете библиотекаря, тем быстрее вы получите свою информацию. Так что вам нужно знать своего библиотекаря, чтобы получить нужные результаты!

Что, если библиотекарь даст вам книгу о «методах глубокого обучения» в ответ на ваш первый запрос об «искусственном интеллекте»? Если вы не знаете предмет, вам нужно сделать второй запрос о «введении в глубокое обучение» и есть ли по этой теме подходящая книга в библиотеке. Этот процесс может повторяться несколько раз; главное – это понять, что информация течет постепенно: вы не загружаете вещи в свой мозг, как это делают персонажи в фильме «*Матрица*». Вместо этого, если вы хотите что-то узнать об искусственном интеллекте, вы можете осознать, что сначала вам нужно познакомиться с глубоким обучением, а для того чтобы сделать это, вы обнаружите, что вам нужно прочитать книги о математическом анализе и линейной алгебре и т. д. Другими словами, вы не знаете всего, что вам нужно, когда спрашиваете впервые.

Подводя итог, можно сказать, что процесс получения информации, которую вы ищете у библиотекаря, имеет некоторые недостатки, вызванные следующими ситуациями:

- библиотекарь не знает вас;
- вы не знаете библиотекаря;
- вам может понадобиться несколько циклов, чтобы получить все, что нужно.

Важно идентифицировать эти проблемы, потому что мы хотим использовать глубокие нейронные сети, чтобы они помогли нам создавать более совершенные поисковые системы, которые можно было бы легко использовать, – и мы хотели бы, чтобы глубокое обучение помогло решать эти проблемы. Понимание данных проблем является первым шагом к их решению.

1.7. Открываем черный ящик поисковой системы

Теперь давайте попытаемся понять, сколько из того, что делает поисковая система, видят пользователи. Важной проблемой при создании эффективных поисковых запросов является то, какой язык запросов вы используете. Несколько лет назад вы вводили одно или несколько ключевых слов в поле поиска, чтобы выполнить запрос. Сегодня технология эволюционировала до такой степени, что вы можете вводить запросы на естественном языке. Некоторые поисковые системы индексируют документы на нескольких языках (например, для поиска в сети) и допускают последующие запросы. Если вы ищете одно и то же, но выражаете это с помощью несколько разных запросов в поисковой системе, такой как Google, то увидите поразительно разные результаты.

Давайте проведем небольшой эксперимент, чтобы увидеть, как изменяются результаты поиска, когда один и тот же запрос выражается по-разному. Если бы вы разговаривали с человеком и задавали один и тот же вопрос по-разному, то ожидали бы всегда получать один и тот же ответ. Например, если вы спросите кого-то: «Каковы “последние достижения в искусственном интеллекте”, по вашему мнению?», то получите ответ на основе его мнения. Если вы спросите того же человека: «Каковы, по вашему мнению, “последние достижения в области искусственного интеллекта”?», то, вероятно, получите точно такой же ответ или тот, который семантически эквивалентен.

Сегодня в случае с поисковыми системами часто бывает не так. В табл. 1.4 представлены результаты поиска «последних достижений в области искусственного интеллекта» и некоторых вариантов в Google.

Таблица 1.4. Сравнение похожих запросов

Запрос	Заголовок первого результата
Latest breakthroughs in artificial intelligence	Academic papers for “latest breakthroughs in artificial intelligence” (Google Scholar)
Latest advancements in artificial intelligence	Google advancements artificial intelligence push with 2 top hires
Latest advancements on artificial intelligence	Images related to “latest advancements on artificial intelligence” (Google Images)
Latest breakthroughs in AI	Artificial Intelligence News—ScienceDaily
Più recenti sviluppi di ricerca sull’intelligenza artificiale	Intelligenza Artificiale (Wikipedia)

Хотя первый результат первого запроса неудивителен, изменение терма «breakthroughs» (прорывы) на один из его синонимов («advancements») приводит к другому результату, который, по-видимому, предполагает, что поисковая система по-разному понимает необходимую информацию: вы не смотрели, как Google

улучшает искусственный интеллект! Третий запрос дает удивительный результат: изображения! У нас нет реального объяснения этому. Изменение сочетания «искусственный интеллект» на его аббревиатуру «ИИ» (AI) приводит к другому, но по-прежнему релевантному результату. А когда вы используете перевод исходного запроса на итальянский, то получаете совершенно иной результат по отношению к запросу на английском языке: страницу Википедии об искусственном интеллекте. Это кажется обобщенным, учитывая, например, что Google Scholar индексирует научные статьи на разных языках.

Рейтинги в поисковых системах могут значительно различаться, как и мнения пользователей. Хотя специалист по разработке поисковых систем может оптимизировать ранжирование, чтобы отвечать на ряд заданных запросов, трудно настроить его для, возможно, десятков или сотен подобных запросов. Поэтому в реальной жизни мы не настраиваем ранжирование результатов поиска вручную; это будет почти невозможно и вряд ли приведет к хорошему ранжированию в общем.

Часто поиск выполняется методом проб и ошибок: вы запускаете первоначальный запрос и получаете слишком много результатов; вы выполняете второй запрос и все равно получаете слишком много результатов; а третий запрос может вернуть банальные результаты, которые вам не интересны. Выражение информационной потребности с помощью поискового запроса не является тривиальной задачей. Часто все заканчивается тем, что выполняете кучу запросов только для того, чтобы получить общее представление о том, что, по вашему мнению, может сделать с ними поисковая система. Это все равно, что пытаться заглянуть в черный ящик: вы почти ничего не видите, но пытаетесь делать предположения относительно того, что происходит внутри.

В большинстве случаев у пользователей нет возможности понять, что делает поисковая система. Хуже того, все сильно меняется в зависимости от того, как пользователь выражает свой запрос.

Теперь, когда вы понимаете, как обычно работает поисковая система, и вы узнали о некоторых важных проблемах, которые еще не были решены до конца, пришло время познакомиться с глубоким обучением и выяснить, как оно может помочь решить или хотя бы смягчить такие проблемы. Мы начнем с общего обзора возможностей глубоких нейронных сетей.

1.8. Глубокое обучение спешит на помощь

До сих пор мы исследовали темы поиска информации, которые необходимы, чтобы подготовить вас к путешествию по поиску на базе нейронных сетей. Теперь вы приступите к знакомству с глубоким обучением, которое может помочь при создании более интеллектуальных поисковых систем. Этот раздел познакомит вас с основными понятиями глубокого обучения.

В прошлом ключевая проблема *компьютерного зрения* (область компьютерных наук, которая занимается обработкой и пониманием визуальных данных, таких как изображения или видео) при работе с изображениями заключалась в том, что было почти невозможно получить представление изображения, содержащее информацию о закрытых объектах и визуальных структурах. Как заставить компьютер сказать, представляет ли изображение бегущего льва, холодильник, стаю обезь-

ян и т. д.? Глубокое обучение помогло решить эту проблему с помощью создания особого типа глубокой нейронной сети, который мог бы изучать представления изображений постепенно, по одной абстракции за раз, как показано на рис. 1.9.

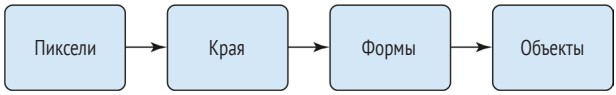


Рис. 1.9 ❖ Постепенное изучение абстракций изображений

Как упоминалось ранее в этой главе, глубокое обучение – это подобласть машинного обучения, которая фокусируется на изучении глубоких представлений текста, изображений или данных в целом путем изучения последовательных абстракций все более значимых представлений. Это делается с помощью глубокой нейронной сети (на рис. 1.10 показана глубокая нейронная сеть с тремя скрытыми слоями). Помните, что нейронная сеть считается *глубокой*, если в ней есть как минимум два скрытых слоя.

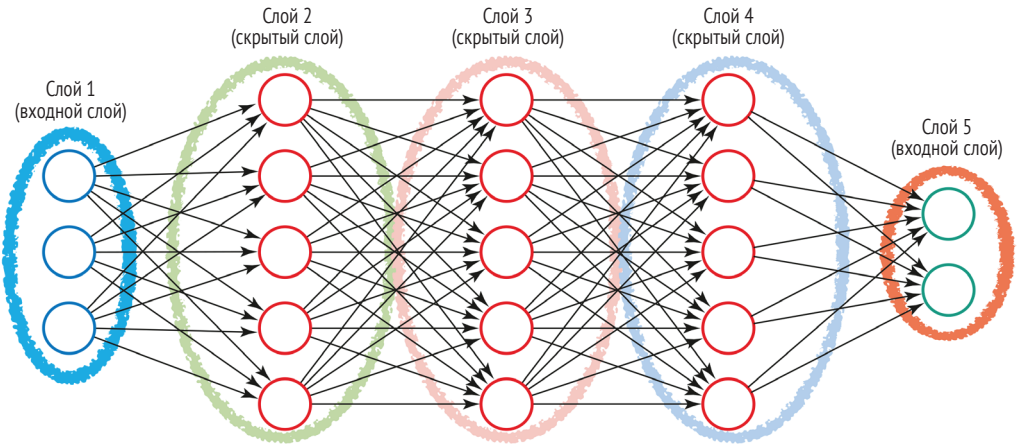


Рис. 1.10 ❖ Глубокая нейронная сеть прямого распространения с тремя скрытыми слоями

На каждом этапе (или слое сети) такие глубокие нейронные сети способны захватывать все более сложные структуры данных. Не случайно компьютерное зрение является одной из областей, которая способствовала разработке и исследованию алгоритмов для изучения представлений при работе с изображениями.

Исследователи обнаружили, что имеет смысл использовать такие глубокие сети, особенно для высококомпозиционных данных. Это означает, что они могут очень помочь, когда вы рассматриваете нечто, что образовано более мелкими частями подобных компонентов. Изображения и текст являются хорошими примерами композиционных данных, потому что их можно постепенно разделить на более мелкие единицы (например, текст → параграфы → предложения → слова). Но (глубокие) нейронные сети не полезны только для изучения представлений; их можно использовать для выполнения множества различных задач в машинном

обучении. Я упомянул, что задача по классификации документов может быть решена с помощью методов машинного обучения.

Несмотря на то что существует множество разных способов создания нейронной сети, нейронные сети обычно состоят из:

- набора нейронов;
- набора связей между всеми или некоторыми нейронами;
- веса (действительное число) для каждой направленной связи между двумя нейронами;
- одной или нескольких функций, которые отображают, как каждый нейрон получает и *распространяет* сигналы по направлению к своим исходящим соединениям;
- при желании набора слоев, которые группируют наборы нейронов, обладающих сходной связностью в нейронной сети.

На рис. 1.10 мы можем идентифицировать 20 нейронов, организованных в набор из 5 слоев. Каждый нейрон в каждом слое связан со всеми нейронами в соседних слоях (предыдущий и последующие слои), за исключением первого и последнего слоев. Традиционно информация начинает течь внутри сети слева направо. Первый слой, который получает входные данные, называется *входным слоем*; и последний слой, называемый *выходным слоем*, выводит результаты нейронной сети. Находящиеся между ними слои называются *скрытыми*.

Представьте, что вы можете применить тот же подход к тексту, чтобы изучить представления документов, которые содержат все более высокие абстракции в документе. Для таких задач существуют методы на базе глубокого обучения, и со временем эти алгоритмы становятся все умнее: их можно использовать для извлечения представлений слов, предложений, абзацев и документов, которые могут охватывать удивительно интересную семантику.

При использовании алгоритма нейронной сети для изучения представлений слов в наборе текстовых документов тесно связанные слова лежат рядом в векторном пространстве. Рассмотрим создание точки на двухмерном графике для каждого слова, содержащегося во фрагменте текста, и посмотрим, как схожие или тесно связанные слова лежат близко друг к другу, как показано на рис. 1.11. Этого можно достичь с помощью алгоритма нейронной сети под названием *word2vec* для изучения таких векторных представлений слов (также называемых *векторами слов*). Обратите внимание, что слова «Информация» и «Поиск» лежат близко друг к другу. Точно так же «word2vec» и «Skip-gram», термы, которые относятся к алгоритмам (неглубокой) нейронной сети, используемым для извлечения векторов слов, находятся рядом друг с другом.

Одной из ключевых идей поиска на основе нейронных сетей является использование таких представлений для повышения эффективности поисковых систем. Было бы неплохо иметь модель поиска, которая опирается на векторы слов и документов (также называющиеся *векторными представлениями*) с этими возможностями, поэтому мы могли бы эффективно рассчитывать и использовать сходства документов и слов, взглянув на *ближайших соседей*. На рис. 1.12 показана глубокая нейронная сеть, используемая для создания представлений слов, содержащихся в проиндексированных документах, которые затем возвращаются в поисковую систему; их можно использовать для настройки порядка результатов поиска.

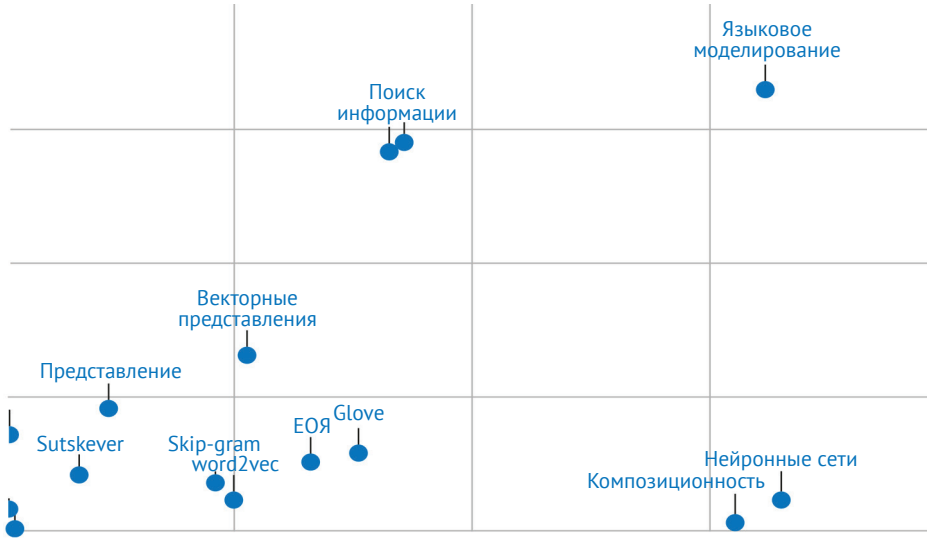


Рис. 1.11 ❖ Векторы слов, полученные из текста исследовательских статей в word2vec

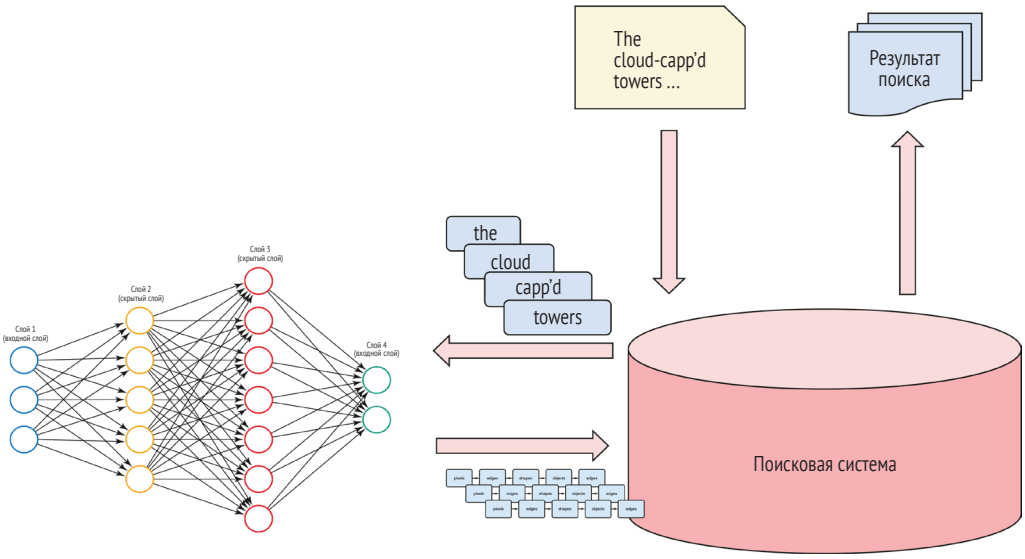


Рис. 1.12 ❖ Приложение для поиска на базе нейронных сетей: использование представлений слов, генерируемых глубокой нейронной сетью, для предоставления более релевантных результатов

В предыдущем разделе анализировалась важность контекста по сравнению со сложностью выражения и понимания информационных потребностей через текстовые запросы. Хорошие семантические представления текста часто строятся с использованием контекста, в котором появляется слово, предложение или

документ, чтобы вывести наиболее подходящее представление. Давайте посмотрим на предыдущий пример, чтобы кратко увидеть, как алгоритмы глубокого обучения могут помочь получить более качественные результаты с релевантностью. Рассмотрим два запроса «последние достижения в области искусственного интеллекта» и «последние достижения в области искусственного интеллекта» из табл. 1.4, предполагая, что мы используем векторную модель. В таких моделях сходство между запросами и документами может сильно варьироваться в зависимости от цепочки анализа текста. Но эта проблема не влияет на векторные представления текста, сгенерированного с помощью современных алгоритмов на базе нейронных сетей. Хотя фраза «искусственный интеллект» и аббревиатура «ИИ» могут находиться далеко друг от друга в векторной модели, они, скорее всего, будут располагаться близко друг к другу, когда построены с использованием представлений слов, генерируемых нейронными сетями. С таким простым изменением мы можем повысить релевантность поисковой системы с помощью более семантически обоснованных представлений слов.

Прежде чем приступить к подробному рассмотрению приложений для поиска на базе нейронных сетей, давайте посмотрим, как поисковые системы и нейронные сети могут работать вместе.

Глубокое обучение и глубокие нейронные сети

Нам нужно провести одно важное различие. Глубокое обучение в основном касается изучения представлений слов, текста, документов и изображений с помощью глубоких нейронных сетей. Глубокие нейронные сети, однако, имеют более широкое распространение: они используются, например, в языковом моделировании, машинном переводе и множестве других задач. В этой книге я буду пояснять, когда мы используем глубокие нейронные сети для изучения представлений и когда мы используем их для других целей. В дополнение к представлениям глубокие нейронные сети могут помочь решить ряд задач, связанных с поиском информации.

1.9. Индекс, пожалуйста, познакомьтесь с нейроном

Искусственная нейронная сеть может научиться прогнозировать результаты на основе обучающего набора с помеченными данными (контролируемое обучение, где каждый ввод снабжен информацией об ожидаемом результате), или она может выполнять обучение без учителя (никакой информации о правильном выводе для каждого ввода не предоставляется), для того чтобы извлекать шаблоны и/или изучать представления. Типичный рабочий процесс поисковой системы включает в себя индексацию и поиск контента; примечательно, что такие задачи могут происходить параллельно. Хотя на данный момент это может показаться технической деталью, то, как вы интегрируете поисковую систему с нейронной сетью, в принципе, важно, поскольку это влияет на эффективность и производительность конструкции поиска. У вас может быть сверхточная система, но если она медлительна, никто не захочет ее использовать! В этой книге вы увидите несколько способов интеграции нейронных сетей и поисковых систем:

- *обучение, затем индексация* – сначала обучите сеть на наборе документов (тексты, изображения), а затем индексируйте те же данные в поисковую

систему и используйте нейронную сеть вместе с поисковой системой во время поиска;

- *индексация, затем обучение* – сначала индексируйте коллекцию документов в поисковой системе; затем обучите нейронную сеть с помощью индексированных данных (в конечном итоге переобучая ее при изменении данных); а потом используйте нейронную сеть вместе с поисковой системой во время поиска;
- *обучение – извлечение – индексация* – сначала обучите сеть на наборе документов и используйте обученную сеть для создания полезных ресурсов, которые будут индексироваться вместе с данными. Поиск происходит как обычно, только с поисковой системой.

Вы увидите, что каждый из этих вариантов в данной книге применяется в правильном контексте. Например, вариант *обучение, затем индексация* будет использоваться в главе 3 для генерации текста, а вариант *индексация, затем обучение* будет использоваться в главе 2 для генерации синонимов из проиндексированных данных. Вариант *обучение – извлечение – индексация* имеет смысл, когда вы используете нейронную сеть для изучения чего-то вроде семантического представления данных, подлежащих индексации; вы будете применять такие представления во время поиска, не прибегая ко взаимодействию с нейронной сетью. Это относится к сценарию, описанному в главе 8 для поиска изображений. В последней главе книги также кратко рассматривается, как справиться с ситуациями, когда поначалу данные доступны не полностью, а поступают в потоковом режиме.

1.10. Обучение нейронной сети

Чтобы использовать мощные возможности обучения нейронной сети, необходимо обучить ее. Обучение сети, подобной той, что показана в предыдущем разделе, происходит с помощью средств контролируемого обучения, предоставляющих входные данные для входного слоя сети, сравнивая (прогнозируемые) результаты сети с известными (целевыми) результатами и позволяя сети учиться на расхождениях между прогнозируемыми и целевыми результатами. Нейронные сети могут легко представлять множество интересных математических функций; это одна из причин, по которой они могут иметь очень высокую верность. Такие математические функции регулируются *весами* соединений и *функциями активации* нейронов. Алгоритм обучения нейронной сети принимает несоответствия между желаемыми и фактическими результатами и корректирует веса каждого слоя, чтобы уменьшить ошибку вывода в будущем. Если вы подадите достаточно данных в сеть, она сможет достичь очень маленького уровня ошибок и, следовательно, хорошо работать. Функции активации влияют на способность нейронной сети выполнять предсказания и скорость обучения; функции активации контролируют, когда и сколько входящего сигнала на нейрон распространяется по всем выходным соединениям.

Наиболее часто используемый алгоритм обучения для нейронных сетей называется методом *обратного распространения ошибки*. При заданных желаемых и фактических результатах алгоритм распространяет *ошибку* каждого нейрона обратно и, следовательно, корректирует свое внутреннее состояние на соединениях

каждого нейрона, по одному слою за раз, от вывода к вводу (назад); *см.* рис. 1.13. Каждый обучающий пример заставляет обратное распространение ошибки «регулировать» состояние каждого нейрона и связи, чтобы уменьшить количество ошибок, создаваемых сетью для этой пары конкретного ввода и желаемого вывода. Это высокоуровневое описание того, как работает алгоритм обратного распространения ошибки; мы подробнее рассмотрим его в последующих главах, когда вы лучше познакомитесь с нейронными сетями.

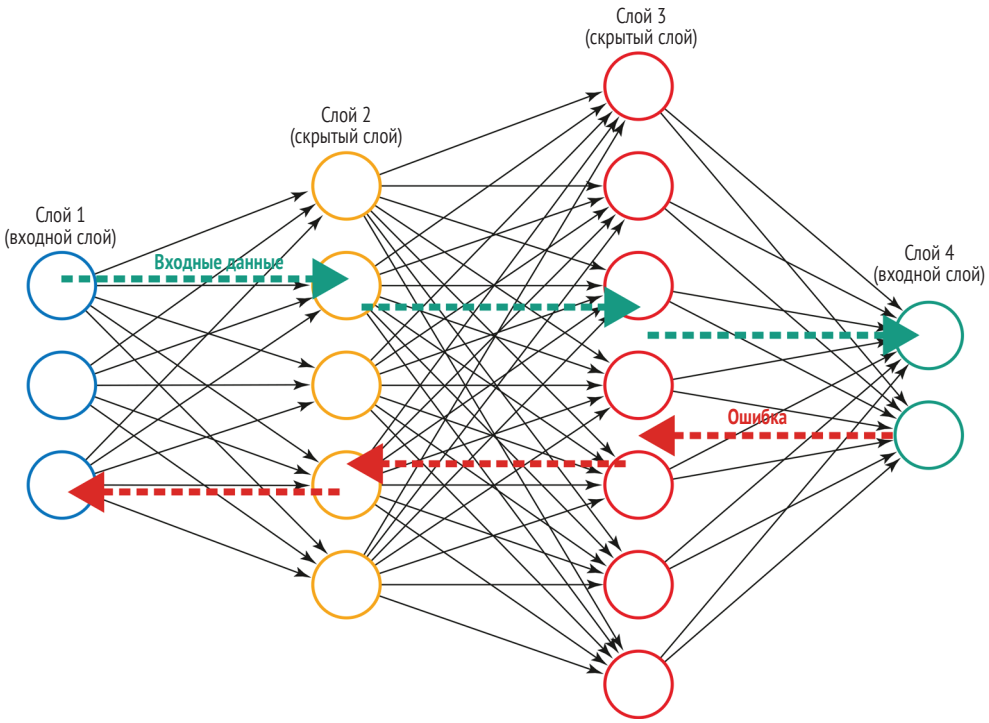


Рис. 1.13 ❖ Шаг вперед (передача входных данных) и шаг назад (обратное распространение ошибки)

Теперь, когда вы понимаете, как обучаются нейронные сети, вам нужно решить, как подключиться к поисковой системе. Поисковые системы могут получать данные для постоянной индексации; поскольку новый контент добавляется, существующий контент обновляется или даже удаляется. Хотя этот процесс относительно легко и быстро поддерживать в поисковой системе, многие алгоритмы машинного обучения создают *статические модели*, которые нельзя быстро адаптировать при изменении данных. Типичный рабочий процесс разработки для задачи машинного обучения включает в себя следующие шаги:

1) выбор и сбор данных для использования в качестве учебного набора;
2) сохранение отдельных частей учебного набора отдельно для оценки и настройки (наборы для тестирования и перекрестной проверки);
3) обучение нескольких моделей машинного обучения в соответствии с алгоритмами (нейронные сети прямого распространения, метод опорных век-

торов и т. д.) и гиперпараметрами (например, количество слоев и количество нейронов в каждом слое для нейронных сетей);
4) оценка и настройка модели для наборов для тестирования и перекрестной проверки;
5) выбор наиболее эффективной модели и использование ее для решения желаемой задачи.

Как видите, этот процесс направлен на создание вычислительной модели, которая будет использоваться для решения определенной задачи или проблемы с использованием статических данных обучения; обновления обучающих наборов (добавленные или измененные входные данные и результаты) таких моделей часто требуют повторения всей последовательности шагов. Это приводит к конфликту с такими системами, как поисковики, которые имеют дело с постоянным потоком новых данных. Например, поисковая система онлайн-газеты будет ежедневно обновляться множеством различных новостей; вы должны принять это во внимание при разработке системы поиска на базе нейронных сетей. Нейронные сети – это модели машинного обучения: вам может потребоваться провести повторное обучение модели или найти решения, чтобы ваша нейронная сеть могла выполнять онлайн-обучение (повторное обучение не требуется)[1].

Рассмотрим, как эволюционировали во времени значения определенных английских слов. Например, слово «cell» сегодня обычно относится к мобильным телефонам или ячейкам с биологической точки зрения; до изобретения мобильных телефонов слово «cell» упоминалось преимущественно по отношению к биологическим клеткам или… тюрьмам! Некоторые концепции тесно связаны со словами только в определенных временных рамках: государственные посты меняются каждые несколько лет, поэтому Барак Обама был президентом Соединенных Штатов в период с 2009 по 2017 год, тогда как термин «президент Соединенных Штатов» применялся по отношению к Джону Фицджеральду Кеннеди между 1961 и 1963 годом. Если подумать о книгах, содержащихся в библиотечном архиве, сколько из них содержат фразу «президент Соединенных Штатов»? Они редко будут относиться к одному и тому же человеку из-за того, что были написаны в разное время.

Я упомянул, что нейронные сети можно использовать для генерации *векторов слов*, которые фиксируют семантику слов, так что слова со сходными значениями будут иметь векторы слов, близкие друг к другу. Как вы думаете, что случится с вектором слов «президент США», если вы обучите модель новостным статьям 1960-х годов и сравните ее с векторами слов, сгенерированными моделью, обученной новостным статьям 2009 года? Будет ли вектор слов «Барак Обама» из последней модели находиться рядом с вектором «президент США» из предыдущей модели? Вероятно, нет, если вы не проинструктируете нейронную сеть, как работать с эволюцией слов во времени[2]. С другой стороны, обычные поисковые

---

[1] *См.*, например: *Andrey Besedin et al.* Evolutive deep models for online learning on data streams with no storage // Workshop on Large-scale Learning from Data Streams in Evolving Environments. 2017 (**http://mng.bz/K14O**) и *Doyen Sahoo et al.* Online Deep Learning: Learning Deep Neural Networks on the Fly (**https://arxiv.org/pdf/1711.03705.pdf**).

[2] *См.*, например: *Zijun Yao et al.* Dynamic Word Embeddings for Evolving Semantic Discovery // International Conference on Web Search and Data Mining. 2018 (**https://arxiv.org/abs/1703.00607**).

системы могут легко работать с такими запросами, как «президент США», и возвращать результаты поиска, содержащие такую фразу, независимо от того, когда они были включены в инвертированный индекс.

1.11. Перспективы поиска на базе нейронных сетей

Поиск на базе нейронных сетей – это интеграция глубокого обучения и глубоких нейронных сетей в поиск на разных этапах. Способность глубокого обучения улавливать глубокую семантику позволяет нам получать релевантные модели и функции ранжирования, которые хорошо адаптируются к базовым данным. Глубокие нейронные сети могут изучить представления изображений, которые дают удивительно хорошие результаты при поиске изображений. Простые меры сходства, такие как косинусное расстояние, могут применяться к представлениям данных, генерируемым с помощью глубокого обучения, для захвата семантически похожих слов, предложений, абзацев и т. д.; здесь есть ряд применений, например на этапе анализа текста и при рекомендации похожих документов. В то же время глубокие нейронные сети могут делать больше, чем «просто» изучать представления; они могут учиться генерировать или переводить текст, а также научиться оптимизировать работу поисковой системы.

Как вы увидите в процессе чтения книги, поисковая система состоит из разных компонентов, которые действуют сообща. Наиболее очевидные части – это ввод данных в поисковую систему и их поиск. Нейронные сети могут использоваться во время индексации для улучшения данных непосредственно перед их поступлением в инвертированный индекс, или их можно использовать для расширения либо указания области действия поискового запроса, чтобы предоставить большее количество результатов или более точные результаты. Но нейронные сети также можно использовать, чтобы давать пользователям умные подсказки, помогая им набирать запросы или переводить свои поисковые запросы «под капотом» и заставить поисковую систему работать с несколькими языками.

Все это звучит потрясающе, но нельзя бросить нейронные сети в поисковую систему и ожидать, что она станет автомагически совершенной. Каждое решение должно приниматься в контексте; а у нейронных сетей есть некоторые ограничения, в том числе стоимость обучения, обновление моделей и многое другое. Но применение поиска на базе нейронных сетей к поисковой системе – это отличный способ сделать ее лучше для пользователей. Это также увлекательное путешествие для специалистов по разработке поисковых систем, которые изучают красоту нейронных сетей.

Резюме

- Поиск – сложная проблема: общие подходы к поиску информации сопряжены с ограничениями и недостатками, и пользователям, и специалистам по разработке поисковых систем может быть трудно заставить все работать так, как ожидалось.
- Анализ текста является важной задачей в поиске как на этапе индексации, так и на этапе поиска, поскольку он подготавливает данные для хранения в ин-

вертированных индексах и оказывает значительное влияние на эффективность поисковой системы.

- Релевантность – это фундаментальная мера того, насколько хорошо поисковая система реагирует на информационные потребности пользователей. Некоторые модели поиска информации могут дать стандартизированную оценку важности результатов для запросов, но универсального средства не существует. Контекст и мнения могут значительно различаться от пользователя к пользователю, поэтому специалисты по разработке поисковых систем должны быть постоянно сосредоточены на измерении релевантности.
- Глубокое обучение – это область машинного обучения, которая использует глубокие нейронные сети для изучения (глубоких) представлений контента (текста, такого как слова, предложения и абзацы, но также и изображений), который может охватывать семантически релевантные меры сходства.
- Поиск на основе нейронных сетей представляет собой мост между поиском и глубокими нейронными сетями с целью использования глубокого обучения, чтобы помочь улучшить различные задачи, связанные с поиском.

Глава 2

Генерируем синонимы

О чем идет речь в этой главе:

- почему и как синонимы используются в поиске;
- краткое введение в Apache Lucene;
- основы нейронных сетей прямого распространения;
- использование алгоритма word2vec;
- генерация синонимов с использованием word2vec.

Первая глава дала вам общий обзор тех возможностей, которые открываются, когда глубокое обучение применяется к поисковым задачам. Эти возможности включают в себя использование глубоких нейронных сетей для поиска изображений с помощью текстового запроса на основе его содержимого, создание текстовых запросов на естественном языке и т. д. Вы также узнали об основах поисковых систем и о том, как они проводят поиск по запросам и предоставляют релевантные результаты. Теперь вы готовы приступить к применению глубоких нейронных сетей для решения поисковых задач.

В этой главе мы начнем с неглубокой нейронной сети, которая поможет вам определить, когда два слова схожи по семантике. Эта, казалось бы, легкая задача крайне важна для того, чтобы дать поисковой системе способность понимать язык.

В поиске информации распространенным методом улучшения количества релевантных результатов запроса является использование *синонимов*. Синонимы позволяют расширить количество возможных способов выражения запроса или фрагмента индексированного документа. Например, вы можете выразить фразу «Мне нравится жить в Риме» как «Я наслаждаюсь жизнью в Вечном городе»: термы «жить» и «наслаждаться», а также «Рим» и «Вечный город» семантически похожи. Таким образом, информация, передаваемая обоими предложениями, в основном одинакова. Синонимы могут помочь решить проблему, которую мы обсуждали в первой главе, когда библиотекарь и студент ищут книгу, понимая друг друга, потому что использование синонимов позволяет людям выражать одну и ту же концепцию по-разному – и тем не менее получать те же результаты!

В этой главе мы начнем работать с синонимами, используя word2vec, один из наиболее распространенных алгоритмов на базе нейронных сетей для изучения представлений слов. Изучение word2vec позволит вам ближе познакомиться с тем, как нейронные сети работают на практике. Для этого вы сначала увидите, как работают нейронные сети *прямого распространения*.

Нейронные сети прямого распространения, один из самых основных типов нейронных сетей, являются основными строительными блоками глубокого обучения. Далее вы узнаете о двух архитектурах этого типа сетей: Skip-gram и CBOW (*Continuous Bag of Words*, «непрерывный мешок со словами», англ. *bag* – мультимножество). Они позволяют узнать, когда два слова имеют схожее значение, и, следовательно, хорошо подходят для понимания того, являются ли два данных слова синонимами. Вы узнаете, как применять их для улучшения полноты поисковой системы, помогая ей избежать пропуска релевантных результатов поиска.

Наконец, вы выполните оценку того, насколько таким образом можно улучшить поисковую систему и какие компромиссы нужно будет учитывать для производственных систем. Понимание этих затрат и выгод важно при принятии решения касательно того, когда и где применять эти методы в реальных сценариях.

2.1. Расширение синонимов. Введение

В предыдущей главе вы видели, как важно иметь хорошие алгоритмы для анализа текста: эти алгоритмы определяют способ разбиения текста на более мелкие фрагменты или термы. Когда дело доходит до выполнения запроса, термы, сгенерированные во время индексации, должны совпадать с термами, извлеченными из запроса. Такое соответствие позволяет найти документ и затем появиться в результатах поиска.

Одним из наиболее частых препятствий, мешающих сопоставлению, является тот факт, что люди могут выражать концепцию несколькими различными способами. Например, фразу «прогулка в горах» также можно выразить словами «хайкинг» или «трекинг». Если автор индексируемого текста использует слово «hike» (пешая экскурсия), но пользователь, выполняющий поиск, вводит слов «treck», пользователь не найдет документ. Вот почему нужно, чтобы поисковая система знала о синонимах.

Я объясню, как можно использовать технику под названием *расширение синонимов*, чтобы выражать одну и ту же потребность в информации несколькими способами. Хотя расширение синонимов является популярным методом, у него есть ряд ограничений: в частности, необходимо поддерживать словарь синонимов, который, вероятно, будет изменяться с течением времени и который часто не совсем подходит для индексируемых данных (такие словари нередко получают из общедоступных данных). Вы увидите, как можно использовать такие алгоритмы, как word2vec, чтобы изучить представления слов, помогающих генерировать синонимы точно на основе данных, которые должны быть проиндексированы.

К концу главы у вас будет поисковая система, которая может использовать нейронную сеть для генерации синонимов, которые затем можно будет использовать для *оформления* текста, подлежащего индексации. Чтобы показать, как это работает, мы будем использовать пример, в котором пользователь отправляет запрос «music is my aircraft» (музыка – это мой самолет) через пользовательский интерфейс поисковой системы. (Скоро я объясню, почему пользователь использует этот конкретный запрос в данный момент.) На рис. 2.1 показано, что у вас в итоге получится.

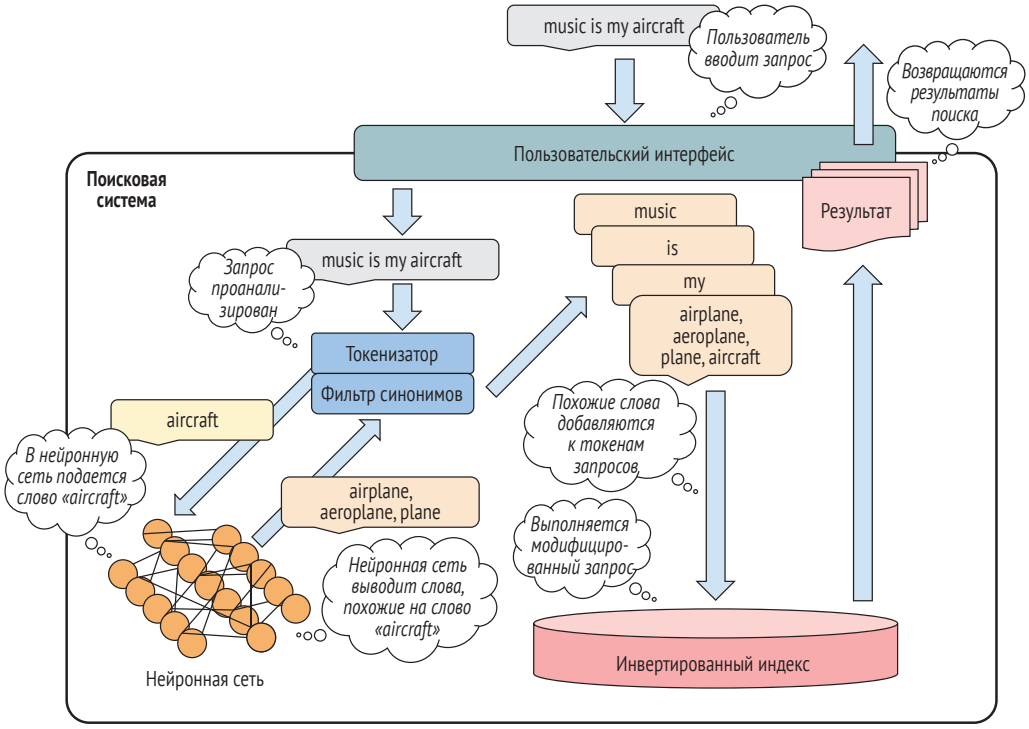


Рис. 2.1 ❖ Расширение синонимов во время поиска с помощью нейронной сети

Вот основные шаги, которые показаны на рисунке. В поисковой системе запрос сначала обрабатывается конвейером анализа текста. *Фильтр синонимов* в конвейере использует нейронную сеть для генерации синонимов. В этом примере нейронная сеть возвращает слова «airplane», «aeroplane» и «plane» в качестве синонимов слова «aircraft» (самолет). Сгенерированные синонимы затем используются вместе с токенами из пользовательского запроса для поиска совпадений в инвертированном индексе. Наконец, результаты поиска собраны. Получилось большое изображение. Не волнуйтесь: теперь мы рассмотрим каждый шаг подробно.

2.1.1. Почему синонимы?

Синонимы – это слова, которые отличаются по написанию и произношению, но имеют одинаковое или очень близкое значение. Например, слова «aircraft» и «airplane» являются синонимами слова «plane». При поиске информации обычно используют синонимы для оформления текста, чтобы увеличить вероятность совпадения соответствующего запроса. Да, здесь мы говорим о вероятности, потому что не можем предвидеть все возможные способы выражения потребности в информации. Этот метод не является универсальным средством, которое позволит вам понять все пользовательские запросы, но уменьшит количество запросов, которые дают слишком мало или ноль результатов.

Давайте посмотрим на пример, где синонимы могут быть полезны. Такое, вероятно, было и у вас: вы смутно помните короткий фрагмент песни или помните что-то из смыслового содержания текста, но не помните точные слова песни, о которой идет речь. Предположим, вам понравилась песня, в припеве которой были строки: «Music is my ...*что-то там*». Что это было? Машина? Лодка? Самолет? Теперь представьте, что у вас есть система, которая собирает тексты песен, и вы хотите, чтобы пользователи могли осуществлять поиск по ней. Если в поисковой системе включено расширение синонимов, поиск по фразе «music is my plane» выдаст фразу, которую вы ищете: «music is my aeroplane»! В этом случае использование синонимов позволяет найти релевантный документ (песню «Aeroplane» группы Red Hot Chili Peppers), используя фрагмент и неверное слово. Без расширения синонимов было бы невозможно получить этот релевантный ответ с такими запросами, как «music is my boat», «music is my plane» и «music is my car».

Это считается улучшением полноты. *Полнота*, которую мы кратко упомянули в первой главе, – это число от 0 до 1, равное количеству найденных и релевантных документов, деленное на количество релевантных документов. Если ни один из найденных документов не релевантен, полнота равна 0. А если все найденные документы релевантны, полнота равна 1.

Общая идея расширения синонимов заключается в том, что когда поисковая система получает поток термов, она может обогатить их, добавив их синонимы, если они существуют, в одной и той же позиции. В примере с «Aeroplane» синонимы термов запроса были расширены: они были потихоньку декорированы словом «aeroplane» в той же позиции, что и «plane» в потоке текста; *см.* рис. 2.2.

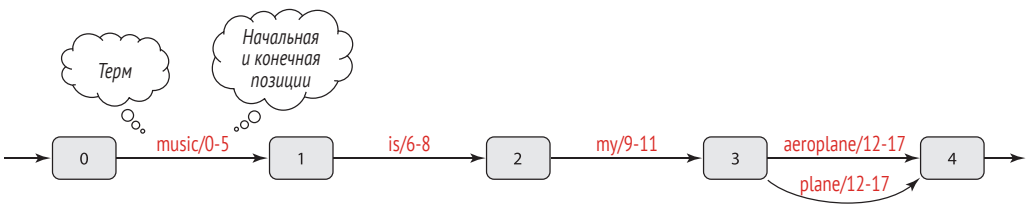


Рис. 2.2 ❖ График расширения синонимов

Можно применить ту же технику во время индексации текста песни «Aeroplane». Расширение синонимов во время индексации немного замедлит индексирование (из-за вызовов word2vec), и индекс неизбежно будет больше (потому что он будет содержать больше термов для хранения). С другой стороны, процесс поиска будет идти быстрее, потому что вызов word2vec не произойдет во время поиска. Решение относительно того, выполнять расширение синонимов во время индексации или во время поиска, может оказать заметное влияние на производительность системы по мере роста ее размера и нагрузки.

Теперь, когда вы поняли, почему синонимы полезны в контексте поиска, давайте посмотрим, как реализовать расширение синонимов, сначала с помощью общепринятых методов, а затем с помощью word2vec. Это поможет вам оценить преимущества использования данного алгоритма.

2.1.2. Сопоставление синонимов на базе словаря

Давайте вначале рассмотрим, как реализовать поисковую систему, используя расширение синонимов, активированное во время индексации. Самый простой и наиболее распространенный подход для реализации синонимов основан на том, что поисковой системе предоставляется словарь, содержащий сопоставления всех слов и связанных с ними синонимов. Такой словарь может выглядеть как таблица, где каждый ключ – это слово, а соответствующие значения – его синонимы:

```
aeroplane -> plane, airplane, aircraft
boat -> ship, vessel
car -> automobile
...
```

Представьте, что вы вводите текст песни «Aeroplane» в поисковую систему для индексации и используете расширение синонимов с помощью предыдущего словаря. Давайте возьмем припев песни – «music is my aeroplane» – и посмотрим, как с этим справляется расширение синонимов. У вас есть простой анализ текста конвейер, состоящий из токенизатора, который создает токен каждый раз, когда находит пробел, в результате чего создается токен для каждого слова в предложении. Таким образом, конвейер анализа текста во время индексации создаст эти токены. Затем вы будете использовать *фильтр токенов* для расширения синонимов: для каждого полученного токена он будет просматривать словарь синонимов и определять, совпадает ли какое-либо из ключевых слов («aeroplane», «boat», «car») с текстом токена. Постинг-лист фрагмента «music is my aeroplane» (отсортированный по возрастанию в алфавитном порядке) будет выглядеть так, как показано в табл. 2.1.

Таблица 2.1. Пост-листинг фрагмента «music is my aeroplane»

Терм	**Документ (позиция)**
aeroplane	1(12,17)
aircraft	1(12,17)
airplane	1(12,17)
is	1(6,8)
music	1(0,5)
my	1(9,11)
plane	1(12,17)

Этот постинг-лист также записывает информацию о позиции вхождения терма в конкретном документе. Эта информация помогает вам визуализировать тот факт, что термы «plane», «airplane» и «airplane», которые не были включены в исходный фрагмент текста, были добавлены в индекс с той же позицией, что и информация, прикрепленная к исходному терму («aeroplane»).

Вы можете записать *позиции* термов в инвертированном индексе, чтобы восстановить порядок, в котором терм появляется в тексте документа. Если вы посмотрите на таблицу инвертированных индексов и выберете термы с более низкими позициями в порядке возрастания, то получите «music is my aeroplane/aircraft/airplane/plane». Синонимы можно легко менять, поэтому в индексе можно пред-

ставить четыре разных варианта: «music is my aeroplane», «music is my aircraft», «music is my airplane» и «music is my plane». Важно подчеркнуть, что хотя вы нашли четыре различные формы, в которых предложение может быть проиндексировано и найдено, если какие-то из них совпадают, поисковая система вернет только один документ: все они ссылаются на документ 1 в постинг-листе.

Теперь, когда вы понимаете, как можно проиндексировать синонимы в поисковой системе, можно попробовать создать свою первую поисковую систему на базе Apache Lucene, которая индексирует тексты песен, настраивая надлежащий анализ текста с помощью расширения синонимов во время индексации.

> **ПРИМЕЧАНИЕ** Далее я буду использовать понятия *Lucene* или *Apache Lucene*, которые являются взаимозаменяемыми, но правильное название торговой марки – Apache Lucene.

Быстрое знакомство с Apache Lucene

Я кратко расскажу о Lucene, перед тем как приступить к расширению синонимов. Это позволит вам больше сосредоточиться на концепциях, а не на API Lucene и деталях реализации.

Где найти Apache Lucene

Последнюю версию Apache Lucene можно скачать по адресу **https://lucene.apache.org/core/downloads.html?**. Вы можете скачать бинарный пакет (.tgz или .zip) либо исходную версию. Бинарный дистрибутив рекомендуется, если вы просто хотите использовать Lucene в собственном проекте; пакет .tgz / .zip содержит JAR-файлы компонентов Lucene. Lucene состоит из различных артефактов: единственный обязательный – `lucene-core`. Остальные – необязательные части, которые можно использовать при необходимости. Базовые сведения, необходимые для того, чтобы начать работу с Lucene, можно найти в официальной документации, доступной по адресу **https://lucene.apache.org/core/7_4_0/index.html**. Пакет с исходным кодом подходит для разработчиков, которые хотят просмотреть код или улучшить его. (Патчи для улучшений, новые функции, исправления ошибок, документация и т. д. – всегда желанные гости на странице **https://issues.apache.org/jira/browse/LUCENE**.) Если вы используете такие инструменты сборки, как Maven, Ant или Gradle, то можете включить Lucene в свой проект, потому что все компоненты выпущены в общедоступных репозиториях, таких как Maven Central (**http://mng.bz/vN1x**).

Apache Lucene – это библиотека поиска с открытым исходным кодом, написанная на Java, лицензированная Apache 2. В Lucene основные сущности, подлежащие индексации и поиску, представлены документами (Documents). Документ в зависимости от вашего варианта использования может обозначать что угодно: страницу, книгу, абзац, изображение и т. д. Что бы это ни было, это вы и получите в результатах поиска. Документ состоит из нескольких полей, которые можно использовать для захвата его различных частей. Например, если ваш документ – это веб-страница, можно подумать о наличии отдельного поля для заголовка страницы, ее содержимого, размера, времени создания и т. д. Основные причины существования полей:

- вы можете настроить конвейеры анализа текста для каждого поля;
- вы можете настроить параметры индексирования, например сохранять в постинг-листе позиции термов или значение исходного текста, к которому относится каждый терм.

К поисковой системе Lucene можно обратиться через каталог (Directory): список файлов, где хранятся инвертированные индексы (и другие структуры данных, используемые, например, для записи позиций). Увидеть каталог для чтения инвертированного индекса можно, открыв `IndexReader`:

```
Path path = Paths.get("/home/lucene/luceneidx");

Directory directory = FSDirectory.open(path);

IndexReader reader = DirectoryReader.open(directory);
```

Путь назначения, в котором инвертированные индексы хранятся в файловой системе

Получает доступное только для чтения представление поисковой системы через IndexReader

Открывает каталог на пути назначения

Можно использовать `IndexReader` для получения полезной статистики для индекса, такой как количество документов, проиндексированных в настоящее время, или наличие каких-либо документов, которые были удалены. Вы также можете получить статистику для поля или конкретного терма. А если вам известен *идентификатор* документа, который вы хотите получить, то можно получить документы непосредственно из `IndexReader`:

```
int identifier = 123;
Document document = reader.document(identifier);
```

`IndexReader` необходим для поиска, потому что позволяет читать индекс. Следовательно, вам нужен `IndexReader` для создания `IndexSearcher`. `IndexSearcher` – это точка входа для выполнения поиска и сбора результатов; запросы, которые будут выполняться через `IndexSearcher`, будут выполняться в индексных данных, предоставляемых `IndexReader`.

Не вдаваясь в программирование запросов, вы можете выполнить запрос, введенный пользователем, с помощью `QueryParser`. При поиске необходимо указать анализ текста. В Lucene задача анализа текста выполняется с помощью API `Analyzer`. `Analyzer` может состоять из токенизатора и (необязательно) компонентов `TokenFilter`; или вы можете использовать готовые реализации, как в этом примере:

```
QueryParser parser = new QueryParser("title",
   new WhitespaceAnalyzer());
Query query = parser.parse("+Deep +search");
```

Создает парсер запросов для поля заголовка с помощью WhitespaceAnalyzer

Выполняет синтаксический анализ введенного пользователем запроса и получает запрос Lucene

В этом случае вы указываете анализатору запросов разбивать токены, когда он находит пробелы, и выполняете запросы к полю с именем `title`. Предположим, пользователь вводит запрос «+ Deep + search». Вы передаете его в `QueryParser` и получаете объект `Query`. Теперь можно выполнить запрос:

```
IndexSearcher searcher = new IndexSearcher(reader);
TopDocs hits = searcher.search(query, 10);
```

Выполняет запрос к IndexSearcher, возвращая первые 10 документов

```
for (int i = 0; i < hits.scoreDocs.length; i++) {
  ScoreDoc scoreDoc = hits.scoreDocs[i];
  Document doc = reader.document(scoreDoc.doc);
  System.out.println(doc.get("title") + " : "
      + scoreDoc.score);
}
```

Перебирает результаты

Извлекает ScoreDoc, который содержит идентификатор возвращаемого документа и его оценку (заданную базовой моделью поиска)

Получает документ, в котором вы можете проверить поля, используя идентификатор документа

Выводит значение поля заголовка возвращаемого документа

Если вы выполните его, то не получите никаких результатов, потому что вы еще ничего не проиндексировали! Давайте исправим это и рассмотрим, как индексировать документы с помощью Lucene. Во-первых, нужно решить, какие поля поместить в свои документы и как должны выглядеть их конвейеры анализа текста (во время индексации). В этом примере мы будем использовать книги. Предположим, что вам нужно удалить какие-то бесполезные слова из содержания книг, используя при этом более простой конвейер анализа текста для заголовка, который ничего не удаляет.

Листинг 2.1 ❖ Создание анализаторов для каждого поля

```
Map<String, Analyzer> perFieldAnalyzers = new HashMap<>();

CharArraySet stopWords = new CharArraySet(Arrays
    .asList("a", "an", "the"), true);

perFieldAnalyzers.put("pages", new StopAnalyzer(
   stopWords));

perFieldAnalyzers.put("title", new WhitespaceAnalyzer());

Analyzer analyzer = new PerFieldAnalyzerWrapper(
    new EnglishAnalyzer(), perFieldAnalyzers);
```

Устанавливает карту, где ключами являются имена полей, а значения – это анализаторы, которые будут использоваться для полей

Создает список стоп-слов токенов, которые нужно удалить из содержимого книг при индексации

Использует StopAnalyzer с заданными стоп-словами для поля «pages»

Использует WhitespaceAnalyzer для поля «title»

Создает анализатор для каждого поля, для которого также требуется анализатор по умолчанию (в данном случае EnglishAnalyzer) для любого другого поля, которое можно добавить в документ

Инвертированные индексы для поисковой системы на базе Lucene записываются на диск в каталоге с помощью `IndexWriter`, который сохраняет документы в соответствии с `Index-WriterConfig`. Эта конфигурация содержит множество опций, но самая важная для вас – анализатор времени индексации. Как только `IndexWriter` будет готов, вы можете создавать документы и добавлять поля.

Листинг 2.2 ❖ Добавление документов в индекс Lucene

```
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter writer = new IndexWriter(directory,
   config);

Document dl4s = new Document();
dl4s.add(new TextField("title", "DL for search",
```

Создает конфигурацию для индексации

Создает IndexWriter для записи документов в каталог на основе IndexWriterConfig

Создает экземпляры документа

```
   Field.Store.YES));
dl4s.add(new TextField("page", "Living in the information age ...",
   Field.Store.YES));

Document rs = new Document();
rs.add(new TextField("title", "Relevant search", Field.Store.YES));
rs.add(new TextField("page", "Getting a search engine to behave ...",
   Field.Store.YES));

writer.addDocument(dl4s);
writer.addDocument(rs);
```

Добавляет поля, каждое из которых имеет имя, значение и параметр для сохранения значения с термами

Добавляет документы в поисковую систему

После добавления нескольких документов в `IndexWriter` вы можете сохранить их в файловой системе, выполнив команду `commit`. Пока вы этого не сделаете, новые `IndexReaders` не увидят добавленные документы:

```
writer.commit();
writer.close();
```

Фиксирует изменения

Закрывает IndexWriter (освобождает ресурсы)

Запустите код еще раз, и вот что вы получите:

```
Deep learning for search : 0.040937614
```

Код находит совпадение по запросу «+ Deep + search» и выводит его заголовок и оценку.

Теперь, когда вы познакомились с Lucene, давайте вернемся к теме расширения синонимов.

Настройка индекса Lucene с использованием расширения синонимов

Сначала вы определите алгоритмы, которые будут применяться для анализа текста во время индексации и поиска. Затем вы добавите тексты песен в инвертированный индекс. Во многих случаях рекомендуется использовать один и тот же токенизатор как при индексировании, так и во время поиска, поэтому текст разбивается по одному и тому же алгоритму. Благодаря этому запросам легче соответствовать фрагментам документов. Мы начнем с простого и настроим:

- анализатор времени поиска, который использует токенизатор, разбивающий токены, когда встречает символ пробела (также известный как *токенизатор пробелов*);
- анализатор времени индексации, использующий токенизатор пробелов и фильтр синонимов.

Причина этого состоит в том, что вам не нужно расширение синонимов и во время запроса, и во время индексации. Для совпадения двух синонимов достаточно выполнить расширение один раз.

Предполагая, что у вас есть два синонима «aeroplane» и «plane», в приведенном ниже листинге мы создадим цепочку анализа текста, которая может взять терм из исходного токена (например, «plane») и сгенерировать еще один терм для его синонима (например, «aeroplane»).

Будут созданы два терма: исходный и новый.

Листинг 2.3 ❖ Настройка расширения синонимов

```
SynonymMap.Builder builder = new SynonymMap.Builder();
builder.add(new CharsRef("aeroplane"), new CharsRef("plane"), true);
final SynonymMap map = builder.build();
```

Программным путем определяет синонимы

```
Analyzer indexTimeAnalyzer = new Analyzer() {
  @Override
  protected TokenStreamComponents createComponents(
        String fieldName) {
    Tokenizer tokenizer = new WhitespaceTokenizer();
    SynonymGraphFilter synFilter = new
        SynonymGraphFilter(tokenizer, map, true);
    return new TokenStreamComponents(tokenizer, synFilter);
  }
};

Analyzer searchTimeAnalyzer = new WhitespaceAnalyzer();
```

Создает пользовательский анализатор для индексации

Создает фильтр синонимов, который получает термы от токенизатора пробелов и расширяет синонимы в соответствии со словом карты, игнорируя регистр

Анализатор пробелов для времени поиска

В этом упрощенном примере мы создали словарь синонимов только с одной записью. Обычно у вас будет больше записей или вы будете читать их из внешнего файла, поэтому вам не нужно будет писать код для каждого синонима.

Вы почти готовы добавить текст песни в указатель, используя `index-TimeAnalyzer`. Прежде чем сделать это, давайте посмотрим, как структурированы тексты песен. У каждой песни есть автор, название, год выхода, текст песни и т. д. Как я уже говорил ранее, важно изучить данные, подлежащие индексации, посмотреть, какие данные у вас есть, и, возможно, придумать аргументированные цепочки анализа текста, которые, как вы ожидаете, будут хорошо работать с этими данными. Вот пример:

```
author: Red Hot Chili Peppers
title: Aeroplane
year: 1995
album: One Hot Minute
text: I like pleasure spiked with pain and music is my aeroplane ...
```

Можете ли вы отследить такую структуру в поисковой системе? Будет ли это полезно?

В большинстве случаев удобно сохранять легковесную структуру документа, потому что каждая его часть передает разную семантику и, следовательно, разные требования в отношении поиска. Например, год всегда будет числовым значением; нет смысла использовать для него токенизатор пробелов, потому что вряд ли какие-либо пробелы появятся в этом поле. Для всех остальных полей, вероятно, можно использовать анализатор, который вы определили ранее для индексации. Собрав все это вместе, вы получите несколько инвертированных индексов (по одному для каждого атрибута), предназначенных для индексации разных частей документа, и все это в одной поисковой системе; *см.* рис. 2.3.

С помощью Lucene можно определить поле для каждого из атрибутов в примере (`автор`, `название`, `год`, `альбом`, `текст`). Вы указываете, что вам нужен отдельный анализатор для поля `год`, которое не касается значения; для всех остальных значений будет использоваться ранее определенный `indexTime-Analyzer` с активированным расширением синонимов.

Листинг 2.4 ❖ Отдельные цепочки анализа для индексации и поиска

```
Directory directory = FSDirectory.open(Paths.get(
    "/path/to/index"));
```

Открывает каталог для индексации

```
Map<String, Analyzer> perFieldAnalyzers =
    new HashMap<>();
```
Создает карту, ключами которой являются имена полей и значения в соответствующей цепочке анализа, которые будут использоваться

```
perFieldAnalyzers.put("year",
    new KeywordAnalyzer());
```
Настраивает другой анализатор (ключевое слово; не касается значение) для поля «год»

```
Analyzer analyzer = new PerFieldAnalyzerWrapper(
    indexTimeAnalyzer, perFieldAnalyzers);
```
Создает оборачивающий анализатор, который может работать с анализаторами для каждого поля

```
IndexWriterConfig config = new IndexWriterConfig(
    analyzer);
```

```
IndexWriter writer = new IndexWriter(
    directory, config);
```
Встраивает все вышеперечисленное в объект конфигурации

Создает IndexWriter, который будет использоваться для индексации

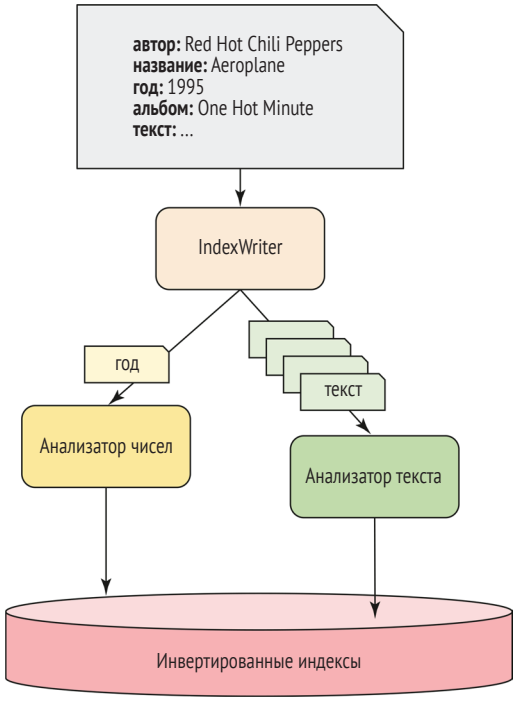


Рис. 2.3 ❖ Разбиение текста на фрагменты в зависимости от типа данных

Этот механизм позволяет индексированию быть гибким относительно, как контент анализируется перед записью в инвертированные индексы; экспериментировать с разными анализаторами, когда вы имеете дело с различными частями документов, и неоднократно менять их, прежде чем найти наиболее подходящую комбинацию для корпуса данных – обычное дело. Даже тогда, в реальной ситуации, вероятно, что такие конфигурации будут нуждаться в корректировке с течением времени. Например, вы можете индексировать песни только на английском языке, а затем в какой-то момент начать добавлять песни на китайском. В этом случае вам придется настроить анализаторы для работы с обоими языками (например, нельзя ожидать, что токенизатор пробельных символов будет хорошо

работать с китайским, японским и корейским языками, где слова часто не разделяются пробелом).

Давайте поместим наш первый документ в индекс Lucene.

Листинг 2.5 ❖ Индексирование документов

```
Document aeroplaneDoc = new Document();
aeroplaneDoc.add(new Field("title", "Aeroplane", type));
aeroplaneDoc.add(new Field("author", "Red Hot Chili Peppers", type));
aeroplaneDoc.add(new Field("year", "1995", type));
aeroplaneDoc.add(new Field("album", "One Hot Minute", type));
aeroplaneDoc.add(new Field("text",
    "I like pleasure spiked with pain and music is my aeroplane ...", type));

writer.addDocument(aeroplaneDoc);
writer.commit();
```

Создает документ для песни «Aeroplane»

Добавляет все поля из текста песни

Добавляет документ

Сохраняет обновленный инвертированный индекс в файловой системе, делая изменения долговечными (и доступными для поиска)

Вы создаете документ, состоящий из нескольких полей, по одному на атрибут песни, а затем добавляете его в `writer`.

Чтобы выполнить поиск, вы (снова) открываете каталог и получаете представление для индекса `IndexReader`, в котором можно осуществлять поиск через `IndexSearcher`. Чтобы убедиться, что расширение синонимов работает должным образом, введите в качестве запроса слово «plane»; вы ожидаете, что будет найдена песня «Aeroplane».

Листинг 2.6 ❖ Поиск по слову «plane»

```
IndexReader reader = DirectoryReader.open(directory);

IndexSearcher searcher = new IndexSearcher(reader);

QueryParser parser = new QueryParser("text",
    searchTimeAnalyzer);

Query query = parser.parse("plane");

TopDocs hits = searcher.search(query, 10);
for (int i = 0; i < hits.scoreDocs.length; i++) {
    ScoreDoc scoreDoc = hits.scoreDocs[i];
    Document doc = searcher.doc(scoreDoc.doc);
    System.out.println(doc.get("title") + " by "
        + doc.get("author"));
}
```

Инстанциирует искателя

Открывает представление для индекса

Создает анализатор запросов, который использует анализатор времени поиска с введенным пользователем запросом для создания поисковых термов

Преобразует введенный пользователем запрос (в виде строки) в правильный объект запроса Lucene, используя QueryParser

Ищет и получает первые 10 результатов

Перебирает результаты

Получает результат поиска

Выводит название и автора возвращаемой песни

Как и ожидалось, результат выглядит следующим образом:

```
Aeroplane by Red Hot Chili Peppers
```

Мы быстро ознакомились с тем, как настроить анализ текста для индексации и поиска, а также как индексировать документы и получать их. Вы также узнали,

как добавить возможность расширения синонимов. Но должно быть ясно, что этот код нельзя использовать в реальной ситуации:

- нельзя написать код для каждого синонима, который хотите добавить;
- вам нужен словарь синонимов, который можно подключить и управлять им отдельно, чтобы избежать необходимости изменять код поиска каждый раз, когда вам нужно его обновить;
- вы должны управлять эволюцией языков – новые слова (и синонимы) добавляются постоянно.

Первый шаг к решению этих проблем – записать синонимы в файл и позволить фильтру синонимов читать их оттуда. Это можно сделать, поместив синонимы в одну строку, разделив их запятыми. Вы создадите анализатор более компактным способом, используя шаблон Строитель (**https://ru.wikipedia.org/wiki/Строитель_(шаблон_проектирования)**).

Листинг 2.7 ❖ Подача синонимов из файла

```
Map<String, String> sffargs = new HashMap<>();
sffargs.put("synonyms", "synonyms.txt");
sffargs.put("ignoreCase", "true");

CustomAnalyzer.Builder builder = CustomAnalyzer.builder()
    .withTokenizer(WhitespaceTokenizerFactory.class)
    .addTokenFilter(SynonymGraphFilterFactory.class, sffargs)
return builder.build();
```

Определяет файл, содержащий синонимы

Определяет анализатор

Позволяет анализатору использовать токенизатор пробелов

Позволяет анализатору использовать фильтр синонимов

Установите синонимы в файле синонимов:

```
plane,aeroplane,aircraft,airplane
boat,vessel,ship
...
```

Таким образом, код остается неизменным независимо от каких-либо изменений в файле синонимов; вы можете обновлять файл столько, сколько вам нужно. Хотя это гораздо лучше, чем писать код для синонимов, вам не нужно писать файл синонимов вручную, если только вы не знаете, что у вас будет всего несколько фиксированных синонимов. К счастью, в наши дни существует много данных, которые можно использовать бесплатно или по очень низкой цене. Хороший, большой ресурс для обработки естественного языка в целом – это проект WordNet (**https://wordnet.princeton.edu**), лексическая база данных для английского языка от Принстонского университета. Вы можете воспользоваться обширным словарем синонимов WordNet, который постоянно обновляется, и включить его в свой конвейер анализа индексации, загрузив его в виде файла (например, synonyms-wn.txt) и указав, что вы хотите использовать формат WordNet.

Листинг 2.8 ❖ Использование синонимов из WordNet

```
Map<String, String> sffargs = new HashMap<>();
sffargs.put("synonyms", "synonyms-wn.txt");
sffargs.put("format", "wordnet");
CustomAnalyzer.Builder builder = CustomAnalyzer.builder()
    .withTokenizer(WhitespaceTokenizerFactory.class)
    .addTokenFilter(SynonymGraphFilterFactory.class, sffargs)
return builder.build();
```

Устанавливает файл синонимов, используя словарь WordNet

Определяет формат WordNet для файла синонимов

После подключения словаря WordNet у вас появляется очень большой, высококачественный источник расширения синонимов, который должен прекрасно подойти для английского языка. Но есть еще несколько проблем. Во-первых, не для всех языков существует ресурс типа WordNet. Во-вторых, даже если вы придерживаетесь английского языка, расширение синонима для слова основано на его *денотации*, как это определено правилами грамматики и словарей английского языка; его *коннотация*, как это определено контекстом, в котором эти слова появляются, не принимается во внимание.

Я описываю разницу между тем, что лингвисты определяют как синоним, основываясь на строгих словарных определениях (денотация), и тем, как люди обычно используют язык и слова в реальной жизни (коннотация). В неформальных контекстах, таких как социальные сети, чаты и встречи с друзьями в реальной жизни, люди могут использовать два слова, как если бы они были синонимами, даже если по правилам грамматики они не являются синонимами. Чтобы решить эту проблему, word2vec включит и предоставит более продвинутый уровень поиска, чем просто расширение синонимов на основе строгого синтаксиса языка. Вы увидите, что использование word2vec позволяет создавать расширения синонимов, не зависящие от языка; он учится у данных, слова которых похожи, не особо заботясь об используемом языке и о том, является ли слово формальным или неформальным. Это полезная функция word2vec: слова с похожим контекстом считаются схожими именно из-за их контекста. Там нет грамматики или синтаксиса. Для каждого слова word2vec просматривает окружающие слова, предполагая, что семантически похожие слова будут появляться в схожих контекстах.

2.2. Важность контекста

Основная проблема описанного подхода заключается в том, что сопоставления синонимов являются статическими и не привязаны к индексированным данным. Например, в случае с WordNet синонимы строго подчиняются семантике грамматики английского языка, но не учитывают сленг или неформальный контекст, где слова часто используются как синонимы, даже если они не являются синонимами согласно строгим правилам грамматики. Еще пример – сокращения, используемые в сеансах чата и электронных письмах. Например, нередко в электронной почте встречаются такие сокращения, как ICYMI (in case you missed it – *если вы не заметили*) и AKA (also known as – *также известен как*). *ICYMI* и «если вы не заметили» нельзя назвать синонимами, и вы, вероятно, не найдете их в словаре, но они означают одно и то же.

Один из подходов к преодолению этих ограничений заключается в том, чтобы иметь способ генерировать синонимы из данных, которые нужно принять. Основная концепция заключается в том, что должна быть возможность извлечь *ближайших соседей* слова, взглянув на его контекст, что означает анализ шаблонов окружающих слов, встречающихся вместе с самим этим словом. В этом случае ближайший сосед слова должен быть его синонимом, даже если он не является строго синонимом с точки зрения грамматики.

Эта идея о том, что слова, которые используются и встречаются в одном и том же контексте, как правило, имеют схожие значения, называется дистрибутивной гипотезой (*см.* **https://aclweb.org/aclwiki/Distributional_Hypothesis**) и является осно-

вой многих алгоритмов глубокого обучения для текстовых представлений. Интересно то, что эта идея игнорирует язык, сленг, стиль и грамматику: каждый бит информации о слове выводится из контекстов слова, которые появляются в тексте. Возьмем, к примеру, то, как часто используются слова, обозначающие города (Рим, Кейптаун, Окленд и т. д.). Давайте посмотрим на несколько предложений:

- Мне нравится жить в Риме, потому что …
- Тем, кто любит серфинг, следует поехать в Кейптаун, потому что …
- Я хотел бы посетить Окленд, чтобы увидеть …
- Движение в Риме сумасшедшее …

Часто названия городов используются рядом с предлогом «в» или на небольшом расстоянии от таких глаголов, как «жить», «посетить» и т. д. Это основная интуиция, лежащая в основе того факта, что контекст предоставляет множество информации о каждом слове.

Учитывая это, вы хотите узнать представления слов для слов в данных, которые нужно проиндексировать, чтобы иметь возможность генерировать синонимы из данных, вместо того чтобы создавать или загружать словарь синонимов вручную. В примере с библиотекой в главе 1 я упомянул, что лучше понять, что находится в библиотеке; в этом случае библиотекарь может более эффективно помочь вам. Студент, который приходит в библиотеку, может попросить у библиотекаря, скажем, «книги об искусственном интеллекте». Предположим также, что в библиотеке есть только одна книга на эту тему, и она называется «*Принципы ИИ*». Если библиотекарь (или студент) выполнял бы поиск по названиям книг, он пропустил бы эту книгу, если бы не знал, что ИИ – это аббревиатура (и, учитывая предыдущие предположения, синоним) сочетания «искусственный интеллект». Помощник, знающий об этих синонимах, будет полезен в подобной ситуации.

Давайте представим два гипотетических типа такого помощника: Джон, специалист по английскому языку, который много лет изучал грамматику и синтаксис английского языка; и Робби, еще один студент, который еженедельно сотрудничает с библиотекарем и имеет возможность читать большинство книг. Джон не смог сказать вам, что ИИ – это искусственный интеллект, потому что его опыт не дает ему этой информации. С другой стороны, у Робби гораздо менее формальное знание английского языка, но он эксперт по книгам, которые есть в библиотеке; он может с легкостью сказать вам, что ИИ – это искусственный интеллект, потому что он прочитал книгу «*Принципы ИИ*» и знает, что это принципы искусственного интеллекта. В этом сценарии Джон ведет себя как словарь WordNet, а Робби – это алгоритм word2vec.

Хотя Джон доказал, что знает язык, Робби может быть более полезным в этой конкретной ситуации.

В главе 1 я упомянул, что нейронные сети хорошо справляются с изучением представлений (в данном случае представлений слов), которые чувствительны к контексту. Это вид возможностей, которые вы будете использовать при работе с word2vec. Говоря кратко, вы будете использовать нейронную сеть word2vec, чтобы изучать представление слов, которые могут сообщить вам наиболее похожее (или ближайшее соседнее) слово для «plane»: «aeroplane». Прежде чем перейти к этому, давайте рассмотрим одну из самых простых форм нейронных сетей: сети прямого распространения.

Нейронные сети прямого распространения являются основой для большинства более сложных архитектур нейронных сетей.

2.3. Нейронные сети прямого распространения

Нейронные сети являются ключевым инструментом для нейронного поиска, и многие архитектуры нейронных сетей исходят от сетей прямого распространения. *Нейронная сеть прямого распространения* – это нейронная сеть, в которой информация передается от входного слоя к скрытым слоям, если таковые имеются, и, в конце, к выходному слою; здесь нет петель, потому что связи между нейронами не формируют цикл. Рассматривайте ее как волшебный черный ящик с входными данными и результатами. Магия в основном происходит внутри сети благодаря тому, как нейроны связаны друг с другом и как они реагируют на свои входные данные. Например, если вы ищете дом для покупки в конкретной стране, то можете использовать «волшебный ящик», чтобы спрогнозировать справедливую цену, которую вы могли бы надеяться заплатить за конкретный дом. Как видно по рис. 2.4, волшебный ящик научился делать прогнозы, используя функции ввода, такие как размер дома, местоположение и рейтинг, предоставленные продавцом.

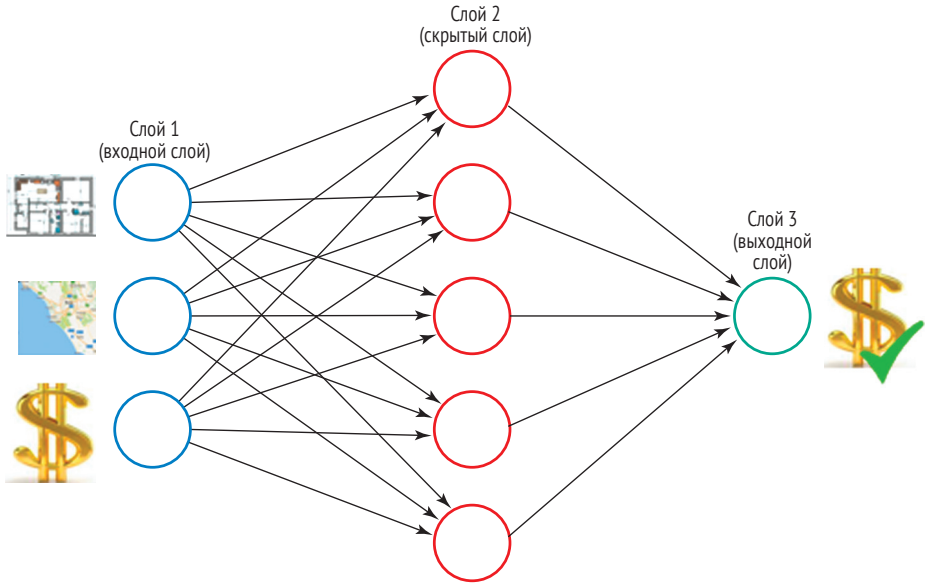


Рис. 2.4 ❖ Прогнозирование цены
с помощью нейронной сети прямого распространения с тремя входами,
пятью скрытыми юнитами и одним выходным юнитом

Нейронная сеть прямого распространения состоит из:

- *входного слоя* – он отвечает за сбор входных данных, предоставленных пользователем. Эти входные данные обычно представлены в виде действительных чисел. В примере с прогнозированием цены на дом у вас есть: размер дома, его местоположение и сумма денег, которую требует продавец. Вы будете кодировать эти входные данные как три действительных числа, поэтому ввод, который вы передадите в сеть, будет трехмерным вектором: [размер, местоположение, цена];

- *одного или нескольких скрытых слоев (необязательно)* – представляет собой более загадочную часть сети. Рассматривайте его как часть сети, которая позволяет ей быть успешной в изучении и прогнозировании. В этом примере в скрытом слое есть пять юнитов, каждый из которых связан с юнитами во входном слое, а также с юнитами в выходном слое. Связность в сети играет фундаментальную роль в динамике сетевой активности. В большинстве случаев все юниты в слое (x) полностью связаны (прямо) с юнитами в следующем слое ($x + 1$);
- *выходной слой* – отвечает за предоставление окончательного результата сети. В примере с ценами на жилье он предоставит действительное число, представляющее, как сеть оценивает правильную стоимость.

ПРИМЕЧАНИЕ Обычно неплохо масштабировать входные данные, чтобы они находились более или менее в том же диапазоне значений, например от –1 до 1. В этом примере размер дома в квадратных метрах составляет от 10 до 200, а диапазон цен составляет порядка десятков тысяч. Предварительная обработка входных данных, чтобы они находились в одинаковых диапазонах значений, позволяет сети учиться быстрее.

Как это работает: веса и функции активации

Как вы уже видели, нейронная сеть прямого распространения получает входные данные и выдает результаты. Фундаментальные строительные блоки этих сетей называются *нейронами* (хотя нейрон мозга гораздо сложнее). Каждый нейрон в нейронной сети прямого распространения:

- принадлежит слою;
- сглаживает каждый ввод по входящему весу;
- распространяет свой вывод в соответствии с функцией активации.

В нейронной сети прямого распространения на рис. 2.5 второй слой состоит только из одного нейрона. Этот нейрон получает ввод от трех нейронов в первом слое и распространяет вывод только на один нейрон в третьем слое. Он имеет ассоциированную функцию активации, а его входящие ссылки с предыдущим уровнем имеют ассоциированные веса (часто это действительные числа от –1 до 1).

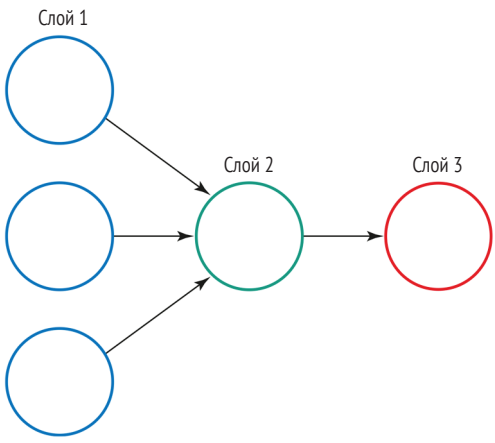


Рис. 2.5 ❖ Распространение сигналов через сеть

Предположим, что все входящие веса нейрона в слое 2 установлены на 0,3 и что он получает от первого слоя входные данные 0,4, 0,5 и 0,6. Каждый вес умножается на его ввод, а результаты суммируются: 0,3 × 0,4 + 0,3 × 0,5 + 0,3 × 0,6 = 0,45. К этому промежуточному результату применяется функция активации, которая затем распространяется на исходящие звенья нейрона. Распространенными функциями активации являются гиперболический тангенс (`tanh`), `sigmoid` и блок линейной ректификации (`ReLU`).

В текущем примере давайте используем функцию `tanh`. У вас будет `tanh (0,45) = 0,4218990053`, поэтому нейрон третьего слоя получит это число в качестве ввода по своей единственной входящей ссылке. Выходной нейрон будет выполнять те же шаги, что и нейрон из слоя 2, используя собственные веса. По данной причине эти сети называются сетями *прямого распространения*: каждый нейрон преобразует и распространяет свои входные данные для подачи нейронов в следующем слое.

Коротко об обратном распространении ошибки

В первой главе я упомянул, что нейронные сети и глубокое обучение относятся к области машинного обучения. Я также затронул основной алгоритм, используемый для обучения нейронных сетей: обратное распространение ошибки. В этом разделе мы рассмотрим его подробнее.

Основной момент при обсуждении развития глубокого обучения связан с тем, насколько хорошо и как быстро нейронные сети могут обучаться. Хотя искусственные нейронные сети – это старая компьютерная парадигма (примерно 1950 г.), в последнее время они стали популярны (примерно 2011 г.), поскольку производительность современных компьютеров улучшилась до уровня, позволившего нейронным сетям эффективно обучаться в разумные сроки.

В предыдущем разделе вы видели, как сеть распространяет информацию из входного слоя в выходной в стиле прямого распространения. С другой стороны, после прямой передачи обратное распространение позволяет сигналу течь обратно из выходного слоя к входному.

Значения активаций нейронов в выходном слое, генерируемые прямой передачей на входе, сравниваются со значениями в желаемом выводе. Это сравнение выполняется *функцией стоимости*, которая вычисляет убыток или стоимость и представляет собой меру того, насколько сеть ошибочна в данном конкретном случае. Такая ошибка отсылается в обратном направлении через входящие соединения выходных нейронов в соответствующие блоки в скрытом слое. На рис. 2.6 видно, что нейрон в выходном слое отправляет свою часть ошибки обратно подключенным блокам в скрытом слое.

Как только блок получает ошибку, он обновляет свои веса в соответствии с *алгоритмом обновления*; алгоритм, который обычно используется, – это *стохастический градиентный спуск*. Это обратное обновление весов происходит до тех пор, пока веса на соединениях входного слоя не будут скорректированы (обратите внимание, что обновления выполняются только для блоков выходных и скрытых слоев, так как блоки ввода не имеют веса), а затем обновление останавливается. Таким образом, запуск обратного распространения обновляет все веса, связанные с существующими соединениями. Логическое обоснование этого алгоритма состоит в том, что каждый вес отвечает за часть ошибки, и поэтому обратное распространение пытается отрегулировать такие веса, чтобы уменьшить ошибку для этой конкретной пары ввода/вывода.

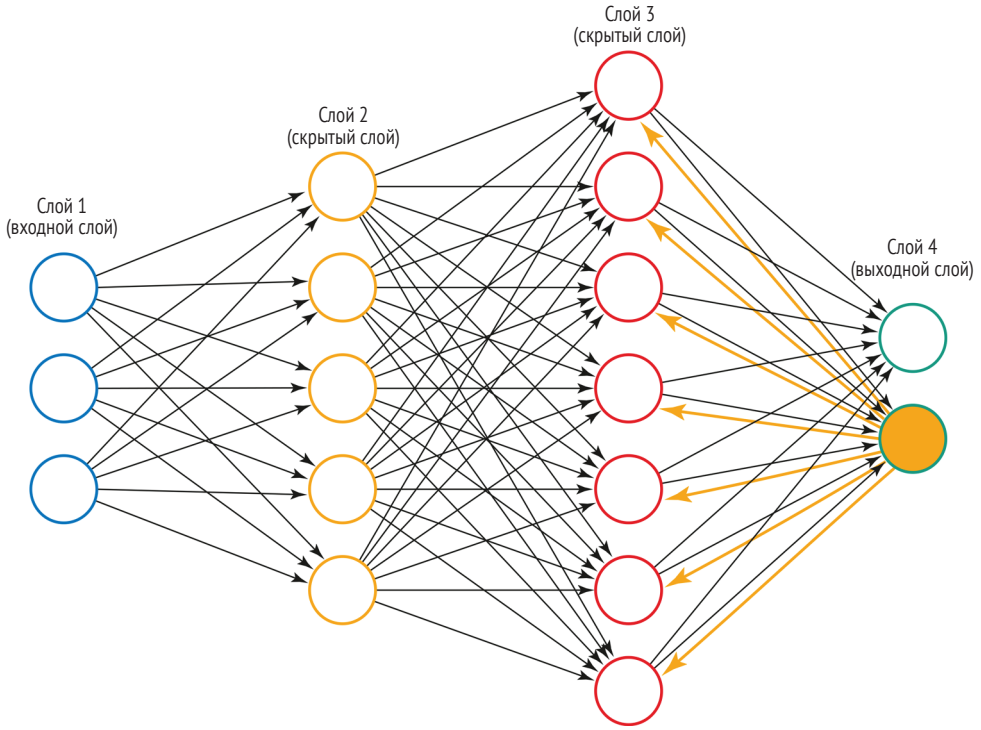


Рис. 2.6 ❖ Обратное распространение сигнала с выходного слоя в скрытый слой

Алгоритм градиентного спуска (или любой другой алгоритм обновления для настройки веса) решает, *как* веса изменяются по отношению к части ошибки, вносимой каждым весом. В этой концепции используется много математики, но вы можете рассматривать ее так, как будто функция стоимости определяет форму, подобную той, что изображена на рис. 2.7, где высота холма определяет величину ошибки. Очень низкая точка соответствует комбинации весов нейронной сети с очень низкой ошибкой:

- *низкая* – точка с наименьшей возможной ошибкой, имеющая оптимальные значения для весов нейронной сети;
- *высокая* – точка с высокой ошибкой; градиентный спуск пытается выполнить спуск к точкам с меньшей ошибкой.

Координаты точки задаются значением весов в нейронной сети, поэтому градиентный спуск пытается найти значение весов (точку) с очень низкой погрешностью (очень низкой высотой) в форме.

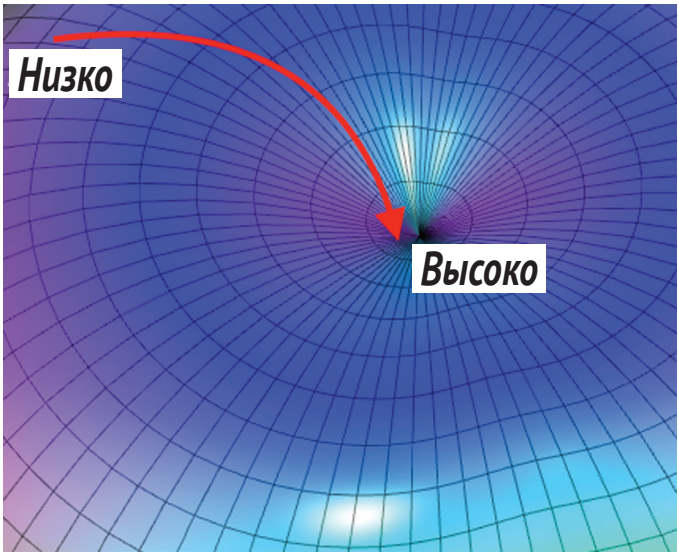


Рис. 2.7 ❖ Геометрическая интерпретация обратного распространения ошибки с градиентным спуском

2.4. Использование word2vec

Теперь, когда вы понимаете, что такое общая сеть прямого распространения, мы можем сосредоточиться на более конкретном алгоритме нейронной сети, основанном на нейронных сетях прямого распространения: word2vec. Хотя его основы довольно просты для понимания, интересно увидеть хорошие результаты (с точки зрения захвата семантики слов в тексте), которых вы можете достичь. Но что он делает, и какую пользу это несет в случае использования расширения синонимов?

Word2vec берет фрагмент текста и выводит серию векторов, по одному на каждое слово в тексте. Когда выходные векторы word2vec построены на двухмерном графе, векторы, слова которых очень похожи с точки зрения семантики, находятся очень близко друг к другу. Можно использовать меры расстояния, такие как косинусное расстояние, чтобы найти наиболее похожие слова по отношению к данному слову. Таким образом, вы можете использовать этот метод, чтобы найти синонимы слова. Говоря кратко, в этом разделе вы настроите модель word2vec, дадите ей текст песни, которую хотите проиндексировать, получите выходные векторы для каждого слова и используете их для поиска синонимов.

В первой главе мы обсуждали использование векторов в контексте поиска, когда говорили о векторной модели и TF-IDF. В некотором смысле word2vec также генерирует векторную модель, векторы которой (по одному на каждое слово) взвешиваются нейронной сетью в процессе обучения. Векторы слов, генерируемые такими алгоритмами, как word2vec, часто называют *векторными представлениями слов*, потому что они отображают статические, дискретные, многомерные представления слов (такие как TFIDF или прямое унитарное кодирование) в другое (непрерывное) векторное пространство с меньшим количеством задействованных измерений.

Вернемся к примеру с песней «Aeroplane». Если вы передадите ее текст в word-2vec, то получите вектор для каждого слова:

```
0.7976110753441061, -1.300175666666296, i
-1.1589942649711316, 0.2550385962680938, like
-1.9136814615251492, 0.0, pleasure
-0.178102361461314, -5.778459658617458, spiked
0.11344064895365787, 0.0, with
0.3778008406249243, -0.11222894354254397, pain
-2.0494382050792344, 0.5871714329463343, and
-1.3652666102221962, -0.4866885862322685, music
-12.878251690899361, 0.7094618209959707, is
0.8220355668636578, -1.2088098678855501, my
-0.37314503461270637, 0.4801501371764839, aeroplane
...
```

Вы можете увидеть их в координатном плане, показанном на рис. 2.8.

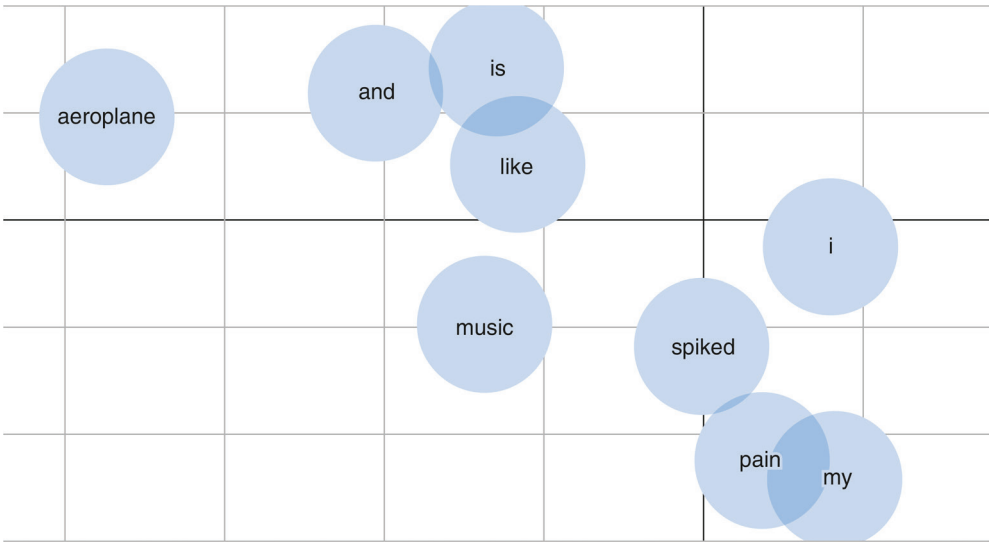


Рис. 2.8 ❖ Векторы слов песни «Aeroplane»

В выходных данных примера были использованы два измерения, чтобы эти векторы было легче отображать на графике. Но на практике обычно используют 100 или более измерений и алгоритм уменьшения размерности, такой как метод главных компонент или t-SNE, для получения двух- или трехмерных векторов, которые легче построить. (Использование множества измерений позволяет собирать больше информации по мере роста объема данных.) На данном этапе мы не будем подробно обсуждать эту настройку, а вернемся к ней позже, когда вы узнаете больше о нейронных сетях.

Использование косинусного сходства для измерения расстояния между каждым из сгенерированных векторов дает несколько интересных результатов:

```
music -> song, view
looking -> view, better
in -> the, like
sitting -> turning, could
```

Как видно, извлечение двух ближайших векторов для нескольких случайных векторов дает хорошие результаты и не очень:

- слова «Music» и «song» очень близки семантически; даже можно сказать, что они синонимы. Но то же самое нельзя сказать о «view»;
- «Looking» и «view» взаимосвязаны, но «better» не имеет ничего общего с «looking»;
- «In», «the» и «like» не близки друг к другу;
- «Sitting» и «turning» – это -ing'овые формы глаголов, но у них слабо связанная семантика. «Coul» – это тоже глагол, но он не имеет ничего общего с «sitting».

В чем проблема? Разве word2vec не подходит для этой задачи?

Тут есть два фактора:

- число измерений (два) сгенерированных векторов слов, вероятно, слишком мало;
- предоставление модели word2vec текста одной песни, вероятно, не обеспечивает достаточного контекста для каждого слова, чтобы получить точное представление.

Модели нужно бóльшее количество примеров контекстов, в которых встречаются слова «better» и «view».

Предположим, что вы снова создаете модель word2vec, на этот раз используя 100 измерений и больший набор текстов песен, взятых из набора данных Billboard Hot 100 (**https://www.kaylinpavlik.com/50-years-of-pop-music/**):

```
music -> song, sing
view -> visions, gaze
sitting -> hanging, lying
in -> with, into
looking -> lookin, lustin
```

Результаты намного лучше и более уместны: почти все из них можно использовать как синонимы в контексте поиска. Представьте себе использование такого метода либо во время запроса, либо во время индексации. Больше никаких словарей для обновления; поисковая система может научиться генерировать синонимы из данных, которые она обрабатывает.

Прямо сейчас у вас может возникнуть пара вопросов: как работает word2vec? и как интегрировать это на практике в поисковик? В статье «Эффективная оценка представлений слов в векторном пространстве»[1] описываются две разные модели нейронной сети для изучения таких представлений слов: *непрерывный мешок слов* (CBOW) и *непрерывный skip-gram*. Мы обсудим их и то, как их реализовать, через минуту. Word2vec выполняет обучение представлений слов без присмотра. Упомянутые модели CBOW и skip-gram просто должны быть снабжены достаточно

[1] *Томас Миколов и др.*, 2013 (**https://arxiv.org/pdf/1301.3781.pdf**).

большим текстом, который правильно закодирован. Основная идея word2vec заключается в том, что нейронной сети предоставляется фрагмент текста, который разбивается на фрагменты определенного размера (также известные как *окна*). Каждый фрагмент подается в сеть в виде пары, состоящей из *целевого слова* и *контекста*. На рис. 2.9 целевое слово – это «aeroplane», а контекст состоит из слов «music», «is» и «my». Скрытый слой сети содержит набор весов (в данном случае 11 из них – количество нейронов в скрытом слое) для каждого слова. Эти векторы будут использоваться как представления слов, когда обучение закончится.

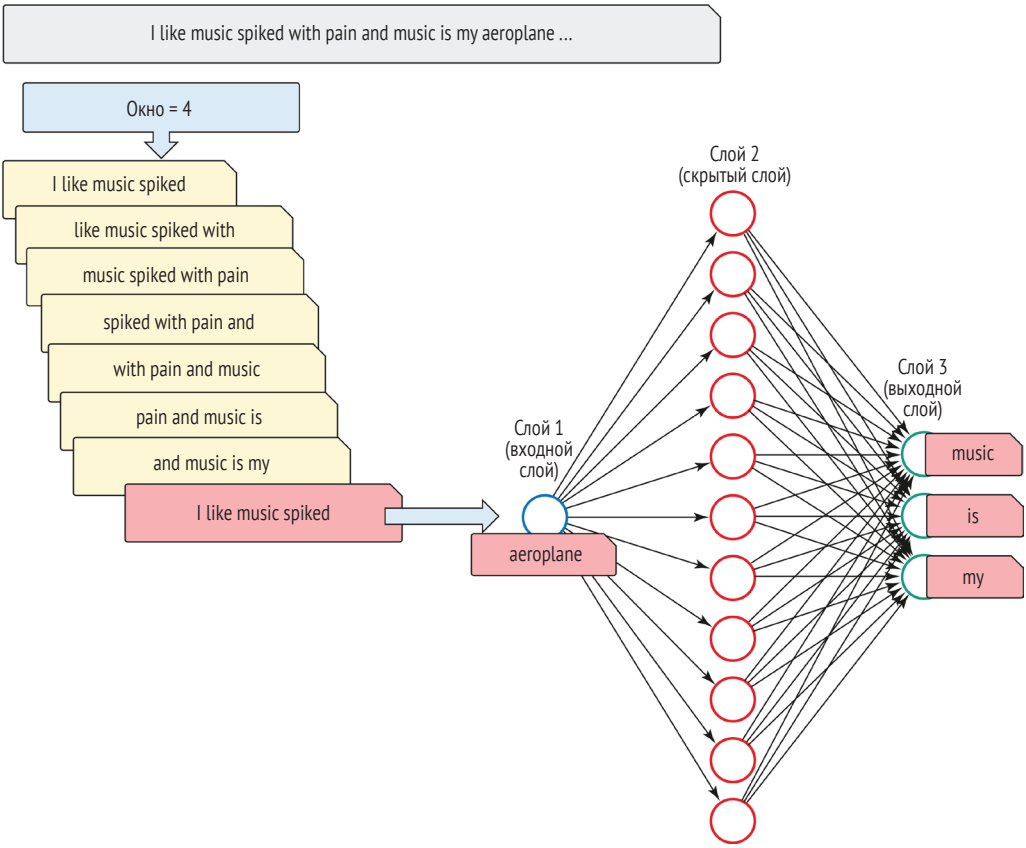


Рис. 2.9 ❖ Передача фрагментов текста (модель skip-gram)

Важным примечанием к word2vec является то, что результаты работы нейронной сети не особо важны. Вместо этого вы извлекаете внутреннее состояние скрытого слоя в конце фазы обучения, что дает ровно одно векторное представление для каждого слова.

Во время обучения часть каждого фрагмента используется в качестве целевого слова, а остальная часть используется в качестве контекста. В модели CBOW целевое слово применяется в качестве вывода сети, а оставшиеся слова текстового фрагмента (контекста) используются в качестве входных данных. В модели непрерывного skip-gram наоборот: целевое слово используется в качестве ввода,

а контекстные слова – в качестве выходных данных (как в примере). На практике обе модели работают хорошо, но skip-gram обычно предпочтительнее, потому что она работает немного лучше с редко используемыми словами.

Например, если взять текст «she keeps moet et chandon in her pretty cabinet let them eat cake she says» из песни «Killer Queen» (группы Queen) и окно 5, модель word2vec на основе CBOW получит образец для каждого фрагмента из пяти слов. Например, для фрагмента | she | keeps | moet | et | chandon | ввод будет состоять из слов | she | keeps | et | chandon |, а вывод – из слова moet.

Как видно по рис. 2.10, нейронная сеть состоит из входного, скрытого и выходного слоев. Этот вид нейронной сети с одним скрытым слоем называется *неглубоким*. Нейронные сети с более чем одним скрытым слоем называются *глубокими*.

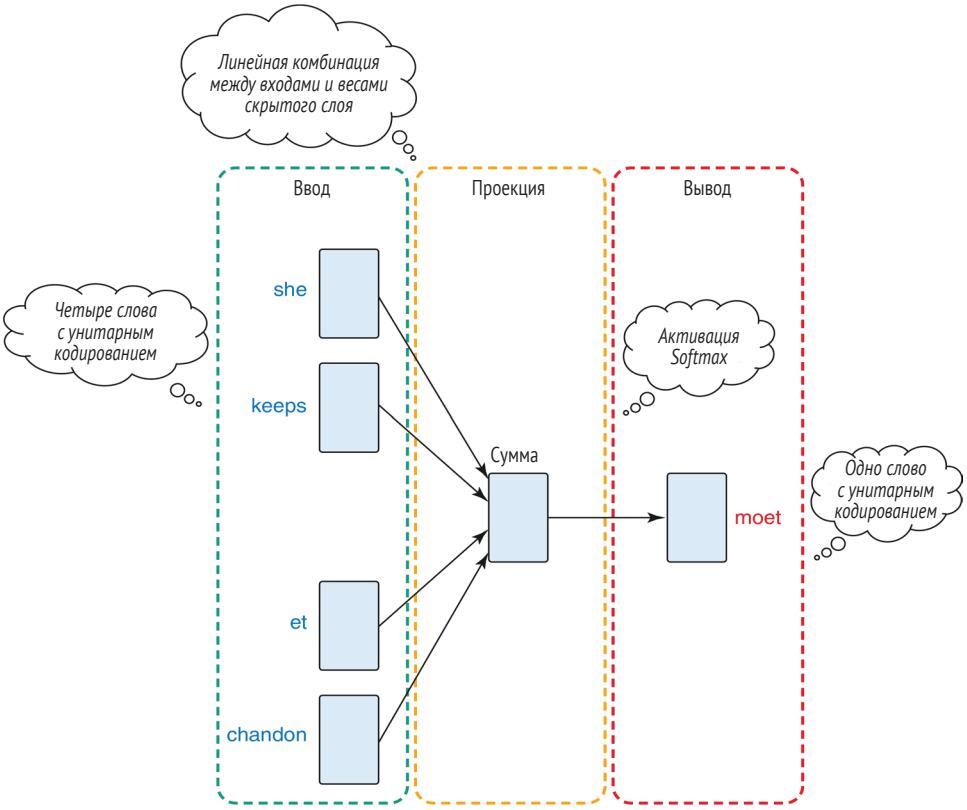


Рис. 2.10 ❖ Модель «непрерывный мешок слов»

Нейроны в скрытом слое не имеют функции активации, поэтому они линейно объединяют веса и входные данные (перемножьте их и суммируйте все результаты). Входной слой имеет количество нейронов, равное количеству слов в тексте для каждого слова; word2vec требует, чтобы каждое слово было представлено в виде вектора, кодированного с помощью прямого унитарного кодирования.

Давайте посмотрим, как выглядит этот вектор. Представьте, что у вас есть набор данных с тремя словами: [cat, dog, mouse]. У вас есть три вектора, в каждом из

которых все значения установлены на 0, кроме одного, который установлен на 1 (тот, что идентифицирует это конкретное слово):

```
dog   : [0,0,1]
cat   : [0,1,0]
mouse : [1,0,0]
```

Если вы добавите слово «lion» в набор данных, векторы с унитарным кодированием для этого набора данных будут иметь размерность 4:

```
lion  : [0,0,0,1]
dog   : [0,0,1,0]
cat   : [0,1,0,0]
mouse : [1,0,0,0]
```

Если у вас есть 100 слов во входном тексте, каждое слово будет представлено в виде 100-мерного вектора. Следовательно, в модели CBOW у вас будет 100 входных нейронов, умноженных на значение параметра `window` минус 1. Таким образом, если `window` равно 4, у вас будет 300 входных нейронов.

Скрытый слой имеет количество нейронов, равное требуемой размерности результирующих векторов слов. Этот параметр должен быть установлен тем, кто настраивает сеть.

Размер выходного слоя равен числу слов во входном тексте: в этом примере – 100. Модель CBOW для входного текста из 100 слов, размерности векторных представлений, равной 50, со значением окна 4 будет иметь 300 входных, 50 скрытых и 100 выходных нейронов. Следует отметить, что хотя размерности ввода и вывода зависят от размера словаря (в данном случае 100) и параметра `window`, размерность векторных представлений слов, генерируемых моделью CBOW, является параметром, который выбирается пользователем. Например, на рис. 2.11 вы видите следующее:

- входной слой имеет размерность `C×V`, где `C` – длина контекста (соответствующая параметру `window` минус 1), а `V` – это размер словаря;
- скрытый слой имеет размерность `N`, определяемую пользователем;
- выходной слой имеет размерность, равную `V`.

В случае с word2vec входные данные модели CBOW распространяются через сеть, сначала умножая векторы с унитарным кодированием входных слов на их веса, находящиеся между входным и скрытым слоями; это можно представить как матрицу, содержащую вес для каждого соединения между вводом и скрытым нейроном. Они объединяются (умножаются) с весами, находящимися между скрытым и выходным слоями, производя результаты, которые затем передаются через функцию softmax.

Softmax «раздавливает» K-мерный вектор (выходной вектор) произвольных действительных значений к K-мерному вектору действительных значений в диапазоне (0, 1), которые складываются в 1, чтобы иметь возможность представлять распределение вероятностей.

Ваша сеть сообщает вам вероятность того, что каждое выходное слово будет выбрано с учетом контекста (сетевой ввод).

Теперь у вас есть нейронная сеть, которая может предсказать наиболее вероятное слово в тексте, учитывая контекст из нескольких слов (параметр `window`). Эта нейронная сеть может сказать вам, что с учетом контекста типа «мне нравится

есть» вы должны ожидать, что следующим словом будет что-то вроде «пиццы». Обратите внимание, что поскольку порядок слов не учитывается, также можно сказать, что, учитывая контекст «я ем пиццу», следующее слово, которое наиболее вероятно появится в тексте, – это «люблю».

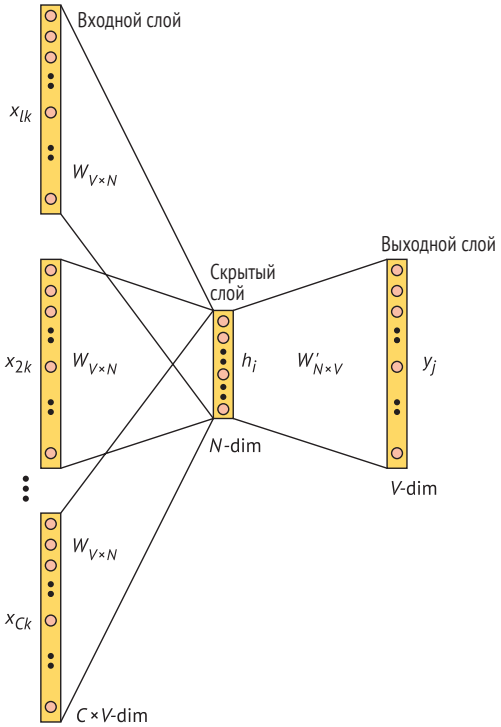


Рис. 2.11 ❖ Веса модели «непрерывный мешок слов»

Но самая важная часть этой нейронной сети для генерации синонимов не учится прогнозировать слова с учетом контекста. Удивительная красота этого метода заключается в том, что внутренне веса скрытого слоя корректируются таким образом, что позволяют определить, когда два слова семантически похожи (потому что они появляются в том же или похожем контексте).

После прямого распространения алгоритм обучения обратного распространения корректирует вес каждого нейрона в разных слоях, поэтому нейронная сеть будет давать более точный результат для каждого нового фрагмента. Когда учебный процесс завершен, веса между скрытым и выходным слоями обозначают векторное представление каждого слова в тексте.

Skip-gram выглядит совсем иначе по отношению к модели CBOW. Тут применимы те же концепции: входные векторы кодируются с помощью унитарной кодировки (по одному на каждое слово), поэтому входной слой имеет количество нейронов, равное количеству слов во входном тексте. Скрытый слой имеет размерность желаемых результирующих векторов слов, а выходной слой имеет количество нейронов, равное количеству слов, умноженному на `window` минус 1. Используя тот же пример, что и раньше, если мы возьмем текст «she keeps moet et chandon in

her pretty cabinet let them eat cake she says» и значение `window` 5, модель word2vec, основанная на модели skip-gram, получит первый образец для | `she` | `keeps` | `moet` | `et` |`chandon` |с вводом `moet` и выводом | `she` | `keeps` | `et` | `chandon` | (*см.* рис. 2.12).

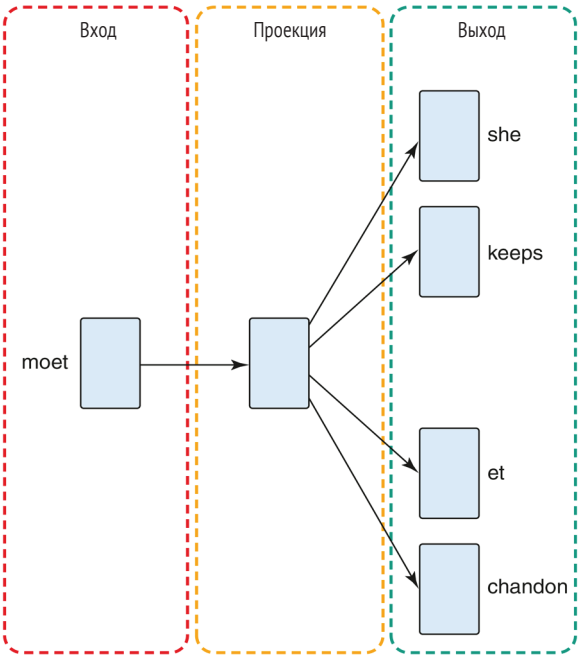


Рис. 2.12 ❖ Модель skip-gram

На рис. 2.13 приведен пример выдержки векторов слов, рассчитанных word2vec для текста набора данных Hot 100 Billboard. Тут показано небольшое подмножество слов, нанесенных на график, чтобы оценить геометрическую семантику слов.

Обратите внимание на ожидаемые закономерности между «me» и «my» в отношении «you» и «your». Также обратите внимание на группы похожих слов или слов, используемых в сходных контекстах, которые являются хорошими кандидатами на синонимы.

Теперь, когда вы немного узнали о том, как работает алгоритм word2vec, давайте напишем код и посмотрим на него в действии. Потом вы сможете объединить его с поисковой системой для расширения синонимов.

Deeplearning4j

Deeplearning4j (DL4J) – это библиотека глубокого обучения для виртуальной машины Java (JVM). Она хорошо зарекомендовала себя среди пользователей Java, и у нее не слишком большой курс обучения для начинающих. Она также поставляется с лицензией Apache 2, что удобно, если вы хотите использовать ее в компании и включить ее в продукт, возможно, с закрытым исходным кодом. Кроме того, в DL4J есть инструменты для импорта моделей, созданных с помощью других фреймворков, таких как Keras, Caffe, TensorFlow, Theano и т. д.

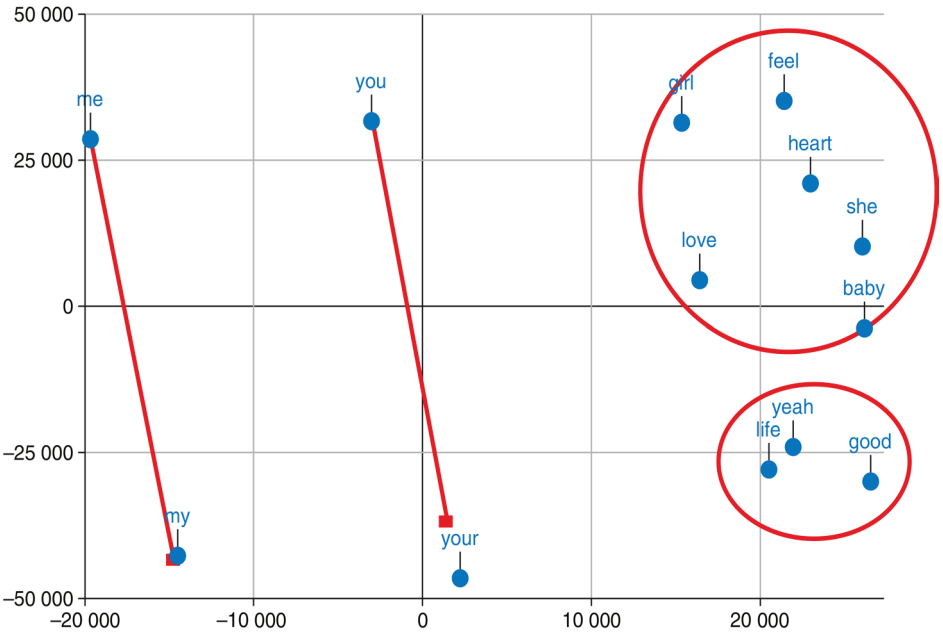


Рис. 2.13 ❖ Отрывки векторов word2vec для набора данных Hot 100 Billboard

2.4.1. Настройка word2vec в Deeplearning4j

В этой книге мы будем использовать DL4J для реализации алгоритмов на базе нейронных сетей. Давайте посмотрим, как применять ее для настройки модели word2vec.

DL4J имеет встроенную реализацию word2vec на базе модели skip-gram. Вам необходимо настроить параметры ее конфигурации и передать входной текст, который вы хотите дать поисковой системе.

Помня об использовании текста песни, давайте предоставим word2vec текстовый файл Billboard Hot 100. Вам нужно вывести векторы слов подходящего размера, поэтому установите для этого параметра конфигурации значение 100, а для размера окна – 5.

Листинг 2.9 ❖ Пример word2vec в DL4J

```
String filePath = new ClassPathResource(
    "billboard_lyrics_1964-2015.txt").getFile()
    .getAbsolutePath();
SentenceIterator iter = new BasicLineIterator(filePath);

Word2Vec vec = new Word2Vec.Builder()
     .layerSize(100)
     .windowSize(5)
     .iterate(iter)
     .elementsLearningAlgorithm(new CBOW<>())
     .build();
vec.fit();
```

Читает корпус текста, содержащий тексты песен

Устанавливает итератор для корпуса

Создает конфигурацию для word2vec

Устанавливает количество измерений, которые должны иметь векторные представления

Устанавливает параметр window

Устанавливает word2vec для итерации выбранного корпуса

Выполняет обучение

Использует модель CBOW

```
String[] words = new String[]{"guitar", "love", "rock"};
for (String w : words) {
    Collection<String> lst = vec.wordsNearest(w, 2);
    System.out.println("2 Words closest to '"
        + w + "': " + lst);
}
```

Получает слова, наиболее близкие к входному слову

Выводит ближайшие слова

Вы получаете вывод, который выглядит довольно прилично:

```
2 Words closest to 'guitar': [giggle, piano]
2 Words closest to 'love': [girl, baby]
2 Words closest to 'rock': [party, hips]
```

Обратите внимание, что вы также можете использовать в качестве альтернативы модель skip-gram, изменив `elements-LearningAlgorithm`.

Листинг 2.10 ❖ Использование модели skip-gram

```
Word2Vec vec = new Word2Vec.Builder()
    .layerSize(...)
    .windowSize(...)
    .iterate(...)
    .elementsLearningAlgorithm(new SkipGram<>())
     .build();
vec.fit();
```

Использует модель skip-gram

Как вы убедились, настроить такую модель и получить результаты за разумное время – просто (обучение модели word2vec заняло около 30 секунд на «обычном» ноутбуке). Имейте в виду, что теперь мы будем стремиться использовать это в сочетании с поисковой системой, которая должна дать более совершенный алгоритм расширения синонимов.

2.4.2. Расширение синонимов на базе Word2vec

Теперь, когда у вас в руках этот мощный инструмент, нужно быть осторожным! При использовании WordNet у вас есть ограниченный набор синонимов, поэтому индекс нельзя взорвать. С векторами слов, сгенерированными word2vec, вы можете попросить модель возвращать самые близкие слова для каждого слова, которое будет проиндексировано. Это может быть неприемлемо с точки зрения производительности (как для времени выполнения, так и для хранения), поэтому вам нужно придумать стратегию ответственного использования word2vec. Например, можно ограничить типы слов, для которых вы просите у word2vec ближайшие слова. При обработке естественного языка обычно помечают каждое слово как *часть речи*, которая обозначает его синтаксическую роль в предложении. Распространенными частями речи являются СУЩЕСТВИТЕЛЬНОЕ, ГЛАГОЛ и ПРИЛАГАТЕЛЬНОЕ; есть также и менее крупные, такие как ИС и ИН (имя собственное и имя нарицательное соответственно). Например, вы можете принять решение использовать word2vec только для слов, чьи части речи – это ИН или ГЛАГОЛ, чтобы избежать раздувания индекса из-за синонимов прилагательных. Еще один метод – посмотреть, насколько информативен документ. Короткий текст имеет относительно низкую вероятность попадания в запрос, поскольку он состоит из нескольких термов. Поэтому вы можете сосредоточиться на таких

документах и расширить их синонимы, вместо того чтобы фокусироваться на более длинных документах.

С другой стороны, «информативность» документа зависит не только от его размера. Таким образом, можно использовать другие методы, такие как просмотр *весов* термов (количество раз, когда терм появляется во фрагменте текста) и пропуск тех, которые имеют малый вес.

Вы также можете использовать результаты word2vec, только если они имеют хороший показатель сходства. Если вы используете косинусное расстояние для измерения ближайших соседей вектора слова, такие соседи могут находиться слишком далеко (низкий показатель сходства), но тем не менее быть ближайшими. В этом случае вы можете принять решение не использовать этих соседей.

Теперь, когда вы обучили модель word2vec на наборе данных Hot 100 Billboard с применением Deeplearning4j, давайте используем ее вместе с поисковой системой для генерации синонимов.

Как было объяснено в главе 1, фильтр токенов выполняет операции для термов, предоставляемых токенизатором, например фильтрует их или, как в этом случае, добавляет другие термы для индексации. `TokenFilter` Lucene основан на API `incrementToken`, который возвращает булево значение, равное false, в конце потока токенов. Средства реализации этого API потребляют один токен за раз (например, фильтруя или расширяя токен). На рис. 2.14 показана схема того, как должно работать расширение синонимов на базе word2vec.

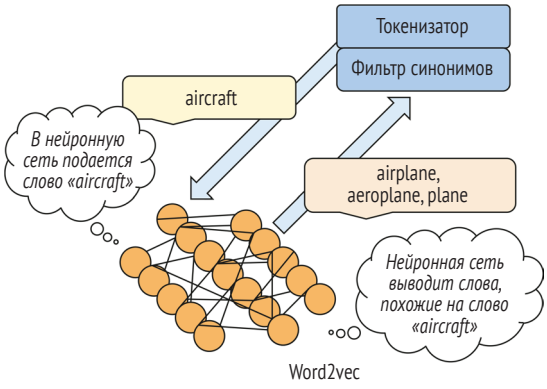


Рис. 2.14 ❖ Расширение синонимов во время поиска с помощью word2vec

Вы закончили обучение для word2vec, поэтому можете создать фильтр синонимов, который будет использовать обученную модель для прогнозирования синонимов термов во время фильтрации. Вы создадите `TokenFilter`, который может использовать DL4J word2vec для входных токенов, что означает реализацию левой части рис. 2.14.

API Lucene для фильтрации токенов требуют реализации метода `incrementToken`. Этот метод возвращает значение `true`, если по-прежнему есть токены, которые нужно потреблять из потока токенов, или значение `false`, если токенов для фильтрации больше нет. Основная идея заключается в том, что фильтр токенов вернет

значение `true` для всех оригинальных токенов и значение `false` для всех связанных синонимов, которые вы получаете от word2vec.

Листинг 2.11 ❖ Фильтр расширения синонимов на базе Word2vec

```
protected W2VSynonymFilter(TokenStream input,
    Word2Vec word2Vec) {
  super(input);
  this.word2Vec = word2Vec;
}

@Override
public boolean incrementToken()
    throws IOException {
  if (!outputs.isEmpty()) {
    ...
  }

  if (!SynonymFilter.TYPE_SYNONYM.equals(
      typeAtt.type())) {
    String word = new String(termAtt.buffer())
        .trim();
    List<String> list = word2Vec.
        similarWordsInVocabTo(word, minAcc);
    int i = 0;
    for (String syn : list) {
      if (i == 2) {
        break;
      }
      if (!syn.equals(word)) {
        CharsRefBuilder charsRefBuilder = new CharsRefBuilder();
        CharsRef cr = charsRefBuilder.append(syn).get();

        State state = captureState();
        outputs.add(new PendingOutput(state, cr));
        i++;
      }
    }
  }
  return !outputs.isEmpty() || input.incrementToken();
}
```

Создает фильтр токенов, который принимает уже обученную модель word2vec

Реализует API Lucene для фильтрации токенов

Добавляет кешированные синонимы в поток токенов (см. следующий листинг кода)

Расширяет токен, только если он не является синонимом (чтобы избежать циклов в расширении)

Для каждого терма используется word2vec, чтобы найти самые близкие слова с верностью выше minAcc (например, 0,35)

Записывает не более двух синонимов для каждого токена

Записывает текущее состояние исходного терма (не синонима) в потоке токенов (например, начальная и конечная позиции)

Записывает значение синонима

Создает объект, содержащий синонимы для добавления в поток токенов, после того как все первоначальные термы были использованы

Этот код перебирает все термы и, когда находит синоним, помещает его в список ожидающих результатов для расширения (список `outputs`). Вы примените эти термы, и они будут добавлены (фактические синонимы) после обработки каждого исходного терма, как показано далее.

Листинг 2.12 ❖ Расширение ожидающих синонимов

```
...
  if (!outputs.isEmpty()) {
    PendingOutput output = outputs.remove(0);
    restoreState(output.state);
    termAtt.copyBuffer(output.charsRef.chars, output
```

Получает первый ожидающий вывод для расширения

Получает состояние исходного терма, включая его текст, положение в текстовом потоке и т. д.

```
      .charsRef.offset, output.charsRef.length);
    typeAtt.setType(SynonymFilter.TYPE_SYNONYM);
    return true;
  }
```

Устанавливает тип терма в качестве синонима

Устанавливает текст синонима, заданный word2vec и ранее сохраненный в ожидающем выводе

Результаты вывода word2vec используются в качестве синонимов, только если они имеют верность, превышающую определенный порог, как обсуждалось в предыдущем разделе. Фильтр выбирает лишь два слова, наиболее близких к данному терму (согласно word2vec), с верностью минимум 0,35 (что не так уж и высоко) для каждого терма, передаваемого токенизатором. Если вы передадите фильтру предложение «I like pleasure spiked with pain and music is my airplane», он расширит слово «airplane» двумя дополнительными словами: «airplanes» и «aeroplane» (*см.* заключительную часть расширенного потока токенов, показанного на рис. 2.15).

Рис. 2.15 ❖ Поток токенов после расширения синонимов с помощью word2vec

2.5. Оценки и сравнения

Как упоминалось в главе 1, обычно можно собирать метрики, включая точность, полноту, запрос с нулевым результатом и т. д., как до, так и после введения расширения запроса. Также обычно полезно определить наиболее подходящий набор конфигурации для всех параметров нейронной сети. Обычная нейронная сеть имеет много параметров, которые можно настроить:

- общая сетевая архитектура, например использование одного или нескольких скрытых слоев;
- преобразования выполняются в каждом слое;
- количество нейронов в каждом слое;
- связи между нейронами, принадлежащими к разным слоям;
- количество раз (также называемых *эпохами*), которое сеть должна прочитать все обучающие наборы, чтобы достичь своего конечного состояния (возможно, с низкой ошибкой и высокой верностью).

Эти параметры также применяются к другим методам машинного обучения. В случае с word2vec вы можете выбрать:

- размер сгенерированных векторных представлений слов;
- окно, используемое для создания фрагментов для обучения моделей без учителя;
- какую архитектуру использовать: CBOW или skip-gram.

Как видите, существует множество возможных настроек параметров.

Перекрестная проверка – это метод оптимизации параметров при одновременном обеспечении того, что модель машинного обучения работает достаточно хорошо для данных, отличающихся от той, которая используется для обучения. При перекрестной проверке исходный набор данных разделяется на три подмножества: обучающий набор, проверочный набор и тестовый. Обучающий набор ис-

пользуется в качестве источника данных для обучения модели. На практике он часто применяется для обучения нескольких отдельных моделей с различными настройками для доступных параметров. Набор перекрестной проверки используется для выбора модели с наиболее эффективными параметрами. Это можно сделать, например, взяв каждую пару входных и требуемых выходных данных в наборе перекрестной проверки и посмотреть, дает ли модель результаты, равные или близкие к желаемому выходному значению, при наличии этого конкретного входного значения. Тестовый набор используется так же, как набор перекрестной проверки, за исключением того, что он применяется только моделью, выбранной путем тестирования на наборе перекрестной проверки. Верность результатов на тестовом наборе можно считать хорошим показателем общей эффективности модели.

2.6. Соображения относительно продукционных систем

В этой главе вы увидели, как использовать word2vec для создания синонимов из данных, которые будут проиндексированы и найдены. Большинство существующих продукционных систем уже содержит множество проиндексированных документов, и в таких случаях зачастую невозможно получить доступ к исходным данным, поскольку они существовали до того, как были проиндексированы. В случае индексации 100 лучших песен года для создания поисковой системы текстов песен нужно учитывать, что рейтинги самых популярных песен меняются каждый день, неделю, месяц и год. Это подразумевает, что набор данных будет меняться со временем; поэтому если вы не будете хранить старые копии в отдельном хранилище, то не сможете создать модель word2vec для всех проиндексированных документов (текстов песен) позже.

Решение данной проблемы заключается в работе с поисковой системой в качестве основного источника данных. Когда вы настраиваете word2vec с использованием DL4J, то выбираете предложения из одного файла:

```
String filePath = new ClassPathResource("billboard_lyrics.txt").getFile()
   .getAbsolutePath();
SentenceIterator iter = new BasicLineIterator(filePath);
```

Учитывая развивающуюся систему, куда тексты песен из разных файлов поступают ежедневно, еженедельно или ежемесячно, вам придется получать предложения непосредственно из поисковой системы. По этой причине мы создадим `SentenceIterator`, который считывает сохраненные значения из индекса Lucene.

Листинг 2.13 ❖ Извлечение предложений для word2vec из индекса Lucene

```
public class FieldValuesSentenceIterator implements
    SentenceIterator {

  private final IndexReader reader;
  private final String field;
  private int currentId;

  public FieldValuesSentenceIterator(
      IndexReader reader, String field) {
    this.reader = reader;
```

Представление индекса, используемого для извлечения значений документа

Специальное поле для извлечения из него значений

Идентификатор текущего документа, который используется, потому что это итератор

```
    this.field = field;
    this.currentId = 0;
  }

  ...

  @Override
  public void reset() {
    currentId = 0;  <-- Первый идентификатор документа – это всегда 0
  }
}
```

В примере с поисковой системой текстов песен текст был проиндексирован в поле `text`. Поэтому вы выбираете предложения и слова, которые будут использоваться для обучения модели word2vec из этого поля.

Листинг 2.14 ❖ Чтение предложений из индекса Lucene

```
Path path = Paths.get("/path/to/index");
Directory directory = FSDirectory.open(path);
IndexReader reader = DirectoryReader.open(directory);
SentenceIterator iter = new FieldValuesSentenceIterator(reader, "text");
```

После того как вы все настроите, вы передадите новый `SentenceIterator` в реализацию word2vec:

```
SentenceIterator iter = new FieldValuesSentenceIterator(reader, "text");
Word2Vec vec = new Word2Vec.Builder()
  .layerSize(100)
  .windowSize(5)
  .iterate(iter)
  .build();
vec.fit();
```

На этапе обучения `SentenceIterator` предлагается выполнить итерацию по строкам.

Листинг 2.15 ❖ Передача значений полей в word2vec для обучения для каждого документа

```
@Override
public String nextSentence() {
  if (!hasNext()) {   <-- Итератор имеет больше предложений, если идентификатор текущего документа не превышает количество документов, содержащихся в индексе
    return null;
  }
  try {
    Document document = reader.document(currentId,
        Collections.singleton(field));   <-- Получает документ с текущим идентификатором (берется только нужное вам поле)
    String sentence = document.getField(field)
        .stringValue();   <-- Получает значение текстового поля из текущего документа Lucene в виде строки
    return preProcessor != null ? preProcessor
        .preProcess(sentence) :
   sentence;   <-- Возвращает предложение, которое предварительно обрабатывается, если вы установили предпроцессор (например, чтобы удалить ненужные символы или токены)
  } catch (IOException e) {
    throw new RuntimeException(e);
  } finally {
```

```
    currentId++;
  }
}
@Override
public boolean hasNext() {
  return currentId < reader.numDocs();
}
```

Увеличивает идентификатор документа для следующей итерации

Таким образом, word2vec можно часто повторно обучать в существующих поисковых системах без необходимости сохранять исходные данные. Фильтр расширения синонимов можно обновлять по мере обновления данных в поисковой системе.

2.6.1. Синонимы против антонимов

Представим, что у вас есть предложения: «Я люблю пиццу», «Я ненавижу пиццу», «Я люблю макароны», «Я ненавижу макароны», «Я люблю макароны» и «Я ем макароны». Это мог бы быть небольшой набор предложений для word2vec, чтобы использовать его для изучения точных представлений в реальной жизни. Но отчетливо видно, что между словами «я» слева и «пицца» и «макароны» справа стоят глаголы. Поскольку word2vec изучает векторные встраивания слов с использованием похожих фрагментов текста, у вас могут получиться схожие векторы слов для глаголов «нравиться», «ненавидеть», «любить» и «есть». Таким образом, word2vec может сообщить, что глагол «любить» близок к глаголу «нравиться» и «есть» (что хорошо, учитывая, что все предложения связаны с едой), но также и глагол «ненавидеть», который определенно не является синонимом слова «любить».

В некоторых случаях эта проблема может быть неважной. Предположим, вы хотите пойти поужинать и ищете хороший ресторан в интернете. Вы пишите в поисковике запрос «обзоры ресторанов, которые любят люди». Если вы получите отзывы о «ресторанах, которые люди ненавидят», вам будет ясно, куда идти не стоит. Но это крайний случай; как правило, вам не нужно, чтобы антонимы (противоположность синониму) расширялись как синонимы.

Не беспокойтесь – как правило, в тексте достаточно информации, чтобы сообщить вам, что хотя слова «ненавидеть» и «любить» появляются в одинаковых контекстах, они не являются правильными синонимами. Тот факт, что этот корпус текста состоит только из таких предложений, как «Я ненавижу пиццу» или «Я люблю макароны», усложняет задачу: обычно глаголы «ненавижу» и «люблю» также появляются в других контекстах, что помогает word2vec выяснить, что они не похожи. Чтобы увидеть это, давайте выполним оценку ближайших слов слова nice (приятный), используя их сходство:

```
String tw = "nice";
Collection<String> wordsNearest = vec.wordsNearest(tw, 3);
System.out.println(tw + " -> " + wordsNearest);
for (String wn : wordsNearest) {
  double similarity = vec.similarity(tw, wn);
  System.out.println("sim(" + tw + "," + wn + ") : " + similarity);
  ...
}
```

Сходство между векторами слов может помочь вам исключить ближайших соседей, которые недостаточно похожи. Образец word2vec указывает, что ближайшие слова «nice» – это «cute», «unfair» и «real»:

```
nice -> [cute, unfair, real]
sim(nice,cute) : 0.6139052510261536
sim(nice,unfair) : 0.5972062945365906
sim(nice,real) : 0.5814308524131775
```

«Cute» – это синоним. «Unfair» – не антоним, а прилагательное, выражающее негативные чувства; это не очень хороший результат, потому что он контрастирует с положительной природой «nice» и «cute». «Real» также не выражает ту же общую семантику, что и «nice». Чтобы это исправить, можно, например, отфильтровать ближайших соседей, чье сходство меньше абсолютного значения 0,5 или меньше наибольшего сходства минус 0,1. Вы предполагаете, что первый ближайший сосед обычно достаточно хорош, если его сходство больше 0,5; когда это применимо, вы исключаете слова, которые находятся слишком далеко от ближайшего соседа. В этом случае, отфильтровывая слова, чье сходство меньше, чем наибольшее сходство ближайшего соседа (0,61) минус 0,1, вы отфильтровываете и слова «unfair» и «real» (у каждого сходство меньше 0,60).

Резюме

- Расширение синонимов может быть удобным способом улучшить полноту и сделать пользователей вашей поисковой системы счастливее.
- Общие методы расширения синонимов основаны на статических словарях, которые могут требовать ручного обслуживания или часто далеки от данных, для которых они используются.
- Нейронные сети прямого распространения являются основой многих архитектур нейронных сетей. В нейронной сети прямого распространения информация передается от входного слоя к выходному; между этими двумя слоями может быть один или несколько скрытых слоев.
- Word2vec – это алгоритм на базе нейронной сети прямого распространения для изучения векторных представлений слов, который можно использовать для поиска слов с похожим значением или слов, которые встречаются в сходных контекстах, поэтому его целесообразно применять и для расширения синонимов.
- Вы можете использовать *непрерывный мешок слов* или skip-gram для word2vec. В CBOW целевое слово используется в качестве выходных данных сети, а оставшиеся слова фрагментов текста – в качестве входных данных. В модели skip-gram целевое слово применяется в качестве входных данных, а контекстные слова являются выходными. Обе эти модели работают хорошо, но skip-gram обычно предпочтительнее, потому что он лучше работает с редкими словами.
- Модели Word2vec могут обеспечить хорошие результаты, но вам нужно управлять значениями слов или частями речи при использовании их в качестве синонимов.
- Работая с word2vec, будьте осторожны, чтобы не допустить использования антонимов в качестве синонимов.

Часть II

ПОДКЛЮЧЕНИЕ НЕЙРОННЫХ СЕТЕЙ ДЛЯ ИСПОЛЬЗОВАНИЯ ИХ В ПОИСКОВОЙ СИСТЕМЕ

Теперь, когда вы уже что-то знаете об основах поиска и глубокого обучения, вы можете приступить к использованию нейронных сетей в поисковой системе везде, где считаете нужным, верно? В теории да; на практике нет. Глубокие нейронные сети – это не магия: нужно проявлять крайнюю осторожность, когда вы решаете, где имеет смысл использовать такие мощные методы и как это делать. В главах 3–6 рассматриваются задачи, которые обычно выполняет каждая современная поисковая система, и освещаются их ограничения. Когда мы их идентифицируем, то рассмотрим, как использовать глубокое обучение для смягчения таких проблем. Вы узнаете, как лучше решить задачу поисковой системы, взглянув на пример с результатами или используя более строгие показатели поиска информации.

Глава 3

От простого поиска к генерации текста

О чем идет речь в этой главе:
- расширение запросов;
- использование журналов поиска для создания обучающих данных;
- рекуррентные нейронные сети;
- генерация альтернативных запросов с помощью рекуррентных нейросетей.

Когда только появился интернет и поисковые системы (конец 1990-х), люди выполняли поиск только по ключевым словам. Пользователи могли набрать фразу «фильм земекис будущее», чтобы найти информацию о фильме «*Назад в будущее*» режиссера Роберта Земекиса. Хотя поисковые системы эволюционировали и сегодня мы можем вводить запросы, используя естественный язык, многие пользователи по-прежнему полагаются на ключевые слова при поиске. Для этих пользователей было бы полезно, если бы поисковая система могла генерировать правильный запрос на основе ключевых слов, которые они вводят: например, взяв фразу «фильм земекис будущее» и сгенерировав «“Назад в будущее” Роберта Земекиса». Давайте назовем сгенерированный запрос *альтернативным запросом* в том смысле, что он представляет собой альтернативное (текстовое) представление потребности в информации, выраженной пользователем.

Эта глава научит вас, как добавить возможности генерации текста в свою поисковую систему, чтобы, учитывая пользовательский запрос, она генерировала несколько альтернативных запросов, которые будут выполняться «под капотом» вместе с оригинальным. Цель состоит в том, чтобы выразить запрос дополнительными способами для расширения сети поиска, не прося пользователя подумать или ввести альтернативные варианты. Чтобы добавить генерацию текста в поисковую систему, вы будете использовать мощную архитектуру для нейронных сетей под названием *рекуррентная нейронная сеть*.

Рекуррентные нейронные сети обладают той же гибкостью, что и сети прямого распространения без излишеств, о которых вы узнали в главе 2. Но РНС также имеют преимущество, которое состоит в том, что они способны работать с длинными последовательностями входных и выходных данных.

Прежде чем вы научитесь использовать рекуррентные нейронные сети, давайте вспомним, что вы делали с сетями прямого распространения. Вы использовали

их с определенной моделью, word2vec, чтобы улучшить расширение синонимов, дабы запрос можно было расширить с использованием (одного или нескольких) его синонимов. Более эффективное расширение синонимов повышает эффективность поисковой системы, возвращая более релевантные документы. Word2vec использует специально разработанную нейронную сеть для генерации плотных векторных представлений слов. Такие векторы можно использовать для вычисления сходства двух слов по расстояниям между их векторами, как в случае расширения синонимов. Но они также могут использоваться в качестве входных данных для более сложных архитектур нейронных сетей, таких как РНС. Именно так вы и будете их использовать в данной главе.

> **ПРИМЕЧАНИЕ** На практике нейронные сети обычно обучают выполнять определенные задачи, организуя функции активации нейронов, слои и их соединения, в зависимости от проблемы. В оставшейся части этой книги вы познакомитесь с различными архитектурами нейронных сетей, каждая из которых решает разные проблемы. Например, в области компьютерного зрения, где входными данными сети обычно являются изображения или видео, обычно используют *сверточные нейронные сети*. В этих сетях каждый слой имеет свою особую специфическую функцию: существуют сверточные слои, слои подвыборки и т. д. В то же время агрегация этих слоев позволяет построить глубокую нейронную сеть, в которой пиксели постепенно преобразуются в нечто более абстрактное: например, пиксели → ребра → объекты →. Мы кратко рассмотрели их в главе 1 и более подробно рассмотрим в главе 8.

В главе 1 вы увидели, как пользователь может выразить потребность в информации в виде нескольких слегка отличающихся версий и как даже небольшие изменения в способе написания запроса могут повлиять на то, какие документы возвращаются первыми. Таким образом, при обучении нейронной сети, чтобы генерировать выходные запросы из входных, полезно выходить за рамки только слов в запросе, отдельно от их контекста. Цель состоит в том, чтобы генерировать текстовые запросы, которые семантически похожи на входной запрос; это позволяет поисковой системе возвращать результаты поиска, основываясь на различных способах выражения одной и той же фундаментальной потребности (через запрос). Вы можете использовать рекуррентную нейросеть для создания текста на естественном языке, а затем интегрировать этот сгенерированный текст в поисковую систему. В оставшейся части этой главы вы узнаете, как работают рекуррентные нейронные сети, как их настраивать, чтобы генерировать альтернативные запросы, и как поисковая система с поддержкой РНС повышает эффективность возврата релевантных результатов для конечных пользователей.

3.1. Информационная потребность в сравнении с запросом: преодоление разрыва

В главе 1 говорилось о фундаментальной проблеме того, как пользователи могут лучше всего выразить потребность в информации. Но, будучи пользователем, вы действительно хотите тратить много времени на размышления о том, как сформулировать запрос? Представьте себе, что вы едете на работу на общественном транспорте рано утром, ища информацию в своем телефоне. У вас нет времени

или умственных способностей (еще раннее утро!), чтобы придумать лучший способ взаимодействия с поисковой системой.

Если вы попросите пользователей объяснить нужную им информацию в трех или четырех предложениях, то, скорее всего, получите подробное объяснение конкретной потребности и подробного контекста. Но если вы попросите одного и того же человека выразить то, что он ищет, в коротком запросе не более чем из пяти или шести слов, велика вероятность того, что он не сможет это сделать, потому что не всегда легко сжать подробное требование в короткую последовательность слов. Будучи специалистами по разработке поисковых систем, нам нужно что-то сделать, чтобы преодолеть этот разрыв между намерениями пользователей и получающимися запросами.

3.1.1. Генерация альтернативных запросов

Хорошо известный метод, помогающий пользователям писать запросы, – это подсказка с текстом, который появляется, пока пользователь печатает запрос. Это позволяет пользовательскому интерфейсу поисковой системы направлять пользователя, пока он пишет. Поисковая система явно пытается помочь пользователю набрать «хороший» запрос (мы подробно рассмотрим, как это делается, в главе 4). Еще один подход к заполнению разрыва между потребностью в информации и введенным пользователем запросом заключается в постобработке запроса сразу после его поступления в поисковую систему, но до того, как он будет выполнен. Ответственность такой постобработки заключается в том, чтобы использовать введенный запрос для создания нового, который в некоторой степени «лучше». Конечно, слово «лучше» может иметь разные значения в этом контексте; в этой главе основное внимание уделяется созданию запроса, который выражает одну и ту же информацию различными способами, чтобы повысить вероятность того, что:

- релевантный документ включен в набор результатов;
- более релевантные документы заняли первое место в результатах поиска.

В наши дни это обычно делается вручную и постепенно – вы можете запустить первый запрос, например о «последних исследованиях в области искусственного интеллекта»; затем второй, например «что такое глубокое обучение»; а потом третий, например «рекуррентные нейронные сети для поиска». Термин *вручную* относится к тому факту, что в этом примере вы выполняете запрос, смотрите на результаты, размышляете над ними, пишите и выполняете еще один запрос, смотрите на результаты, размышляете над ними и т. д., пока вы либо не получите то, что вам нужно, либо не сдадитесь.

Цель состоит в том, чтобы создать набор альтернативных запросов без какого-либо взаимодействия с пользователем. Такие запросы должны иметь то же или похожее значение по отношению к исходному запросу, но используя другие слова (при этом они должны быть правильно написаны). Чтобы увидеть, как это должно работать, давайте вернемся к примеру с запросом «фильм Земекис будущее». Если вы вводите эту фразу, поисковая система должна сделать следующее:

1) принять введенный пользователем запрос «фильм Земекис будущее»;
2) передать запрос через цепочку анализа времени и создать преобразованную версию пользовательского запроса – в данном случае мы предполагаем, что вы настроили фильтр для строчных букв в нижнем регистре;

3) передать отфильтрованный запрос «фильм Земекис будущее» рекуррентной нейросети и получить в качестве выходных данных один или несколько альтернативных запросов, например «“Назад в будущее” Роберта Земекиса»;
4) преобразовать исходный отфильтрованный запрос и сгенерированный альтернативный запрос в форму, которая является реализационно-специфической для поисковой системы (*проанализированный* запрос);
5) выполнить запросы к инвертированным индексам.

Как видно по рис. 3.1, вы будете настраивать поисковую систему для использования нейронной сети во время поиска, чтобы генерировать соответствующие альтернативные запросы для добавления к запросу, введенному пользователем. Вы сохраните исходный запрос в том виде, в котором он был написан пользователем, и добавите сгенерированные запросы в качестве дополнительных *необязательных* запросов. К концу главы мы обсудим, как лучше всего использовать сгенерированные запросы.

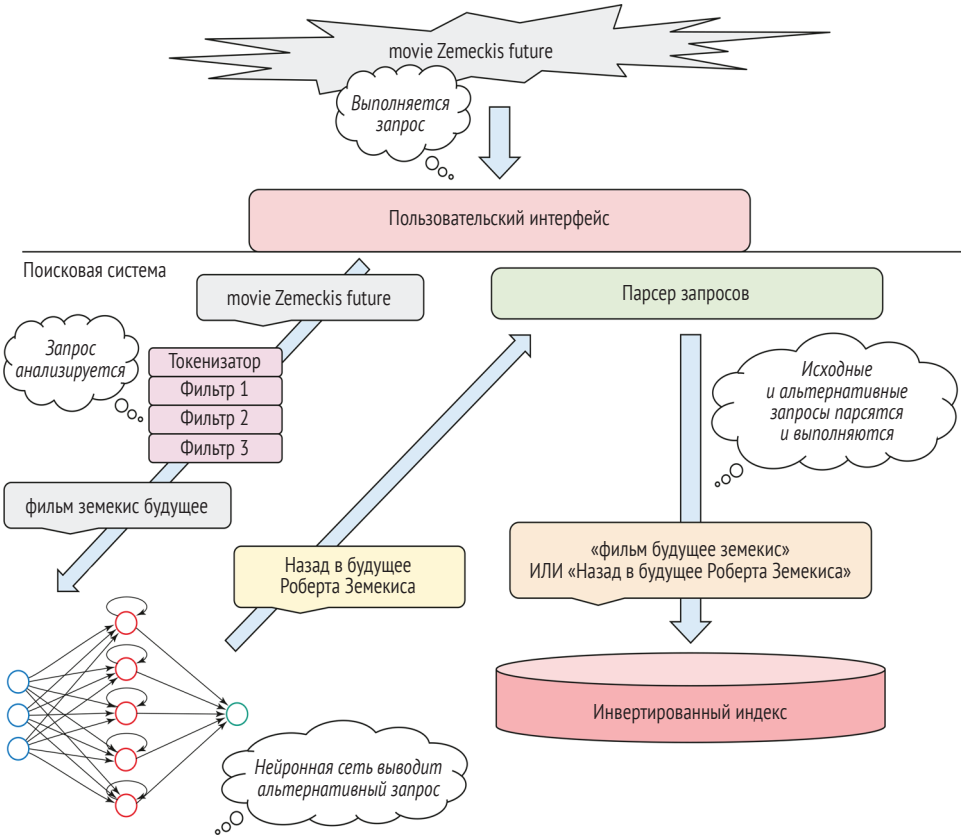


Рис. 3.1 ❖ Генерация альтернативных запросов

Автоматическое расширение запросов – это название метода генерации (порций) запросов под капотом, чтобы максимизировать количество релевантных результатов для конечного пользователя. В некотором смысле расширение сино-

нимов (которое вы видели в главе 2) представляет собой особый случай автоматического расширения запросов, если вы используете его только во время запроса (не для индексации синонимов, а только для расширения синонимов термов в запросе).

Ваша цель – использовать эту функцию расширения запросов, чтобы улучшить механизм запросов, следующим образом:

- минимизация запросов с нулевым результатом. Предоставление альтернативного текстового представления для запроса с большей вероятностью приведет к попаданию в результаты поиска;
- улучшение полноты (доля релевантных документов, которые были получены относительно определенного запроса) за счет включения результатов, которые вы в противном случае пропустили бы;
- повышение точности за счет повышения результатов, соответствующих как исходному запросу, так и альтернативному (что подразумевает, что альтернативные запросы близки к исходному).

> **ПРИМЕЧАНИЕ** Расширение запросов применяется не только для нейронных сетей; этот подход может быть реализован с использованием различных алгоритмов. Теоретически вы можете заменить нейронную сеть в модели расширения запроса на черный ящик. До появления (глубоких) рекуррентных нейросетей существовали другие подходы для создания естественного языка (это подобласть обработки естественного языка, называемая генерацией естественного языка). В конце главы я приведу краткое сравнение с другими методами, чтобы проиллюстрировать «необоснованную эффективность рекуррентных нейронных сетей»[1].

Прежде чем увидеть РНС в действии, как это имеет место во многих сценариях машинного обучения, важно внимательно посмотреть, как вы обучаете модель, а также какие данные вы должны использовать и почему. Как вы помните, при контролируемом обучении вы сообщаете алгоритму, как вы хотите, чтобы модель производила результаты относительно определенных входных данных.

Таким образом, способ, которым вы структурируете входные данные и результаты, во многом зависит от того, чего вы хотите достичь. В следующем разделе представлен краткий обзор трех возможных способов подготовки данных для передачи их в рекуррентную нейросеть.

3.1.2. Подготовка данных

Я выбрал рекуррентные нейросети для реализации расширения запросов, потому что они удивительно хороши и гибки в обучении генерации последовательностей текста, включая последовательности, которые не появляются в обучающих данных, но которые по-прежнему «имеют смысл». Кроме того, РНС обычно требуют меньше настройки по сравнению с другими алгоритмами генерации естественного языка, которые используют грамматику, цепи Маркова и т. д. Все это звучит замечательно, но что вы ожидаете получить при создании альтернативных запросов на практике? Как должны выглядеть сгенерированные запросы? Как это часто бывает в компьютерных науках, ответ – … *как сказать*!

---

[1] *См.*: *Andrej Karpathy*. The Unreasonable Effectiveness of Recurrent Neural Networks. 2015. May 21 (**http://mng.bz/Mxl2**).

Важно определить, чего вы хотите достичь. Если рассмотреть случай, когда пользователь вводит запрос «книги об искусственном интеллекте», вы можете предоставить другие запросы (или предложения), содержащие ту же семантическую информацию, например «публикации из области искусственного интеллекта» или «книги, посвященные теме умных машин». В то же время нужно подумать о том, насколько полезными будут такие альтернативные представления в вашей поисковой системе – возможные альтернативные запросы могут дать нулевые результаты, если у вас нет документов, посвященных теме искусственного интеллекта!

Вы не хотите создавать альтернативное представление запроса, которое идеально, но бесполезно. Вместо этого вы можете внимательно изучить запросы пользователей и предоставить альтернативные представления, основанные на информации, которую они содержат; или вы можете заставить алгоритм генерации запроса получать информацию из индексированных, а не пользовательских данных, чтобы сгенерированные альтернативные запросы лучше отражали то, что уже есть в поисковой системе (и сглаживали проблему альтернативного запроса, который не возвращает результаты).

В реальной жизни у вас часто есть доступ к *журналам запросов*, которые представляют собой простые записи того, что пользователи запрашивают через поисковую систему, с минимальной информацией о результатах. Можно получить много идей, просматривая эти журналы. Например, можно отчетливо увидеть, когда люди не могут найти то, что они ищут, потому что они будут отправлять запросы, похожие по смыслу. Вы также можете наблюдать, как пользователи переключаются с поиска одной темы на другую. В качестве примера предположим, что вы создаете поисковую систему для медиакомпании, которая предоставляет пользователям политические, культурные и новости мира моды.

Вот пример журнала запросов:

```
time: 2017/01/06 09:06:41, query:{"artificial intelligence"}, results:
    {size=10, ids:["doc1","doc5", ...]}
time: 2017/01/06 09:08:12, query:{"books about AI"}, results:
    {size=1, ids:["doc5"]}
time: 2017/01/06 19:21:45, query:{"artificial intelligence hype"}, results:
    {size=3, ids:["doc1","doc8", ...]}
time: 2017/05/04 14:12:31, query:{"covfefe"}, results:
    {size=100, ids:["doc113","doc588", ...]}
time: 2017/10/08 13:26:01, query:{"latest trends"}, results:
    {size=15, ids:["doc113","doc23", ...]}
...
```

Запрос "covfefe" вернул 100 результатов, а первые два итоговых идентификатора документа – doc113 и doc588

Предположим, что это – часть огромного журнала запросов активности пользователей в поисковой системе. Теперь представьте, что вам нужно построить *обучающий набор* из этого журнала запросов – набор примеров входных данных, связанных с желаемыми результатами, – коррелируя аналогичные запросы, чтобы вы могли создавать обучающие примеры, где входные данные – это запрос, а целевой выход – один или более высокое число коррелированных запросов. В этом случае каждый пример будет состоять из одного входного запроса и одного или нескольких выходных запросов. На практике часто используют журналы запросов для таких задач обучения, потому что:

- журналы запросов отражают поведение пользователей в этой конкретной системе, поэтому итоговая модель будет вести себя относительно близко к фактическим пользователям и данным;
- использование или создание других наборов данных может повлечь за собой дополнительные затраты при возможном обучении модели, основанной на разных данных, пользователях, доменах и т. д.

В текущем примере представим, что у вас есть два связанных запроса: «мужская одежда последние тенденции» и «неделя моды в Париже». Вы можете использовать их взаимозаменяемо в качестве входных и выходных данных для обучения нейронной сети. Нетривиальное решение, которое вам нужно принять, – как измерить корреляцию (сходство) двух запросов. Ваши общие знания говорят о том, что оба запроса схожи в том смысле, что мероприятие, приуроченное к неделе моды в Париже, оказывает значительное влияние на тенденции в одежде (моде) (как для мужчин, так и для женщин), поэтому вы можете выбрать «неделю моды в Париже» как альтернативное представление запроса «мужская одежда последние тенденции»; *см.* рис. 3.2. Но в этом контексте ни поисковая система, ни нейронная сеть ничего не знают о теме моды – они просто видят входные и выходные тексты и векторы.

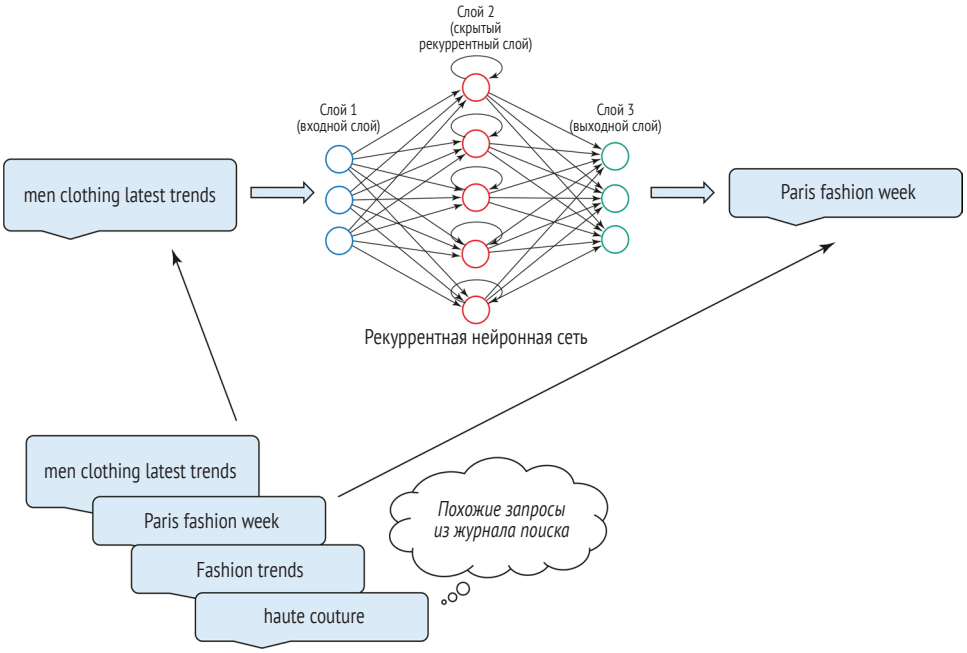


Рис. 3.2 ❖ Обучение на основе запросов

Каждая строка из журнала запросов содержит введенный пользователем запрос, связанный с его результатами поиска: точнее, идентификаторы документов соответствующих результатов. Но это не то, что вам нужно. Ваши обучающие примеры должны состоять из входного запроса и одного или нескольких выходных запросов, которые похожи или каким-либо образом связаны с входными данными. Поэтому, прежде чем вы сможете обучать сеть, вам нужно обработать строки

журнала поиска и создать обучающий набор. Этот тип работы, который включает в себя манипулирование данными и их обработку, часто называют *подготовкой*, или *предварительной обработкой данных*. Хотя это может показаться немного утомительным, но это крайне важно для эффективности любой ассоциированной задачи машинного обучения.

В следующих разделах рассматриваются три различных способа выбора входных и выходных последовательностей для нейронной сети, которые можно использовать для обучения созданию альтернативных запросов: сопоставляя запросы, которые генерируют похожие наборы результатов поиска, поступающие от одних и тех же пользователей в определенные временные окна, или те, что содержат похожие поисковые термы. Каждый из этих вариантов даст определенные побочные эффекты, связанные с тем, как нейронная сеть будет учиться генерировать новые запросы.

Сопоставление запросов, которые генерируют похожие наборы результатов поиска

При первом подходе группируются запросы, которые совместно используют часть связанных с ними результатов поиска. Например, вот что можно извлечь из журнала запросов.

Листинг 3.1 ❖ Сопоставление запросов с использованием общих результатов

```
query:{"artificial intelligence"} -> {"books about AI"
    , "artificial intelligence hype"}   <-- Использует doc1 и doc5
query:{"books about AI"} -> {
    "artificial intelligence"}          <-- Использует doc5
query:{"artificial intelligence hype"} -> {
    "artificial intelligence"}          <-- Использует doc1
query:{"covfefe"} -> {"latest trends"}  |
query:{"latest trends"} -> {"covfefe"}  |   Использует doc113
```

Сопоставляя запросы с общими документами в журнале поиска, вы видите, что запрос «последние тенденции» может генерировать запрос «covfefe» и наоборот, а запросы, связанные с искусственным интеллектом, предлагают хорошие альтернативы.

Обратите внимание, что запрос «последние тренды» относится к относительной концепции: последние тренды в один прекрасный день могут (или будут) существенно отличаться от тенденций завтрашнего дня или тех, что будут на следующей неделе. Если предположить, что тенденция *covfefe* длилась одну неделю, нейронной сети не стоит генерировать «covfefe» в качестве альтернативного запроса для запроса «последних тенденций» через месяц после того, как *covfefe* появился в новостях. Поскольку реальный мир за пределами поисковой системы меняется, вы должны быть осторожны с использованием актуальных данных или, по крайней мере, избегать потенциальных проблем, удаляя обучающие примеры, которые могут привести к плохим результатам, как в этом случае.

Сопоставление запросов, которые приходят от одних и тех же пользователей в определенных временных окнах

Второй потенциальный подход основан на предположении, что пользователи ищут похожие вещи в небольших временных окнах. Например, если вы ищете

«тот конкретный ресторан, в который я ходил, но не могу вспомнить его название», вы будете выполнять несколько поисков, связанных с одной и той же информационной потребностью. Ключевым моментом этого подхода является выявление точных временных окон в журналах запросов, таким образом, запросы, связанные с одной и той же потребностью, можно сгруппировать (независимо от их результатов). На практике определение поисковых сеансов, связанных с одной и той же потребностью, не обязательно будет простым и зависит от того, насколько информативны журналы поиска. Например, если журнал поиска представляет собой плоский список одновременных анонимных поисков от всех пользователей, будет трудно сказать, какие запросы были выполнены одним пользователем. Напротив, если у вас есть информация о каждом пользователе, например его IP-адрес, вы можете попытаться определить сеанс поиска по теме.

Предположим, что образец журнала поиска поступил от одного пользователя. Информация о времени в каждой строке указывает на то, что первые два запроса были выполнены в двухминутном окне, тогда как остальные выполнялись в течение длительного времени. Таким образом, вы можете соотнести первые два запроса – «искусственный интеллект» и «книги об искусственном интеллекте» – и пропустить остальные. Но в реальной жизни люди могут делать несколько вещей одновременно, например желая получить информацию о какой-то технической теме, отправляясь на работу, но им также нужна информация о расписании общественного транспорта или о движении на шоссе. В таких случаях трудно определить, какие запросы семантически сопоставлены, если не смотреть на термы запроса, что вы и сделаете в третьем подходе.

Сопоставление запросов, содержащих похожие поисковые термы

Использовать похожие термы для сопоставления запросов сложно. С одной стороны, это звучит просто. Вы можете найти общие термы среди запросов в журнале поиска, как показано ниже.

Листинг 3.2 ❖ Сопоставление запросов с использованием поисковых термов

```
query:{"artificial intelligence"} ->
    {"artificial intelligence hype"} <------ Использует термы «искусственный» и «интеллект»
query:{"books about AI"} -> {}       <------ Не использует ничего
query:{"artificial intelligence hype"} ->
    {"artificial intelligence"}      <------ Использует термы «искусственный» и «интеллект»
query:{"covfefe"} -> {}              <------ Не использует ничего
query:{"latest trends"} -> {}        <------ Не использует ничего
```

Здесь вы потеряли информацию, которая была передана результатами запроса, как вы можете видеть в сравнении с предыдущим листингом; кроме того, обучающий набор намного меньше и беднее. Давайте посмотрим на запрос «книги об ИИ». Он, безусловно, связан с «искусственным интеллектом» и, возможно, с «шумихой вокруг искусственного интеллекта». Но простое сопоставление термов не отражает того факта, что ИИ – это аббревиатура слов «искусственный интеллект». Можно смягчить эту проблему, применяя методы расширения синонимов, о которых вы узнали в главе 2; для этого требуется дополнительный шаг предварительной обработки, чтобы сгенерировать новые строки в журнале поиска, в которых синонимы расширяются. В этом примере, если ваш алгоритм расширения

синонимов может сопоставить терм «ИИ» с составным термом «искусственный интеллект», вы получите следующие пары ввода/вывода.

Листинг 3.3 ❖ Сопоставление запросов с использованием поисковых термов и расширения синонимов

```
query:{"artificial intelligence"} -> {"artificial intelligence hype"}
query:{"books about AI"} -> {}
query:{"books about artificial intelligence"} ->
    {"artificial intelligence",
    "artificial intelligence hype"}
query:{"artificial intelligence hype"} -> {"artificial intelligence"}
query:{"covfefe"} -> {}
query:{"latest trends"} -> {}
```

Дополнительное сопоставление; совместно использует термы «искусственный» и «интеллект»

Что касается предыдущих результатов, теперь у вас есть дополнительное сопоставление: используя синонимы, сгенерированные новым входным запросом «книги об искусственном интеллекте», которого не было в исходном журнале поиска. Хотя это и выглядит прекрасно, будьте осторожны, поскольку в каждом запросе может быть несколько синонимов для каждого терма. Это часто случается с большими словарями, такими как WordNet, а также при использовании векторных представлений слов на базе сходства (например, word2vec) для расширения синонимов. Обычно желательно иметь больше данных для обучения нейронных сетей, но они должны быть хорошего качества, чтобы давать хорошие результаты. Не будем забывать, что это этап предварительной обработки для обучения нейронной сети, которая будет использоваться для генерации последовательностей. Если вы даете нейронной сети текстовые последовательности, которые не имеют большого смысла (не все синонимы определенного слова хорошо вписываются в любой возможный контекст), они будут генерировать последовательности с небольшим значением или без него.

Если вы планируете использовать расширение синонимов, вам, вероятно, *не* следует расширять каждый возможный синоним; вместо этого можно делать это только для входных запросов, у которых нет соответствующего альтернативного запроса, такого как «книги об ИИ» в предыдущем примере.

Выбор выходных последовательностей из индексированных данных

Если методы, описанные до сих пор, не достаточно хорошо подходят для ваших данных – например, введенные пользователем запросы часто дают слишком мало или ноль результатов, – вы можете получить помощь от индексированных данных. Во многих реальных сценариях индексированные документы имеют обычно относительно короткий заголовок. Такой заголовок можно использовать в качестве запроса, если он тесно связан с исходным входным запросом. Давайте снова выберем запрос «фильм Земекис будущее». Если ввести его в поисковой системе фильмов (такой как IMDB), то мы, вероятно, получим что-то вроде этого:

```
title: Back to the Future
director: Robert Zemeckis
year: 1985
writers: Robert Zemeckis, Bob Gale
stars: Michael J. Fox, Christopher Lloyd, Lea Thompson, ...
```

Давайте представим, как был получен этот документ:

- терм «фильм» находится в списке стоп-слов в поисковой системе фильмов, поэтому совпадения не было;
- терм «Земекис» – совпадение в полях `writers` и `director`;
- терм «будущее» – совпадение в поле `title`.

Поставьте себя на место того, кто рассматривает и запросы, и результаты: по мере того как пользователь вводит запрос, если вы увидели, что пользователь вводит «фильм Земекис будущее», можно сразу сказать, что ему следовало напечатать запрос типа «назад в будущее». Именно такой пример обучения, состоящий из входных данных («фильм Земекис будущее») и целевых выходных данных («назад в будущее»), вы можете передать в нейронную сеть. Можно предварительно обработать журнал поиска, чтобы целевой альтернативный запрос, генерируемый нейронной сетью, был запросом, который вернул бы самый подходящий результат. Вероятно, это поможет сократить количество запросов с нулевыми результатами, поскольку подсказки в альтернативных запросах исходят не из пользовательских запросов, а скорее из текста релевантных документов. Для построения обучающих примеров вы связываете запрос с заголовками двух или трех релевантных документов из журнала поиска, как показано на рис. 3.3.

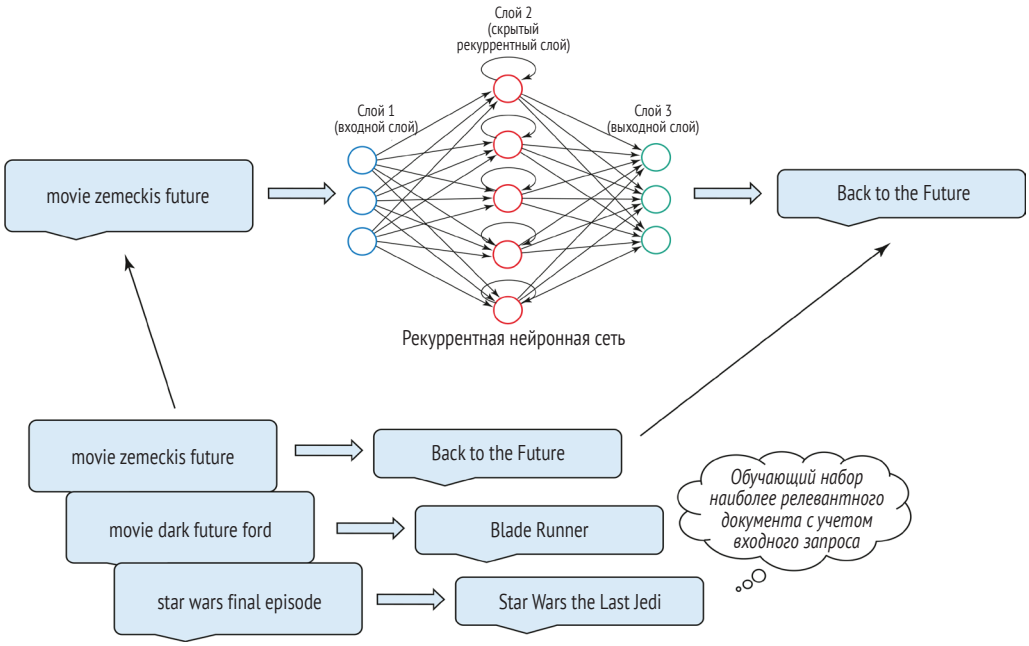


Рис. 3.3 ❖ Обучение на основе названий релевантных документов

Вы можете спросить: почему бы не использовать поисковую систему вместо нейронной сети для генерации альтернативных запросов? Такой подход ограничил бы набор альтернативных запросов для определенного входного текста тем, что поисковая система уже может сделать с точки зрения совпадения. Например, запрос «фильм Земекис будущее» всегда будет давать один и тот же набор альтер-

нативных запросов, если вы используете поисковую систему для их генерации. В случае с примером запроса это бы сработало – но что, если пользователь введет запрос «фильм спилберг будущее» (спутав продюсера фильма с его режиссером)? В поисковой системе нет совпадения с термом «спилберг». Таким образом, поисковая система может вернуть в ответ множество фильмов, режиссером которых был Стивен Спилберг, с термом «будущее», но фильма *Назад в будущее* среди них не будет. Ключевым выводом является то, что вы не ограничены использованием запросов для обучения нейронной сети, если целевой результат тесно связан с входными данными таким образом, чтобы это было полезно для представления альтернативного запроса.

Неконтролируемые потоки текстовых последовательностей

Совершенно иной подход к работе с рекуррентной нейросетью для генерации текста заключается в проведении неконтролируемого обучения с потоками текста. Как упоминалось в главе 1, это форма машинного обучения, при которой алгоритму обучения ничего не сообщается о хорошем (или плохом) результате; алгоритм просто строит модель данных максимально точно.

Вы увидите, что это, пожалуй, самый удивительный способ, с помощью которого рекуррентные нейросети могут научиться генерировать текст: никто не говорит им, что такое хороший результат, поэтому они учатся воспроизводить текстовые последовательности хорошего качества на основе входных данных.

В примере с журналом поиска вы принимаете запросы один за другим, удаляя все остальное:

```
artificial intelligence
books about AI
artificial intelligence hype
covfefe
latest trends
```

Как видите, это простой текст. Все, что вам нужно сделать, – это решить, как определить конец запроса. В этом случае вы можете использовать символ возврата каретки (`\n`) в качестве разделителя двух последовательных запросов, а алгоритм генерации текста будет останавливаться всякий раз, когда генерируется возврат каретки. Это заманчивый подход, потому что он почти не требует предварительной обработки: используемые данные могут поступать откуда угодно, поскольку это просто текст. Позже вы увидите плюсы и минусы данного подхода.

3.1.3. Подведем итог

Вот краткое изложение того, что мы обсуждали в этом разделе:

- выполнение контролируемого обучения по схожим запросам дает вам преимущество в том, что вы можете указать, что считаете хорошими, похожими запросами. Недостатком является то, что эффективность нейронной сети будет зависеть от того, насколько хорошо вы определяете, когда два запроса похожи на этапе подготовки данных;
- возможно, вы не захотите явно указывать, когда два запроса похожи, а вместо этого позволите соответствующим документам запроса предоставить альтернативный текст запроса. Это заставит нейронную сеть генерировать

альтернативные запросы, текст которых поступает из проиндексированных документов (например, заголовки документов), и, вероятно, уменьшит количество запросов с небольшим или нулевым результатом;

- при неконтролируемом подходе поток запросов из журнала поиска рассматривается как последовательность правдоподобных последовательных слов, поэтому требуется небольшая подготовка данных. Преимущества этого подхода состоят в том, что он прост в реализации и позволяет точно определить, какие последовательные запросы (а следовательно, и темы) интересуют пользователей.

Есть много альтернатив и много возможностей для творчества, чтобы создать новые способы генерирования данных, которые соответствуют потребностям ваших пользователей. Ключевой момент – это осторожность при подготовке данных для вашей системы. Мы предполагаем, что вы выбрали один из подходов, обсуждаемых здесь; далее мы рассмотрим, как рекуррентные нейросети учатся генерировать текстовые последовательности.

3.2. Обучение на последовательностях

В главе 1 вы увидели, как выглядит общая архитектура нейронной сети с входными и выходными слоями по краям сети и скрытыми слоями между ними. Затем, во второй главе, мы начали изучать две менее общие модели нейронных сетей (непрерывный мешок слов и skip-gram), используемые для реализации алгоритма word2vec. Обсуждаемые до сих пор архитектуры могут использоваться для моделирования того, как можно отобразить входные данные в соответствующие результаты. В случае с моделью skip-gram вы отображаете входной вектор, обозначающий определенное слово, в выходной вектор, обозначающий фиксированное количество слов.

Давайте рассмотрим простую нейронную сеть прямого распространения, которую вы могли бы применять для определения языка, используемого в текстовых предложениях: например, для четырех языков: английского, немецкого, португальского и итальянского. Это называется *задачей многоклассовой классификации,* где вход – это фрагмент текста, а выход – один из трех или более классов, назначенных этому входу (пример категоризации документов из главы 1 также является задачей многоклассовой классификации). В этом примере нейронная сеть, которая может выполнять такую задачу, будет иметь четыре выходных нейрона, по одному для каждого класса (языка). Только для одного выходного нейрона будет установлено значение 1 в выходном слое, чтобы сигнализировать, что вход принадлежит определенному классу. Например, если значение выходного нейрона 1 равно 1, входной текст классифицируется как текст, написанный на английском; если значение выходного нейрона 2 равно 1, то входной текст классифицируется как текст на немецком, и т. д.

Определить размер входного слоя намного сложнее. Если предположить, что вы работаете с текстовыми последовательностями фиксированного размера, вы можете соответствующим образом спроектировать входной слой. Для определения языка вам понадобится несколько слов, поэтому давайте предположим, что вы настроите входной слой с девятью нейронами: по одному на каждое входное слово; см. рис. 3.4.

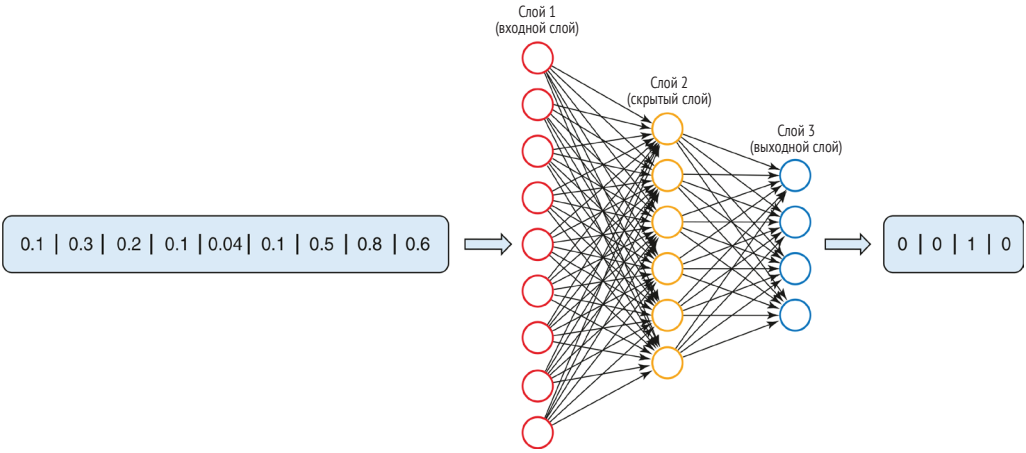


Рис. 3.4 ❖ Нейронная сеть прямого распространения с девятью входами и четырьмя выходами для определения языка

ПРИМЕЧАНИЕ На практике было бы трудно использовать такое отображение «один к одному» между словами и нейронами: как вы видели в случае с методом унитарного кодирования, описанного для word2vec, каждое слово представлено как вектор всех нулей, кроме одного, размер которого равен размеру всего словаря. В этом случае, если бы вы использовали унитарное кодирование, входной слой содержал бы нейроны *размера 9* *. Но поскольку мы фокусируемся на входах фиксированного размера, здесь это не важно.

Ясно, что у вас есть проблема, если текстовая последовательность содержит менее девяти слов: вам нужно наполнить ее поддельными словами-заполнителями. Для более длинных последовательностей вы будете определять язык по девять слов за раз. Рассмотрим текст обзора фильма. Содержимое может быть на одном языке, например итальянском, но в нем может идти речь о фильме, название которого указано на языке оригинала, например на английском. Если разделить текст обзора на последовательности из девяти слов, результаты могут быть «на итальянском» или «английском», в зависимости от того, какая часть текста подается в нейронную сеть.

Учитывая это, как заставить нейронную сеть учиться на последовательностях входных данных, размер которых неизвестен заранее? Если бы вы знали размер каждой последовательности, которую вы хотите, чтобы сеть изучила, то могли бы сделать входной слой достаточно длинным, чтобы включить всю последовательность. Но это ударит по производительности в случае длинных последовательностей, потому что для обучения с большим входом требуется больше нейронов в скрытом слое, чтобы сеть давала точные результаты. Таким образом, это решение будет плохо масштабироваться. Рекуррентные нейросети могут обрабатывать неограниченные последовательности текста, сохраняя фиксированный размер входного и выходного слоев, поэтому они идеально подходят для обучения генерации последовательностей текста в случае автоматического расширения запросов.

3.3. Рекуррентные нейронные сети

Можно рассматривать рекуррентную нейросеть как нейронную сеть, которая может запоминать информацию о своих входных данных по мере их обработки; таким образом, результаты, создаваемые последующими входными данными, также зависят от ранее увиденных входных данных. В то же время размер входного слоя (и выходного слоя, если РНС генерирует последовательности) является фиксированным.

Пока это немного абстрактно, но вы поймете, как это работает на практике и почему это важно. Давайте попробуем сгенерировать текстовые последовательности *без* РНС, используя нейронную сеть прямого распространения с пятью входами и четырьмя выходами. В примере с определением языка использовался один вход для каждого слова, но на практике зачастую удобнее использовать символы вместо строк. Причина этого состоит в том, что количество возможных слов намного больше, чем количество доступных символов, и сети может быть проще научиться обрабатывать все возможные комбинации из 255 символов, чем все возможные комбинации из более чем 300 000 слов[1]. При использовании метода унитарного кодирования символ будет представлен вектором размером 255, а слово, взятое из *Оксфордского словаря английского языка*, будет представлено как вектор размером 301 000! Входному слою нейронной сети понадобится 301 000 нейронов для одного слова, в отличие от 255 нейронов для одного символа. С другой стороны, слово представляет собой комбинацию символов, которая имеет значение. На уровне символов такая информация недоступна, и поэтому нейронная сеть с символьными входами должна сначала научиться генерировать значимые слова из символов; если вы используете слова в качестве входных данных, это уже другой случай. В конце концов, это компромисс.

Например, при использовании символов предложение «большая бурая лисица перепрыгнула через ленивого пса» можно разбить на куски, состоящие из пяти символов. Затем каждый вход подается в нейронную сеть с пятью входными нейронами; *см*. рис. 3.5. Можно передать всю последовательность в сеть независимо от размера входного слоя. Кажется, что вы можете использовать «простую» нейронную сеть, – вам не нужна РНС.

Но представим, что люди, слушающие кого-то, должны понимать, что говорит этот человек, только слыша слова, состоящие из пяти символов, и забывая каждую последовательность, как только они услышат следующую. Например, если бы кто-то сказал: «my name is Yoda» (меня зовут Йода), вы получили бы каждую из приведенных ниже последовательностей, не помня все остальные:

```
my na
 y nam
  name
 name
ame i
```

[1] Это число постоянно увеличивается; *см.* Оксфордский словарь английского языка (**www.oed.com**).

```
me is
e is
 is Y
is Yo
s Yod
 Yoda
```

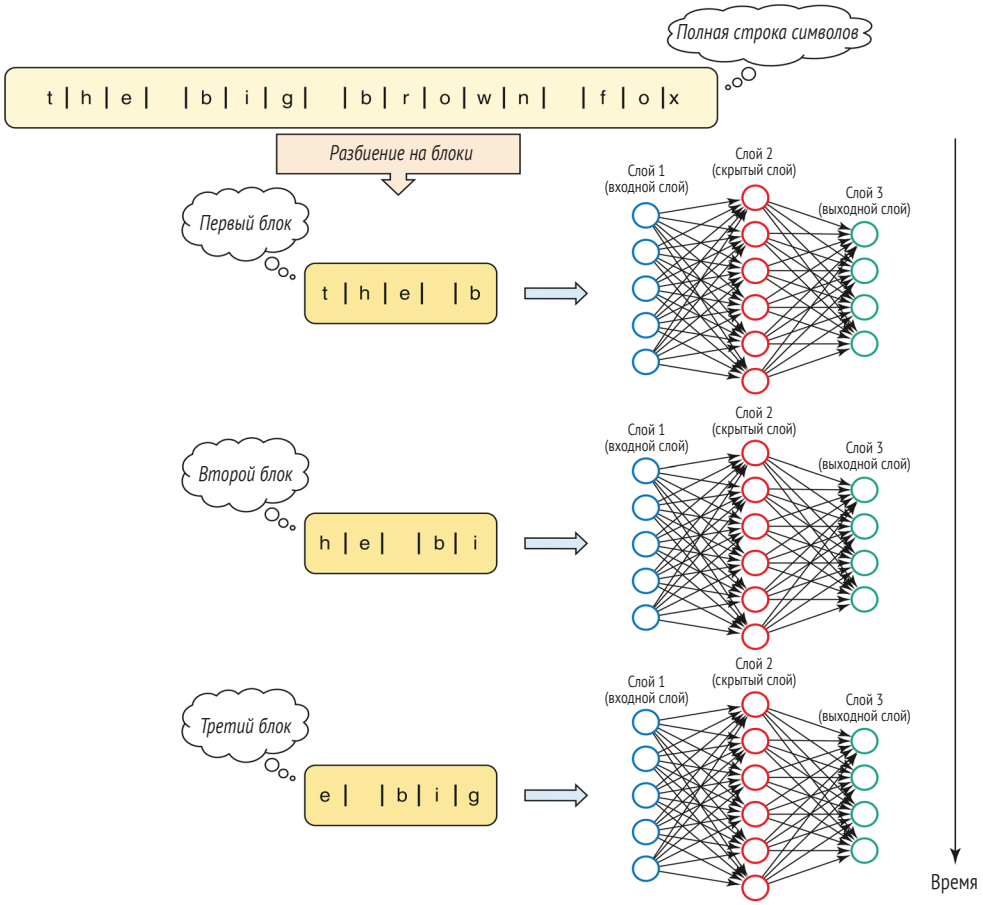


Рис. 3.5 ❖ Нейронная сеть, принимающая входную последовательность с фиксированным входным слоем из пяти нейронов

Теперь вас просят повторить то, что вы услышали. Странно! При таком коротком фиксированном входе вы можете редко получать целые слова, и каждый вход всегда отделен от остальной части предложения.

Что позволяет понять предложение, так это то, что каждый раз, когда вы слышите последовательность из пяти символов, вы отслеживаете то, что получили непосредственно перед этим.

Допустим, у вас есть память размером в 10 символов:

```
my na ()
y nam (m)
 name (my)
name  (my )
ame i (my n)
me is (my na)
e is  (my nam)
 is Y (my name)
is Yo (my name )
s Yod (my name i)
 Yoda (my name is)
```

Это сильное упрощение того, как люди и нейронные сети работают с вводом и памятью, но этого должно быть достаточно для того, чтобы вы увидели обоснование эффективности рекуррентных нейросетей (простая сеть показана на рис. 3.6) при работе с последовательностями по сравнению с обычными нейронными сетями прямого распространения.

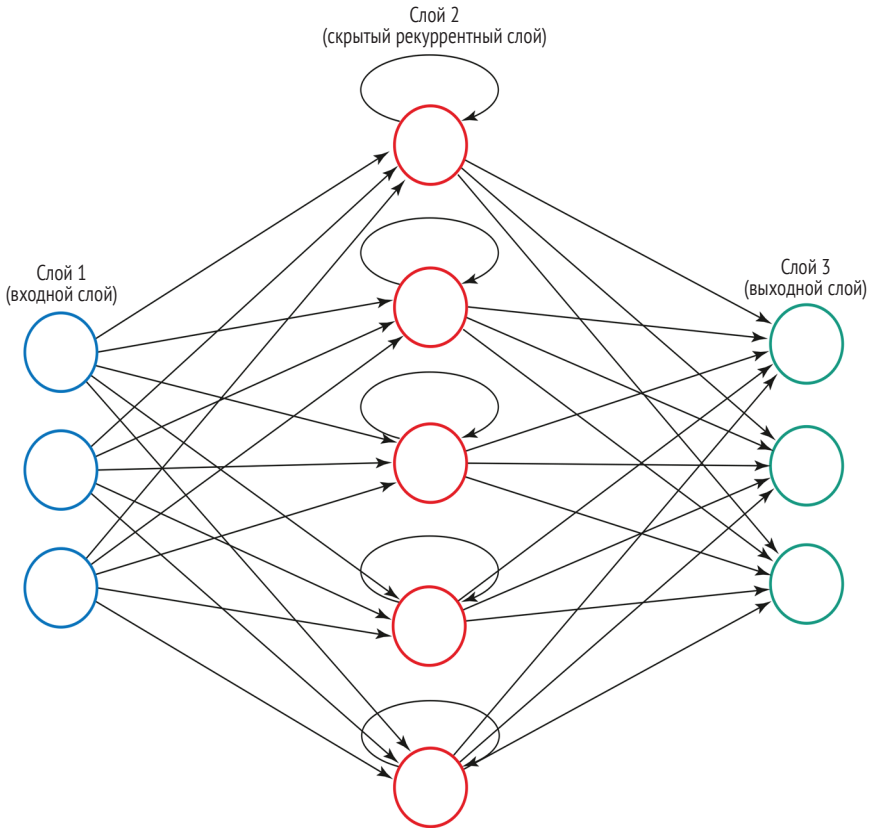


Рис. 3.6 ❖ Рекуррентная нейронная сеть

3.1.1. Внутреннее устройство и динамика РНС

Эти специальные нейронные сети называются *рекуррентными*, потому что с помощью простых циклических соединений в нейронах скрытого слоя сеть становится способной к операциям, которые зависят от текущего входа и предыдущего состояния сети по отношению к предыдущему входу. В случае с обучением генерации текста «меня зовут Йода» внутреннее состояние РНС можно рассматривать как память, которая позволяет понять предложение. Давайте выберем один нейрон в скрытом слое в РНС, как показано на рис. 3.7.

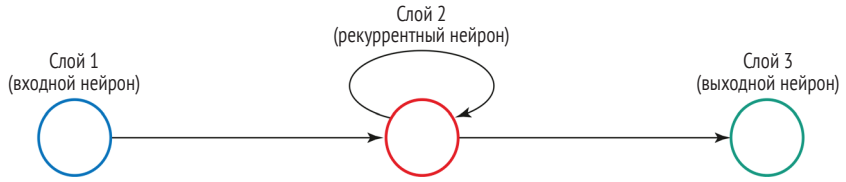


Рис. 3.7 ❖ Рекуррентный нейрон в скрытом слое РНС

Рекуррентный нейрон объединяет сигнал от входного нейрона (стрелка от нейрона слева) *и* сигнал, сохраняемый внутри (стрелка в форме петли), которая играет роль памяти в примере с Йодой. Как видно, этот одиночный нейрон обрабатывает вход, превращая его в выход, учитывая его внутреннее состояние (веса скрытого слоя и функция активации). Он также обновляет свое состояние в зависимости от нового ввода и его текущего состояния. Это именно то, что нужно сделать нейрону, чтобы научиться связывать последующие входные данные. Говоря «связывать», я подразумеваю, что во время обучения сеть узнает, например, что символы, которые образуют значащие слова, с большей вероятностью будут появляться поблизости.

Вернемся к примеру с Йодой. Сеть узнает, что, увидев символы *Y* и *o*, наиболее вероятный символ для генерации – это *d*, потому что последовательность «Yod» уже была видна. Это значительное упрощение динамики обучения РНС, но дает вам общее представление.

Функции потерь

Как и во многих алгоритмах машинного обучения, нейронная сеть учится минимизировать ошибки, которые она совершает при попытке создать «хорошие» результаты из входных данных. Хорошие результаты, которые вы предоставляете во время обучения, вместе с входными данными сообщают сети, насколько это неправильно, когда она затем выполняет прогноз. Величина такой ошибки обычно измеряется с помощью *функции потерь*. Целью алгоритма обучения является оптимизация параметров алгоритма (в случае с нейронной сетью – оптимизация весов), чтобы потери были как можно ниже.

Ранее я упоминал, что РНС, используемая для генерации текста, неявно узнает, насколько вероятны определенные последовательности текста с точки зрения вероятности. В предыдущем примере последовательность «Yoda» могла иметь вероятность 0,7, тогда как последовательность «ode» может иметь вероятность 0,01. Соответствующая функция потерь сравнивает вероятности, рассчитанные нейронной сетью (с ее текущими весами), с фактическими вероятностями во

входном тексте; например, последовательность «Yoda» будет иметь фактическую вероятность около 1 в примере текста, что дает сумму потери (ошибку). Существует несколько различных функций потерь, но та, которая интуитивно выполняет этот тип сравнения, называется *функцией потерь на основе перекрестной энтропии*; мы будем использовать ее в примерах с РНС. Можно рассматривать такую функцию как измерение того, насколько вероятности, рассчитанные нейронной сетью, отличаются от того, какими они должны быть относительно определенного результата. Например, если сеть, изучающая предложение с Йодой, говорит, что вероятность слова «Yoda» равна 0,00000001, это, возможно, приведет к большим потерям: правильная вероятность должна быть высокой, потому что «Yoda» является одной из немногих известных хороших последовательностей во входном тексте.

Функции потерь играют ключевую роль в машинном обучении, потому что они определяют цель алгоритма обучения. Для разных типов задач используются разные функции потерь.

Например, функция потерь на основе перекрестной энтропии полезна для задач классификации, тогда как функция потерь на основе *среднего квадрата ошибки* полезна, когда нейронной сети необходимо прогнозировать реальные значения.

Чтобы рассказать о математических основах функций потерь, вероятно, потребовалась бы целая глава; поскольку в центре внимания этой книги находятся приложения глубокого обучения для поиска, мы не будем вдаваться в подробности. Но я буду приводить подходящую для использования функцию потерь в зависимости от конкретной решаемой проблемы по мере прохождения книги.

Развертывание РНС

Возможно, вы заметили, что единственное эстетическое различие между сетью прямого распространения и рекуррентной нейросетью заключается в стрелках в форме петли в скрытых слоях. Слово «рекуррентный» относится к таким петлям.

Лучший способ увидеть, как РНС работает на практике, – развернуть ее. Представьте себе развертывание РНС в набор конечных связанных копий одной и той же сети. Это полезно на практике при реализации данной сети, а также облегчает понимание того, как РНС естественным образом вписываются в обучение с использованием последовательностей.

В примере с Йодой я сказал, что память из 10 символов помогает вам помнить ранее введенные символы, когда вы видите новые входные данные. РНС обладает способностью отслеживать предыдущие входные данные (с учетом контекста) посредством рекуррентных нейронов или слоев. Если вы позволите рекуррентному слою РНС «вылиться» в набор из 10 копий слоя, то *развернете* сеть на 10 (*см.* рис. 3.8).

Вы подаете предложение «меня зовут Йода» рекуррентной сети, развернутой на 10 шагов.

Давайте сосредоточимся на выделенном узле на рис. 3.8: видно, что он получает входные данные от своего ввода (символ *s*) и предыдущего узла в скрытом (развернутом) слое, который, в свою очередь, получает входные данные от символа *i* и предыдущего узла в скрытом слое; так продолжается до первого ввода. Идея состоит в том, что каждый узел получает информацию о простом вводе (символ последовательности) и, наоборот, из предыдущих входных данных и внутренних состояний сети для таких предыдущих данных.

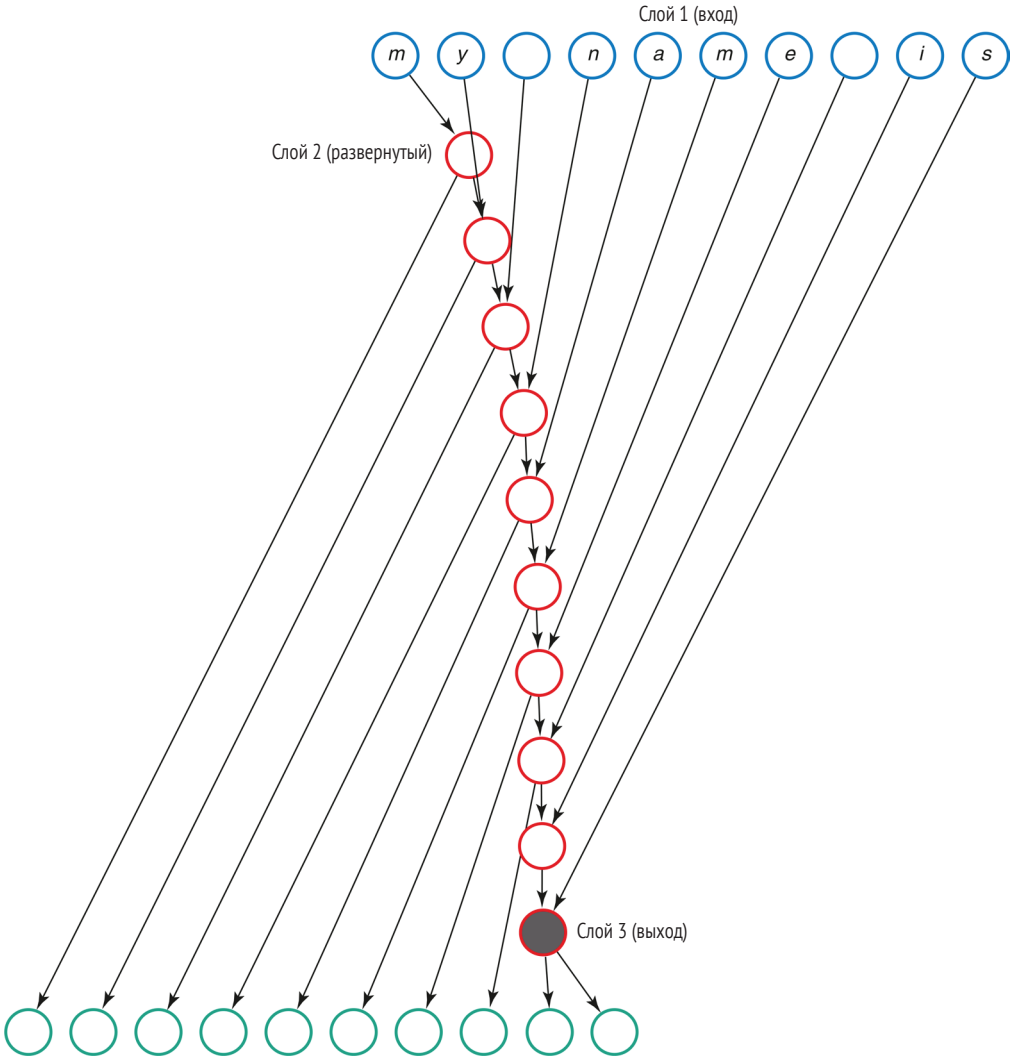


Рис. 3.8 ❖ Развернутая рекуррентная нейронная сеть, читающая фразу «my name is Yoda»

С другой стороны, продвигаясь вперед, видно, что вывод первого символа (*m*) зависит только от ввода и внутреннего состояния (весов) сети; тогда как вывод символа *y* зависит от ввода, текущего состояния *и* предыдущего состояния, как это было в случае с первым символом, *m*.

Таким образом, параметр `unrolls` – это количество шагов, на которое сеть может оглянуться назад во время генерации выхода для текущего входа. На практике при настройке рекуррентных нейросетей вы можете решить, сколько шагов хотите использовать для развертывания сети. Чем больше у вас шагов, тем лучше сети смогут обрабатывать более длинные последовательности, хотя им также потребуется больше данных и больше времени для обучения. Теперь у вас должно сложиться базовое представление о том, как PHC обрабатывает последовательно-

сти входных данных, таких как текст, и отслеживает прошлые последовательности при генерации значений в выходных слоях.

Обратное распространение ошибки во времени: как обучаются рекуррентные нейросети

В главе 2 мы кратко познакомились с методом обратного распространения ошибки, наиболее широко используемым алгоритмом обучения нейронной сети прямого распространения. Рекуррентные нейронные сети можно рассматривать как сети прямого распространения с дополнительным измерением: временем. Эффективность рекуррентных сетей заключается в их способности учиться правильно учитывать информацию из предыдущего ввода, используя алгоритм обучения под названием *обратное распространение ошибки во времени*. По сути, это расширение простого обратного распространения, где количество изучаемых весов намного больше, чем в простых нейронных сетях прямого распространения, из-за циклов в рекуррентном слое, потому что у рекуррентных сетей есть веса, которые контролируют прохождение прошлой информации. Мы только что рассмотрели концепцию развертывания RNC. Обратное распространение ошибки во времени регулирует веса рекуррентных слоев; таким образом, чем больше у вас развертываний, тем больше параметров нужно отрегулировать, чтобы получить хорошие результаты. По сути, обратное распространение ошибки во времени заставляет (рекуррентную) нейронную сеть автоматически изучать не только веса на связях между нейронами, принадлежащими к разным слоям, но и то, как нужно объединять прошлую информацию с текущими входными данными с помощью дополнительных весов.

Причины развертывания рекуррентной сети теперь должны быть понятнее. Это способ ограничить количество рекурсий, которые цикл выполняет в рекуррентном нейроне или слое, поэтому обучение и прогнозирование ограничены и не повторяются бесконечно (что может затруднить вычисление значения в рекуррентном нейроне).

3.3.2. Долгосрочные зависимости

Давайте рассмотрим, как будет выглядеть рекуррентная сеть для генерации запросов. Представим, что у вас есть два похожих запроса, таких как «книги об искусственном интеллекте» и «книги о машинном обучении». (Это простой пример: две последовательности имеют одинаковую длину.) Первое, что нужно сделать, – это выбрать размер скрытых слоев и количество развертываний. В предыдущем разделе вы узнали, что количество развертываний определяет, насколько далеко сеть может оглянуться назад во времени. Для того чтобы это работало должным образом, сеть должна быть достаточно мощной, а это означает, что ей нужно больше нейронов в скрытом слое, чтобы правильно обрабатывать информацию, приходящую из прошлого, по мере того как количество развертываний растет. Количество нейронов в слое определяет максимальную *мощность* сети. Также важно отметить, что если вам нужна сеть со множеством нейронов (и слоев), вам нужно будет предоставить большое количество данных, чтобы сеть работала хорошо с точки зрения верности выходных данных.

Количество развертываний связано с *долгосрочной зависимостью*: это сценарий, при котором слова могут иметь семантические корреляции, даже если они

появляются дальше друг от друга в текстовой последовательности. Например, посмотрите на приведенное ниже предложение, где слова, которые удалены друг от друга, сильно коррелируют:

В 2017 году, несмотря на то что произошло во время финала 2016 года, команда Golden State Warriors снова выиграла чемпионат.

Прочитав эту фразу, вы с легкостью поймете, что слово «чемпионат» относится к «2017 году». Но не очень умный алгоритм может связать слово «чемпионат» с «2016 годом», потому что такую пару также можно сгенерировать. Этот алгоритм не сможет принять во внимание тот факт, что слово «2016» относится к слову «финал» в дополнительном предложении. Это пример долгосрочной зависимости. В зависимости от данных, с которыми вы имеете дело, вам может потребоваться учитывать это, чтобы заставить РНС работать эффективно.

Использование большего количества развертываний помогает смягчить проблемы, связанные с долговременными зависимостями, но в целом вы никогда не узнаете, насколько далеко друг от друга могут находиться два взаимосвязанных слова (или символы, или даже фразы). Чтобы решить эту проблему, исследователи разработали улучшенную архитектуру рекуррентных нейронных сетей под названием *долгая краткосрочная память* (LSTM).

3.3.3. LSTM-сети

До сих пор вы видели, что слой в обычной рекуррентной сети состоит из нескольких нейронов с петлевыми связями. С другой стороны, слой LSTM-сети немного сложнее.

Слои LSTM могут определять:

- какая информация должна пройти через следующее развертывание;
- какую информацию следует использовать для обновления значений внутреннего состояния LSTM;
- какую информацию следует использовать в качестве следующего возможного внутреннего состояния;
- какую информацию вывести.

Что касается ванильных рекуррентных нейросетей (самая основная форма, как показано в предыдущем разделе), в LSTM есть гораздо больше параметров для изучения. Это эквивалентно звукорежиссеру в студии звукозаписи, который настраивает эквалайзер (LSTM), а не поворачивает регулятор громкости (РНС): эквалайзер гораздо сложнее в эксплуатации, но если вы правильно настроите его, то сможете получить звук гораздо лучшего качества. Нейроны LSTM-слоя имеют больше весов, которые настраивают, чтобы они учились, когда запоминать информацию, а когда – забывать. Это делает обучение LSTM-сетей более затратным в вычислительном отношении процессом, нежели обучение РНС.

Легковесная версия LSTM-нейронов, но все же немного более сложная, чем нейроны ванильных РНС, – это управляемый рекуррентный блок[1]. О LSTM-сетях можно узнать намного больше, но ключевой момент здесь заключается в том, что

[1] *См.*: *Кён Хён Чхо и др.* Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. 2014. September 3 (**https://arxiv.org/abs/1406.1078v3**).

они очень хорошо работают с долгосрочными зависимостями, а следовательно, хорошо подходят при использовании генерации запросов.

3.4. LSTM-сети для генерации текста без контроля

В Deeplearning4j можно использовать готовую реализацию LSTM-сетей. Давайте настроим простую конфигурацию для рекуррентной нейронной сети с одним скрытым LSTM-уровнем. Вы создадите рекуррентную сеть, которая может отобрать 50 символов текста. Хотя это не длинная последовательность, этого должно быть достаточно для обработки коротких текстовых запросов (например, запрос «книги об искусственном интеллекте» – это 33 символа).

Параметр unroll в идеале должен быть больше целевого образца (вывода) текста, чтобы можно было обрабатывать более длинные последовательности ввода. Приведенный ниже код будет использоваться для конфигурации рекуррентной сети с 50 нейронами во входном и выходном слоях и 200 нейронами в скрытом (рекуррентном) слое. Развертывание сети будет осуществляться за 10 временных шагов.

Листинг 3.4 ❖ Пример конфигурации LSTM-сети

```
int lstmLayerSize = 200;
int sequenceSize = 50;
int unrollSize = 10;
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .list()
    .layer(0, new LSTM.Builder()
        .nIn(sequenceSize)
        .nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunctions
            .LossFunction.MCXENT)
        .activation(Activation.SOFTMAX)
        .nIn(lstmLayerSize)
        .nOut(sequenceSize).build())
    .backpropType(BackpropType.TruncatedBPTT)
        .tBPTTForwardLength(unrollSize)
            .tBPTTBackwardLength(unrollSize)
    .build();
```

Количество нейронов в скрытом (LSTM) слое

Количество нейронов во входном и выходном слоях

Количество развертываний PHC

Объявляет уровень LSTM с 50 входами (nIn) и 200 выходами (nOut), используя функцию активации tanh

Объявляет выходной слой с 200 входами (nIn) и 50 выходами (nOut), используя функцию активации softmax. Функция потерь здесь также объявлена

Объявляет измерение времени PHC (LSTM) с unrollSize в качестве параметра алгоритма обратного распространения ошибки во времени

Важно отметить ряд деталей, касающихся этой архитектуры:

- вы указываете параметр функции потерь для функции на основе перекрестной энтропии;
- вы используете функцию активации tanh на входных и скрытых слоях;
- вы используете функцию активации softmax в выходном слое.

Использование функции потерь на основе перекрестной энтропии тесно связано с применением функции softmax в выходном слое. Функция softmax в выходном слое преобразует каждый из его входящих сигналов в оценочную вероятность относительно других сигналов, генерируя *распределение вероятностей*, где каждое такое значение находится между 0 и 1, а сумма всех полученных значений равна 1.

В контексте генерации текста на уровне символов у вас будет один нейрон для каждого символа в данных, используемых для обучения сети. Как только функция softmax будет применена к значениям, сгенерированным скрытым слоем LSTM-сети, каждому символу будет назначена вероятность (число от 0 до 1). В примере с Йодой данные состоят из 10 символов, поэтому выходной слой будет содержать 10 нейронов. Функция softmax заставляет выходной слой содержать вероятность для каждого символа:

```
m -> 0.031
y -> 0.001
n -> 0.022
a -> 0.088
e -> 0.077
i -> 0.063
s -> 0.181
Y -> 0.009
o -> 0.120
d -> 0.408
```

Как видите, наиболее вероятный символ происходит от нейрона, связанного с символом *d* (вероятность = 0,408).

Давайте передадим образец текста в эту LSTM-сеть и посмотрим, что она научится генерировать.

Прежде чем создавать текст для ваших запросов, сначала попробуем что-нибудь более простое для понимания. Это поможет вам убедиться, что сеть делает свою работу правильно.

Мы будем использовать текст, написанный на естественном языке: в частности, литературные произведения, взятые из проекта Гутенберга (**www.gutenberg.org**), такие как: «Queen. This is mere madness; And thus a while the fit will work on him». Мы научим рекуррентную сеть, как (пере)писать поэмы и комедии Шекспира (*см.* рис. 3.9)!

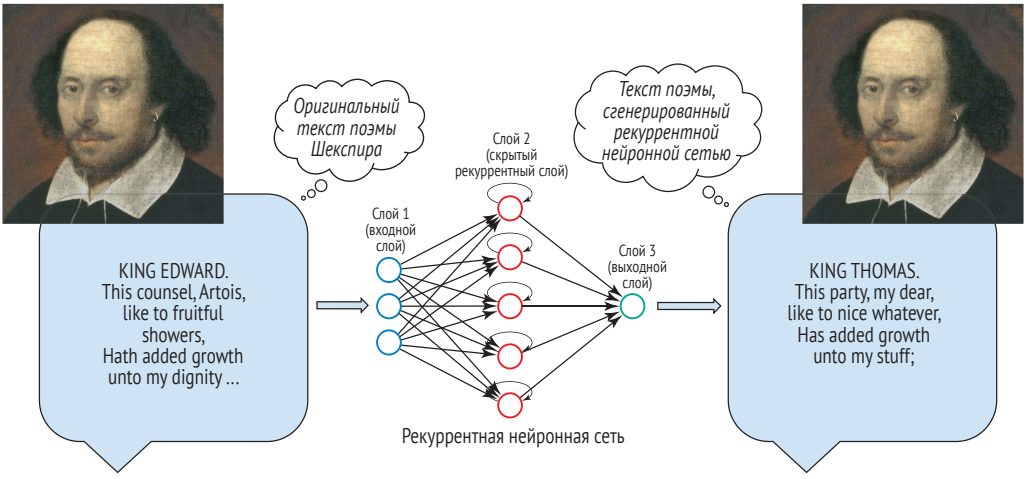


Рис. 3.9 ❖ Генерация шекспировского текста

Это будет ваш первый опыт работы с РНС, поэтому лучше начать с самого простого подхода к ее обучению. Вы будете проводить обучение сети без учителя, подавая текст из произведений Шекспира по одной строке за раз, как показано на рис. 3.10. (Размеры входного и выходного слоев установлены на 10 для удобства чтения.) Просматривая текст произведений Шекспира, вы берете отрывки «*размер развертки* + 1» и подаете их, по одному символу за раз, во входной слой. Ожидаемый результат в выходном слое – следующий символ во входном фрагменте: например, с учетом предложения «work on him» вы увидите, что входные данные

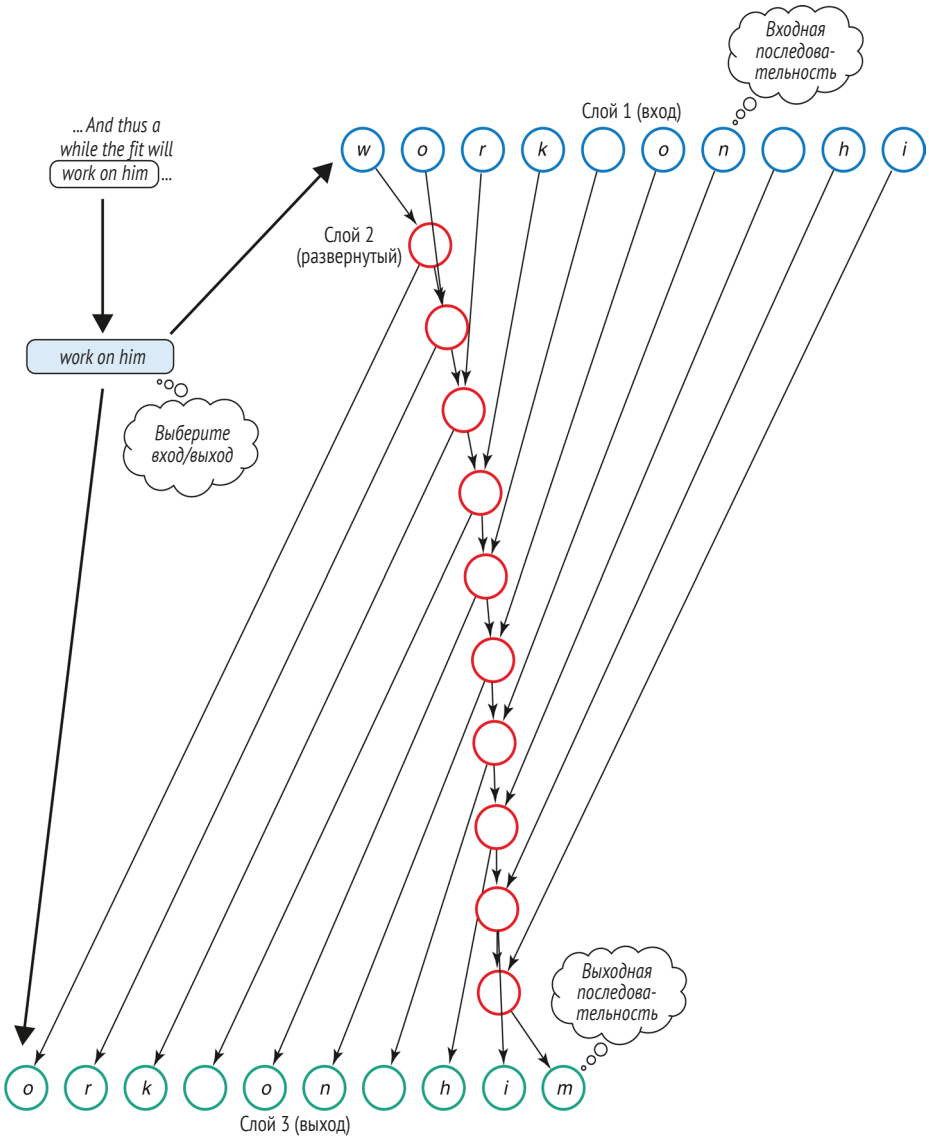


Рис. 3.10 ❖ Передача текста в развернутую рекуррентную сеть с использованием неконтролируемого обучения с последовательностью

получают символы «work on hi», и соответствующие результаты «ork with him». Таким образом, вы обучаете сеть генерировать следующий символ, также оглядываясь на предыдущие 10 символов.

Вы настроили LSTM-сеть ранее; теперь вы будете обучать ее, перебирая последовательности символов из текстов Шекспира. Сначала вы инициализируете сеть, используя ранее определенную конфигурацию:

```
MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();
```

Как уже упоминалось, вы создаете рекуррентную нейронную сеть, которая генерирует текстовые последовательности по одному символу за раз. Поэтому вы будете использовать `DataSetIterator` (API DL4J для перебора наборов данных), который создает последовательности символов: `CharacterIterator` (**http://mng.bz/y1ZJ**). Некоторые детали, касающиеся `CharacterIterator`, можно пропустить. Он инициализируется с помощью:

- исходного файла, содержащего текст для проведения обучения без учителя;
- количества примеров, которые должны быть переданы в сеть, прежде чем она обновит свои веса (так называемый параметр *мини-пакета*);
- длины каждого примера последовательности.

Вот код для перебора символов текста произведения Шекспира:

```
CharacterIterator iter = new CharacterIterator("/path/to/shakespeare.txt",
    miniBatchSize, exampleLength);
```

Теперь у вас есть все кусочки головоломки для обучения сети. Обучение `MultiLayer-Network` осуществляется с помощью метода `fit(Dataset)`:

```
MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();
net.setListeners(new ScoreIterationListener(1));
while (iter.hasNext()) {
    net.fit(iter);
}
```

Можно настроить слушателей на изучение процесса обучения (например, чтобы убедиться, что потери со временем снижаются)

Перебирает содержимое набора данных

Обучает сеть для каждой части набора данных

Вам нужно проверить, что величина потерь, генерируемых сетью во время обучения, со временем неуклонно снижается. Это полезно в качестве проверки работоспособности: в нейронной сети с соответствующими настройками эта цифра будет постоянно уменьшаться. В приведенном ниже журнале показано, что за 10 итераций потери сократились с 4176 до 3490 (со взлетами и падениями в промежутках):

```
Score at iteration 46 is 4176.819462796047
Score at iteration 47 is 3445.1558312409256
Score at iteration 48 is 3930.8510119434372
Score at iteration 49 is 3368.7542747804177
Score at iteration 50 is 3839.2150762596357
Score at iteration 51 is 3212.1088334832025
Score at iteration 52 is 3785.1824493103672
Score at iteration 53 is 3104.690257065846
Score at iteration 54 is 3648.584794826596
```

```
Score at iteration 55 is 3064.9664614373564
Score at iteration 56 is 3490.8566755252486
```

Если построить график для большего количества таких значений (например, 100), то можно увидеть нечто наподобие того, что изображено на рис. 3.11.

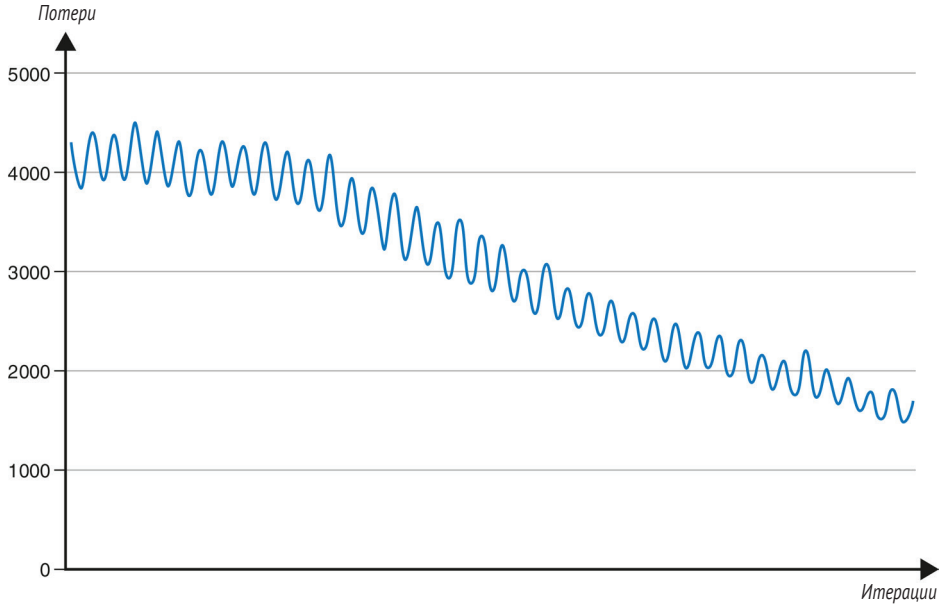


Рис. 3.11 ❖ График потерь

Давайте посмотрим на некоторые последовательности (по 50 символов в каждой), сгенерированные этой рекуррентной сетью после нескольких минут обучения:

- ...o me a fool of s itter thou go A known that fig..
- ..ou hepive beirel true; They truth fllowsus; and..
- ..ot; suck you a lingerity again! That is abys. T...
- ..old told thy denuless fress When now Majester s...

Хотя вы можете признать, что грамматика не так уж плоха и некоторые части могут даже иметь смысл, отчетливо видно, что качество этих последовательностей оставляет желать лучшего. Вы, вероятно, не захотите использовать эту сеть для написания запроса на естественном языке для конечного пользователя, учитывая ее плохие результаты. Полный пример генерации текста произведения Шекспира с помощью похожей LSTM-сети (с одним скрытым рекуррентным слоем) можно найти на странице **http://mng.bz/7ew9**.

Один приятный момент, касающийся рекуррентных нейронных сетей, состоит в том, что мы продемонстрировали, что добавление большего количества скрытых слоев часто повышает верность сгенерированных результатов[1]. Это означает,

[1] *См.*: *Разван Паскану и др.* How to Construct Deep Recurrent Neural Networks. 2014. 24 апр. (**https://arxiv.org/abs/1312.6026**).

что при наличии достаточного количества данных увеличение количества скрытых слоев может заставить более глубокие рекуррентные сети работать лучше. Чтобы увидеть, применимо ли это в данном случае, давайте создадим LSTM-сеть с двумя скрытыми слоями.

Листинг 3.5 ❖ Настройка LSTM-сети с двумя скрытыми слоями

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .list()
        .layer(0, new LSTM.Builder()
        .nIn(sequenceSize)
        .nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(1, new LSTM.Builder()
        .nIn(lstmLayerSize)
        .nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
        .activation(Activation.SOFTMAX)
        .nIn(lstmLayerSize)
        .nOut(sequenceSize).build())
    .backpropType(BackpropType.TruncatedBPTT)
        .tBPTTForwardLength(unrollSize).tBPTTBackwardLength(unrollSize)
    .build();
```

В этой новой конфигурации вы добавляете второй скрытый слой LSTM-сети, идентичный первому

С помощью этой конфигурации вы снова обучаете нейронную сеть, используя тот же набор данных, поэтому код для обучения остается тем же. Обратите внимание, как вы генерируете выходной текст из обученной сети. Поскольку это RNC, вы используете API-интерфейс DL4J `network.rnnTime-Step (INDArray)`, который принимает входной вектор и создает выходной вектор, используя предыдущее состояние рекуррентной сети, а затем обновляет его. Дальнейший вызов `rnnTimeStep` будет использовать это ранее сохраненное внутреннее состояние для получения вывода.

Как обсуждалось ранее, вход в эту сеть представляет собой последовательность символов, каждый из которых представлен способом унитарного кодирования. Шекспировский текст содержит 255 различных символов, поэтому ввод символов будет представлен вектором размером 255, все значения которого равны 0, за исключением того, что имеет значение 1. Каждая позиция соответствует символу, поэтому если вы установите для вектора в определенном положении значение 1, это будет означать, что входной вектор представляет этот конкретный символ. Выход, сгенерированный RNC относительно входа, будут распределением вероятности, потому что вы используете функцию активации softmax в выходном слое. Такое распределение скажет вам, какие символы с большей вероятностью будут сгенерированы в ответ на соответствующий входной символ (и предыдущие выходы, согласно информации, хранящейся на уровне RNC). Распределение вероятностей похоже на математическую функцию, которая может выводить все возможные символы, но при этом вероятность выдачи одних выше, чем других. Например, в векторе, сгенерированном сетью, обученной на предложении «меня зовут Йода», символ *у*, скорее всего, будет сгенерирован таким распределением, чем символ *n*, если предыдущий входной символ – это *m* (и, следовательно, последовательность *my* более вероятна, чем *mn*).

Подобное распределение вероятностей используется для генерации выходного символа.

Сначала вы преобразуете последовательность символов инициализации (например, пользовательский запрос) в последовательность векторов символов.

Листинг 3.6 ❖ Унитарное кодирование последовательности символов

```
INDArray input = Nd4j.zeros(sequenceSize,
    initialization.length());
char[] init = initialization.toCharArray();
for (int i = 0; i < init.length; i++) {

  int idx = characterIterator.convertCharacterToIndex(
    init[i]);

  input.putScalar(new int[] {idx, i}, 1.0f);
}
```

Создает входной вектор необходимого размера

Перебирает каждый символ во входной последовательности

Получает индекс каждого символа

Создает вектор с унитарным кодированием для каждого символа со значением в позиции «index», установленным в 1

Для каждого символьного вектора вы генерируете выходной вектор символьных вероятностей и конвертируете его в реальный символ путем выборки (извлечения вероятного результата) из сгенерированного распределения:

```
INDArray output = network.rnnTimeStep(input);

int sampledCharacterIdx = sampleFromDistribution(
    output);

char c = characterIterator.convertIndexToCharacter(
    sampledCharacterIdx);
```

Прогнозирует распределение вероятностей для заданного входного символа (вектора)

Отбирает вероятный символ из сгенерированного распределения

Преобразует индекс отобранного символа в фактический символ

В шекспировском тексте вы инициализируете входную последовательность случайным символом, а затем рекуррентная сеть генерирует последующие символы. После этого вы видите, что наличие двух скрытых слоев LSTM-сети дает лучшие результаты:

- … ou for Sir Cathar Will I have in Lewfork what lies …
- … , like end. OTHELLO. I speak on, come go'ds, and …
- … , we have berowire to my years sword; And more …
- … Oh! nor he did he see our strengh …
- … WARDEER. This graver lord. CAMILL. Would I am be …
- … WALD. Husky so shall we have said? MACBETH. She h …

Как и ожидалось, сгенерированный текст выглядит более точным, чем текст, созданный с помощью первой LSTM-сети, в которой был один скрытый слой. В этот момент вам, наверное, интересно, что произошло бы, если бы вы добавили еще один скрытый слой. Будут ли результаты еще лучше? Сколько скрытых слоев должна иметь идеальная сеть для генерации текста в этом случае? Вы можете легко ответить на первый вопрос, попробовав пример с сетью, которая имеет три скрытых слоя. Труднее и, может быть, невозможно дать точный ответ на второй вопрос. Поиск наиболее подходящей архитектуры и сетевых настроек – сложный

процесс; вы узнаете больше подробностей о рекуррентных сетях в конце этой главы, когда мы поговорим об их использовании в производстве.

Используя ту же конфигурацию, что и ранее, но с дополнительным (третьим) скрытым слоем, примеры выглядят так:

- … J3K. Why, the saunt thou his died There is hast …
- … RICHERS. Ha, she will travel, Kate. Make you about …
- … or beyond There the own smag; know it is that l …
- … or him stepping I saw, above a world's best fly …

Учитывая параметры, заданные вами в нейронной сети (размер слоя, размер последовательности, размер развертывания и т. д.), добавление четвертого скрытого слоя не улучшит результаты. На самом деле они были бы немного хуже (например, «… CHOPY. Wencome. My lord 'tM times our mabultion …»): добавление большего количества слоев означает увеличение мощности, но также сложности сети. Обучение требует все больше и больше времени и данных; иногда невозможно получить более качественные результаты, просто добавив еще один скрытый слой. В главе 9 мы обсудим несколько методов, чтобы решить вопрос баланса между потребностями вычислительных ресурсов (процессор, данные, время) и верностью результатов на практике.

3.4.1. Неуправляемое расширение запроса

Теперь, когда вы увидели, как работает РНС на базе долгой краткосрочной памяти в случае литературного текста, давайте соберем сеть для генерации альтернативных запросов. В предыдущем примере вы передали текст в рекуррентную сеть (обучение без учителя), потому что это был самый простой способ понять и увидеть, как работает такая сеть. Теперь давайте рассмотрим использование этого же подхода для расширения запросов. Можно опробовать его на общедоступных ресурсах, таких как набор данных web09-bst (**http://boston.lti.cs.cmu.edu/Data/web08-bst/planning.html**), где содержатся запросы из актуальных информационно-поисковых систем. Вы ожидаете, что рекуррентная сеть научится генерировать запросы, похожие на те, что находятся в журнале поиска, по одному на строку.

Следовательно, задача подготовки данных состоит в том, чтобы извлечь все запросы из журнала поиска и записать их в один файл, один за другим.

Вот фрагмент из журнала запросов:

```
query:{"artificial intelligence"}, results:{
    size=10, ids:["doc1","doc5", ...]}
query:{"books about AI"}, results:{
    size=1, ids:["doc5"]}
query:{"artificial intelligence hype"}, results:{
    size=3, ids:["doc1","doc8", ...]}
query:{"covfefe"}, results:{size=100, ids:["doc113","doc588", ...]}
query:{"latest trends"}, results:{size=15, ids:["doc113","doc23", ...]}
...
```

Часть запроса состоит из «искусственного интеллекта»

Часть запроса состоит из «книг об ИИ»

Используя только часть запроса каждой строки, вы получаете текстовый файл наподобие этого:

```
artificial intelligence
books about AI
artificial intelligence hype
```

```
covfefe
latest trends
...
```

Получив его, вы можете передать его в LSTM-сеть, как описано в предыдущем разделе. Количество скрытых слоев зависит от различных ограничений; два слоя – обычно хорошее стартовое значение. Как показано на рис. 3.1, вы создадите свой алгоритм расширения запроса в анализатор запросов, чтобы пользователь не подвергался генерированию альтернативных запросов. Для этого примера вы расширите `Query-Parser` от Lucene, в обязанности которого входит создание запроса из строки (в данном случае пользовательского запроса).

Листинг 3.7 ❖ Анализатор запросов Lucene для расширения альтернативных запросов

```
public class AltQueriesQueryParser
    extends QueryParser {

  private final MultiLayerNetwork rnn;
  private CharacterIterator characterIterator;

  public AltQueriesQueryParser (String field, Analyzer a,
         MultiLayerNetwork rnn, CharacterIterator characterIterator) {
      super(field, a);
      this.rnn = rnn;
      this.characterIterator = characterIterator;
    }

  @Override
  public Query parse(String query) throws ParseException {
    BooleanQuery.Builder builder =
        new BooleanQuery.Builder();
    builder.add(new BooleanClause(super.parse(
        query), BooleanClause.Occur.MUST));

    String[] samples = sampleFromNetwork(query);

    for (String sample : samples) {
      builder.add(new BooleanClause(super.parse(
        sample), BooleanClause.Occur.SHOULD));
    }

    return builder.build();
  }

  private String[] sampleFromNetwork(String query) {
    // место, где происходит "волшебство" ...
  }
}
```

Анализатор запросов транслирует строку в разобранный запрос для запуска с индексом Lucene

RNC, используемая пользовательским анализатором запросов для генерации альтернативных запросов

Инициализирует булев запрос Lucene, содержащий исходный введенный пользователем запрос и дополнительные запросы, созданные RNC

Добавляет обязательный оператор для введенного пользователем запроса (необходимо отобразить результаты этого запроса)

Позволяет RNC генерировать несколько выборок, чтобы использовать их в качестве дополнительных запросов

Анализирует текст, сгенерированный RNC, и включает его в качестве необязательного оператора

Создает и возвращает окончательный запрос в виде комбинации введенного пользователем запроса и запросов, созданных RNC

Этот метод выполняет кодирование запросов, прогнозирование с помощью RNC и декодирование выходных данных в новый запрос, как в примере с текстом Шекспира

Вы инициализируете анализатор запросов с помощью RNC и используете его для создания ряда альтернативных запросов, которые добавляются в качестве необязательных операторов, присоединенных к исходному запросу. Все волшебство содержится в той части кода, которая генерирует строки нового запроса из исходного.

РНС получает введенный пользователем запрос в качестве входных данных и создает новый запрос в качестве результатов. Помните, что нейронные сети «общаются» посредством векторов, поэтому вам необходимо преобразовать текстовый запрос в вектор. Вы выполняете унитарное кодирование символов введенного пользователем запроса. Как только входной текст преобразуется в вектор, вы отбираете выходной запрос по одному символу за раз. Вернемся к примеру с текстом из произведения Шекспира. Вы сделали следующее:

- закодировали введенный пользователем запрос в серию символьных векторов с использованием унитарного кодирования;
- отправили эту последовательность в сеть;
- получили первый выходной символьный вектор, преобразовали его в символ и передали этот сгенерированный символ обратно в сеть;
- итерировали предыдущий шаг, пока не был найден конечный символ (например, символ возврата каретки в данном случае).

На практике это означает, что если вы передаете РНС введенный пользователем запрос, который похож на распространенный терм, сеть, вероятно, «дополнит» запрос, добавив соответствующие термы. Если вместо этого вы передаете РНС запрос, который выглядит как законченный запрос, сеть, вероятно, сгенерирует запрос, который вы можете найти рядом с введенным пользователем запросом в журнале поиска. После этого вы теперь можете генерировать альтернативные запросы, используя приведенные ниже настройки.

Листинг 3.8 ❖ Испытываем AltQueriesQueryParser, используя LSTM-сеть с двумя скрытыми слоями

```
int lstmLayerSize = 150;
int miniBatchSize = 10;
int exampleLength = 50;
int tbpttLength = 40;
int epochs = 1;
int noOfHiddenLayers = 2;
double learningRate = 0.1

String file = getClass().getResource("/queries.txt")
    .getFile();
CharacterIterator iter = new CharacterIterator(file,
    miniBatchSize, exampleLength);

MultiLayerNetwork net = NeuralNetworksUtils
    .trainLSTM(
  lstmLayerSize, tbpttLength, epochs, noOfHiddenLayers, iter, learningRate,
  WeightInit.XAVIER,
  Updater.RMSPROP,
  Activation.TANH,
```

Количество примеров, которые будут помещены в мини-пакет

Длина каждой входной последовательности, чтобы научить РНС генерировать новые

Размер слоев LSTM-сети

Размер развертывания (как параметр обратного распространения ошибки во времени)

Сколько раз РНС должна перебирать одни и те же данные

Количество скрытых LSTM-слоев в РНС

Скорость обучения градиентного спуска

Исходный файл, содержащий запросы

Создает итератор для текстовых символов файла, содержащего запросы

Алгоритм, используемый для инициализации весов сети

Алгоритм обновления, используемый для обновления параметров при выполнении градиентного спуска

Функция активации, которая будет использоваться в скрытых слоях

```
  new ScoreIterationListener(10));

Analyzer analyzer = new EnglishAnalyzer(null);
AltQueriesQueryParser altQueriesQueryParser =
    new AltQueriesQueryParser ("text",
        analyzer, net, iter);

String[] queries = new String[] {"latest trends",
    "covfefe", "concerts", "music events"};

for (String query : queries) {
  System.out.println(altQueriesQueryParser
    .parse(query));
}
```

Устанавливает слушателя счет-итераций, который выводит значение потерь каждые 10 итераций (обратного распространения во времени)

Анализатор, используемый для определения термов в тексте запроса

Инстанциирует AltQueriesQueryParser

Создает несколько примеров запросов

Выводит альтернативные запросы, сгенерированные пользовательским парсером

Вот что будет содержаться в стандартном выводе:

```
latest trends -> (latest trends) about AI,
    (latest trends) about artificial intelligence

covfefe -> books about coffee

concerts -> gigs in santa monica
music events -> concerts in California
```

Запрос «latest trends» расширен в более конкретный запрос о тенденциях в области ИИ; это повышает результаты, связанные с ИИ

Второй запрос выглядит странно, но не с точки зрения РНС: символы, составляющие запросы «covfefe» и «coffee», практически идентичны и находятся в одинаковых позициях

Альтернативный запрос для «music events» похож, но более конкретен

Обратите внимание, что термы используются совместно между входными и выходными запросами.

Первые сгенерированные альтернативные запросы похожи на более конкретные версии оригинала, что может не соответствовать желанию пользователя. Видно, что запрос «latest trends» заключен в скобки: РНС генерирует запросы «about AI» и «about artificial intelligence», чтобы завершить предложение. Если вы зададите общий вопрос о «последних тенденциях», синтаксический анализатор запросов будет вести себя осторожно при создании более специфичных версий исходного запроса, если нет другого контекста (в этом примере запрос «latest trends» слишком обобщенный). Если вам не нужны альтернативные запросы, как те, что указаны для первого запроса, можно воспользоваться одним трюком, чтобы намекнуть РНС, что она должна попытаться сгенерировать совершенно новый запрос. Данные, которые вы даете рекуррентной сети, разбиваются на последовательности, по одной на строку, ограниченные возвратом каретки, и вот в чем состоит этот трюк: добавьте символ возврата каретки в конце введенного пользователем запроса. РНС используется для наблюдения за последовательностями вида `wordA wordB wordC CR` (или, точнее, символьных потоков, где часто есть пробел), где `CR` – это возврат каретки. Неявно символ `CR` сообщает сети, что последовательность текста перед `CR` завершена и начинается новая последовательность. Если вы возьмете введенный пользователем запрос «latest trends» и разрешите анализатору запросов добавить в конце этого запроса `CR`, рекуррентная сеть попытается сгенерировать новую последовательность, начиная с символа возврата каретки.

Это значительно повышает вероятность того, что РНС будет генерировать текст, похожий на новый запрос, а не на более конкретную версию исходного запроса.

3.5. От неконтролируемой до контролируемой генерации текста

Подход, используемый для создания альтернативных запросов, который вы только что видели, прекрасен, но нам нужно нечто получше; вы сосредоточены на предоставлении инструмента, который меняет жизнь ваших конечных пользователей, и хотите убедиться, что поисковая система работает лучше прежнего, или все эти усилия будут бесполезны.

В случае использования расширения запроса ключевая роль определяется способом обучения РНС. Вы видели, как рекуррентная сеть выполняет неконтролируемое обучение из текстового файла, содержащего множество пользовательских запросов, не связанных напрямую. В разделе 3.1.2 также упоминаются более сложные альтернативы, когда создаются примеры, имеющие требуемый альтернативный запрос относительно определенного входного запроса.

В этом разделе я кратко расскажу о контролируемой генерации текста для поиска (используя журналы поиска, например) с помощью двух разных алгоритмов.

3.5.1. Создание моделей sequence-to-sequence

Вы познакомились с LSTM-сетями и узнали о том, насколько хорошо они справляются с обработкой последовательностей. Выполнение контролируемого обучения при построении альтернативных запросов требует предоставления желаемой целевой последовательности, которая будет сгенерирована относительно входной последовательности. В разделе 3.1.2, где мы обсуждали подготовку данных, вы увидели, что можно получить обучающие примеры, выводя их из журналов поиска.

Поэтому если у вас есть пары типа «последние исследования в области искусственного интеллекта» → «последние публикации в области искусственного интеллекта», можно использовать их в архитектуре РНС, как показано на рис. 3.12.

С такими парами ввода/вывода РНС (или LSTM-сети) гораздо труднее обучаться. В предыдущем подходе с использованием неконтролируемого обучения сеть училась генерировать следующий символ в последовательности, чтобы научить РНС воспроизводить входную последовательность.

При использовании контролируемого обучения вы пытаетесь научить нейронную сеть генерировать последовательность выходных символов, которая может полностью отличаться от входных символов. Рассмотрим один пример. Если у вас есть входная последовательность «latest resea», легко догадаться, что следующим символом будет *r*. Вывод РНС, который будет использоваться для обучения, выглядит на один символ вперед во времени:

```
l -> a
la -> at
lat -> ate
late -> ates
lates -> atest
```

```
latest -> atest
latest  -> atest r
latest r -> atest re
latest re -> atest res
latest res -> atest rese
latest rese -> atest resea
latest resea -> atest resear
```

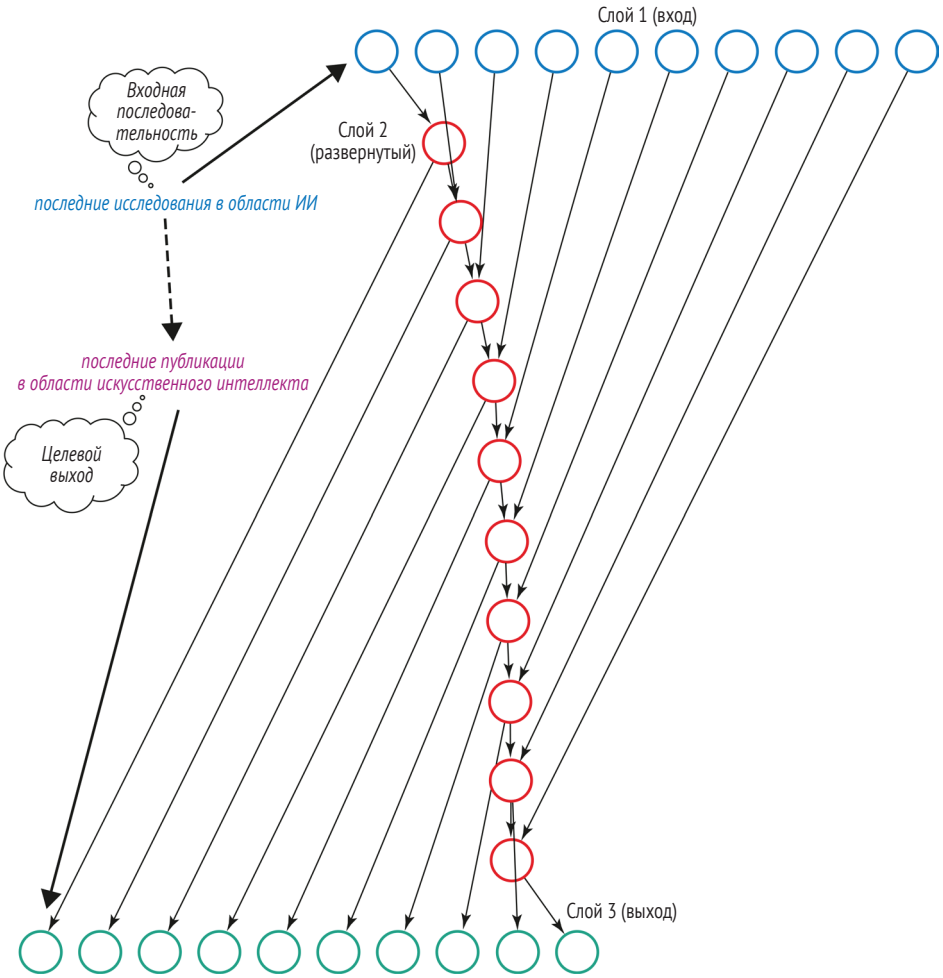


Рис. 3.12 ❖ Контролируемое обучение при помощи LSTM с применением последовательности

С другой стороны, если вы используете часть предложения «recent pub» в качестве целевого вывода, должно получиться что-то наподобие этого:

```
l -> r
la -> re
lat -> rec
late -> rece
```

```
lates -> recen
latest -> recent
latest  -> recent
latest r -> recent p
latest re -> recent pu
latest res -> recent pub
latest rese -> recent publ
latest resea -> recent public
```

Эта задача явно намного сложнее, поэтому сейчас я познакомлю вас с интересной архитектурой, именуемой моделями *sequence-to-sequence*. Эта архитектура использует две LSTM-сети:

- *кодировщик* принимает входную последовательность в качестве последовательности векторов слов (не символов). Он генерирует выходной вектор, называемый *вектором мысли*, который соответствует последнему скрытому состоянию LSTM-сети, вместо того чтобы генерировать распределение вероятностей, как в предыдущей модели;
- *декодер* принимает мысленный вектор в качестве входа и генерирует выходную последовательность, представляющую распределение вероятности, которое будет использоваться для выборки выходной последовательности.

Эта архитектура также носит название *seq2seq* (*см.* рис. 3.13). Мы рассмотрим ее более подробно в главе 7, поскольку она тоже используется для выполнения машинного перевода (преобразуя последовательность, написанную на определенном языке, в соответствующую последовательность на другом целевом языке). Seq2seq также часто применяется для построения диалоговых моделей для чатботов. В контексте поиска интересна концепция вектора мысли: векторизованное представление намерений пользователя. В этой области проводится много исследований[1]. Хотя это и называется вектором мысли, то, что изучает PHC, основано на заданных входах и выходах. В этом случае если вход – это запрос, а выход – еще один запрос, вектор мысли можно рассматривать как вектор, который может сопоставить входной запрос с выходным. Если выходной запрос является релевантным по отношению к входному, вектор мысли кодирует информацию о том, как из этого входного запроса можно сгенерировать соответствующий альтернативный запрос. Это распределенное представление намерения пользователя.

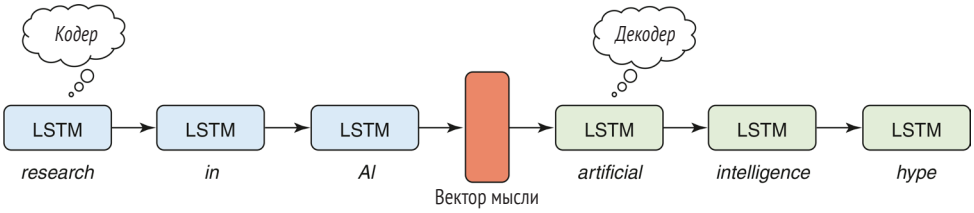


Рис. 3.13 ❖ Модель seq2seq для запросов

[1] *См.*, например: *Райан Кирос и др.* Skip-Thought Vectors. 2015. June 22 (**https://arxiv.org/pdf/1506.06726v1.pdf**); *Шуай Тан и др.* Trimming and Improving Skip-thought Vectors. 2017. June 9 (**https://arxiv.org/abs/1706.03148**); *Йошуа Бенджио*. The Consciousness Prior. 2017. September 25 (**https://arxiv.org/abs/1709.08568**).

Поскольку в главе 7 мы более подробно рассмотрим модели sequence-to-sequence, сейчас мы будем использовать ранее обученную модель seq2seq, в которой связанные входные и желаемые выходные запросы были извлечены из журнала поиска на основе двух показателей:

- как близко во времени они выполнены, как видно из журнала поиска;
- используют ли они совместно хотя бы один результат поиска.

В DL4J вы загружаете ранее созданную модель из файловой системы и передаете ее ранее определенному анализатору `AltQueriesQueryParser`:

```
MultiLayerNetwork net = ModelSerializer
    .restoreMultiLayerNetwork(
    "/path/to/seq2seq.zip");
AltQueriesQueryParser altQueriesQueryParser = new
    AltQueriesQueryParser ("text", new
    EnglishAnalyzer(null), net, null);
```

Восстанавливает ранее сохраненную модель нейронной сети из файла

Создает AltQueriesQueryParser с использованием нейронной сети, реализующей модель seq2seq. Обратите внимание, что вам больше не нужен CharacterIterator

Чтобы использовать модель seq2seq, нужно изменить способ генерации последовательности. При неконтролируемом подходе вы выбирали символы из выходных распределений вероятностей. В этом случае вы будете генерировать последовательность из LSTM-сети-декодера на уровне слов. Вот результаты, полученные `Alt-QueryParser` с использованием модели seq2seq:

Этот результат на первый взгляд может показаться странным, но на самом деле в Чикаго есть фонд Музея современного искусства

```
museum of contemporary art chicago -> foundation
```

```
joshua music festival -> houston monmouth
```

Входной запрос о музыкальном событии генерирует запрос, который содержит город и название другого события (хотя Монмутский музыкальный фестиваль проходит в Орегоне)

```
mattel toys -> mexican yellow shoes
```

Запрос о детских игрушках генерирует запрос о мексиканских желтых туфлях. Если сейчас Рождество, то это хороший результат (подарок для детей и для… кого-то, кому могут понравиться желтые туфли)!

3.6. Соображения относительно продукционных систем

Обучение рекуррентны нейронных сетей было утомительным, а в случае с LSTM-сетями было еще хуже. В настоящее время у нас есть фреймворки, такие как DL4J, которые могут работать на центральных или графических процессорах или даже распределенно (например, через Apache Spark). Другие фреймворки, такие как TensorFlow, имеют выделенное аппаратное обеспечение (тензорные процессоры!) и т. д. Однако настройка РНС, чтобы она хорошо работала, – это не пустяки. Возможно, вам придется обучить несколько разных моделей, чтобы создать ту, которая лучше всего работает с вашими данными. Кстати, существуют не только теоретические ограничения по настройке LSTM-сетей. Данные, которые вы используете для обучения, также определяют, что они могут делать во время тестирования: например, при использовании их в невидимых запросах.

На практике потребовалось несколько часов проб и ошибок, чтобы найти правильные настройки для различных параметров при неконтролируемом подходе. Этот процесс займет меньше времени, когда вы станете лучше разбираться в динамике LSTM-сетей (и нейронных сетей в целом). Например, в текстах Шекспира, с которыми мы работали, содержатся последовательности, которые намного длиннее запросов. Запросы короткие – в среднем от 10 до 50 символов, тогда как строки из пьесы *Макбет* могут содержать 300 символов. Таким образом, параметр длины для примера с текстом Шекспира (200) длиннее, чем тот, что используется для обучения генерации запросов (50).

Также обратите внимание на скрытые структуры в тексте. Текст из шекспировских комедий обычно имеет следующий шаблон: CHARACTERNAME: SOMETEXT PUNCTUATION CR, тогда как запросы – это просто последовательности слов, за которыми следует возврат каретки. Запросы могут содержать как формальные, так и неформальные предложения со словами типа «myspaceeee», которые могут сбить с толку PHC. Таким образом, в то время как шекспировскому тексту нужен был только один скрытый слой, чтобы получить хорошие результаты, LSTM-сети требовалось, по крайней мере, два скрытых слоя, чтобы они работали эффективно.

Решение о том, следует ли проводить неконтролируемое обучение LSTM-сети на символах или использовать модель seq2seq, зависит в первую очередь от имеющихся у вас данных.

Если вы не можете генерировать хорошие обучающие примеры (где выходной запрос является соответствующей альтернативой входному запросу), вероятно, вам следует придерживаться неконтролируемого подхода. Эта архитектура также более легкая, и обучение, вероятно, займет меньше времени.

Ключевой момент, который необходимо учитывать: во время обучения следует отслеживать значения потерь, чтобы убедиться, что они неуклонно снижаются. Вы видели график потерь, сгенерированный путем построения значений, выводимых `ScoreIteredListener` во время обучения неконтролируемой LSTM-сети. Это полезно делать, чтобы убедиться, что обучение идет хорошо. Если потери начинают увеличиваться или перестают уменьшаться при значении, далеком от нуля, вам, вероятно, необходимо настроить параметры сети.

Наиболее важным параметром является скорость обучения. Это значение (обычно между 0 и 1) определяет скорость, с которой алгоритм градиентного спуска идет вниз по направлению к точкам, где ошибка низкая. Если скорость обучения слишком высока (ближе к 1: например, 0,9), это приведет к тому, что потери начнут расходиться (увеличиваясь до бесконечности). Если скорость обучения, наоборот, слишком низкая (ближе к 0: например, 0,0000001), градиентный спуск может занять слишком много времени, чтобы достичь точки с низкой ошибкой.

Резюме

- Нейронные сети могут научиться генерировать текст даже в виде естественного языка. Это полезно для тихого создания запросов, которые выполняются вместе с введенными пользователем запросами, чтобы обеспечить более подходящие результаты поиска.
- Рекуррентные нейронные сети полезны для генерации текста, потому что они способны обрабатывать даже длинные последовательности текста.

- LSTM-сети являются разновидностью рекуррентных сетей, которые могут работать с долговременными зависимостями. Они работают лучше, чем простые РНС, когда имеют дело с текстом на естественном языке, где связанные понятия или слова могут находиться на значительном расстоянии друг от друга в предложении.
- Предоставление более глубоких слоев в нейронных сетях может помочь в тех случаях, когда сети требуется больше вычислительной мощности для обработки больших наборов данных и/или более сложных шаблонов.
- Иногда полезно внимательно посмотреть, как нейронная сеть генерирует свои результаты. Небольшие корректировки (например, трюк с `CR`) могут повлиять на качество результатов.
- Модели seq2seq и векторы мысли являются мощными инструментами для обучения генерации текстовых последовательностей контролируемым образом.

Глава 4

Более чувствительные поисковые подсказки

О чем идет речь в этой главе:

- общепринятые подходы к составлению поисковых подсказок;
- нейронные языковые модели на уровне символов;
- параметры настройки в нейронных сетях.

Мы обсудили основы нейронных сетей и рассмотрели создание как неглубоких, так и глубоких архитектур этих сетей. В практическом плане вы теперь знаете, как интегрировать нейронные сети в поисковую систему, чтобы улучшить ее работу с помощью двух ключевых функций: расширения синонимов и генерации альтернативных запросов. Обе эти функции работают в поисковой системе, чтобы сделать ее умнее и заставить возвращать пользователю более подходящие результаты. Но можете ли вы сделать что-либо, чтобы улучшить формулировку самого запроса? В частности, можно ли сделать что-либо, чтобы помочь пользователю лучше писать запросы – запросы, которые дают результаты, наиболее близкие к тому, что ищет пользователь?

Ответ – конечно же, да. Вы, несомненно, привыкли к поисковой системе, которая предлагает вам подсказки при вводе запроса. Эта функция автозаполнения предназначена для ускорения процесса ввода запроса, предлагая слова или предложения, которые могут сформировать значимый запрос. Например, если пользователь начинает вводить «boo», функция автозаполнения может предоставить остальную часть слова, которое пользователь, вероятно, будет писать: например, «book» или полное предложение, начинающееся с «boo», например «books about deep learning» (книги о глубоком обучении). Помощь пользователям в составлении запросов может ускорить процесс и помочь пользователям избежать опечаток и похожих ошибок. Но эта функциональность также дает поисковой системе возможность предоставлять подсказки, чтобы помочь пользователю составить более подходящий запрос. Эти подсказки – слова или предложения, имеющие смысл в контексте конкретного запроса, который пишет пользователь. Слова «book» и «boomerang» начинаются с «boo», поэтому если пользователь начинает вводить «boo», поисковая система может предложить выбрать «book» или «bomerang» для завершения запроса. Но если пользователь вводит запрос «big parks where I can play boo» (большие парки, где я могу играть в бу), ясно, что слово «boomerang» будет иметь больше смысла, нежели слово «book».

Генерируя эти подсказки, автозаполнение также влияет на эффективность поисковой системы. Представьте себе, что вместо «большие парки, где я могу играть в бумеранг» поисковая система дает подсказку «большие парки, где я могу играть в книги». Безусловно, это даст меньше релевантных результатов поиска.

Подсказки также дают поисковой системе шанс отдавать предпочтение определенным запросам (а следовательно, соответствующим документам), по сравнению с другими. Это может быть полезно, например, для маркетинга. Если владелец поисковой системы на сайте интернет-магазина хочет продавать книги больше, чем бумеранги, то там, возможно, будет подсказка «большие парки, где я могу играть в книги» вместо «большие парки, где я могу играть в бумеранг». Если вы знаете, что пользователи ищут чаще всего, то можете чаще давать подсказки, связанные с этими повторяющимися запросами.

Автозаполнение является обычной функцией в поисковых системах, поэтому для ее создания уже существует множество алгоритмов. Чем нейронные сети могут помочь вам в этом? Отвечаем одним словом: чувствительностью. *Чувствительная* подсказка – это подсказка, которая точно интерпретирует то, что ищет пользователь, и перефразирует это таким образом, с помощью которого, скорее всего, можно будет получить релевантные результаты. Эта глава будет основываться на том, что вы узнали о нейронных сетях, чтобы заставить их генерировать более чувствительные подсказки.

4.1. Генерация поисковых подсказок

Из главы 3 вы знаете, что глубокие нейронные сети могут научиться генерировать текст, который выглядит так, как будто он был написан человеком. Вы убедились в этом на практике, когда генерировали альтернативные запросы. Теперь вы увидите, как использовать и расширять такие нейронные сети, чтобы они могли превзойти современные наиболее распространенные алгоритмы автозаполнения, создавая более качественные и более чувствительные поисковые подсказки.

4.1.1. Подсказки при составлении запросов

В главе 2 мы обсуждали, как помочь пользователям поисковой системы искать текст песни в распространенном случае, когда пользователь не может точно вспомнить название песни. В этом контексте мы познакомились с техникой расширения синонимов, чтобы позволить пользователям делать, возможно, неполный или неправильный запрос (например, «music is my aircraft»), который мы исправили с помощью расширения синонимов «под капотом» («music is my aeroplane»), используя алгоритм word2vec. Расширение синонимов – полезный метод, но, возможно, можно было бы сделать что-нибудь попроще, чтобы помочь пользователю вспомнить, что припев песни – «music is my *aeroplane*», а не «music is my aeroplane», подсказывая правильные слова, пока пользователь вводит запрос. Можно не дать пользователю выполнить неоптимальный запрос в том смысле, что он уже знает, что «aircraft» – это не то слово.

Наличие хороших алгоритмов автозаполнения дает два преимущества:

- меньше запросов с небольшим или нулевым результатом (влияет на полноту);
- меньше запросов с низкой релевантностью (влияет на точность).

Если алгоритм *подсказки слов* работает хорошо, он не будет предлагать несуществующие слова или термы, которые никогда не встречались в индексированных данных. Это означает, что маловероятно, что запрос с использованием термов, предложенных таким алгоритмом, не даст результатов. Давайте рассмотрим пример запроса «music is my aircraft». Если расширение синонимов у вас не активировано, возможно, что такой песни, которая содержит все эти термы, нет; следовательно, самые подходящие результаты будут содержать слова «music» и «my» или «my» и «aircraft», с низкой релевантностью к информационной потребности пользователя (и, следовательно, с низкой *оценкой*). В идеале, когда пользователь вводит запрос «music is my», алгоритм подсказки предложит подсказку «aeroplane», потому что это предложение, которое поисковая система уже видела (проиндексировала).

Мы только что затронули важный вопрос, который играет ключевую роль в создании эффективных подсказок: откуда поступают подсказки? Чаще всего они идут из:

- статических (составленных вручную) словарей слов или предложений, которые будут использоваться для подсказок;
- хронологии ранее введенных запросов (например, взятых из журнала запросов);
- индексированных документов, взятых из различных частей документов (заголовок, основное содержание, авторы и т. д.).

В оставшейся части данной главы мы рассмотрим получение подсказок из этих источников, используя общепринятые методы из области поиска информации и обработки естественного языка (NLP). Вы также увидите их в сравнении с алгоритмами подсказки, основанными на языковых моделях нейронных сетей, старом NLP-методе, реализованном посредством нейронных сетей, с точки зрения особенностей и верности результатов.

4.1.2. Подсказчики на базе словаря

В прежние времена, когда поисковым системам требовалось множество вручную созданных алгоритмов, был распространен подход, заключавшийся в создании словаря слов, который можно было бы использовать, чтобы помочь пользователям вводить запросы. Подобные словари обычно содержали только важные слова, такие как основные понятия, тесно связанные с данной конкретной областью. Например, поисковая система магазина, где продавались музыкальные инструменты, могла использовать словарь, содержащий такие термы, как «гитара», «бас-гитара», «ударные» и «пианино». Было бы очень трудно заполнить словарь всеми релевантными словами путем компилирования его вручную. Вместо этого можно сделать так, чтобы такие словари создавались самостоятельно (например, с помощью скрипта), просматривая журналы запросов, получая введенные пользователем запросы и извлекая список из 1000 (например) наиболее часто используемых термов. Таким образом, можно избежать наличия в словаре слов с ошибками с помощью порогового значения частоты (надеясь, что люди чаще всего вводят запросы без опечаток). При таком сценарии словари могут по-прежнему быть хорошим ресурсом для подсказок, основанных на истории запросов: вы можете использовать эти данные, чтобы предлагать те же запросы или их части.

Давайте создадим алгоритм подсказок на базе словаря, используя API Lucene, с термами из предыдущих запросов. В течение данной главы вы будете реализовывать этот API с использованием различных источников и алгоритмов подсказок; это поможет вам сравнить их и оценить, какой из них выбрать, в зависимости от варианта использования.

4.2. Lookup API

Функции подсказки и автозаполнения предоставляются с помощью API `Lookup` в Apache Lucene (**http://mng.bz/zM0a**). Жизненный цикл поиска обычно включает в себя следующие фазы:

- *сборка* – поиск собирается из источника данных (например, словаря);
- *поиск* – поиск используется для предоставления подсказок, основанных на последовательности символов (и некоторых других, необязательных параметрах);
- *повторная сборка* – поиск собирается заново, если данные, которые будут использоваться для подсказки, обновляются, или необходимо использовать новый источник;
- *сохранение и загрузка* – поиск сохраняется (например, для повторного использования в будущем) и загружается (например, из ранее сохраненного поиска на диске).

Давайте создадим поиск, используя словарь. Вы будете использовать файл, содержащий 1000 ранее введенных запросов, как записано в журнале поисковой системы. Вот как выглядит файл questions.txt с одним запросом на строку:

```
...
popular quizzes
music downloads
music lyrics
outerspace bedroom
high school musical sound track
listen to high school musical soundtrack
...
```

Вы можете собрать словарь из этого простого текстового файла и передать его в `Lookup`, чтобы создать подсказчика на базе словаря:

```
Lookup lookup = new JaspellLookup ();
Path path = Paths.get("queries.txt");
Dictionary dictionary = new
    PlainTextDictionary(path);
lookup.build(dictionary);
```

Инстанциирует поиск

Находит входной файл, содержащий запросы (по одному на строку)

Создает простой текстовый словарь, который выполняет чтение из файла запросов

Создает поиск, используя данные из словаря

Видно, что в реализацию Lookup под названием `JaspellLookup`, которая основана на *троичном дереве поиска*, передаются данные из словаря, содержащего прошлые запросы. Троичное дерево поиска (TST; **https://en.wikipedia.org/wiki/Ternary_search_tree**), подобное тому, что показано на рис. 4.1, – это структура данных, в которой строки хранятся в виде, напоминающем дерево. TST – это особый тип

дерева, именуемый *префиксным деревом*, где каждый узел в дереве – это символ, и у него максимум три дочерних узла.

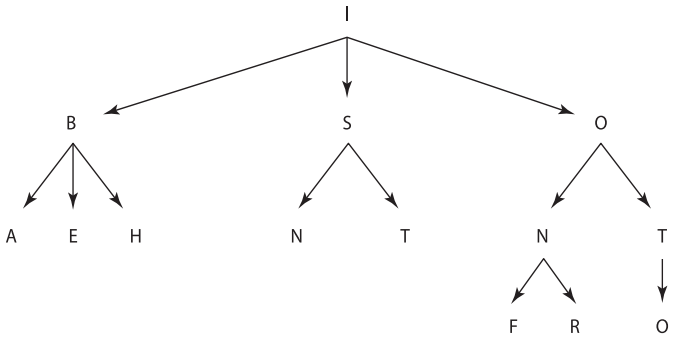


Рис. 4.1 ❖ Троичное дерево поиска

Такие структуры данных особенно полезны для автозаполнения, потому что они эффективны с точки зрения скорости при поиске строк, которые имеют определенный префикс. Вот почему префиксные деревья часто используются в контексте автозаполнения: когда пользователь ищет «mu», дерево может эффективно возвращать все строки, которые начинаются с «mu».

Теперь, когда вы создали свой первый подсказчик, давайте посмотрим, как он работает. Вы разделите запрос «music is my aircraft» на все более крупные последовательности и передадите их поиску, чтобы получить подсказки, имитирующие способ, с помощью которого пользователь вводит запрос в поисковой системе. Вы начнете с «m», затем «mu», «mus», «musi» и т. д. И увидите, какие результаты вы получите на основании предыдущих запросов. Для генерации таких *инкрементных входных данных* используйте приведенные ниже код:

```
List<String> inputs = new LinkedList<>();
for (int i = 1; i < input.length(); i++) {
  inputs.add(input.substring(0, i));
}
```

С каждым шагом создается подстрока исходного ввода, где конечный индекс i больше

API `Lookup#lookup` принимает последовательность символов (ввод пользователя, печатающего запрос) и ряд других параметров, например если вам нужны только более популярные подсказки (допустим, строки, которые чаще встречаются в словаре), и максимальное количество таких подсказок для извлечения. Используя список инкрементных входных данных, можно сгенерировать подсказки для каждой такой подстроки:

```
List<Lookup.LookupResult> lookupResults = lookup.lookup(substring, false, 2);
```

Использует Lookup, чтобы получить максимум два результата для данной подстроки (например, «mu»), независимо от их частоты (morePopular имеет значение false)

Вы получаете список результатов `LookupResults`, каждый из которых состоит из ключа, который является строкой-подсказкой, и значения, которое является весом этой подсказки; этот вес можно рассматривать как меру того, насколько реле-

вантной или частой считает связанную строку реализация подсказчика, поэтому ее значение может варьироваться в зависимости от используемой реализации поиска. Давайте покажем каждый результат предложения вместе с весом:

```
for (Lookup.LookupResult result : lookupResults) {
    System.out.println("--> " + result.key + "(" + result.value + ")");
}
```

Если вы передадите подсказчику все сгенерированные подстроки «music is my aircraft», результаты будут следующими:

```
'm'
--> m
--> m &
----
'mu'
--> mu
--> mu alumni events
----
'mus'
--> musak
--> musc
----
'musi'
--> musi
--> musi for wish you could see me now
----
'music'
--> music
--> music &dvd whereeaglesdare
----
'music '
--> music &dvd whereeaglesdare
--> music - mfs curtains up
----
'music i'
--> music i can download for free no credit cards and music parental advisory
--> music in atlanta
----
'music is'
----
... ◄——— Подсказок больше нет
```

Вы не получаете подсказок для входных данных, помимо «music i». И это не очень хорошо. Причина состоит в том, что вы создали поиск, основанный исключительно на целых строках запроса; вы не предоставили подсказчику возможность разбивать такие строки на более мелкие текстовые единицы. Поиск не смог дать подсказку «is» после слова «music», потому что ни один ранее введенный запрос не начинался со слов «music is».

Это существенное ограничение. С другой стороны, данный тип подсказки удобен для хронологического автозаполнения, когда пользователь видит запросы, которые он вводил в прошлом, как только начинает набирать новый запрос. На-

пример, если бы пользователь выполнил тот же запрос, что и неделю назад, это выглядело бы как подсказка, если бы при реализации использовался словарь ранее введенных запросов.

Но вы хотите сделать больше:

- предлагать в качестве подсказки не только целые строки, которые пользователь вводил в прошлом, но и слова, из которых состояли прошлые запросы (например, «music», «is», «my» и «aircraft»);
- предлагать строки запроса, даже если пользователь вводит слово, которое находится в середине ранее введенного запроса. Например, предыдущий метод дает результаты, если строка запроса *начинается* с того, что печатает пользователь, но вы хотите предложить в качестве подсказки фразу «music is my aircraft», даже если пользователь вводит «my a»;
- предлагать последовательности слов, которые грамматически и семантически верны, но, возможно, ранее не были набраны ни одним пользователем.

Функциональность механизма подсказок должна быть в состоянии составить естественный язык, чтобы помочь пользователям писать более подходящие по звучанию запросы:

- создавать подсказки, которые отражают данные из поисковой системы. Пользователю было бы крайне неприятно, если бы подсказка привела к пустому списку результатов;
- помочь пользователям устранить неоднозначность, когда запрос может иметь разные области действия среди возможных интерпретаций.

Возьмем такой запрос, как «связность нейронов», который может относиться как к области нейронаук, так и к искусственным нейронным сетям. Было бы полезно дать пользователю подсказку касательно того, что такой запрос может попасть в самые разные области, и позволить ему отфильтровать результаты перед отправкой запроса.

В следующих разделах мы рассмотрим каждый из этих пунктов и увидим, как использование нейронных сетей позволяет получать более точные подсказки, по сравнению с другими методами.

4.3. Проанализированные подсказчики

Рассмотрим набор запроса в поисковой системе. Во многих случаях вы не знаете весь запрос целиком, который собираетесь написать. Много лет назад было подругому, когда большая часть поиска в сети основывалась на ключевых словах и людям приходилось думать заранее: «Какие самые важные слова мне нужно искать, чтобы получить релевантные результаты поиска?» Этот подход включал в себя гораздо больше проб и ошибок, чем сейчас. Сегодня хорошие поисковые системы дают полезные подсказки, когда вы набираете запрос; поэтому вы печатаете, просматриваете подсказки, выбираете одну из них, начинаете печатать снова, ищите дополнительные подсказки, выбираете еще одну и т. д.

Давайте проведем простой эксперимент и посмотрим, какие подсказки мне были предложены, когда я искал «books about search and deep learning» (книги о поиске и глубоком обучении) в Google. Когда я набрал слово «book», результаты были обобщенными, как показано на рис. 4.2 (этого и следовало ожидать, потому что у слова «book» может быть много разных значений в разных контекстах). Од-

ной из подсказок было бронирование отелей для тех, кто едет в отпуск в Италию (Рим, Искья, Сардиния, Флоренция, Понца). На этом этапе подсказки не сильно отличались от того, что мы создали с помощью подсказчика на базе словаря с помощью Lucene в предыдущем разделе: все подсказки начинались со слова «book».

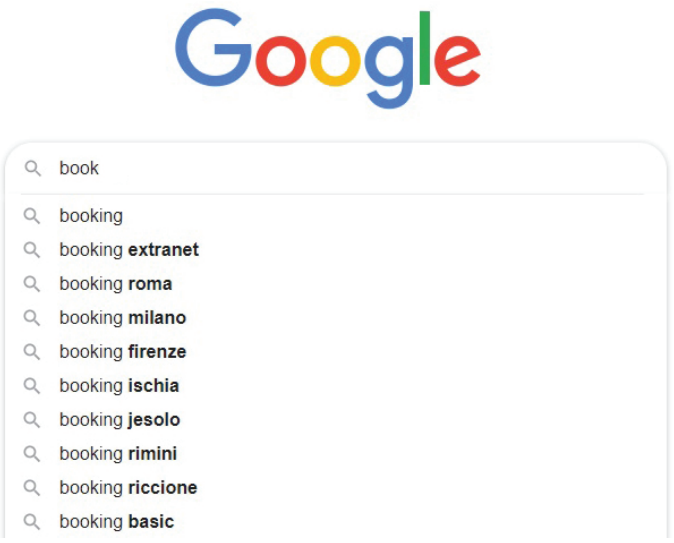


Рис. 4.2 ❖ Подсказки для слова «book»

Я не выбрал ни одну из подсказок, потому что ни одна из них не соответствовала целям моего поиска. Поэтому я продолжал печатать: «books about sear» (см. рис. 4.3).

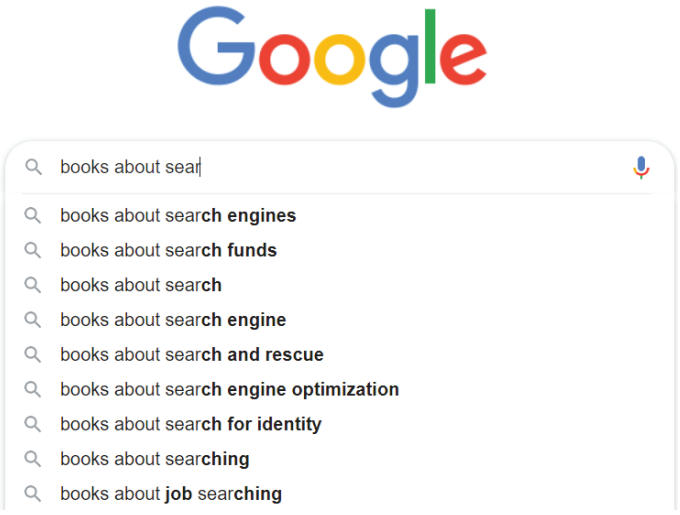


Рис. 4.3 ❖ Подсказки для фразы «books about sear»

Подсказки стали более значимыми и лучше соответствовали моим целям, хотя первые результаты не были релевантными (книги о поисковой оптимизации, книги о поиске личности, книги о поиске и спасении). Пятая подсказка была, вероятно, самой близкой. Интересно отметить, что я также получил следующее:

- подсказку с *инфиксом* (строка подсказки, содержащая новые токены, помещенные между двумя существующими токенами исходной строки). В подсказке «book about google search» слово «google» находится между словами «about» и «sear» в набранном мной запросе. Имейте это в виду, потому что это то, чего вам нужно будет добиться позже; но сейчас мы это пропустим;
- подсказка, где пропущено слово «about» (последние три, «books search ...»). Также имейте это в виду; вы можете отказаться от термов из запроса, давая подсказки.

Я выбрал подсказку «books about search engines», набрал «and» и получил результаты, приведенные на рис. 4.4. Глядя на них, вы, вероятно, понимаете, что по теме интеграции поисковых систем и глубокого обучения книг не много: ни одна из подсказок не намекает на «глубокое изучение». Более важная деталь, на которую следует обратить внимание, заключается в том, что подсказчик, по-видимому, отбрасывает часть текста запроса, когда дает мне подсказки; в окне подсказок все результаты начинаются со слов «engine and». Возможно, это проблема пользовательского интерфейса, потому что подсказки кажутся точными; они не относятся к движкам в общем, а достаточно четко отражают, что слово *engine* относится к поисковику. Вот еще одна идея, о которой следует помнить на будущее: можно отказаться от части текста запроса, когда он становится длиннее.

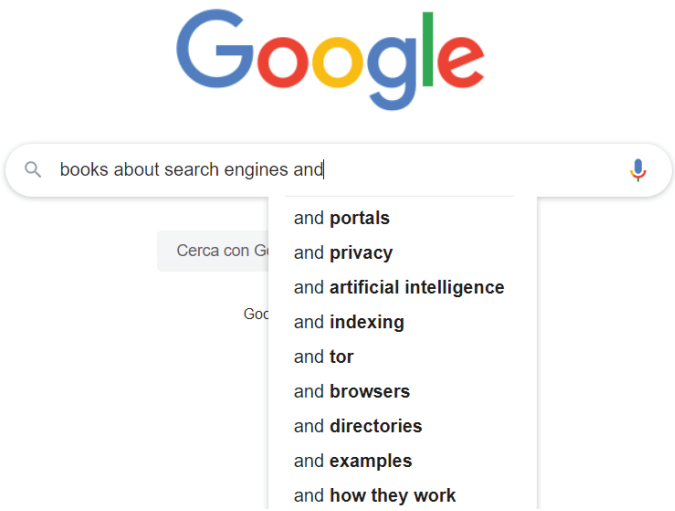


Рис. 4.4 ❖ Подсказки для фразы «books about search engines and»

Я продолжил попытки. Последней подсказкой, показанной на рис. 4.5, был запрос, который я намеревался набрать изначально, с небольшим изменением: я планировал набрать «books about search and deep learning», а подсказкой была фраза «books about search engines and deep learning».

Рис. 4.5 ❖ Подсказки для фразы «books about search engines and dee»

Данный эксперимент был предназначен не для того, чтобы продемонстрировать, как поисковая система Google реализует автозаполнение. Скорее, мы хотели понаблюдать за некоторыми возможностями при работе с автозаполнением:

- подсказки из отдельных слов («books»);
- подсказки из нескольких слов («search engines»);
- подсказки из целых фраз.

Это поможет вам обдумать и решить, что на практике полезно для ваших поисковых систем.

Помимо детализации подсказок (одно слово, несколько слов, предложение и т. д.), мы заметили, что некоторые подсказки имеют следующие характеристики:

- слова, удаленные из запроса («books search engines»);
- подсказки с инфиксами («books about google search»);
- удаленный префикс («books about» не были частью окончательных подсказок).

Все это и многое другое возможно благодаря применению анализа текста к входящему запросу и данным из словаря, который вы используете для создания подсказчика. Например, вы можете удалить некоторые термы, используя фильтр стоп-слов. Или можно разбить длинные запросы на несколько подпоследовательностей и сгенерировать подсказки для каждой подпоследовательности, используя фильтр, который разбивает текстовый поток при определенной длине. Это прекрасно согласуется с тем фактом, что анализ текста активно используется в поисковых системах. У Lucene есть такая реализация поиска, которая называется `AnalyzingSuggester`. Вместо того чтобы полагаться на фиксированную структуру данных, он использует анализ текста, чтобы позволить вам определить, как следует манипулировать текстом, сначала при построении поиска и еще раз при передаче фрагмента текста в поиск, чтобы получить подсказки:

```
Analyzer buildTimeAnalyzer =
    new StandardAnalyzer();

Analyzer suggestTimeAnalyzer =
    new StandardAnalyzer();

Directory dir = FSDirectory.open(
```

Когда вы создаете поиск, то используете StandardAnalyzer, который удаляет стоп-слова и разбивает токены на пробелы

Когда вы ищете подсказки, то используете тот же анализатор, который применялся во время сборки

```
    Paths.get("suggestDirectory"));

AnalyzingSuggester lookup = new AnalyzingSuggester(
    dir, "prefix", buildTimeAnalyzer,
    suggestTimeAnalyzer));
```

Вам необходимо предоставить каталог в файловой системе, потому что AnalyzingSuggester использует его для создания необходимых структур данных для генерации подсказок

Создает экземпляр AnalyzingSuggester

`AnalyzingSuggester` можно создать с использованием отдельных анализаторов для времени сборки и поиска; это позволяет вам быть креативным при настройке подсказчика.

Внутри этой реализации поиска используется *конечный преобразователь*: структура данных, используемая в Lucene. Можно рассматривать этот преобразователь в виде графа, в котором каждое ребро связано с символом и, возможно, весом (см. рис. 4.6). Во время сборки все возможные подсказки, которые приходят в ходе применения анализатора к словарным статьям, компилируются в большой конечный преобразователь. Во время запроса при обходе преобразователя с помощью (проанализированного) входного запроса будут получены все возможные пути и, следовательно, строки подсказок для вывода:

```
'm'
--> m
--> .m
----
'mu'
--> mu
--> mu'
----
'mus'
--> musak
--> musc
----
'musi'
--> musi
--> musi for wish you could see me now
----
'music'
--> music
--> music'
----
'music '
--> music'
--> music by the the
----
'music i'
--> music i can download for free no credit cards and music parental advisory
--> music industry careers
----
'music is'
--> music'
--> music by the the
----
'music is '
```

```
--> music'
--> music by the the
----
'music is m'
--> music by mack taunton
--> music that matters
----
'music is my'
--> music of my heart by nicole c mullen
--> music in my life by bette midler
----
'music is my '
--> music of my heart by nicole c mullen
--> music in my life by bette midler
----
'music is my a'
--> music of my heart by nicole c mullen
--> music in my life by bette midler
----
'music is my ai'
----
...
```

Подсказчик на базе словаря не смог предоставить подсказки после этой точки

Подсказчик на базе словаря не смог предоставить подсказки после этой точки

Подсказок больше нет

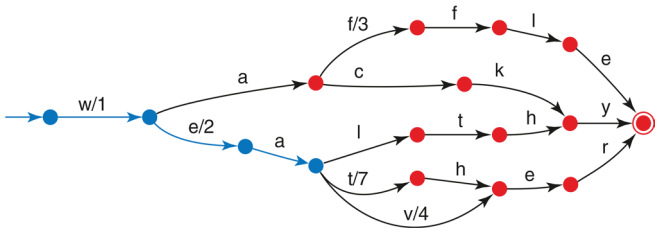


Рис. 4.6 ❖ Конечный преобразователь

Ранее используемый подсказчик на базе троичного дерева поиска прекратил давать подсказки после «music i», поскольку ни одна запись в словаре не начиналась со слов «music is». Но проанализированный подсказчик, даже если словарь тот же, способен предоставить больше подсказок.

В случае с фразой «music is» токен «music» соответствует нескольким подсказкам, и поэтому предоставляются соответствующие результаты, даже если для «is» нет никаких подсказок.

Еще более интересен тот факт, что когда запрос превращается в «music is my», некоторые подсказки содержат и слово «music», и слово «my». Но в определенный момент, когда есть слишком много токенов, которые не совпадают (начиная с «music is my ai»), поиск перестает предоставлять подсказки, поскольку они могут быть слишком слабо связаны с данным запросом. Это определенное улучшение предыдущей реализации, которое решает одну из проблем: вы можете предоставлять подсказки на базе отдельных токенов, а не только целых строк.

Вы также можете улучшить ситуацию, используя слегка измененную версию `Analyzing-Suggester`, которая лучше работает с подсказками с инфиксами:

```
AnalyzingInfixSuggester lookup = new AnalyzingInfixSuggester(dir,
    buildTimeAnalyzer, lookupTimeAnalyzer, ... );
```

Используя этот подсказчик, вы получите более интересные результаты:

```
'm'
--> 2007 s550 mercedes
--> 2007 qualifying times for the boston marathon
----
'mu'
--> 2007 nissan murano
--> 2007 mustang rims com
----
'mus'
--> 2007 mustang rims com
--> 2007 mustang
```

У вас нет результатов, начинающихся с «m», «mu» или «mus»; вместо этого такие последовательности используются для совпадения с самой важной частью строки, например «2007 s550 mercedes», «2007 qualifying times for the boston marathon», «2007 nissan murano» и «2007 mustang rims com». Еще заметное отличие заключается в том, что сопоставление токенов может происходить в середине подсказки (именно поэтому оно называется *инфиксом*):

```
'music is my'
--> 1990's music for myspace
--> words to music my humps
----
'music is my '
--> words to music my humps
--> where can i upload my music
----
'music is my a'
--> words to music my humps
--> where can i upload my music
```

С помощью `AnalyzingInfixSuggester` вы получаете подсказки с инфиксами. Он берет входную последовательность, анализирует ее, чтобы создать токены, а затем предлагает совпадения на базе совпадений префиксов любых таких токенов. Но у вас по-прежнему есть проблемы с тем, чтобы сделать подсказки ближе к данным, хранящимся в поисковой системе, чтобы они были больше похожи на естественный язык и имелась возможность более эффективно устранять неоднозначность, когда два слова имеют разные значения. Кроме того, вы не получаете никаких подсказок, когда начинаете вводить слово «aircraft», поскольку недостаточно токенов совпадает.

Теперь, когда у вас есть небольшой опыт в решении проблем, связанных с предоставлением подходящих подсказок, мы обсудим языковые модели. Сначала рассмотрим модели, реализованные с помощью обработки естественного языка (*N-граммы*), а затем модели, реализованные с помощью нейронных сетей (нейронные языковые модели).

4.4. Использование языковых моделей

В подсказках, показанных в предыдущих разделах, некоторые текстовые последовательности не имели большого смысла: например, «music by the». Вы дали подсказчику данные из ранее введенных запросов, поэтому в какой-то записи пользователь должен был ошибиться, набрав артикль «the» дважды. Кроме того, вы предоставили подсказки, состоящие из всего запроса целиком. Хотя это прекрасно подходит, если вы хотите использовать автозаполнение для возврата всего текста предыдущих запросов (это может быть полезно, если вы ищете книгу в книжном онлайн-магазине), но не очень хорошо подходит для составления новых запросов.

В средних и крупных поисковых системах журналы поиска содержат огромное количество разнообразных запросов – придумать хороший алгоритм подсказки сложно из-за количества и разнообразия таких текстовых последовательностей. Например, если вы посмотрите на набор данных web09-bst (**http://boston.lti.cs.cmu.edu/Data/web08-bst/planning.html**), то найдете такие запросы, как «hobbs police department», «ipod file sharing» и «liz taylor's biography». Эти запросы выглядят неплохо и могут быть использованы в качестве источников для алгоритма подсказок. С другой стороны, вы также можете найти такие запросы, как «hhhhh», «hqwebdev» и «hhht hootdithuinshithins». У вас вряд ли возникнет желание, чтобы подсказчик предлагал похожие подсказки! Проблема состоит не в том, чтобы отфильтровать запрос «hhhh», который можно убрать из набора данных, удалив все строки или слова, содержащие три или более одинаковых последовательных символов. Отфильтровать запрос «hqwebdev» гораздо сложнее: он содержит слово «webdev» (сокращенный вариант записи слова «web developer») с префиксом «hq». Такой запрос может иметь смысл (например, есть сайт с таким именем), но вы не хотите использовать слишком конкретные подсказки для универсального сервиса подсказок. Задача состоит в том, чтобы работать с различными текстовыми последовательностями, некоторые из которых, возможно, не имеет смысла использовать, потому что они слишком специфичны и, следовательно, редки. Одним из способов решения этой проблемы является использование *языковых моделей*.

Языковые модели

В области естественной обработки языка основной задачей языковой модели является прогнозирование вероятности определенной последовательности текста. Вероятность – это мера вероятности того или иного события. Она колеблется от 0 до 1. Поэтому если вы возьмете причудливый запрос, который мы видели ранее – «music by the the», – и передадите его в языковую модель, то получите низкую вероятность (например, 0,05). Языковые модели представляют распределение вероятностей и, следовательно, могут помочь спрогнозировать вероятность определенного слова или последовательности символов в определенном контексте. Языковые модели могут помочь исключить последовательности, которые маловероятны (низкая вероятность), и создать ранее невидимые последовательности слов, поскольку они фиксируют, какие последовательности наиболее вероятны (даже если они могут не отображаться в тексте).

Языковые модели часто реализуются путем вычисления вероятностей N-грамм.

N-граммы

N-грамма – это последовательность символов, состоящая из *n* последовательных *единиц*, где единицей может быть символ («a», «b», «c», ...) или слово («music», «is», «my», «...»). Представьте себе N-граммную модель языка (используя слова в качестве единиц), где *n* = 2. N-грамма, где *n* = 2, также называется *биграммой*; N-грамма, где *n* = 3, также известна как *триграмма*. Биграммная языковая модель может оценивать вероятность пар слов, таких как «музыкальный концерт» или «музыкальная софа». Хорошая языковая модель назначает вероятность биграмме «музыкальный концерт», который выше, по сравнению с вероятностью биграммы «музыкальная софа».

В качестве примечания по реализации вероятность (последовательность) N-грамм для языковой модели можно рассчитать несколькими способами. Большинство из них полагается на *допущение Маркова*, когда вероятность какого-либо события в будущем (например, следующего символа или слова) зависит только от ограниченной истории предшествующих событий (символов или слов). Поэтому если вы используете N-граммную модель, где *n* = 2, которая также носит название *биграммной модели*, вероятность *следующего* слова, заданного *текущим* словом, определяется путем подсчета количества вхождений двух слов «music is» и деления этого результата на количество вхождений только одного текущего слова («music»). Например, вероятность того, что следующее слово – это «is», учитывая текущее слово «music», можно записать как *P (is | music)*. Чтобы вычислить вероятность слова с заданной последовательностью слов, превышающей два, например вероятность того, что «aeroplane» с учетом «music is my», вы разбили это предложение на биграммы, вычислили вероятности всех таких биграмм и перемножили их:

P(*music is my aeroplane*) = P(*is*|*music*) * P(*my*|*is*) * P(*aeroplane*|*my*).

Для справки: во многих N-граммных моделях языка используется несколько более продвинутый метод под названием *глупый откат*[1], который сначала пытается вычислить вероятность N-грамм с большим *n* (например, *n* = 3), а затем рекурсивно возвращается к меньшим N-граммам (таким как *n* = 2), если N-граммы с текущим *n* не существует в данных. Такие вероятности не учитываются, поэтому вероятности от более крупных N-грамм оказывают более положительное влияние на общий показатель вероятности. У Lucene есть поиск с языковой моделью на базе N-грамм, называющийся `FreeTextSuggester` (с использованием алгоритма глупого отката), который использует анализатор, чтобы решить, как разбивать N-граммы:

```
Lookup lookup = new FreeTextSuggester (new WhitespaceAnalyzer());
```

Давайте посмотрим, как это работает, установив для *n* значение 2, в запросе «music is my aircraft»:

```
'm'
--> my
```

[1] См. раздел 4 книги: *Thorsten Brants et al.* Large Language Models in Machine Translation (**http://www.aclweb.org/anthology/D07-1090.pdf**).

```
--> music
----
'mu'
--> music
--> museum
----
'mus'
--> music
--> museum
----
'musi'
--> music
--> musical
----
'music'
--> music
--> musical
----
'music '
--> music video
--> music for
----
'music i'
--> music in
--> music industry
----
'music is'
--> island
--> music is
----
'music is '
--> is the
--> is a
----
'music is m'
--> is my
--> is missing
----
'music is my'
--> is my
--> is myspace.com
----
'music is my '
--> my space
--> my life
----
'music is my a'
--> my account
--> my aol
----
'music is my ai'
--> my aim
```

Одна из подсказок для запроса «music is m» совпала с желаемым запросом («is my») на один символ вперед

Подсказки для запроса «music is my» («my space», «my life») – это не то, что вы ищете, но звучит хорошо

Подсказки для запроса «music is my ai» не очень подходят («my aim», «air»), но они ближе к тому, что вам нужно

```
--> air
----
'music is my air'
--> air
--> airport
----
'music is my airc'
--> aircraft
--> airconditioning
----
'music is my aircr'
--> aircraft
--> aircraftbarnstormer.com
----
...
```

Подсказки для запроса «music is my airc» дали совпадение на четыре символа вперед («aircraft») и забавное слово («airconditioning»)

Положительным моментом является то, что подсказчик на базе языковой модели всегда дает подсказки. Когда конечный пользователь не может рассчитывать на подсказки, даже если они не особенно точны, в этом нет смысла. Это преимущество, по сравнению с предыдущими методами. Самое главное, вы можете увидеть поток подсказок от слова «music» далее.

> **ПРИМЕЧАНИЕ** Вам, наверное, интересно, как модели на основе биграмм могут прогнозировать целые слова из частей слов. По аналогии с `AnalyzingSuggester`, `FreeText-Suggester` собирает конечный преобразователь из N-грамм.

С помощью N-граммной модели языка можно создавать запросы типа «music is my space», «music is my life» и даже «music is my airconditioning», которые не отображаются в журнале поиска. Итак, вы достигли цели создания новых последовательностей слов. Но из-за природы N-грамм (фиксированная последовательность токенов) для более длинных запросов полные подсказки не предоставляются: таким образом, фраза «music is my aircraft» не была включена в подсказки на заключительных этапах, просто «aircraft». Это не обязательно плохо, но это подчеркивает тот факт, что такие N-граммные языковые модели не очень эффективны для расчета хороших вероятностей длинных предложений; поэтому они могут давать странные подсказки, такие как «music is my airconditioning».

Все, что вы только что узнали, относится к существующим методам создания подсказок. Я хотел, чтобы вы увидели все проблемы, которые влияют на эти подходы, прежде чем погрузиться в изучение нейронных языковых моделей, которые объединяют возможности каждого из этих методов. Еще один недостаток этих моделей, который мы до сих пор игнорировали, заключается в том, что им нужны словари, созданные вручную, как вы видели в примере с word2vec, – то, что не является устойчивым на практике. Вам необходимы решения, которые автоматически адаптируются к изменяющимся данным и не требуют ручного вмешательства. Для этого вы будете использовать поисковую систему, чтобы снабжать подсказчика. Подсказки, полученные с помощью таких данных, будут основаны на проиндексированном контенте. Если документы будут проиндексированы, подсказчик также будет обновлен. В следующем разделе мы рассмотрим эти подсказчики на базе контента.

4.5. Подсказчики на базе контента

В случае с подсказчиками на базе контента последний поступает непосредственно из поисковой системы. Рассмотрим поисковую систему для книжного магазина. Вероятно, что пользователи будут искать названия книг или авторов гораздо чаще, чем выполнять поиск по тексту книги.

Каждая индексируемая книга имеет отдельные поля для названия, автора(ов) и в конечном итоге текста книги. Кроме того, по мере того как новые книги индексируются, а старые уже не выпускаются, вам необходимо добавлять в поисковую систему новые документы и удалять те, что относятся к книгам, которые больше не продаются. То же самое должно происходить и с подсказками: вы хотите давать подсказки для новых названий и не хотите предлагать названия книг, которые больше не продаются.

Поэтому подсказчик должен быть в курсе. Если какой-либо документ будет удален из индекса, подсказчик может сохранить подсказки, созданные на основе этого текста, но они могут оказаться бесполезными. Предположим, что были проиндексированы две книги: *Lucene в действии* и *Oauth2 в действии*. Подсказчик, использующий только текст из названий книг, будет основываться на термах (написаны строчными буквами): «lucene», «in», «action», «oauth2». Если вы удалите книгу *Lucene в действии*, список термов будет сокращен до «in», «action», «oauth2». Можно оставить токен «lucene» в подсказчике; в этом случае, если пользователь введет «L», подсказчик предложит слово «lucene». Проблема состоит в том, что запрос для слова «lucene» не даст никаких результатов. Вот почему вы должны удалить термы из подсказчика, если они не имеют возможного соответствия во время поиска.

Вы можете получить доступ к инвертированному индексу, который содержит данные о названиях книг, и использовать эти термы так же, как используете строки статического словаря. В Lucene подачу данных из индекса можно осуществлять с использованием `DocumentDictionary`. `DocumentDictionary` считывает данные из поисковой системы, в частности из `Index-Reader` (представление в поисковой системе в определенный момент времени), используя одно поле для выборки термов (которые будут использоваться для подсказок) и еще одно поле для окончательного расчета весов подсказок (насколько важна подсказка).

Давайте создадим словарь из данных, проиндексированных в поле `title` в поисковой системе. Мы будем придавать большее значение названиям, рейтинг которых выше. Подсказки, поступающие от книг с более высоким рейтингом, будут показаны первыми:

```
IndexReader reader = DirectoryReader.open(
    directory);
```
← Получает представление (IndexReader) в поисковой системе

```
Dictionary dictionary = new DocumentDictionary(
    reader, "title", "rating");
```
← Создает словарь на основе содержимого поля title и позволяет рейтингу определять, какой вес имеет подсказка

```
lookup.build(dictionary);
```
← Создает поиск с помощью данных из индекса, так же как и при участии статического словаря

Вы можете направить пользователя к выбору результатов поиска, которые вы хотите, чтобы он нашел, – например, будучи владельцем книжного магазина, вам

будет приятно, если книги с более высоким рейтингом будут показываться чаще. Другие метрики для стимулирования подсказок могут быть связаны с ценами, поэтому пользователю предоставляются более частые подсказки книг с более высокими или более низкими ценами.

Теперь, когда вы готовы к получению данных для подсказок от поисковой системы, можно взглянуть на нейронные языковые модели. Мы ожидаем, что они смогут с большей верностью объединить все хорошее, взятое из методов, обсуждавшихся до сих пор, составляя запросы, которые выглядят так, как будто они были набраны человеком.

4.6. Нейронные языковые модели

Предполагается, что нейронная языковая модель обладает теми же возможностями, что и другие типы языковых моделей, например N-граммные модели. Разница заключается в том, как они учатся предсказывать вероятности и насколько их прогнозы лучше. В главе 3 мы познакомились с рекуррентной нейронной сетью, которая пыталась воспроизвести текст из произведений Шекспира. Мы были сосредоточены на том, как работают рекуррентные сети, но на практике вы настраивали *нейронную языковую модель на уровне символов*! Вы убедились, что РНС очень хорошо справляются с изучением последовательностей текста без учителя и таким образом могут генерировать неплохие новые последовательности на основе увиденных ранее. Языковая модель учится получать точные вероятности текстовых последовательностей, поэтому это идеально подходит для РНС.

Давайте начнем с простой рекуррентной сети, которая *не* является глубокой и реализует языковую модель на уровне символов: эта модель будет прогнозировать вероятности всех возможных выходных символов, учитывая последовательность входных символов. Давайте визуализируем ее:

```
LanguageModel lm = ...
for (char c : chars) {
    System.out.println("mus" + c + ":" + lm.getProbs("mus"+c));
}

....

musa:0.01
musb:0.003
musc:0.02
musd:0.005
muse:0.02
musf:0.001
musg:0.0005
mush:...
musi:...
...
```

Вы знаете, что нейронная сеть использует векторы для входных данных и результатов; выходной слой РНС, который вы использовали для генерации текста в главе 3, создал вектор, содержащий действительное число (от 0 до 1) для каждого возможного выходного символа. Это число обозначает вероятность вывода сим-

вола из сети. Вы также видели, что генерация распределения вероятностей (в данном случае вероятности для всех возможных символов) выполняется функцией softmax. Теперь, когда вы знаете, что делает выходной слой, вы можете добавить рекуррентный слой посередине, в обязанности которого входит запоминание ранее увиденных последовательностей, и входной слой для отправки входных символов в сеть. Результат иллюстрируется диаграммой на рис. 4.7.

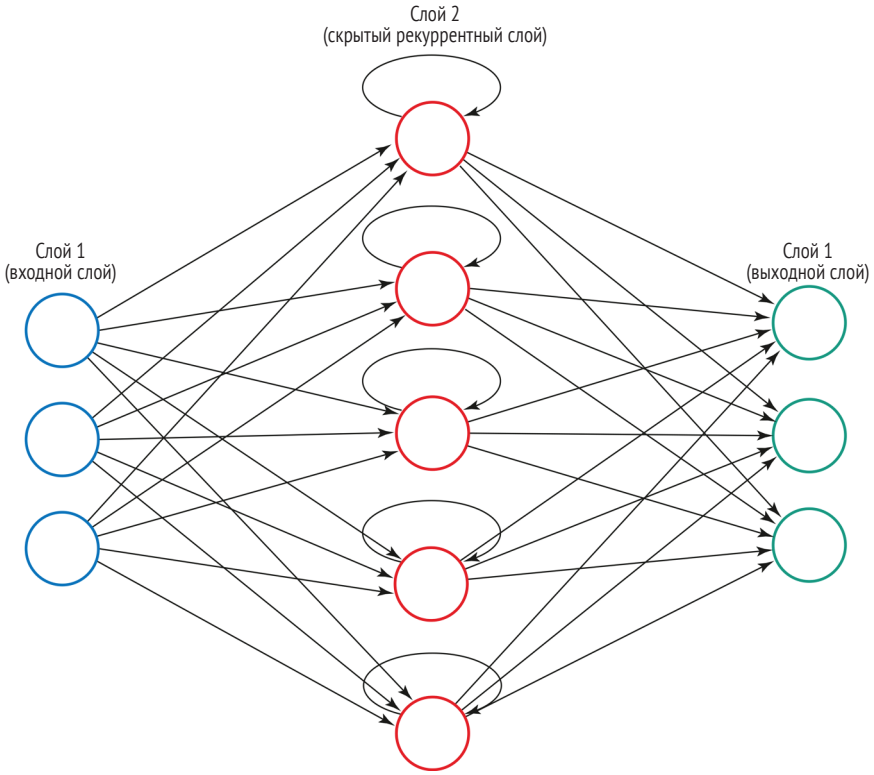


Рис. 4.7 ❖ Рекуррентная сеть для обучения с использованием последовательности

Работая с DL4J, вы настраивали такую сеть при создании альтернативных запросов в главе 3:

```
int layerSize = 50;                    <-- Размер скрытого слоя
int sequenceSize = chars.length();     <-- Входной и выходной размеры
int unrollSize = 100                   <-- Число развертываний PHC
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.layer(0, new LSTM.Builder().nIn(sequenceSize).nOut(layerSize)
    .activation(Activation.TANH).build())
.layer(1, new RnnOutputLayer.Builder(LossFunction.MCXENT).activation(
    Activation.SOFTMAX).nIn(layerSize).nOut(sequenceSize).build())
.backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(unrollSize)
    .tBPTTBackwardLength(unrollSize)
.build();
```

Хотя основная архитектура та же (LSTM-сеть с одним или несколькими скрытыми слоями), цель здесь отличается от того, чего вы хотите достичь в случае использования генерации альтернативных запросов. В случае с альтернативными запросами вам нужна РНС, чтобы получить запрос и вывести новый. В этом случае вам нужно, чтобы сеть угадала подходящее завершение запроса, который пишет пользователь, прежде чем он закончит набирать его. Это та же самая архитектура РНС, что использовалась для генерации текста из произведений Шекспира.

4.7. Нейронная языковая модель на базе символов для создания подсказок

В главе 3 вы передали рекуррентной сети `CharacterIterator`, который перебирает символы в файле. Итак, вы создали подсказки из текстовых файлов. План состоит в том, чтобы использовать нейронную сеть как инструмент для помощи поисковой системы, поэтому данные для подачи должны поступать из самой поисковой системы. Давайте проиндексируем набор данных Hot 100 Billboard:

```
IndexWriter writer = new IndexWriter(directory, new IndexWriterConfig());

for (String line :
     IOUtils.readLines(getClass().getResourceAsStream("/billboard_lyrics_1964
     -2015.csv"))) {
  if (!line.startsWith("\"R")) {
    String[] fields = line.split(",");
    Document doc = new Document();
    doc.add(new TextField("rank", fields[0],
        Field.Store.YES));
    doc.add(new TextField("song", fields[1],
        Field.Store.YES));
    doc.add(new TextField("artist", fields[2],
        Field.Store.YES));
    doc.add(new TextField("lyrics", fields[3],
        Field.Store.YES));
    writer.addDocument(doc);
  }
}
writer.commit();
```

Создает IndexWriter для помещения документов в индекс

Читает каждую строку набора данных, по одной за раз

Не использует строку заголовка

Каждая строка в файле имеет следующие атрибуты, разделенные запятой: Ранг, Песня, Исполнитель, Год, Текст песни, Источник

Индексирует ранг песни в специальное поле (с сохраненным значением)

Индексирует название песни в специальное поле (с сохраненным значением)

Индексирует исполнителя песни в специальное поле (с сохраненным значением)

Индексирует текст песни в специальное поле (с сохраненным значением)

Добавляет созданный документ Lucene в индекс

Сохраняет индекс в файловой системе

Вы можете использовать индексированные данные для построения символьной реализации поиска на базе LSTM-сети под названием `CharLSTMNeuralLookup`. Аналогично тому, что вы делали для `Free-TextSuggester`, вы можете использовать `DocumentDictionary` для подачи данных в `CharLSTMNeural-Lookup`:

```
Dictionary dictionary = new DocumentDictionary(reader, "lyrics", null);
Lookup lookup = new CharLSTMNeuralLookup (...);
lookup.build(dictionary);
```

Создает DocumentDictionary, содержимое которого извлекается из проиндексированного текста песни

Создает поиск на базе charLSTM

Обучает поиск на базе charLSTM

DocumentDictionary извлечет текст из поля lyrics. Для создания экземпляра CharLSTMNeuralLookup необходимо передать конфигурацию сети в качестве параметра конструктора, чтобы:

- во время сборки LSTM-сеть перебирала символы значений документов Lucene и училась генерировать аналогичные последовательности;
- во время выполнения LSTM-сеть генерировала символы на основе части запроса, уже написанного пользователем.

Завершаем предыдущий код. Конструктору CharLSTMNeuralLookup необходимы параметры для построения и обучения LSTM-сети:

```
int lstmLayerSize = 100;
int miniBatchSize = 40;
int exampleLength = 1000;
int tbpttLength = 50;
int numEpochs = 10;
int noOfHiddenLayers = 1;
double learningRate = 0.1;
WeightInit weightInit = WeightInit.XAVIER;
Updater updater = Updater.RMSPROP;
Activation activation = Activation.TANH;

Lookup lookup = new CharLSTMNeuralLookup (lstmLayerSize, miniBatchSize,
    exampleLength, tbpttLength, numEpochs, noOfHiddenLayers,
    learningRate, weightInit, updater, activation);
```

Как упоминалось ранее, нейронным сетям необходимо большое количество данных, чтобы давать хорошие результаты. Будьте внимательны при выборе конфигурации нейронной сети для работы с этими наборами данных. В частности, обычно конфигурация, в которой нейронная сеть хорошо работает на одном наборе данных, не приводит к тому же качеству на другом наборе данных. Рассмотрим количество обучающих образцов по отношению к количеству весов нейронной сети, которые должны быть изучены сетью. Количество примеров всегда должно быть больше количества изучаемых параметров: весов нейронной сети.

Если у вас есть MultiLayerNetwork и DataSet, можно сравнить их:

```
MultiLayerNetwork net = new MultiLayerNetwork(...);
DataSet dataset = ...;
System.out.println("params :" + net.numParams() + ," examples: "
    + dataset.numExamples());
```

Еще один аспект, который мы еще не рассматривали, – это *инициализация* весов сети. Каковы начальные значения весов, когда вы начинаете обучение нейронной сети? Установка одного и того же значения для весов (ноль еще хуже) и случайных значений – плохая идея. Схемы инициализации весов чрезвычайно важны для способности нейронной сети к быстрому обучению. В этом случае хорошими схемами инициализации веса являются NORMAL и XAVIER. Они относятся к распределениям вероятностей с определенными свойствами; о них можно прочитать на странице **http://mng.bz/K19K**.

Для прогнозирования выходов из нейронной сети вы используете тот же код, который применялся для генерации альтернативных запросов. Поскольку эта LSTM-сеть работает на уровне символов, вы выводите по одному символу за раз:

```
INDArray output = network.rnnTimeStep(input);
```
Прогнозирует распределение вероятностей для заданного входного символа (вектора)

```
int sampledCharacterIdx = sampleFromDistribution(
    output);
```
Отбирает вероятный символ из сгенерированного распределения

```
char c = characterIterator.convertIndexToCharacter(
    sampledCharacterIdx);
```
Преобразует индекс выбранного символа в фактический

Теперь вы можете реализовать API `Lookup#lookup`, используя нейронную языковую модель. У этой модели есть базовая нейронная сеть и объект (`CharacterIterator`), который обращается к набору данных, используемому для обучения. Основная причина, по которой следует обратиться к нему, – это сопоставление. Например, вам необходимо иметь возможность восстановить, какой символ соответствует определенному вектору с унитарным кодированием (и наоборот):

```
public class CharLSTMNeuralLookup extends Lookup {

  private CharacterIterator characterIterator;
  private MultiLayerNetwork network;

  public CharLSTMNeuralLookup (MultiLayerNetwork net,
      CharacterIterator iter) {
    network = net;
    characterIterator = iter;
  }

  @Override
  public List<LookupResult> lookup(CharSequence key,
        boolean onlyMorePopular, int num) throws IOException {
    List<LookupResult> results = new
        LinkedList<>();
    Map<String, Double> output = NeuralNetworksUtils
        .sampleFromNetwork(network, characterIterator,
        key.toString(), num);
    for (Map.Entry<String, Double> entry : output.entrySet()) {
      results.add(new LookupResult(entry.getKey(),
        entry.getValue().longValue()));
    }
    return results;
  }
  ...
```
Отбирает текстовые последовательности из сети, учитывая введенную пользователем строку ввода

Подготавливает список результатов

Добавляет отобранные выходные данные в список результатов, используя их вероятности (из функции softmax) в качестве весов подсказок

`CharLSTMNeuralLookup` также должен реализовывать API сборки, где нейронная сеть будет обучаться (или переучиваться):

```
IndexReader reader = DirectoryReader.open(directory);
Dictionary dictionary = new DocumentDictionary(reader,
    "lyrics", "rank");
lookup.build(dictionary);
```
Извлекает текст, который будет использоваться для подсказок из поля lyrics, взвешенный по значению ранга песни

Поскольку символьная LSTM0-сеть использует `CharacterIterator`, вы преобразуете данные из словаря (объект `InputIterator`) в `CharacterIterator` и передаете их в нейронную сеть для обучения (в качестве побочного эффекта это означает на-

личие временного файла на диске для хранения данных, извлеченных из индекса для обучения сети):

```
@Override
public void build(Dictionary dictionary) throws IOException {
  Path tempFile = Files.createTempFile("chars",
    ".txt");
  FileOutputStream outputStream = new FileOutputStream(tempFile.toFile());
  for (BytesRef surfaceForm; (surfaceForm = dictionary
        .getInputIterator().next()) != null;) {
    outputStream.write(surfaceForm.bytes);
  }
  outputStream.flush();
  outputStream.close();
  characterIterator = new CharacterIterator(tempFile
    .toAbsolutePath().toString(), miniBatchSize,
    exampleLength);
  this.network = NeuralNetworksUtils.trainLSTM(
    lstmLayerSize, tbpttLength, numEpochs, noOfHiddenLayers, ...);
  FileUtils.forceDeleteOnExit(tempFile.toFile());
}
```

Создает временный файл

Извлекает текст, поступающий из поля lyrics в индексе Lucene (Lucene использует BytesRef вместо String по соображениям производительности)

Записывает текст во временный файл

Освобождает ресурсы для записи во временный файл

Создает CharacterIterator (используя параметры конфигурации CharLSTMNeuralLookup)

Создает и обучает LSTM-сеть (используя параметры конфигурации CharLSTMNeuralLookup)

Удаляет временный файл

Прежде чем идти дальше и использовать этот поиск в поисковом приложении, необходимо убедиться, что нейронная языковая модель работает хорошо и дает хорошие результаты. Как и другие алгоритмы в информатике, нейронные сети не волшебники: нужно правильно их настроить, если вы хотите, чтобы они хорошо работали.

4.8. Настройка языковой модели

Вместо того чтобы делать то, чем вы занимались в главе 3, и добавлять дополнительные слои в сеть, мы начнем с одного слоя, отрегулируем другие параметры и посмотрим, достаточно ли одного слоя. Наиболее важной причиной для этого является тот факт, что по мере роста сложности сети (например, увеличения количества слоев) данные и время, необходимые для фазы обучения, чтобы сгенерировать хорошую модель (которая дает хорошие результаты), также растут. Так что хотя маленькие, неглубокие сети и не могут обогнать более глубокие сети с большим количеством различных данных, данный пример моделирования языка – неплохое место, чтобы научиться начинать с малого и идти дальше только тогда, когда это необходимо.

По мере того как вы будете больше работать с нейронными сетями, вы узнаете, как их лучше настроить. На данный момент вы знаете, что когда данных много и они разные, возможно, неплохо иметь глубокую РНС для языкового моделирования. Но давайте будем прагматичными и посмотрим, так ли это на самом деле. Для этого вам нужен способ оценить процесс обучения нейронной сети. Обучение нейронной сети – это *проблема оптимизации*, когда вы хотите оптимизировать веса в связях между нейронами, чтобы они могли генерировать желаемые результаты. На практике это означает, что у вас есть начальный набор весов в каждом

слое в соответствии с выбранной схемой их инициализации. Эти веса корректируются во время обучения, поэтому ошибка, которую сеть допускает при попытке спрогнозировать результаты, уменьшается с продолжением обучения. Если ошибка, допущенная сетью, не уменьшается или, что еще хуже, увеличивается, вы сделали что-то не так в настройке. В главе 3 мы говорили о *функциях потерь*, которые измеряют такую ошибку, и о том, что цель алгоритма обучения нейросети – минимизировать эти функции. Хороший способ начать измерять, хорошо ли проходит обучение, – это составить график потерь сети с течением времени и убедиться, что они продолжают уменьшаться по мере продолжения обратного распространения ошибки.

Чтобы ваша языковая модель дала хорошие результаты, вам нужно отслеживать, снижаются ли потери. В случае с DL4J можно использовать слушателей `TrainingListener`, например `ScoreIterationListener` из главы 3 (куда записывается значение потерь) или, что еще лучше, `StatsListener`, который имеет надлежащий пользовательский интерфейс и будет собирать и отправлять статистику на удаленный сервер, чтобы вы могли лучше контролировать учебный процесс. На рис. 4.8 показано, как такой сервер отображает процесс обучения.

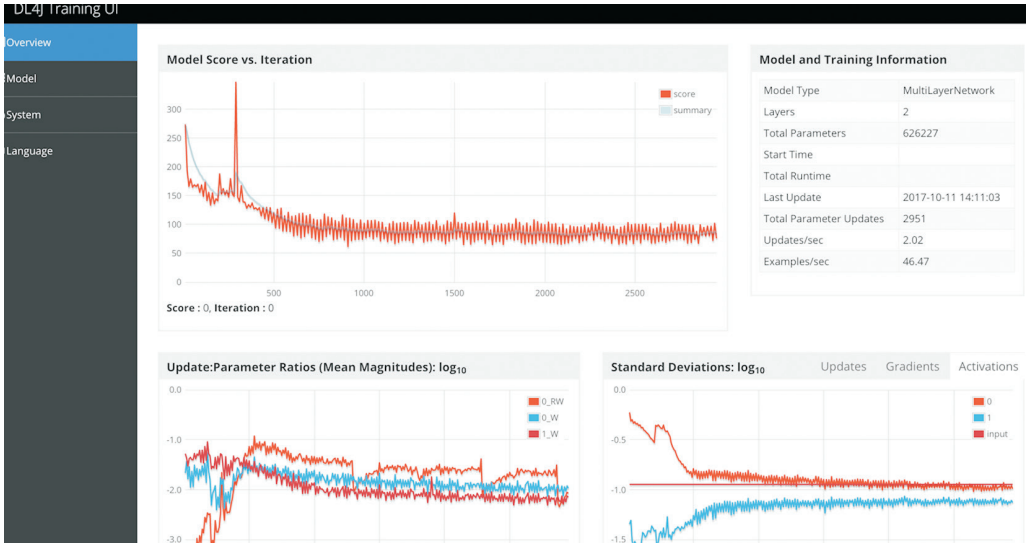


Рис. 4.8 ❖ Обучающий пользовательский интерфейс DL4J

На странице **Overview** (Обзор) DL4J Training UI содержится много информации о процессе обучения; на данный момент мы сосредоточимся на панели **Model Score vs. Iteration** в левом верхнем углу. Здесь оценка должна уменьшаться по мере увеличения количества итераций, в идеале до нуля. В правом верхнем углу вы видите общую информацию о параметрах сети и скорости обучения. Мы пропустим графики в нижней части страницы, потому что они показывают более подробную информацию о размере параметров (например, веса) и их изменении со временем.

Настроить этот пользовательский интерфейс легко:

```
UIServer uiServer = UIServer.getInstance();
```
Инициализирует бэкенд пользовательского интерфейса

```
StatsStorage statsStorage = new InMemoryStatsStorage();
```

```
uiServer.attach(statsStorage);
```
Прикрепляет экземпляр StatsStorage к пользовательскому интерфейсу, поэтому содержимое StatsStorage будет отображено

Настраивает, где должна храниться информация о сети, – в данном случае в памяти

После настройки и запуска UI-сервера вы теперь велите нейронной сети отправлять на него статистику, добавляя `StatsListener`:

```
MultiLayerNetwork net = new MultiLayerNetwork(conf);
```
Нейронная сеть для мониторинга

```
net.init();
```
Инициализирует сеть (например, устанавливает начальные веса в слоях)

```
net.setListeners(new StatsListener(statsStorage));
```
Использует StatsListener

```
net.fit()
```
Запускает обучение

Как только начинается обучение, вы можете получить доступ к пользовательскому интерфейсу DL4J по адресу http://localhost:9000 из веб-браузера. Когда вы это сделаете, то увидите страницу «Обзор».

Давайте начнем с символьной LSTM-сети, у которой есть два скрытых слоя, и посмотрим, как она работает с набором данных запросов, посмотрев на DL4J Training UI (*см.* рис. 4.9).

Рис. 4.9 ❖ Нейронная языковая модель LSTM на уровне символов с двумя скрытыми слоями (по 300 нейронов каждый)

Как вы видите, количество очков уменьшается с количеством итераций, а это означает, что вы, вероятно, не получите хороших результатов. Чрезмерная оптимизация нейронной сети – распространенная ошибка: вы начали с двух 300-мерных скрытых слоев, и, возможно, это слишком много. Ранее в этой главе я упоминал о том, что не стоит иметь весов больше, чем обучающих образцов. Давайте еще раз проверим журналы:

```
...
INFO o.d.n.m.MultiLayerNetwork - Starting MultiLayerNetwork ...
INFO c.m.d.u.NeuralNetworksUtils - params :1.197.977, examples: 77.141
INFO o.d.o.l.ScoreIterationListener - Score at iteration 0 is 174.1792
....
```

Количество обучающих примеров в 100 раз меньше количества изучаемых параметров. Из-за этого маловероятно, что при обучении вы получите хороший набор весов. У вас недостаточно данных!

Вам нужно либо получить больше данных, либо использовать более простую нейронную сеть с меньшим количеством параметров, которые нужно изучить. Предполагая, что вы не можете осуществить первый вариант, давайте перейдем ко второму: настройте более простую, меньшую нейронную сеть, у которой один скрытый слой с 80 нейронами, и снова проверьте журналы:

```
...
INFO o.d.n.m.MultiLayerNetwork - Starting MultiLayerNetwork ...
INFO c.m.d.u.NeuralNetworksUtils - params :56.797, examples: 77.141
INFO o.d.o.l.ScoreIterationListener - Score at iteration 0 is 173.4444
...
```

На рис. 4.10 показана более подходящая кривая потерь; она плавно снижается, хотя конечная точка не близка к нулю.

Цель состоит в том, чтобы конечная потеря достигла значения, которое постоянно остается близким к нулю. В любом случае, давайте проверим это с помощью запроса «music is my aircraft» – можно ожидать субоптимальных результатов, потому что нейронная сеть не нашла комбинацию весов с низкой стоимостью:

```
'm'
--> musorida hosking floa
--> miesxams reald 20
----
...
----
'music '
--> music tents in sauraborls
--> music kart
----
'music i'
--> music instente rairs
--> music in toff chare sive he
----
'music is'
--> music island kn5 stendattion
--> music is losting clutple
```

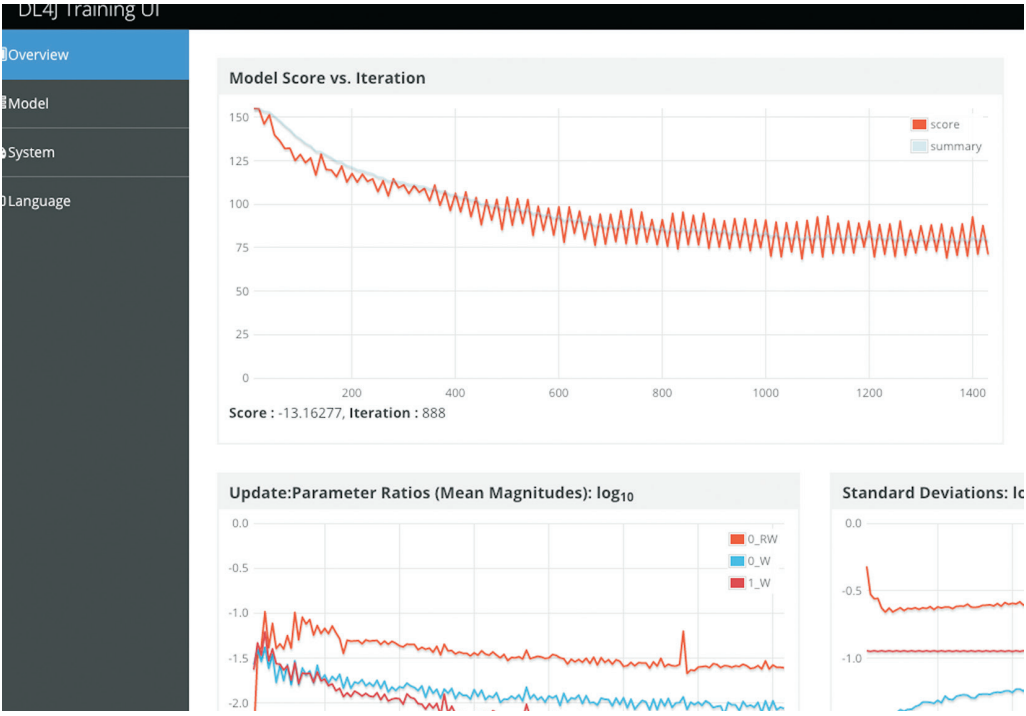


Рис. 4.10 ❖ Нейронная языковая модель LSTM на уровне символов с одним скрытым слоем (80 нейронов)

```
----
'music is '
--> music is seill butter
--> music is the amehia faches of
----
...
----
'music is my ai'
--> music is my airborty cioderopaship
--> music is my air dea a
----
'music is my air'
--> music is my air met
--> music is my air college
----
'music is my airc'
--> music is my aircentival ad distures
--> music is my aircomute in fresight op
----
'music is my aircr'
--> music is my aircrichs of nwire
--> music is my aircric of
----
'music is my aircra'
```

```
--> music is my aircrations sime
--> music is my aircracts fast
----
'music is my aircraf'
--> music is my aircraffems 2
--> music is my aircrafthons and parin
----
'music is my aircraft'
--> music is my aircrafted
--> music is my aircrafts njrmen
```

Эти результаты хуже, чем у предыдущих решений, которые не основаны на нейронных сетях! Давайте сравним результаты первой языковой модели с результатами N-граммной модели и результатами `AnalyzingSuggester`. В табл. 4.1 показано, что хотя нейронная языковая модель всегда дает результаты, многие из них не имеют особого смысла.

Таблица 4.1 Подсказчик

Ввод	Нейронная языковая модель	N-граммная модель	AnalyzingSuggester
“m”	musorida hosking floa	my	m
“music”	music tents in sauraborls	music	music
“music is”	music island kn5 stendattion	Island	music
“music is my ai”	music is my airborty cioderopaship	my aim	
“music is my aircr”	music is my aircrichs of nwire	aircraft	

Что такое «sauraborls» в подсказке «music tents in sauraborls»? И что такое «stendattion» из подсказки «music island kn5 stendattion»? По мере того как длина прогнозируемого текста увеличивается, нейронная языковая модель начинает возвращать последовательности символов, которые формируют бессмысленные слова, – ей не удается выполнить оценку правильных вероятностей для более длинных входных данных. Это именно то, чего вы ожидали после наблюдения за кривой обучения.

Вы хотите, чтобы сеть обучалась лучше, поэтому давайте посмотрим на один из самых важных параметров конфигурации при настройке обучения нейронной сети: *скорость обучения*. Скорость обучения определяет, сколько весов нейронной сети изменилось относительно потерь (градиента). Высокая скорость обучения может привести к тому, что нейронная сеть так и не найдет хороший набор весов, потому что веса меняются слишком сильно, а хорошая комбинация не найдена. Низкая скорость обучения может настолько замедлить обучение, что хороший набор весов не будет найден, пока для обучения не будут использованы все данные.

Давайте немного увеличим число нейронов в слое до 90 и начнем обучение снова:

```
...
INFO o.d.n.m.MultiLayerNetwork - Starting MultiLayerNetwork ...
INFO c.m.d.u.NeuralNetworksUtils - params :67.487, examples: 77.141
INFO o.d.o.l.ScoreIterationListener - Score at iteration 0 is 173.9821
...
```

Количество параметров нейронной сети немного меньше, чем количество доступных обучающих примеров, поэтому вам не нужно добавлять дополнительных параметров в будущем.

Когда вы закончите обучение, давайте посмотрим на результаты поиска:

```
'm'
--> month jeans of saids
--> mie free in manufact
----
'mu'
--> musications head socie
--> musican toels
----
'mus'
--> muse sc
--> muse germany nc
----
'musi'
--> musical federations
--> musicating outlet
----
'music'
--> musican 2006
--> musical swin daith program
----
'music '
--> music on the grade county
--> music of after
----
'music i'
--> music island fire grin school
--> music insurance
----
'music is'
--> music ish
--> music island recipe
----
'music is '
--> music is befied
--> music is an
----
'music is m'
--> music is michigan rup dogs
--> music is math sandthome
----
'music is my'
--> music is my labs
--> music is my less
----
'music is my '
--> music is my free
--> music is my hamby bar finance
```

```
----
'music is my a'
--> music is my acket
--> music is my appedia
----
'music is my ai'
--> music is my air brown
--> music is my air jerseys
----
'music is my air'
--> music is my air bar nude
--> music is my air ambrank
----
'music is my airc'
--> music is my airclass
--> music is my aircicle
----
'music is my aircr'
--> music is my aircraft
--> music is my aircross of mortgage choo
----
'music is my aircra'
--> music is my aircraft
--> music is my aircraft popper
----
'music is my aircraf'
--> music is my aircraft in star
--> music is my aircraft bouble
----
'music is my aircraft'
--> music is my aircraft
--> music is my aircraftless theatre
```

Качество результатов улучшилось. Многие из них состоят из правильных слов английского языка; некоторые из них даже забавные, например «music is my aircraft popper», «music is my aircraftless theatre»! Давайте еще раз посмотрим на вкладку «Обзор» только что обученной языковой модели (см. рис. 4.11).

Потери уменьшаются уже лучше, но они по-прежнему не достигли достаточно малого значения, поэтому скорость обучения, вероятно, еще не установлена правильно. Давайте ускорим этот процесс, установив более высокую скорость обучения. Ее значение было установлено на 0,1, поэтому попробуем 0,4 – очень высокое значение! На рис. 4.12 показано, что сеть снова обучается.

Результат – потери снизились, и нейронная сеть достигла этого с бóльшим количеством параметров. Это означает, что она знает больше о данных для обучения. Тут мы остановимся и решим, что довольны этими результатами.

Для оптимального обучения потребуется больше итераций; регулировка других параметров может дать более привлекательные формы и более качественные подсказки. Мы обсудим настройку нейронной сети в последней главе книги.

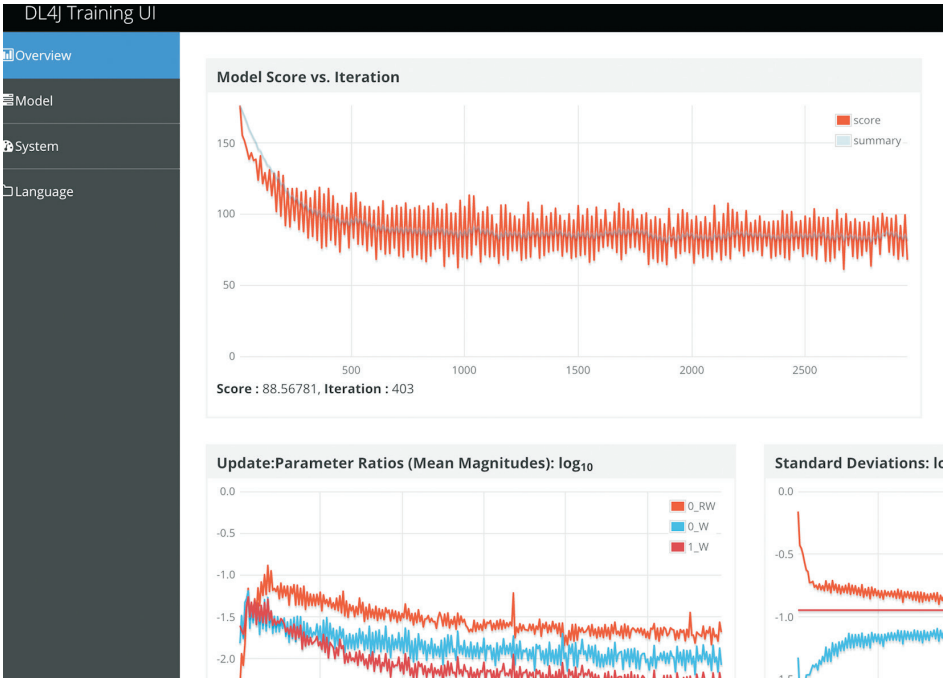


Рис. 4.11 ❖ Параметров больше, но конвергенция по-прежнему неоптимальная

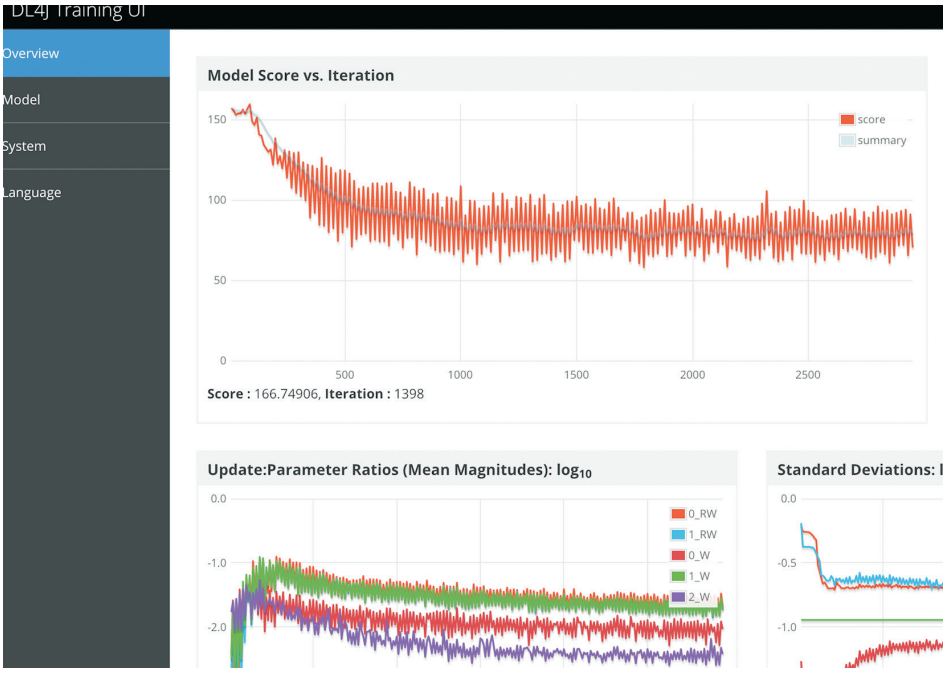


Рис. 4.12 ❖ Более высокая скорость обучения

4.9. Вносим разнообразие в подсказки, используя векторные представления слов

В главе 2 вы убедились, насколько полезно использовать векторные представления слов для расширения синонимов. В этом разделе показано, как соединить их с результатами сгенерированных LSTM-сетью подсказок, чтобы предоставить конечному пользователю более разнообразные подсказки. В продукционных системах результаты разных моделей обычно объединяют, чтобы обеспечить подходящий пользовательский опыт.

Модель word2vec позволяет создавать векторизованное представление слова. Такие векторы изучаются неглубокой нейронной сетью, глядя на окружающий контекст (другие близкие слова) каждого слова. Чудесная особенность word2vec и подобных алгоритмов векторного представления слов состоит в том, что они помещают сходные слова близко друг к другу в векторном пространстве: например, векторы, обозначающие «aircraft» и «airplane», будут находиться очень близко друг к другу.

Давайте создадим модель word2vec из индекса Lucene, содержащего текст песни, аналогично тому, как вы это делали в главе 2:

```
CharacterIterator iterator = ...
MultiLayerNetwork network = ...

FieldValuesSentenceIterator iterator = new
    FieldValuesSentenceIterator(reader, "lyrics");
Word2Vec vec = new Word2Vec.Builder()
    .layerSize(100)
    .iterate(iterator)
    .build();
vec.fit();

Lookup lookup = new CharLSTMWord2VecLookup (network,
    iterator, vec);
```

Создает DataSetIterator для содержимого поля lyrics

Выполняет конфигурацию модели word2vec, используя векторы слов размером 100

Выполняет обучение модели word2vec

Собирает нейронную языковую модель с помощью ранее обученной LSTM-сети, CharacterIterator и модель word2vec

Используя модель word2vec, обученную на тех же данных, вы теперь можете комбинировать ее с `CharLSTMNeuralLookup` и генерировать дополнительные подсказки. Вы определите `CharLSTMWord2VecLookup`, который расширяет класс `CharLSTMNeuralLookup`. Для этой реализации требуется экземпляр `Word2Vec`. Во время поиска он перебирает строку, предложенную LSTM-сетью, а затем модель word2vec используется для поиска ближайшего соседа (соседей) для каждого слова в строке. Эти ближайшие соседи используются для создания новой подсказки. Например, последовательность «music is my aircraft», сгенерированная LSTM-сетью, будет разбита на токены «music», «is», «my» и «aircraft». Модель word2vec будет проверять, например, ближайших соседей, слова «aircraft» и «aeroplane», а затем создаст дополнительную подсказку «music is my aeroplane».

Листинг 4.1 ❖ Расширенная нейронная языковая модель с Word2Vec

```
public class CharLSTMWord2VecLookup extends CharLSTMNeuralLookup {

  private final Word2Vec word2Vec;

  public CharLSTMWord2VecLookup (MultiLayerNetwork net,
        CharacterIterator iter, Word2Vec word2Vec) {
    super(net, iter);
    this.word2Vec = word2Vec;
  }

  @Override
  public List<LookupResult> lookup(CharSequence key, Set<BytesRef> contexts,
        boolean onlyMorePopular, int num) throws IOException {
    Set<LookupResult> results = Sets.
        newCopyOnWriteArraySet(super.lookup(key,
        contexts, onlyMorePopular, num));
    for (LookupResult lr : results) {
      String suggestionString = lr.key.toString();
      for (String word : word2Vec.
            getTokenizerFactory().create(
            suggestionString).getTokens()) {
        Collection<String> nearestWords = word2Vec
            .wordsNearest(word, 2);
        for (String nearestWord : nearestWords) {
          if (word2Vec.similarity(word, nearestWord)
              > 0.7) {
            results.addAll(enhanceSuggestion(lr,
                word, nearestWord));
          }
        }
      }
    }
    return new ArrayList<>(results);
  }

  private Collection<LookupResult> enhanceSuggestion(LookupResult lr,
        String word, String nearestWord) {
    return Collections.singletonList(new LookupResult(
        lr.key.toString().replace(word, nearestWord),
         (long) (lr.value * 0.7)));
  }
}
```

Получает подсказки, сгенерированные LSTM-сетью

Делит строку подсказки на токены (слова)

Находит двух верхних ближайших соседей каждого токена

Каждый ближайший сосед проверяется на предмет того, достаточно ли он похож на входное слово

Создает улучшенную подсказку, используя слово, предложенное word2vec

Простая реализация улучшения подсказки: замена исходного слова ближайшим соседним словом

Еще в начале главы 2 пользователь хотел найти текст песни, название которой он не мог точно вспомнить. Используя модель word2vec для расширения синонимов, вы можете вернуть нужную песню, даже если запрос не соответствует названию, с помощью сгенерированных синонимов. Благодаря этой комбинации нейронной языковой модели и word2vec для генерации подсказок вам удается позволить пользователю полностью избегать поиска: пользователь вводит «music is my airc...» и получает подсказку «music is my aeroplane», поэтому фактически поиск не выполняется, при этом потребность пользователя в информации удовлетворена!

Резюме

- Поисковые подсказки важны, чтобы помочь пользователям писать хорошие запросы.
- Данные для генерации таких подсказок могут быть статическими (например, словари ранее введенных запросов) или динамическими (например, документы, хранящиеся в поисковой системе).
- Вы можете использовать анализ текста и/или N-граммные языковые модели для создания хороших алгоритмов подсказок.
- Нейронные языковые модели – это языковые модели на базе нейронных сетей, таких как PHC (или LSTM-сеть).
- Используя нейронные языковые модели, вы можете получить более приемлемые подсказки.
- Важно следить за процессом обучения нейронной сети, чтобы убедиться, что вы получаете хорошие результаты.
- Можно объединить результаты исходного подсказчика с векторами слов, чтобы сделать подсказки более разнообразными.

Глава 5

Ранжирование результатов поиска с помощью векторных представлений слов

О чем идет речь в этой главе:

- статистические и вероятностные модели поиска;
- работа с алгоритмом ранжирования в Lucene;
- нейронные модели информационного поиска;
- использование усредненных векторных представлений слов для ранжирования результатов поиска.

Со второй главы мы создаем компоненты на базе нейронных сетей, которые могут улучшить работу поисковой системы. Эти компоненты нацелены на то, чтобы помочь поисковой системе лучше понять намерения пользователя путем расширения синонимов, создания представлений альтернативных запросов и предоставления более умных подсказок, когда пользователь печатает запрос. Как показывают эти подходы, запрос можно расширить, адаптировать и преобразовать перед выполнением сопоставления с термами, хранящимися в инвертированных индексах. Затем, как упоминалось в главе 1, термы запроса используются для поиска подходящих документов.

Эти подходящие документы, также известные как *результаты поиска*, сортируются в соответствии с тем, насколько близко они прогнозируют соответствие входному запросу. Задача сортировки результатов известна как *ранжирование*. Функция ранжирования имеет фундаментальное влияние на *релевантность* результатов поиска, поэтому правильная работа с ней означает, что поисковая система будет иметь более высокую *точность*, и пользователи будут получать самую релевантную и важную информацию в первую очередь. Правильное ранжирование – это не одноразовый процесс, а, скорее, дополнительный. В реальной жизни вы будете использовать существующий алгоритм ранжирования, создавать новый или использовать комбинацию существующих и новых функций ранжирова-

ния. Вам не раз придется настраивать их так, чтобы они точно отражали то, что ищут ваши пользователи, как они пишут запросы, и т. д.

В этой главе вы узнаете о распространенных функциях ранжирования, моделях поиска информации и о том, как поисковая система «решает», какие результаты показывать первыми. Затем я покажу вам, как улучшить функции ранжирования своей поисковой системы, используя плотные векторные представления текста (слова, предложения, документы и т. д.). Векторные представления текста могут помочь вашим функциям ранжирования лучше выполнять сопоставление и оценку документов в соответствии с намерениями пользователя.

5.1. Важность ранжирования

Немного забавный мем, который какое-то время гулял по интернету, гласил: «Лучшее место, чтобы спрятать труп, – это страница Google номер 2». Конечно же, это гипербола, которая в основном относится к поиску в сети (поиск контента, например страниц веб-сайтов). Но это говорит многое о степени, в которой пользователи ожидают, что поисковые системы будут хорошо справляться с предоставлением релевантных результатов. Часто пользователю мысленно проще написать более качественный запрос, чем прокрутить страницу вниз и нажать кнопку «Страница 2» на странице результатов. Вышеприведенный мем можно перефразировать следующим образом: «Если он не появился на первой странице, он не может быть релевантным». Это объясняет, почему важна релевантность. Можно предположить, что:

- *пользователи ленивы*. Они не хотят прокручивать страницу вниз или просматривать более двух или трех результатов, прежде чем решить, являются ли результаты поиска хорошими. Зачастую возвращать результаты, количество которых исчисляется тысячами, бесполезно;
- *пользователи не информированы*. Они не знают, как работает поисковая система; они просто пишут запрос и надеются получить хорошие результаты.

Если функция ранжирования в поисковой системе работает хорошо, вы можете вернуть от 10 до 20 лучших результатов, и пользователь останется доволен. Обратите внимание, что этот подход также может оказать положительное влияние на производительность поисковой системы, поскольку пользователь не будет просматривать все документы, по которым есть совпадения.

Хотя вам, наверное, интересно, применима ли проблема релевантности во всех случаях. Например, если у вас короткий запрос, состоящий из одного или двух слов, четко идентифицирующий небольшой набор результатов поиска, проблема релевантности менее очевидна. Подумайте обо всех поисковых запросах, которые вы делали в Google, просто чтобы получить страницу Википедии. Например, представьте, что вы хотите найти страницу с описанием Бернхарда Римана. Набирать URL-адрес en.wikipedia.org, вводить `Бернхард Риманн` в текстовое поле поиска в Википедии и нажимать кнопку с изображением увеличительного стекла, чтобы получить результаты, – все это раздражает. Гораздо быстрее ввести `Бернхард Риман` в поле поиска Google, и вы, скорее всего, получите страницу Википедии в качестве первого или второго результата поиска на первой странице. Это пример, когда вы (как вам кажется) знаете заранее, что хотите найти (вы ленивы, но вас проинформировали относительно того, что вам было нужно, и из предыдущего опыта вы

знали, как обычно работает поисковая система при поиске людей). Но во многих случаях это не работает. Поставьте себя на место студента последнего курса факультета математики, который не интересуется общей информацией о Римане, а хочет понять, почему его работы считаются важными в различных областях науки. Студент заранее не знает, какие конкретные ресурсы ему нужны; он знает *тип* необходимого ресурса и на основе этого будет вводить запрос. Таким образом, такой студент может ввести запрос, например, `важность работ Бернхарда Римана` или `влияние Бернхарда Римана в научных исследованиях`. Если вы выполните два этих запроса в Google, то:

- увидите разные результаты поиска для каждого запроса;
- результаты поиска, которые появляются в обоих случаях, находятся в разном порядке.

Более того, на момент написания этих строк при первом запросе в качестве первого результата возвращается страница Википедии, тогда как первым результатом второго запроса является фраза «влияние Гербарта на Бернхарда Римана». Это странно, поскольку переворачивает то, что было нужно пользователю, с ног на голову: студент хотел узнать, как Риман влиял на других, а не наоборот (второй результат, «вклад Римана в дифференциальную геометрию», звучит гораздо более актуально). Это проблема, которая затрудняет ранжирование результатов поиска.

Давайте теперь посмотрим, как ранжирование вступает в игру в жизненном цикле запроса (см. также рис. 5.1):

1) запрос, написанный пользователем, парсится, анализируется и разбивается на набор термов (закодированный запрос);
2) закодированный запрос выполняется для структур данных поисковой системы (для каждого терма выполняется поиск в таблице инвертированного индекса);
3) совпадающие документы собираются и передаются в функцию ранжирования;
4) каждый документ оценивается функцией ранжирования;
5) как правило, список результатов поиска состоит из таких документов, отсортированных в соответствии с их оценкой в порядке убывания (первый результат имеет наивысшую оценку).

Функция ранжирования берет результаты поиска и присваивает каждому из них оценочное значение, которое является показателем его важности по отношению к входному запросу. Чем выше оценка, тем важнее документ.

Кроме того, при ранжировании результатов умная поисковая система должна учитывать следующее:

- *историю пользователя* – запишите прошлую активность пользователя и учитывайте ее при ранжировании. Например, повторяющиеся термы в прошлых запросах могут указывать на интерес пользователя к определенной теме, поэтому результаты поиска по этой же теме должны иметь более высокий рейтинг;
- *географическое местоположение пользователя* – запишите местоположение пользователя и повысьте рейтинг результатов поиска, написанных на соответствующем языке;
- *временные изменения в информации* – вспомните запрос «последние тенденции» из главы 3. Такой запрос должен соответствовать не только словам

«последние» и/или «тенденции», он также должен повышать рейтинг новых документов (более свежая информация);

- *все возможные контекстные подсказки* – ищите сигналы, чтобы предоставить запросу дополнительный контекст. Например, посмотрите журналы поиска, чтобы увидеть, делался ли запрос ранее; если да, проверьте следующий запрос в журнале поиска, чтобы увидеть, есть ли какие-нибудь общие результаты, и дайте им более высокий рейтинг.

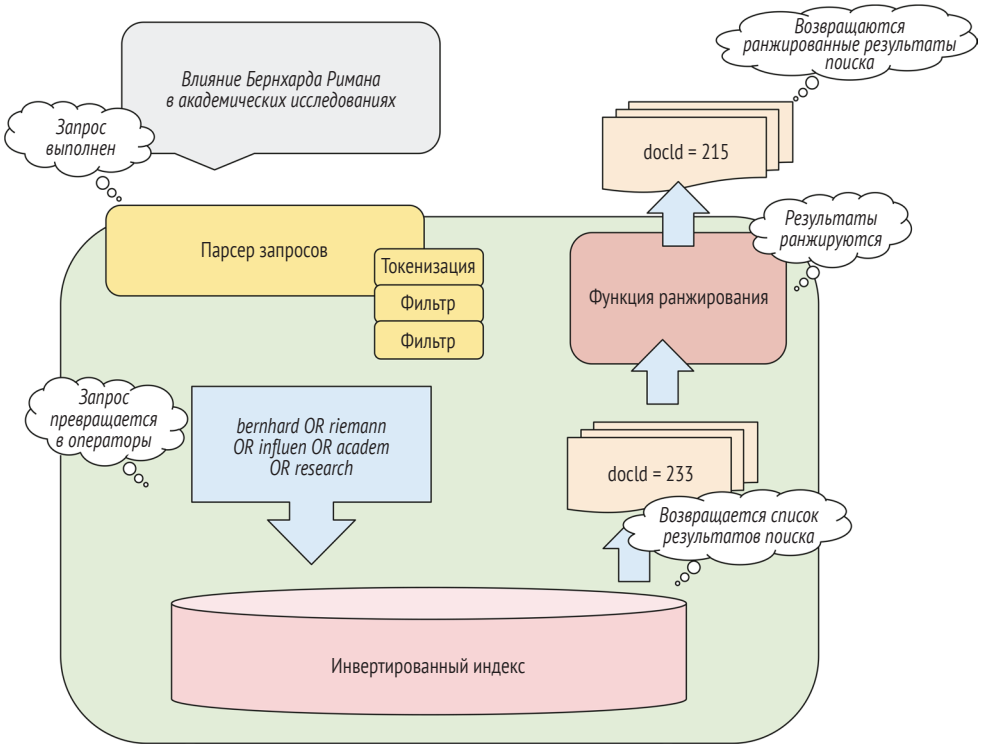


Рис. 5.1 ❖ Выполнение запросов, извлечение и ранжирование

Теперь мы займемся ответом на ключевой вопрос: как поисковая система решает, как ранжировать результаты поиска по заданному запросу?

5.2. Модели поиска

До сих пор мы говорили о задаче ранжирования документа как функции, которая принимает документ в качестве входных данных и генерирует оценочное значение, обозначающее релевантность документа. На практике функции ранжирования часто являются частью *модели поиска информации*. Такая модель определяет, как поисковая система решает проблему предоставления релевантных результатов в отношении информационной потребности: от анализа запроса до сопоставления, поиска и ранжирования результатов поиска. Обоснование наличия модели заключается в том, что трудно придумать функцию ранжирования, которая дает

точную оценку, не зная, как поисковая система обрабатывает запрос. В запросе типа «+ riemann -influenced influencing», если документ содержит термы «riemann» и «influencing», итоговая оценка должна быть комбинацией оценок для первого и второго термов (оценка = оценка (riemann) \+ оценка (influencing)); но терм «riemann» имеет ограничение обязательности (знак \+), поэтому должен давать более высокую оценку, чем «influencing», что является необязательным.

Таким образом, способ, которым поисковая система вычисляет релевантность документа в отношении запроса, влияет на дизайн и инфраструктуру поисковой системы. Начиная с главы 1 мы предполагали, что когда текст подается в поисковую систему, он анализируется и разбивается на куски, которые можно изменять в зависимости от токенизаторов и фильтров токенов. Эта цепочка анализа текста генерирует термы, которые попадают в инвертированные индексы, также известные как *постинг-листы*. Вариант использования «поиск по ключевому слову» мотивировал выбор постинг-листов для эффективного извлечения документов путем сопоставления термов. Точно так же выбор порядка ранжирования пар типа «запрос–документ» может повлиять на системные требования: например, функции ранжирования может потребоваться доступ к дополнительной информации об индексированных данных, чем просто наличие или отсутствие терма в постинг-листе. Широко используемый набор поисковых моделей, именуемых *статистическими моделями*, принимает решения о ранжировании определенного документа, основываясь на том, как часто соответствующий терм появляется в конкретном документе и во всем наборе документов.

В предыдущих главах мы уже выходили за рамки простого сопоставления термов между запросами и документами. Мы использовали расширение синонимов для генерации термов синонимов: например, во время поиска, чтобы расширить число возможных способов, с помощью которых пользователь может «сказать» одно и то же (на уровне слова). Мы расширили этот подход в главе 3, создав новые альтернативные запросы в дополнение к исходному запросу, введенному пользователем.

Вся эта работа направлена на создание поисковой системы, которая стремится понять семантику текста:

- *в случае расширения синонимов* – вводите ли вы «hello» или «hi», вы *семантически* говорите одно и то же;
- *в случае расширения альтернативного запроса* – если вы введете запрос «последние тенденции», то получите альтернативные запросы, которые написаны по-другому, но семантически близки к оригиналу.

В целом (упрощенная) идея заключается в том, что документ, который релевантен к определенному запросу, должен быть возвращен, даже если между запросом и индексированными термами нет точного соответствия. Синонимы и представления альтернативных запросов предоставляют более широкий диапазон релевантных термов запроса, которые могут совпадать с термами документа. Эти методы повышают вероятность того, что вы найдете документ с использованием семантически похожих слов или запросов. В идеальном случае поисковая система выходит за рамки соответствия термов типа «документ–запрос» и понимает потребность пользователя в информации. Исходя из этого, она будет возвращать релевантные результаты, опять же, не ограничивая поиск соответствием термов.

Создать поисковик с хорошими семантическими возможностями понимания не просто. Хорошая новость состоит в том, что, как вы убедитесь, методы, основанные на глубоком обучении, могут здорово помочь сократить разрыв между простой строкой запроса и фактическим намерением пользователя. Возьмем вектор мысли, с которым вы вкратце ознакомились в главе 3, когда разбирали модели seq2seq. Можно рассматривать его как своего рода представление намерения пользователя, которое вам нужно, чтобы выйти за рамки простого соответствия термов.

Хорошая поисковая модель должна учитывать семантику. Как вы можете себе представить, эта семантическая перспектива применима и к ранжированию документов. Например:

- при ранжировании результата, чьи совпадающие термы получены из одного из альтернативных запросов, сгенерированных LSTM-сетью, должны ли такие документы оцениваться иначе, чем документы, которые сопоставляются на основе термов из исходного запроса пользователя?
- если вы планируете использовать представления, сгенерированные посредством глубокого обучения (например, векторы мышления), чтобы понять намерения пользователя, как использовать их для получения и ранжирования результатов?

Теперь мы начнем исследование, которое коснется следующих аспектов:

- более традиционные поисковые модели;
- расширение традиционных моделей, которые используют векторные представления текста, изученные посредством нейронных сетей (это будет нашей основной целью);
- нейронные информационно-поисковые модели, которые полагаются исключительно на глубокие нейронные сети.

5.2.1. TF-IDF и модель векторного пространства

В главе 1 я упомянул меру TF-IDF и модель векторного пространства. Давайте внимательнее посмотрим на них, чтобы понять, как они работают. Основная цель функции ранжирования – присвоить оценку паре «запрос–документ». Распространенный способ измерения важности документа по отношению к запросу основан на вычислении и получении статистики для термов запроса и документа. Такие поисковые модели называются *статистическими моделями для поиска информации*.

Предположим, у вас есть запрос «bernhard riemann influence» (влияние бернхарда римана) и два полученных документа: документ1 = «riemann bernhard – life and works of bernhard riemann» (риман бернхард – жизнь и творчество бернхарда римана) и документ2 = «thomas bernhard biography – bio and influence in literature» (биография томаса бернхарда – биография и влияние в литературе). И запрос, и документы состоят из термов. Когда вы смотрите, какой из них совпал, то видите следующее:

- документ1 соответствует термам «reimann» и «bernhard». Оба терма совпали дважды;
- документ2 соответствует термам «bernhard» и «influence». Оба терма совпали один раз;
- частота терма для документа1 равна 2 для каждого совпадающего терма, а частота терма документа 2 для двух совпадающих термов равна 1;

- частота документа для слова «bernhard» составляет 2 (оно появляется в обоих документах; вы не учитываете повторяющиеся вхождения в отдельном документе). Частота документа для слова «reimann» равна 1, а частота документа для слова «influence» равна 1.

Частота терма и частота документа

Часто статистические модели сочетают в себе *частоту терма* и *частоту документа*, чтобы определить меру релевантности документа с учетом запроса. Обоснование выбора этих метрик заключается в том, что вычисление частот и статистики по термам дает вам меру того, насколько информативен каждый из них. Говоря более конкретно, количество раз, когда терм запроса появляется в документе, дает меру того, насколько уместен данный документ для данного запроса; это *частота терма.* С другой стороны, термы, которые редко появляются в индексированных данных, считаются более важными и информативными, чем более распространенные термы (такие термы, как «the» и «in», как правило, не являются информативными, поскольку они слишком распространены). Частота терма во всех проиндексированных документах называется *частотой документа.*

Если вы суммируете все частоты термов каждого совпадающего терма, для документа1 оценка будет равна 4, для документа2 – 2.

Давайте добавим документ3, содержание которого: «riemann hypothesis – a deep dive into a mathematical mystery» (гипотеза римана – глубокое погружение в математическую тайну), и сопоставим его с тем же запросом. Документ3 имеет оценку, равную 1, потому что совпадает только терм «riemann». Это плохо, поскольку документ3 более релевантен, чем документ2, хотя и не имеет отношения к влиянию Римана.

Более подходящий способ выразить ранжирование – это оценить каждый документ, используя сумму логарифмов частоты термов, деленную на логарифм частоты документа. Эта известная схема взвешивания называется TF-IDF:

вес(терм) = (1 + log(tf(терм))) * log(*N*/df(терм)),

N – количество проиндексированных документов. После добавления нового документа3 частота документа для терма «riemann» теперь равна 2. Используя предыдущее уравнение для каждого совпадающего терма, вы добавляете TF-IDF и получаете следующие оценки:

оценка(документ1) = tf-idf(*riemann*) \+ tf-idf(*bernhard*) = 1.28 \+ 1.28 = 2.56
оценка(документ 2) = tf-idf(*bernhard*) \+ tf-idf(*infuence*) = 1 \+ 1 = 2
оценка(документ 3) = tf-idf(*riemann*) = 1

Вы только что видели, что оценка на базе TF-IDF основана только на чистых частотах термов, поэтому документ, который не является релевантным (документ2), оценивается выше, чем какой-либо соответствующий документ (документ3). Это тот случай, когда в модели поиска отсутствует семантическое понимание намерения запроса, что обсуждалось в предыдущем разделе.

До сих пор в этой книге вы много раз встречались с векторами. Использование их в поиске информации не является новой идеей; модель векторного простран-

ства полагается на представление запросов и документов в качестве векторов и измеряет их сходство на основе схемы взвешивания TF-IDF. Каждый документ может быть представлен одномерным вектором с размером, равным количеству существующих термов в индексе. Каждая позиция в векторе представляет терм, имеющий значение, равное значению TF-IDF этого документа данного терма.

То же самое можно сделать для запросов, потому что они также состоят из термов; единственное отличие заключается в том, что частоты термов могут быть либо локальными (частота термов запроса в том виде, в каком они появляются в запросе), либо из индекса (частота термов запроса в том виде, в каком они появляются в проиндексированных данных). Таким образом, вы представляете документы и запросы в виде векторов. Это представление носит название *мешок слов*, поскольку информация о позициях термов теряется – каждый документ или запрос представлен в виде набора слов, как в табл. 5.1.

Таблица 5.1. Мешок слов

Термы	bernhard	bio	dive	hypothesis	in	influence	into	life	mathematical	riemann
документ1	1.28	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.28
документ2	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0
документ3	0.0	0.0	1.0	1.0	0.0	0.0	1.0	0.0	1.0	1.0

Векторы «bernhard riemann influence» и «riemann influence bernhard» выглядят совершенно одинаково: факты того, что два запроса различаются и первый запрос является более значимым, чем второй, не зафиксированы. Теперь, когда документы и запросы представлены в векторном пространстве, вам нужно вычислить, какой документ лучше всего соответствует входному запросу. Это делается путем вычисления *косинусного сходства* между каждым документом и входным запросом; это даст вам окончательное ранжирование для каждого документа. Косинусное сходство является мерой амплитуды угла между документом и векторами запроса. На рис. 5.2 показаны векторы входного запроса, документ1 и документ2 в (упрощенном, двумерном) векторном пространстве, в котором рассматриваются только термы «bernhard» и «riemann».

Сходство между вектором запроса и документом оценивается путем рассмотрения существующего угла между двумя векторами. Чем меньше угол, тем больше сходство двух векторов. Применяя это к векторам из табл. 5.1, вы получаете следующие оценки сходства:

```
cosineSimilarity(query,doc1) = 0.51
cosineSimilarity(query,doc2) = 0.38
cosineSimilarity(query,doc3) = 0.17
```

При наличии всего трех документов размер результирующего вектора равен 10 (количество столбцов равно количеству термов, используемых в документах). В продукционных системах это значение будет намного выше. Таким образом, одна проблема с представлением мешка слов состоит в том, что размер векторов растет линейно с количеством существующих термов (все отдельные слова, содержащиеся в проиндексированных документах). Это еще одна причина, по которой векторы слов, подобные тем, что генерируются word2vec, лучше, чем векторы мешка слов. Сгенерированные word2vec векторы имеют фиксированный размер, поэтому они

не растут с количеством термов в поисковой системе; следовательно, потребление ресурсов намного ниже при их использовании. (Сгенерированные Word2 векторы лучше справляются с захватом семантики слов, как описано в главе 2.)

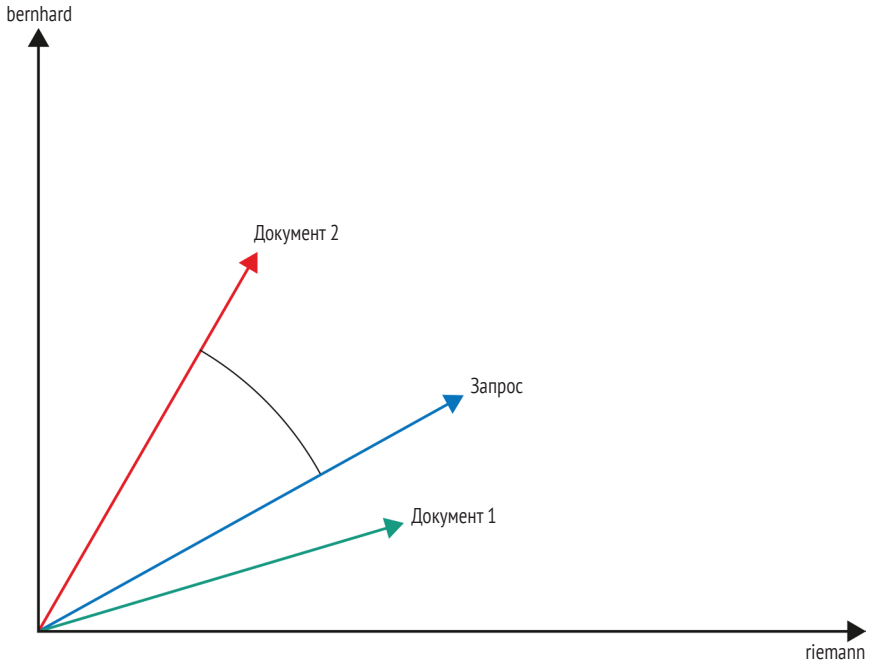


Рис. 5.2 ❖ Косинусное сходство

Несмотря на эти ограничения, модель векторного пространства и мера TF-IDF часто используются с хорошими результатами во многих продукционных системах. Прежде чем обсуждать другие модели поиска информации, давайте действовать прагматично. Мы воспользуемся Lucene и посмотрим, как эти документы ранжируются с помощью TF-IDF и векторной модели.

5.2.2. Ранжирование документов в Lucene

В Lucene API `Similarity` служит базой для функций ранжирования. Lucene поставляется с моделями поиска информации «из коробки», такими как модель векторного пространства с TF-IDF (которая использовалась по умолчанию до версии 5), Okapi BM25, DFR (Divergence from Randomness), языковые модели и другие. `Similarity` нужно установить на время индексации и время поиска. В Lucene 7 сходство VSM + TF-IDF – это `Classic-Similarity`.

Во время индексации `Similarity` устанавливается в `IndexWriterConfig`:

```
IndexWriterConfig config = new IndexWriterConfig();          ← Создает конфигурацию для индексации
config.setSimilarity(new ClassicSimilarity());               ← Устанавливает ClassicSimilarity
IndexWriter writer = new IndexWriter(directory, config);     ← Создает IndexWriter, используя
                                                               сконфигурированное сходство
```

Во время поиска Similarity устанавливается в IndexSearcher:

```
IndexReader reader = DirectoryReader.open(directory);
IndexSearcher searcher = new IndexSearcher(reader);
searcher.setSimilarity(new ClassicSimilarity);
```

Открывает IndexReader
Создает IndexSearcher
Устанавливает Similarity в IndexSearcher

Если вы выполнили индексирование и поиск по первым трем документам, то можете увидеть, ведет ли ранжирование себя так, как и ожидалось:

```
FieldType fieldType = ...
Document doc1 = new Document();
doc1.add(new Field("title",
    "riemann bernhard - life and works of bernhard riemann", ft));
Document doc2 = new Document();
doc2.add(new Field("title",
    "thomas bernhard biography - bio and influence in literature", ft));
Document doc3 = new Document();
doc3.add(new Field("title",
    "riemann hypothesis - a deep dive into a mathematical mystery", ft));
writer.addDocument(doc1);
writer.addDocument(doc2);
writer.addDocument(doc3);
writer.commit();
```

Вы можете определить свойства поля Lucene самостоятельно (сохранение значений, сохранение позиций термов и т. д.)
Для каждого документа создает новый документ и добавляет содержимое в поле title
Добавляет все три документа и фиксирует изменения

Чтобы проверить, как каждый результат поиска ранжируется классом Similarity по отношению к запросу, вы можете попросить Lucene «объяснить» его. Вывод объяснения состоит из текста, описывающего, как каждый совпадающий терм влияет на итоговую оценку каждого результата поиска:

```
String queryString = "bernhard riemann influence";
QueryParser parser = new QueryParser("title", new WhitespaceAnalyzer());
Query query = parser.parse(queryString);
TopDocs hits = searcher.search(query, 3);
for (int i = 0; i < hits.scoreDocs.length; i++) {
  ScoreDoc scoreDoc = hits.scoreDocs[i];
  Document doc = searcher.doc(scoreDoc.doc);
  String title = doc.get("title");
  System.out.println(title + " : " + scoreDoc.score);
  System.out.println("--");
  Explanation explanation = searcher.explain(query, scoreDoc.doc);
  System.out.println(explanation);
}
```

Пишет запрос
Анализирует введенный пользователем запрос
Выполняет поиск
Выводит заголовок документа и оценку для стандартного вывода
Получает объяснение того, как была рассчитана оценка

В случае с ClassicSimilarity вы получаете следующее «объяснение»:

```
riemann bernhard - life and works of bernhard riemann : 1.2140384
--
1.2140384 = sum of:
  0.6070192 = weight(title:bernhard in 0) [ClassicSimilarity], result of:
    0.6070192 = fieldWeight in 0, product of:
      ...
  0.6070192 = weight(title:riemann in 0) [ClassicSimilarity], result of:
```

```
    0.6070192 = fieldWeight in 0, product of:
      ...
--
thomas bernhard biography - bio and influence in literature : 0.9936098
--
0.9936098 = sum of:
  0.42922735 = weight(title:bernhard in 1) [ClassicSimilarity], result of:
    0.42922735 = fieldWeight in 1, product of:
      ...
  0.56438243 = weight(title:influence in 1) [ClassicSimilarity], result of:
    0.56438243 = fieldWeight in 1, product of:
      ...
--
riemann hypothesis - a deep dive into a mathematical mystery : 0.4072008
--
0.4072008 = sum of:
  0.4072008 = weight(title:riemann in 2) [ClassicSimilarity], result of:
    0.4072008 = fieldWeight in 2, product of:
     ...
```

Как и ожидалось, ранжирование учитывает то, что было описано в предыдущем разделе. Из объяснения видно, что каждый терм, соответствующий запросу, вносит свой вклад в соответствии с его весом:

```
0.9936098 = sum of:
  0.42922735 = weight(title:bernhard in 1)...
  0.56438243 = weight(title:influence in 1)...
```

С другой стороны, оценки не совсем такие же, как при ручном вычислении весов TF-IDF для термов. Причина состоит в том, что существует много возможных вариантов схем TF-IDF. Например, здесь Lucene вычисляет обратную частоту документа как

$$\log(N+1)/(\text{df}(\text{терм})+1)$$

вместо

$$\log(N/\text{df}(\text{терм}))$$

Кроме того, Lucene не принимает логарифм частоты терма, а использует частоту терма как есть. Lucene также использует *нормализацию*, метод, позволяющий смягчить тот факт, что документы с большим количеством термов будут ранжироваться слишком высоко, по сравнению с короткими документами (с меньшим количеством термов), что приблизительно может быть равно `1,0/Math.sqrt(числоТермов)`). Используя метод нормализации, вычисление косинусного сходства между вектором запроса и вектором документа эквивалентно вычислению их скалярного произведения:

```
score(query,document1) = tf-idf(query, bernhard) * tf-idf(document1,bernhard)
    + tf-idf(query, riemann) * tf-idf(document, riemann)
```

Lucene не хранит векторы. Достаточно иметь возможность вычислить TF-IDF для каждого совпадающего терма и объединить результаты для вычисления оценки.

5.2.3. Вероятностные модели

Вы узнали о модели векторного пространства и о том, как она применяется на практике в Lucene. Вы также видели, как рассчитываются оценки с использованием статистики по термам. В этом разделе вы узнаете о вероятностных моделях поиска, где оценки по-прежнему рассчитываются на основе вероятностей. Поисковая система ранжирует документ по вероятности релевантности по отношению к запросу.

Вероятности являются мощным инструментом для устранения неопределенности. Мы уже обсуждали, насколько сложно преодолеть разрыв между намерениями пользователя и релевантными результатами поиска. Вероятностные модели пытаются моделировать ранжирование путем измерения вероятности того, насколько определенный документ релевантен по отношению к входному запросу. Если вы бросаете шестигранный кубик, у каждой стороны будет 1/6 вероятности быть результатом: например, вероятность броска 3 равна P (3) = 0,16. Но на практике, если вы бросите кубик шесть раз, то, вероятно, не получите все шесть разных результатов. Вероятность – это оценка вероятности того, что определенное событие может произойти – это не значит, что оно будет происходить так часто.

Безусловная вероятность выпадения любого числа на кубике равна 1/6, но как насчет вероятности выпадения двух одинаковых результатов подряд? Такая условная вероятность может быть выражена как P (событие | условие). В случае с ранжированием можно оценить вероятность того, что определенный документ является релевантным (по отношению к заданному запросу). Это обозначается как P (r = 1 | x), где r – бинарная мера релевантности:

r = 1: релевантно, r = 0: не релевантно.

В вероятностной модели поиска вы обычно ранжируете все документы по заданному запросу по P (r = 1 | x). Лучше всего это выражается *принципом ранжирования по вероятности*: если извлеченные документы упорядочены по уменьшению вероятности релевантности доступным данным, то эффективность системы является наилучшей, которую можно получить для этих данных.

Одна из наиболее известных и широко распространенных вероятностных моделей – это *Okapi BM25*. Говоря кратко, она пытается смягчить два ограничения меры TF-IDF:

- ограничить влияние частоты термов, чтобы избежать чрезмерного ранжирования на основе часто повторяющихся термов;
- предоставить более точную оценку важности частоты документа определенного терма.

BM25 выражает условную вероятность P (r = 1 | x) посредством двух вероятностей, которые зависят от частот термов. Таким образом, BM25 аппроксимирует вероятности путем расчета распределения вероятностей по частотам термов.

Рассмотрим пример с «bernhard riemann influence». В классической схеме TF-IDF высокая частота терма может привести к высокой оценке. Таким образом, если у вас есть фиктивный документ4, который содержит большое количество вхождений «bernhard» («bernhard bernhard bernhard bernhard bernhard bernhard bernhard bernhard bernhard bernhard»), его оценка может быть выше, чем у более

актуальных документов. Если вы проиндексируете его в ранее созданном индексе, то получите следующие результаты, используя меру TF-IDF и модель векторного пространства (`ClassicSimilarity`):

```
riemann bernhard - life and works of bernhard riemann : 1.2888055
bernhard bernhard bernhard bernhard bernhard bernhard ... : 1.2231436
thomas bernhard biography - bio and influence in literature : 1.0464782
riemann hypothesis - a deep dive into a mathematical mystery : 0.47776502
```

Как видите, фиктивный документ возвращается в качестве второго результата, что странно. Кроме того, оценка документа4 почти равна первому результату: поисковая система оценила этот фиктивный документ как важный, но это не так. Давайте установим в Lucene `BM25Similarity` (по умолчанию начиная с версии 6), используя тот же код, что и для тестов `ClassicSimilarity`:

```
searcher.setSimilarity(new BM25Similarity ());
```

После этого ранжирование выглядит следующим образом:

```
riemann bernhard - life and works of bernhard riemann : 1.6426628
thomas bernhard biography - bio and influence in literature : 1.5724708
bernhard bernhard bernhard bernhard bernhard bernhard ... : 0.9965918
riemann hypothesis - a deep dive into a mathematical mystery : 0.68797445
```

Фиктивный документ занимает третье место вместо второго. Хотя это не является оптимальным, оценка значительно снизилась, по сравнению с наиболее релевантным документом. Причина в том, что BM25 «сдавливает» частоту терма, чтобы поддерживать ее ниже определенного настраиваемого порога. В этом случае BM25 смягчил влияние высокой частоты для терма «bernhard».

Еще один положительный момент касательно BM25 заключается в том, что она пытается оценить вероятность появления термов в документе. Частота документа количества термов в документе определяется суммой журналов вероятности появления каждого отдельного терма в этом документе.

Но BM25 также имеет некоторые ограничения:

- как и TF-IDF, BM25 – это модель «мешок слов», поэтому при ранжировании не учитывает порядок термов;
- хотя в целом она работает хорошо, BM25 основана на эвристике (функции, достигающие довольно хорошего результата, но в целом их работа не гарантирована), которая может не очень хорошо применяться к вашим данным (возможно, вам придется скорректировать эту эвристику);
- BM25 выполняет аппроксимацию и упрощение при оценке вероятности, что в некоторых случаях приводит к менее приемлемым результатам (она плохо работает с длинными документами).

Другие вероятностные подходы к ранжированию, основанные на языковых моделях, обычно лучше подходят при простых оценках вероятности, но это не всегда приводит к более качественному ранжированию. В общем, BM25 – это хорошая базовая функция ранжирования.

Теперь, когда мы изучили некоторые наиболее часто используемые модели ранжирования для поисковых систем, давайте рассмотрим, как нейронные сети могут помочь нам улучшить эти модели и предоставить совершенно новые (и лучшие) модели ранжирования.

5.3. Поиск информации на базе нейронных сетей

До сих пор мы решали проблему эффективного ранжирования, рассматривая термы и их локальную (на документ) и глобальную (на коллекцию) частоты. Если вы хотите использовать нейронные сети для получения более подходящей функции ранжирования, вам нужно мыслить с точки зрения векторов. На самом деле это относится не только к нейронным сетям. Вы видели, что даже классическая модель векторного пространства рассматривает документы и запросы как векторы и измеряет их сходство, используя косинусное расстояние. Одна из проблем заключается в том, что размер таких векторов может сильно возрастать (линейно) с количеством проиндексированных слов.

По того как появился поиск информации на базе нейронных сетей, были разработаны другие методы, обеспечивающие более компактное (фиксированное) представление слов. Главным образом они основывались на алгоритмах матричной факторизации, таких как алгоритм *латентно-семантической индексации*, которая основана на *сингулярном разложении*. Говоря кратко, в латентно-семантической индексации вы создаете матрицу термов и документов для каждой строки документа: ставите 1 в каждом элементе, где документ содержит соответствующий терм, и 0 для всех остальных. Затем вы преобразовываете (выполняете разложение) эту разреженную матрицу (множество нулей) с помощью сокращенного метода сингулярного разложения, получив в результате три (более плотные) матрицы, произведение которых является хорошим приближением к оригиналу. Каждая результирующая строка документа имеет фиксированную размерность и больше не является разреженной. Векторы запроса также можно преобразовать с использованием разложенных SVD-матриц. (Несколько похожий метод называется *латентным размещением Дирихле*.) Суть здесь состоит в том, что совпадения термов не требуется; векторы запросов и документов сравниваются, и наиболее схожие векторы документов ранжируются первыми.

Изучение хороших представлений данных – одна из задач, которая глубокому обучению удается лучше всего. Теперь мы рассмотрим использование подобных векторных представлений для ранжирования. Вы уже знакомы с алгоритмом, который мы будем использовать, – word2vec, – который изучает распределенные представления слов. Векторы слов располагаются близко друг к другу, когда слова, которые они представляют, появляются в сходных контекстах и, следовательно, имеют сходную семантику.

5.4. От векторов слов к векторам документов

Давайте приступим к созданию поисковой системы, основанной на векторах, сгенерированных word2vec. Цель состоит в том, чтобы ранжировать документы по запросам, но word2vec дает векторы для слов, а не для их последовательности. Поэтому первое, что нужно сделать, – это найти способ использовать эти векторы слов для обозначения документов и запросов. Запрос обычно состоит из более чем одного слова, как и проиндексированные документы. Например, давайте возьмем векторы слов для каждого из термов в запросе «влияние Бернхарда Римана» и нанесем их на график, как показано на рис. 5.3.

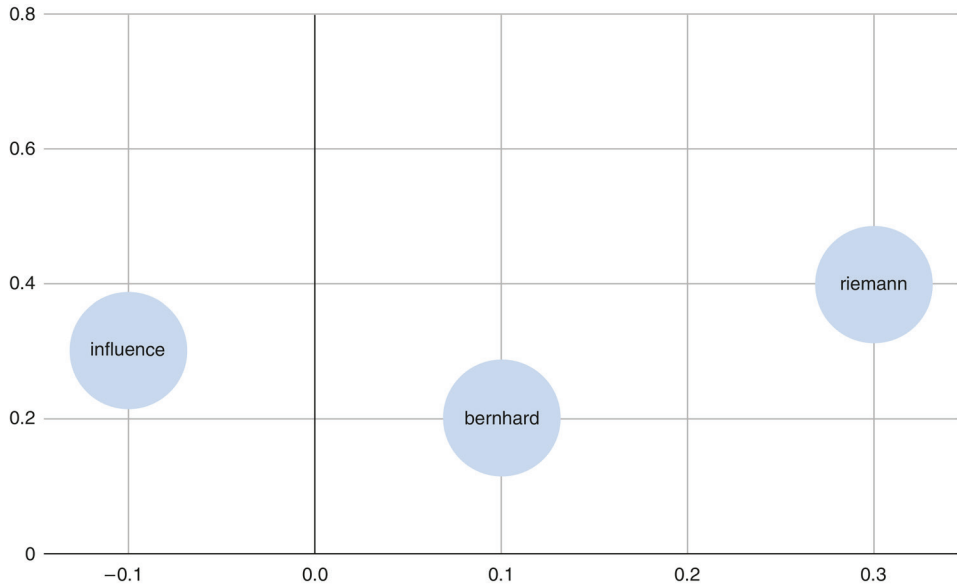


Рис. 5.3 ❖ Векторы слов «Bernhard», «Riemann» и «influence»

Простой метод создания векторов документов из векторов слов – это усреднение векторов слов в единый вектор документа. Это простая математическая операция: добавляется каждый элемент в позиции *j* в каждом векторе, а затем сумма делится на количество усредняемых векторов (это то же самое, что и операция арифметического усреднения). Это можно сделать с помощью векторов DL4J (объектов `INDArrays`) следующим образом:

```
public static INDArray toDenseAverageVector(Word2Vec word2Vec,
    String... terms) {
  return word2Vec.getWordVectorsMean(Arrays.asList(terms));
}
```

Вектор `mean` является результатом операции усреднения. На рис. 5.4, как и ожидалось, средний вектор находится в центре трех векторов слов.

Обратите внимание, что этот метод может применяться как к документам, так и к запросам, потому что они представляют собой композиции слов. Для каждой пары «документ–запрос» можно вычислить векторы документа путем усреднения векторов слов, а затем назначить оценку каждому документу на основе того, насколько близки их соответствующие усредненные векторы слов. Это похоже на то, что вы делали в случае с моделью векторного пространства; отличие состоит в том, что значения этих векторов документов не рассчитываются с использованием TF-IDF, а получены из усреднения векторов word2vec. Таким образом, эти плотные векторы менее тяжелые с точки зрения требуемой памяти (и места, если они хранятся на диске) и более информативны с точки зрения семантики.

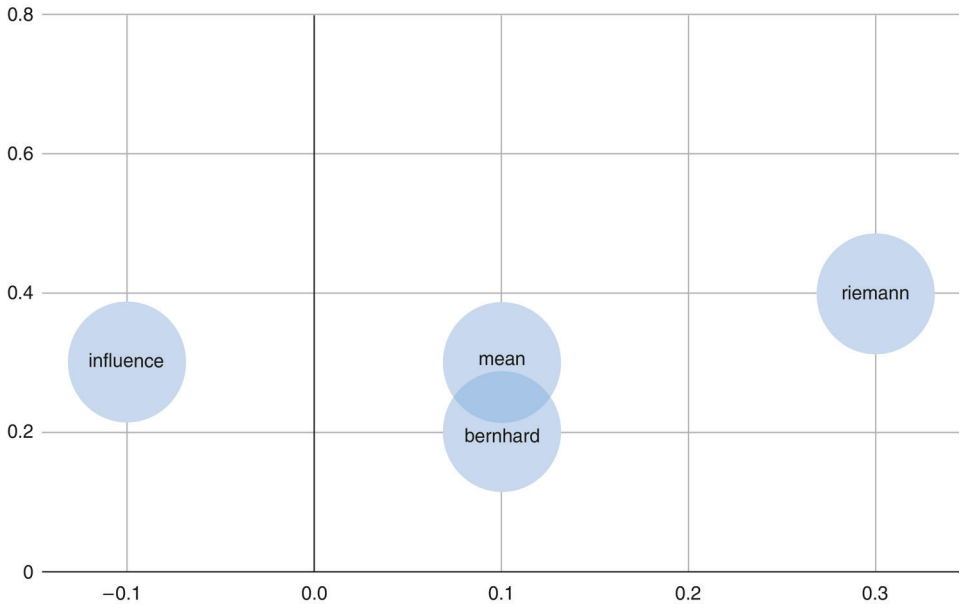


Рис. 5.4 ❖ Усреднение векторов слов «Bernhard», «Riemann» и «influence»

Давайте повторим предыдущий эксперимент, но будем ранжировать документы, используя усредненные векторы слов. Сначала вы вводите данные word2vec из поисковой системы:

```
IndexReader reader = DirectoryReader.open(
    directory);
```
← Создает ридер для набора документов поисковой системы

```
FieldValuesSentenceIterator iterator = new
    FieldValuesSentenceIterator(reader, "title");
```
← Создает итератор DL4J, который может читать данные из ридера в поле title

```
Word2Vec vec = new Word2Vec.Builder()
```
← Настраивает word2vec

```
  .layerSize(3)
  .windowSize(3)
  .tokenizerFactory(new DefaultTokenizerFactory())
  .iterate(iterator)
  .build();
```
← Вы работаете с очень маленьким набором данных, поэтому используете очень маленькие векторы

```
vec.fit();
```
← Позволяет word2vec изучать векторы слов

После того как вы извлекли векторы слов, вы можете создать векторы запросов и документов:

Массив, содержащий термы, введенные в запросе («бернхард», «риман» и «влияние»)

```
String[] terms = ...
```

```
INDArray queryVector = toDenseAverageVector(vec,
    terms);
```
← Преобразует термы запроса в вектор запроса путем усреднения векторов слов термов запроса

```
for (int i = 0; i < hits.scoreDocs.length; i++) {
  ScoreDoc scoreDoc = hits.scoreDocs[i];
  Document doc = searcher.doc(scoreDoc.doc);
```
← Для каждого результата поиска: игнорирует оценку, указанную Lucene, и преобразует результаты в векторы документов

```
  String title = doc.get("title");
  Terms docTerms = reader.getTermVector(scoreDoc.doc,
    "title");
  INDArray denseDocumentVector = VectorizeUtils
    .toDenseAverageVector(docTerms, vec);
  double sim = Transforms.cosineSim(denseQueryVector,
    denseDocumentVector)
  System.out.println(title + " : " + sim);
}
```

Получает заголовок документа

Извлекает термы, содержащиеся в этом документе (используя IndexReader#getTermVector API)

Преобразует термы документа в вектор документа, используя метод усреднения, показанный ранее

Вычисляет косинусное сходство между векторами запроса и документа и выводит его

Для удобства чтения результаты показаны, начиная с наивысшего значения:

```
riemann hypothesis - a deep dive into a
    mathematical mystery : 0.6171551942825317
thomas bernhard biography - bio and influence
    in literature : 0.4961382746696472
bernhard bernhard bernhard bernhard bernhard
    bernhard ... : 0.32834646105766296
riemann bernhard - life and works of bernhard
    riemann : 0.2925628423690796
```

Документ, идущий в самом начале, является релевантным, независимо от частоты терма

Второй документ не является релевантным относительно использования

Фиктивный документ

(Вероятно) наиболее релевантный документ имеет самый низкий балл

Это странно: метод, который, как вы ожидали, поможет вам получить более качественное ранжирование, оценил фиктивный документ лучше, чем самый релевантный! Причины этого следующие:

- word2vec недостаточно обучающих данных, чтобы предоставить векторы слов, которые тщательно представляют семантику слов. Четыре коротких документа содержат слишком мало пар «слово–контекст» для нейронной сети word2vec, чтобы точно отрегулировать веса скрытого слоя;
- если вы выберете вектор документа с наивысшим рейтингом, он будет равен вектору слова «bernhard». Вектор запроса представляет собой средний вектор векторов для «бернхард», «риман» и «влияние»; следовательно, эти векторы всегда будут близко друг к другу в векторном пространстве.

Давайте визуализируем второе утверждение путем построения сгенерированных векторов типа «запрос–документ» в (уменьшенном) двумерном пространстве: *см.* рис. 5.5. Как и ожидалось, векторные представления документа4 и запроса настолько близки, что их метки почти перекрываются.

Один из способов улучшить эти результаты – убедиться, что алгоритм word2vec содержит больше обучающих данных. Например, можно начать с английского дампа из Википедии и проиндексировать заголовок и содержание каждой страницы в Lucene. Кроме того, можно уменьшить влияние текстовых фрагментов, таких как фрагмент документа4, содержащих в основном (или только) отдельные термы, которые также появляются в запросе. Обычный способ сделать это – сгладить усредненные векторы документа, используя частоту терма. Вместо того чтобы делить каждый вектор слова на длину документа, вы делите его на его частоту терма в соответствии с приведенным ниже псевдокодом:

```
documentVector(wordA wordB) = wordVector(wordA)/termFreq(wordA) +
    wordVector(wordB)/termFreq(wordB)
```

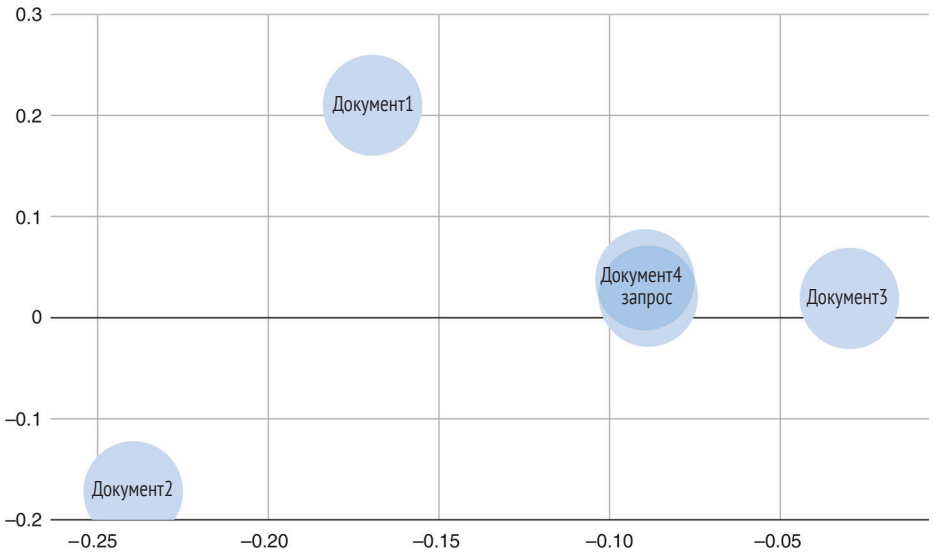


Рис. 5.5 ❖ Сходство между векторными представлениями запросов и документов

Это может быть реализовано в Lucene и DL4J следующим образом:

```
public static INDArray toDenseAverageTFVector(Terms docTerms, Terms
    fieldTerms, Word2Vec word2Vec) throws IOException {
  INDArray vector = Nd4j.zeros(word2Vec
    .getLayerSize());
  TermsEnum docTermsEnum = docTerms.iterator();
  BytesRef term;
  while ((term = docTermsEnum.next()) != null) {
    long termFreq = docTermsEnum.totalTermFreq();
    INDArray wordVector = word2Vec.getLookupTable().
        vector(term.utf8ToString()).div(termFreq);
    vector.addi(wordVector);
  }
  return vector;
}
```

Инициализирует все значения вектора до нуля

Перебирает все существующие термы текущего документа

Получает следующий терм

Получает значение частоты текущего терма

Извлекает векторное представление слова для текущего терма, а затем делит его значения на частоту терма

Суммирует текущий вектор текущего терма с вектором, который будет возвращен

Когда я познакомил вас с усредненными векторами слов, вы увидели, что такие векторы документов располагаются прямо в центре их составных векторов слов. На рис. 5.6 видно, что сглаживание по частоте терма может помочь отделить сгенерированные векторы документов от центрального расположения в векторах слов ближе к менее частому (и, надеюсь, более важному) слову.

Термы «бернхард» и «риман» встречаются чаще, чем «влияние», а сгенерированный вектор документа `tf` ближе к вектору слова «влияние». Это имеет положительное влияние: документы с низкой частотой терма имеют более высокий рейтинг, но по-прежнему находятся достаточно близко к вектору запроса:

```
riemann hypothesis - a deep dive into a mathematical
    mystery : 0.6436703205108643
```

```
thomas bernhard biography - bio and influence in
    literature : 0.527758002281189
riemann bernhard - life and works of bernhard
    riemann : 0.2937617599964142
bernhard bernhard bernhard bernhard bernhard
    bernhard ...: 0.2569074332714081
```

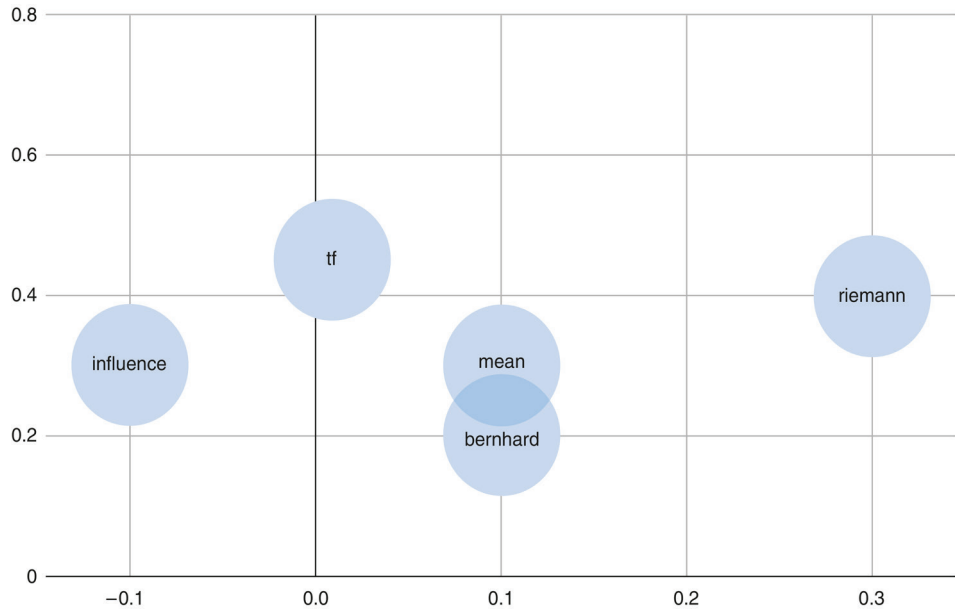


Рис. 5.6 ❖ Усредненный вектор слов, сглаженный частотами термов

Впервые фиктивный документ получает самый низкий балл. Если вы переключитесь с частоты простого терма на TF-IDF в качестве коэффициентов сглаживания для генерации усредненных векторов документов из векторных представлений слов, то получите такое ранжирование:

```
riemann hypothesis - a deep dive into a mathematical
    mystery : 0.7083401679992676
riemann bernhard - life and works of bernhard
    riemann : 0.4424433362483978
thomas bernhard biography - bio and influence in
    literature : 0.3514146476984024
bernhard bernhard bernhard bernhard bernhard
    bernhard ... : 0.09490833431482315
```

Благодаря сглаживанию на базе TF-IDF (*см.*, например, рис. 5.7) ранжирование документов – лучшее, чего вы можете добиться. Вы избежали строгого сходства, основанного на взвешивании термов: наиболее релевантный документ имеет частоту 1 для терма «риман», тогда как документ с самой высокой частотой терма имеет наименьшую оценку. С семантической точки зрения наиболее важные документы оцениваются выше, чем другие.

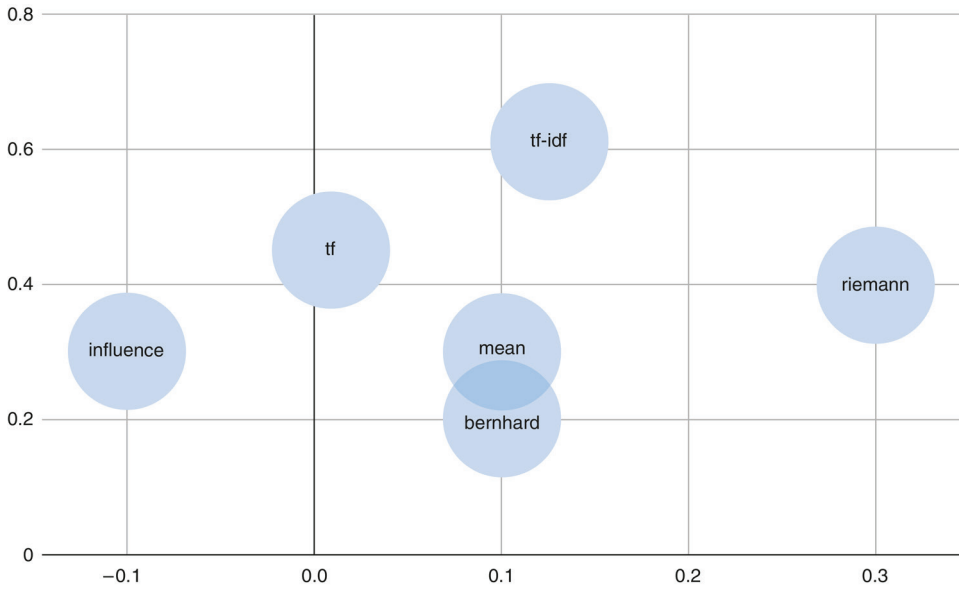


Рис. 5.7 ❖ Усредненный вектор слов, сглаженный с помощью TF-IDF

5.5. Оценки и сравнения

Довольны ли вы этим средством ранжирования документов на основе усредненных векторов слов с использованием TF-IDF? В предыдущем примере вы обучили word2vec, используя конкретные настройки: размер слоя, равный 60, модель skip-gram, размер окна, равный 6, и т. д. Ранжирование было оптимизировано с учетом конкретного запроса и набора из четырех документов. Хотя это полезное упражнение для изучения плюсов и минусов различных подходов, нельзя выполнить такую детальную оптимизацию для всех возможных входных запросов, особенно для больших баз знаний. Принимая во внимание то, что релевантность очень трудно понять, было бы неплохо найти способы автоматизировать оценку эффективности ранжирования. Поэтому, прежде чем перейти к другим способам работы с ранжированием (например, с помощью векторных представлений текста на базе нейронных сетей), давайте быстро познакомимся с инструментами для ускорения оценки функций ранжирования.

Хорошим инструментом для оценки эффективности поисковых систем на базе Lucene является Lucene for Information Retrieval (Lucene4IR). Он появился в результате сотрудничества лиц из исследовательских и отраслевых кругов[1]. Краткое руководство можно найти по адресу **http://mng.bz/YP7N**. Lucene4IR позволяет опробовать различные стратегии индексирования, поиска и ранжирования, по сравнению со стандартными наборами данных для поиска информации. Чтобы испробовать его, можно последовательно запустить `IndexerApp`, `RetrievalApp`

[1] *См.*: *Лейф Аццопарди и др.* Lucene4IR: Developing Information Retrieval Evaluation Resources using Lucene // ACM SIGIR Forum 50. 2016. December. № 2 (**http://sigir.org/wp-content/uploads/2017/01/p058.pdf**).

и ExampleStatsApp. Это приведет к индексации, поиску и записи статистики по возвращенным и соответствующим результатам: например, в соответствии с выбранной конфигурацией Lucene (Similarity, Analyzers и т. д.). По умолчанию эти приложения запускаются в наборе данных CACM (**http://mng.bz/GWZq**) с использованием BM25Shapsity.

После того как вы выполнили оценку данных с помощью инструментов Lucen4IR, вы можете измерить точность, полноту и другие метрики, используя утилиту trec_eval (разработана для измерения качества результатов поиска по данным из серии конференций TREC; **http://trec.nist.gov**). Вот пример вывода терминала trec_eval в наборе данных CACM с использованием ранжирования BM25:

```
./trec_eval ~/lucene4ir/data/cacm/cacm.qrels
    ~/lucene4ir/data/cacm/bm25"results.res
...
num_q           all 51      <-- Количество выполненных запросов
num_ret         all 5067    <-- Количество возвращенных результатов
num_rel         all 793     <-- Количество релевантных результатов
num_rel_ret     all 341     <-- Количество возвращенных результатов, которые также являются релевантными
map             all 0.2430  <-- Средняя точность
Rprec           all 0.2634  <-- R-точность
P_5             all 0.3608  | P_5, P_10 и т. д. дают точность на 5, 10 и т. д.
P_10            all 0.2745  | найденных документах
```

Если вы измените параметр Similarity в файле конфигурации Lucene4IR и снова запустите RetrievalApp и ExampleStatsApp, то сможете наблюдать, как точность, полнота и другие показатели, обычно используемые в поиске информации, изменяются в наборе данных. Вот пример вывода терминала trec_eval в наборе данных CACM с использованием ранжирования на базе языковой модели (LMJelinekMercerSimilarity[1]):

```
./trec_eval ~/lucene4ir/data/cacm/cacm.qrels
    ~/lucene4ir/data/cacm/bm25_results.res
...
map             all 0.2292
Rprec           all 0.2552
P_5             all 0.3373
P_10            all 0.2529
```

В этом случае Similarity было переключено на использование языковых моделей для оценки вероятностей релевантности. Результаты хуже, чем при использовании BM25: все показатели имеют немного более низкие значения.

Преимущество совместного использования этих инструментов состоит в том, что вы можете оценить, насколько хорошо ваши решения влияют на верность результатов поиска, выполнив ряд быстрых и простых шагов. Это не гарантирует, что вы можете достичь идеального ранжирования, но вы можете использовать данный подход, чтобы определить базовую функцию ранжирования для вашей поисковой системы и данных. После короткого знакомства с Lucene4IR вам будет

[1] *См.*: *Chengxiang Zhai and John Lafferty*. A Study of Smoothing Methods for Language Models Applied to Ad Hoc Information Retrieval (**http://mng.bz/zM8a**).

предложено разработать собственный класс Similarity, например на основе word2vec, и посмотреть, имеет ли это значение относительно BM25Similarity и т. д.

5.5.1. Класс Similarity, основанный на усредненных векторных представлениях слов

Вы увидели эффективность векторных представлений документов, созданных с использованием векторов слов, в небольшом эксперименте с образцом запроса «влияние бернхарда римана». В то же время в реальной жизни вам необходимы более качественные доказательства эффективности поисковой модели. В этом разделе вы будете работать с реализациями класса Similarity, основанными на усредненных векторах слов word2vec. Затем вы оцените их эффективность в небольшом наборе данных с помощью проекта Lucene4IR. Это даст вам представление о том, насколько хорошо эти модели ранжирования ведут себя в целом.

Правильное расширение класса Similarity – сложная задача, требующая понимания того, как работает Lucene. Мы сконцентрируемся на соответствующих фрагментах API Similarity, чтобы использовать векторные представления документов для оценки документов по запросам. Давайте начнем с создания класса WordEmbeddingsS Similarity, который создает векторные представления документов с помощью усредненных векторных представлений слов. Для этого требуется обученная модель word2vec, метод сглаживания для усреднения векторов слов, чтобы объединить их в вектор документа, и поле Lucene, из которого можно извлечь содержимое документа:

```
public class WordEmbeddingsSimilarity extends Similarity {

  public WordEmbeddingsSimilarity(Word2Vec word2Vec,
        String fieldName, Smoothing smoothing) {
    this.word2Vec = word2Vec;
    this.fieldName = fieldName;
    this.smoothing = smoothing;
  }
```

В Similarity будут реализованы следующие два метода:

```
  @Override
  public SimWeight computeWeight(float boost,
        CollectionStatistics collectionStats, TermStatistics... termStats) {
    return new EmbeddingsSimWeight(boost, collectionStats, termStats);
  }

  @Override
  public SimScorer simScorer(SimWeight weight,
        LeafReaderContext context) throws IOException {
    return new EmbeddingsSimScorer(weight, context);
  }
```

Наиболее важной частью этой задачи является реализация класса EmbeddingsSimScorer, который отвечает за ранжирование документов:

```
  private class EmbeddingsSimScorer extends SimScorer {
    @Override
    public float score(int doc, float freq) throws IOException {
```

```
      INDArray denseQueryVector = getQueryVector();  <-- Генерирует вектор запроса
      INDArray denseDocumentVector = VectorizeUtils
        .toDenseAverageVector(reader.getTermVector(doc,
         fieldName), reader.numDocs(),
            word2Vec, smoothing);  <-- Генерирует вектор документа
      return (float) Transforms.cosineSim(
        denseQueryVector, denseDocumentVector);  <-- Вычисляет косинусное сходство между
    }                                                векторами документа и запроса
  }                                                  и использует это как оценку документа
```

Как вы видите, метод `score` делает то, что вы делали в предыдущем разделе, но внутри класса `Similarity`. Единственное отличие, по сравнению с предыдущим подходом, состоит в том, что служебный класс `toDenseAverageVector` также принимает параметр `Smoothing`, который указывает, как усреднять векторы слов:

```
public static INDArray toDenseAverageVector(Terms docTerms, double n,
    Word2Vec word2Vec, WordEmbeddingsSimilarity.Smoothing smoothing)
        throws IOException {
  INDArray vector = Nd4j.zeros(word2Vec.getLayerSize());
  if (docTerms != null) {
    TermsEnum docTermsEnum = docTerms.iterator();
    BytesRef term;
    while ((term = docTermsEnum.next()) != null) {
      INDArray wordVector = word2Vec.getLookupTable().vector(
        term.utf8ToString());
      if (wordVector != null) {
        double smooth;
        switch (smoothing) {
          case MEAN:
            smooth = docTerms.size();
            break;
          case TF:
            smooth = docTermsEnum.totalTermFreq();
            break;
          case IDF:
            smooth = docTermsEnum.docFreq();
            break;
          case TF_IDF:
            smooth = VectorizeUtils.tfIdf(n, docTermsEnum.totalTermFreq(),
                docTermsEnum.docFreq());
            break;
          default:
            smooth = VectorizeUtils.tfIdf(n, docTermsEnum.totalTermFreq(),
                docTermsEnum.docFreq());
        }
        vector.addi(wordVector.div(smooth));
      }
    }
  }
  return vector;
}
```

`getQueryVector` делает то же самое, но вместо перебора `docTerms` он выполняет перебор термов в запросе.

Проект Lucene4IR поставляется с инструментами для запуска оценок по набору данных CACM, которые вы можете сделать, используя различные классы `Similarity`. Следуя инструкциям в файле README Lucene4IR (**http://mng.bz/0WGx**), можно сгенерировать статистику для оценки различных ранжирований. Например, вот точность первых пяти результатов с использованием разных классов `Similarity`:

```
WordEmbeddingsSimilarity:  0.2993
ClassicSimilarity:          0.2784
BM25Similarity:             0.2706
LMJelinekMercerSimilarity:  0.2588
```

Вот некоторые интересные цифры. Во-первых, модель векторного пространства с методом взвешивания TF-IDF – не самый плохой результат. У `WordEmbeddingsSimilarity` результат на 2 % лучше, чем у других; неплохо. Но один простой вывод из этой быстрой оценки заключается в том, что эффективность модели ранжирования может меняться в зависимости от данных, поэтому вам следует соблюдать осторожность при выборе модели. Теоретические результаты и оценки всегда нужно сравнивать с использованием вашей поисковой системы в реальной ситуации.

Также важно решить, для чего оптимизировать ранжирование. Зачастую, например, трудно получить высокую точность вместе с высокой полнотой. Давайте познакомимся с еще одной метрикой для оценки эффективности модели ранжирования: NDCG (*normalized discounted cumulative gain* – нормализованный дисконтированный прирост). NDCG измеряет полезность или прирост документа на основе его позиции в списке результатов. Прирост накапливается сверху вниз в списке результатов, поэтому прирост, вносимый каждым результатом, уменьшается с ростом ранжирования. Если вы оцените NDCG предыдущих классов `Similarity` по набору данных CACM, результаты будут еще интереснее:

```
WordEmbeddingsSimilarity:  0.3894
BM25Similarity:            0.3805
ClassicSimilarity:         0.3805
LMJelinekMercerSimilarity: 0.3684
```

Модель векторного пространства и BM25 работают одинаково; функция ранжирования на базе векторных представлений слов получила немного лучшее значение NDCG. Поэтому если вы заинтересованы в более точном ранжировании по первым пяти результатам, вам, вероятно, следует выбрать ранжирование на базе векторных представлений слов, но эта оценка предполагает, что для общего более высокого NDCG это может не иметь существенного значения.

Кроме того, хорошее решение, которое также поддерживается недавними исследованиями, может заключаться в том, чтобы смешивать классические и нейронные модели ранжирования, используя несколько функций ранжирования одновременно[1]. Это можно сделать с помощью класса `MultiSimilarity` в Lucene. Если

[1] *Dwaipayan Roy et al.* Representing Documents and Queries as Sets of Word Embedded Vectors for Information Retrieval // Neu-IR '16 SIGIR Workshop on Neural Information Retrieval. Pisa, Italy, 2016. July 21 (**https://arxiv.org/abs/1606.07869**).

вы выполните одну и ту же оценку, но с разными вариантами `MultiSimilarity`, то увидите, что смешивание языкового моделирования и векторов слов дает самое подходящее значение NDCG:

```
WV+BM25 :       0.4229
WV+LM :         0.4073
WV+Classic :    0.3973
BM25+LM :       0.3927
Classic+LM :    0.3698
Classic+BM25 :  0.3698
```

Резюме

- Классические модели поиска, такие как модель векторного пространства и BM25, обеспечивают хорошую основу для ранжирования документов, но им не хватает семантического понимания возможностей текста.
- Нейронные модели поиска информации нацелены на обеспечение лучшего семантического понимания возможностей для ранжирования документов.
- Распределенные представления слов (например, сгенерированные word2vec) могут быть объединены для создания векторных представлений документов для запросов и документов.
- Усредненные векторные представления слов могут использоваться для создания эффективных классов `Similarity` в Lucene, которые могут достигать хороших результатов при сравнении с наборами данных для поиска информации.

Глава 6

Векторные представления документов для ранжирования и рекомендаций

О чем идет речь в этой главе:

- генерация векторных представлений документов с использованием векторов абзацев;
- использование векторов абзаца для ранжирования;
- поиск сопутствующего контента;
- улучшение поиска сопутствующего контента с помощью векторов абзаца.

В предыдущей главе я познакомил вас с моделями поиска информации на базе нейронных сетей, построив функцию ранжирования на основе усредненных векторных представлений слов. Вы усредняли векторные представления, сгенерированные word2vec, чтобы получить *векторное представление документа*, плотное представление последовательности слов, демонстрирующее высокую точность ранжирования документов в соответствии с намерениями пользователя.

Однако недостаток таких распространенных поисковых моделей, как модель векторного пространства с TFIDF и BM25, заключается в том, что при ранжировании документов они учитывают только отдельные термы. Такой подход может привести к неоптимальным результатам, потому что контекстная информация – часть этих термов – отбрасывается. Учитывая этот недостаток, давайте посмотрим, как генерировать векторные представления документов, которые смотрят не только на отдельные слова, но и на целые фрагменты текста, окружающие эти слова. Созданное векторное представление будет содержать столько семантической информации, сколько возможно, тем самым повышая точность функции ранжирования.

Векторные представления слов очень хорошо подходят для захвата семантики слов, но значение и глубокая семантика текстовых документов не зависят от значения слов. Было бы хорошо иметь возможность изучать семантику фраз или более длинных фрагментов текста, а не только слов. В предыдущей главе вы делали это путем усреднения векторных представлений слов. Продвигаясь вперед, вы обнаружите, что можете добиться большего успеха с точки зрения верности.

В этой главе мы рассмотрим методику непосредственного изучения векторных представлений документов. Используя расширения алгоритмов обучения нейронных сетей word2vec, вы можете создавать векторные представления документов для текстовых последовательностей различной степени детализации (предложения, абзацы, документы и т. д.). Вы поэкспериментируете с этой техникой и продемонстрируете, что она дает лучшие показатели, когда используется для ранжирования.

Кроме того, вы узнаете, как использовать векторные представления документов для поиска сопутствующего контента. *Сопутствующий контент* состоит из документов (тексты, видео и т. д.), которые скоррелированы семантически. Когда вы показываете один результат поиска (например, когда пользователь щелкает по нему на странице результатов поиска), обычно отображается другой контент, например связанный с похожими темами или созданный тем же автором. Это полезно для привлечения внимания пользователей и предоставления им контента, который им может понравиться, но который может не отображаться на первой странице результатов поиска.

6.1. От векторных представлений слов к векторным представлениям документов

В этом разделе я познакомлю вас с расширением word2vec, целью которого является векторное представление документов во время обучения нейронной сети. Это отличается от ранее использовавшегося метода смешивания векторов слов (их усреднения и в конечном итоге сглаживания, например используя веса TF-IDF) и часто дает лучшие результаты при захвате семантики документа[1]. Этот метод, также известный как *векторы абзацев*, расширяет две архитектуры word2vec (непрерывный мешок слов [CBOW] и skip-gram), включая информацию о текущем документе в контексте[2]. Word2vec выполняет неконтролируемое обучение векторных представлений слов с использованием фрагментов текстов определенного размера, именуемых *окном*, для обучения нейронной сети, либо предсказывает контекст по данному слову, принадлежащему этому контексту, либо предсказывает слово по контексту, к которому оно принадлежит.

В частности, нейронная сеть с архитектурой CBOW имеет три уровня (см. рис. 6.1):

- входной слой, содержащий слова контекста;
- скрытый слой, содержащий один вектор для каждого слова;
- выходной слой, содержащий слово для прогнозирования.

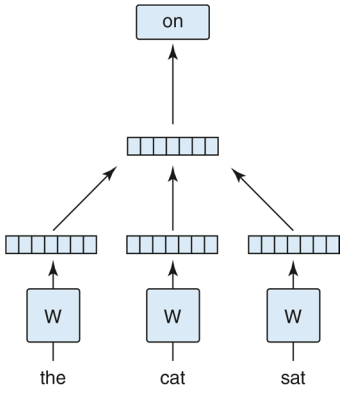


Рис. 6.1 ❖ Модель CBOW

[1] См. сравнения в статье: *Andrew M. Dai, Christopher Olah, and Quoc V. Le.* Document Embedding with Paragraph Vectors (**https://arxiv.org/pdf/1507.07998.pdf**).

[2] *См.*: *Quoc Le and Tomas Mikolov*. Distributed Representations of Sentences and Documents (**http://cs.stanford.edu/~quocle/paragraph_vector.pdf**).

Интуиция, обеспечиваемая методами на базе вектора абзаца, может либо украшать, либо заменять контекст меткой, обозначающей документ, поэтому нейронная сеть научится соотносить слова и контексты с метками, а не слова с другими словами.

Модель CBOW расширена, поэтому входной слой также содержит метку документа, содержащего текущий фрагмент текста. Во время обучения каждый фрагмент текста помечается меткой. Такие текстовые фрагменты могут быть целыми документами или частями документа, как разделы, абзацы либо предложения. *Значение* метки, как правило, не важно[1]; метка может быть *doc_123456* или *тег-foo-bar*, или любая другая сгенерированная машиной строка. Важно то, что метки должны быть уникальными в документе: два разных фрагмента текста не должны быть помечены одной и той же меткой, если они не принадлежат одному и тому же фрагменту текста.

Как видно на рис. 6.2, архитектура этой модели похожа на CBOW; она просто добавляет входную метку, представляющую документ во входном слое. Следовательно, скрытый слой должен быть снабжен вектором для каждой метки, чтобы в конце обучения у вас было представление вектора каждой метки. Интересным в этом подходе является то, что он позволяет обрабатывать документы различной степени детализации. Вы можете использовать метки как для целых документов, так и для небольших их частей, таких как абзацы или предложения. Эти метки действуют как своего рода память, которая связывает контексты с (отсутствующими) словами; поэтому этот метод называется *моделью распределенной памяти векторов абзаца* (PV-DM).

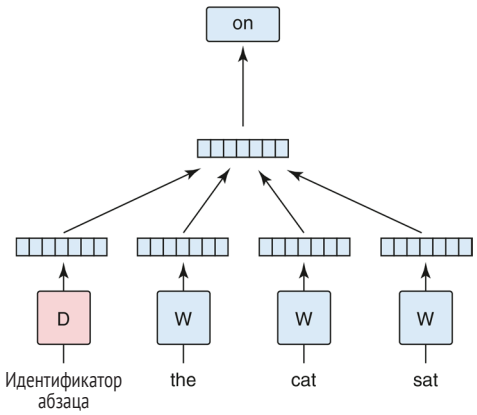


Рис. 6.2 ❖ Модель с распределенной памятью векторов абзаца

В случае с такими документами, как «гипотеза римана – глубокое погружение в математическую тайну», имеет смысл использовать одну метку, потому что текст относительно короткий. Но для более длинных документов, таких как страницы

[1] Только если вы не собираетесь использовать ее где-то еще, кроме обучения, например использовать метки, сгенерированные сетью, в качестве идентификаторов документов при их индексации после завершения обучения.

Википедии, может быть полезно создать ярлык для каждого абзаца или предложения. Давайте выберем первый абзац страницы в Википедии, посвященный Риману: «Георг Фридрих Бернхард Риман (17 сентября 1826 – 20 июля 1866) был немецким математиком, который внес вклад в анализ, теорию чисел и дифференциальную геометрию. В области реального анализа он в основном известен первой строгой формулировкой интеграла, интегралом Римана и его работами над рядами Фурье». Можно пометить каждое предложение разной меткой и создать векторное представление, которое поможет найти похожие предложения вместо похожих страниц Википедии.

Векторы абзацев также расширяют модель skip-gram с помощью модели *распределенного мешка слов* (PV-DBOW). Модель архитектуры skip-gram использует нейронную сеть с тремя слоями:

- входной слой с одним входным словом;
- скрытый слой, содержащий векторное представление для каждого слова в словаре;
- выходной слой, содержащий количество слов, обозначающих прогнозируемый контекст относительно входного слова.

Модель DBOW с векторами абзацев (*см.* рис. 6.3) вводит метки вместо слов, поэтому сеть учится предсказывать части текста, принадлежащие документу, абзацу или предложению, у которого есть эта метка.

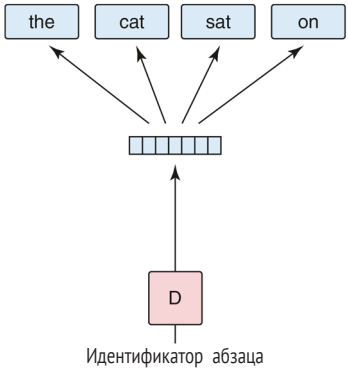


Рис. 6.3 ❖ Модель распределенного мешка слов с векторами абзаца

Модели PV-DBOW и PV-DM могут использоваться для расчета сходства между помеченными документами. Как и в word2vec, они добиваются удивительно хороших результатов при захвате семантики документа. Давайте попробуем использовать векторы абзацев в примере с реализацией `ParagraphVectors` в DL4J:

```
ParagraphVectors paragraphVectors = new ParagraphVectors.Builder()
  .iterate(iterator)
  .layerSize(50)
  .minWordFrequency(7)
  .sequenceLearningAlgorithm(new DM<>())
  .tokenizerFactory(new DefaultTokenizerFactory())
  .build();

paragraphVectors.fit();
```

← Настраивает векторы абзаца

← Устанавливает размеры векторного представления документа

Как и в word2vec, вы можете установить минимальный порог частоты для слова, которое будет использоваться во время обучения

← Завершает настройку

Выбирает выбранную модель вектора абзаца: в данном случае PV-DM

← Выполняет (без учителя) обучение по входным данным

Как и в случае с word2vec, вы можете задать вопросы:

- каковы ближайшие метки для пометки xyz? Это позволит вам найти наиболее похожие документы (потому что каждый документ помечен меткой);
- каковы ближайшие метки, учитывая новый кусок текста? Это позволит использовать векторы абзацев в документах или запросах, которые не являются частью обучающего набора.

Если вы используете заголовки со страниц Википедии для обучения векторов абзацев, можно поискать страницы в Википедии, заголовки которых семантически похожи на вводимый текст. Предположим, вы хотите получить информацию о вашем следующем большом путешествии в Южную Америку. Вы можете получить три самых близких документа к предложению «Путешествие по Южной Америке» из модели векторного абзаца, которую вы только что обучили:

```
Collection<String> strings = paragraphVectors
    .nearestLabels("Travelling in South America"
    , 3);
for (String s : strings) {
  int docId = Integer.parseInt(s.substring(4));

  Document document = reader.document(docId);

  System.out.println(document.get(fieldName));
}
```

Получает метки, ближайшие к указанной входной строке

Каждая метка имеет вид «doc _» + documentId, поэтому вы получаете только часть идентификатора документа для извлечения документа из индекса

Извлекает документ Lucene, имеющий заданный идентификатор

Выводит заголовок документа в консоль

Вот как выглядит вывод:

```
Transport in São Tomé and Príncipe
Transport in South Africa
Telecommunications in São Tomé and Príncipe
```

Не очень релевантно

Информация о транспорте и телекоммуникациях Сан-Томе и Принсипи

Если вы обучаете векторы абзацев, используя весь текст страниц Википедии, а не только заголовок, то получите более релевантные результаты. Это происходит главным образом из-за того, что векторы абзацев, такие как word2vec, изучают текстовые представления, глядя на контекст, а с более коротким текстом (заголовками) это сложнее, чем с более длинным текстом (страница Википедии целиком).

При обучении с использованием всего текста страниц Википедии результат выглядит так:

```
Latin America
Crime and violence in Latin America
Overseas Adventure Travel
```

Векторные представления документов, подобные тем, что создаются векторами абзацев, направлены на то, чтобы обеспечить хорошее представление семантики всего текста в форме вектора. Вы можете использовать их в контексте поиска для решения проблемы семантического понимания в ранжировании. Сходство между такими представлениями больше зависит от значения текста и меньше – от простого сопоставления термов.

6.2. Использование векторов абзацев в ранжировании

Использовать векторы абзацев при ранжировании очень просто: вы можете попросить модель, либо предоставить вектор для уже обученной метки или документа, либо обучить новый вектор для нового фрагмента текста (такого как невидимый документ или запрос). Принимая во внимание тот факт, что, работая с векторами слов, вы должны решить, как их объединить (вы делали это во время

ранжирования, но могли бы делать это во время индексации), модели на основе векторов абзацев позволяют легко извлекать векторные представления запросов и документов, сравнивать и ранжировать их.

Прежде чем перейти к использованию векторов абзацев для ранжирования, давайте сделаем шаг назад. В предыдущем разделе говорилось об использовании данных, проиндексированных в Lucene, для обучения модели векторов абзаца. Это можно сделать, реализовав `LabelAwareIterator`: итератор содержимого документа, который также назначает метку каждому документу Lucene. Вы помечаете каждый документ Lucene внутренним идентификатором документа Lucene, в результате чего получается метка, которая выглядит как *doc_1234*:

```
public class FieldValuesLabelAwareIterator implements LabelAwareIterator {

  private final IndexReader reader;
  private final String field;
  private int currentId = 0;

  @Override
  public boolean hasNextDocument() {
    return currentId < reader.numDocs();
  }

  @Override
  public LabelledDocument nextDocument() {
    if (!hasNextDocument()) {
      return null;
    }
    try {
      Document document = reader.document(currentId,
        Collections.singleton(field));

      LabelledDocument labelledDocument = new
        LabelledDocument();
        labelledDocument.addLabel("doc_"
          + currentId);
        labelledDocument.setContent(document
          .getField(field).stringValue());
        return labelledDocument;
      } catch (IOException e) {
        throw new RuntimeException(e);
      } finally {
        currentId++;
    }
  }
  ...
}
```

FieldValuesLabelAwareIterator извлекает последовательности из IndexReader (представление чтения в поисковой системе)

Контент будет получен из одного поля, а не из всех возможных полей в документе Lucene

Идентификатор текущего документа, который извлекается, инициализируется как 0

Если текущий идентификатор меньше количества документов в индексе, есть дополнительные документы для чтения

Извлекает контент из индекса Lucene

Создает новый LabelledDocument, который будет использоваться для обучения ParagraphVectors. Внутренний идентификатор Lucene используется в качестве метки

Устанавливает содержимое указанного поля Lucene в LabelledDocument

Итератор для векторов абзаца инициализируется следующим образом:

```
IndexReader reader = DirectoryReader.open(writer);
String fieldName = "title";
FieldValuesLabelAwareIterator iterator = new
    FieldValuesLabelAwareIterator (reader, fieldName);
ParagraphVectors paragraphVectors = new ParagraphVectors.Builder()
```

Создает IndexReader

Определяет поле, которое будет использоваться

Создает итератор

```
  .iterate(iterator)          ← Устанавливает итератор в ParagraphVectors
  .build();                   ← Создает модель векторов абзаца (которую еще нужно обучить)

paragraphVectors.fit();       ← Позволяет векторам абзаца выполнять обучение (без учителя)
```

После того как модель закончит обучение, вы можете использовать векторы абзацев для повторного ранжирования документов после фазы поиска:

```
                                  Создает IndexSearcher для выполнения
                                  первого запроса, который идентифицирует
                                  набор результатов
IndexSearcher searcher = new IndexSearcher(reader);

INDArray queryParagraphVector = paragraphVectors
    .getLookupTable().vector(queryString);
if (queryParagraphVector == null) {
  queryParagraphVector = paragraphVectors
    .inferVector(queryString);
}
QueryParser parser = new QueryParser(fieldName, new WhitespaceAnalyzer());
Query query = parser.parse(queryString);
TopDocs hits = searcher.search(query, 10);
for (int i = 0; i < hits.scoreDocs.length; i++) {
  ScoreDoc scoreDoc = hits.scoreDocs[i];
  Document doc = searcher.doc(scoreDoc.doc);

  String label = "doc_" + scoreDoc.doc;

  INDArray documentParagraphVector = paragraphVectors
    .getLookupTable().vector(label);
  double score = Transforms.cosineSim(
    queryParagraphVector, documentParagraphVector);

  String title = doc.get(fieldName);
  System.out.println(title + " : " + score);
}
```

Пытается извлечь существующее векторное представление текущего запроса. Это может не сработать, потому что вы обучали модель на содержании поисковой системы, а не на запросах

Если вектор запроса не существует, позволяет базовой нейронной сети обучаться и выводить вектор для этого нового фрагмента текста (чья метка – это весь текст строки)

Выполняет поиск

Создает метку текущего документа

Выбирает существующий вектор для документа с указанной меткой

Рассчитывает оценку как косинусное сходство между векторами запроса и документа

Выводит результаты в консоль

Этот код показывает, насколько легко получить распределенное представление для запросов и документов без необходимости работать со вложениями слов. Для удобства чтения результаты снова идут от лучших к худшим (даже если код этого не делает). Ранжирование соответствует фактической релевантности возвращенных документов, а оценки соответствуют релевантности документа:

```
riemann hypothesis - a deep dive into a mathematical mystery : 0.77497977
riemann bernhard - life and works of bernhard riemann : 0.76711642
thomas bernhard biography - bio and influence in literature : 0.32464843
bernhard bernhard bernhard bernhard bernhard bernhard ... : 0.03593694
```

Два наиболее релевантных документа имеют высокие (и очень близкие) оценки, а у третьего значительно более низкая оценка; это нормально, потому что он не релевантен. Наконец, фиктивный документ имеет оценку, близкую к нулю.

6.2.1. ParagraphVectorsSimilarity

Можно применить `ParagraphVectorsSimility`, который использует векторы абзаца для измерения сходства между запросом и документом. Его интересной особенностью является реализация API `SimScorer#score`:

```
@Override
public float score(int docId, float freq) throws IOException {
  INDArray denseQueryVector = paragraphVectors
    .inferVector(query);
  String label = "doc_" + docId;
  INDArray documentParagraphVector = paragraphVectors
    .getLookupTable().vector(label);
  if (documentParagraphVector == null) {
    LabelledDocument document = new LabelledDocument();
    document.setLabels(Collections.singletonList(label));
    document.setContent(reader.document(docId).getField(fieldName)
        .stringValue());
    documentParagraphVector = paragraphVectors
        .inferVector(document);
  }
  return (float) Transforms.cosineSim(
    denseQueryVector, documentParagraphVector);
}
```

Извлекает вектор абзаца для текста запроса. Если запрос не был замечен ранее, это будет означать выполнение этапа обучения для сети с моделью векторов абзацев

Извлекает вектор абзаца для документа с меткой, равной идентификатору документа

Если вектор с заданной меткой (docId) нельзя найти, выполняется обучающий этап для сети с моделью векторов абзацев, чтобы извлечь новый вектор

Вычисляет косинусное сходство между векторами абзаца запроса и документа и использует его в качестве оценки для данного документа

6.3. Векторные представления документов и сопутствующий контент

Как пользователь вы, возможно, испытывали чувство, что определенный результат поиска *почти* хорош, но по какой-то причине он недостаточно хорош. Рассмотрим запрос «книги о реализации алгоритмов нейронных сетей» на розничном сайте. Вы получаете результаты поиска: первая книга называется *Обучение программированию нейронных сетей*, поэтому вы нажимаете на этот результат и попадаете на страницу, содержащую более подробную информацию о данной книге. Вы понимаете, что вам нравится ее содержание. Автор является признанным авторитетом в этом вопросе, но он использует Python в качестве языка программирования для своих учебных примеров, который вы недостаточно хорошо знаете. Вы можете спросить: «Есть ли похожая книга, где используется Java для обучения программированию нейронных сетей?» Сайт розничной торговли может показать вам список похожих книг в надежде, что если вы не захотите купить книгу с примерами, написанными на Python, вместо этого вы можете купить другую книгу с похожим содержанием (это может быть книга с примерами на языке Java).

В этом разделе вы узнаете, как предоставить такой сопутствующий контент, найдя в поисковой системе дополнительные документы, которые похожи не только потому, что принадлежат одному и тому же автору или имеют несколько общих слов, но и потому, что между двумя такими документами существует более значимая семантическая корреляция. Это должно напомнить вам о проблеме семантического понимания, которую мы рассматривали при ранжировании функций с использованием векторных представлений документов, изученных с помощью векторов абзацев.

6.3.1. Поиск, рекомендации и сопутствующий контент

Чтобы проиллюстрировать, насколько важно указывать сопутствующий контент в поисковой системе, давайте рассмотрим поток действий, которые пользователь выполняет на платформе обмена видео (например, YouTube). Основной (или даже единственный) интерфейс – это окно поиска, куда пользователи вводят запрос. Предположим, что пользователь вводит учебное пособие по Lucene в поле поиска и нажимает кнопку поиска. Отображается список результатов поиска, и пользователь в конечном итоге выбирает тот, который он считает интересным. С этого момента пользователь обычно прекращает поиск и вместо этого щелкает видео в поле или столбце «Related». Типичными рекомендациями для видео под названием «Учебное пособие по Lucene» могут быть видеоролики с такими названиями, как «Lucene для начинающих», «Знакомство с поисковыми системами» и «Создание рекомендательных систем с помощью Lucene». Пользователь может нажать любую из этих рекомендаций; например, если он узнал достаточно из видео «Учебное пособие по Lucene», он может перейти к просмотру более продвинутого видео; в противном случае может захотеть посмотреть еще одно вступительное видео или видео, которое знакомит с поисковыми системами, если поймет, что необходимы дополнительные предварительные знания, чтобы понять, как использовать Lucene. Этот процесс использования найденного контента и последующей навигации по сопутствующему контенту может продолжаться бесконечно. Таким образом, предоставление соответствующего сопутствующего контента имеет первостепенное значение для лучшего удовлетворения потребностей пользователей.

Содержимое поля «Related» может даже изменить намерение пользователя в направлении тем, которые далеки от первоначального запроса. В предыдущем примере пользователь хотел узнать, как использовать Lucene. Поисковая система предоставила сопутствующий элемент, основная тема которого не была напрямую связана с Lucene: речь шла о создании системы машинного обучения для генерации рекомендаций на базе Lucene. Это большой переход от потребности в информации о работе с Lucene для новичка к изучению рекомендательных систем на базе Lucene (более сложная тема).

Этот краткий пример также относится к сайтам электронной коммерции. Основная цель таких сайтов – продать вам что-то. Поэтому хотя вам и предлагается искать продукт, который вам (возможно) нужен, вам также предлагается множество «рекомендованных для вас» товаров. Эти рекомендации основаны на следующих факторах:

- какие продукты вы искали в прошлом;
- какие темы вы ищете больше всего;
- новые продукты;
- какие продукты вы видели (просматривали или нажимали) недавно.

Одним из основных пунктов этого потока рекомендаций является *удержание пользователей*: сайт электронной коммерции хочет, чтобы вы как можно дольше просматривали и искали, надеясь, что любой из продаваемых ими продуктов будет достаточно интересным для его покупки.

Это выходит за рамки покупки и продажи. Такие возможности очень важны для многих применений, например, в области здравоохранения: врач, просматривающий медицинские записи пациента, извлек бы выгоду из возможности просмат-

ривать аналогичные медицинские записи других пациентов (и их истории), чтобы поставить более точный диагноз. Теперь мы сосредоточимся на реализации алгоритмов поиска связанных или похожих документов по отношению к входному документу на основе их содержимого. Сначала мы рассмотрим, как заставить поисковую систему извлекать сопутствующий контент, а затем вы узнаете, как использовать различные подходы для создания векторных представлений документов, чтобы преодолеть ограничения первого подхода; *см.* рис. 6.4. Мы также воспользуемся этой возможностью, чтобы обсудить, как использовать векторы абзацев для классификации документов, что полезно в контексте предоставления семантически релевантных предложений.

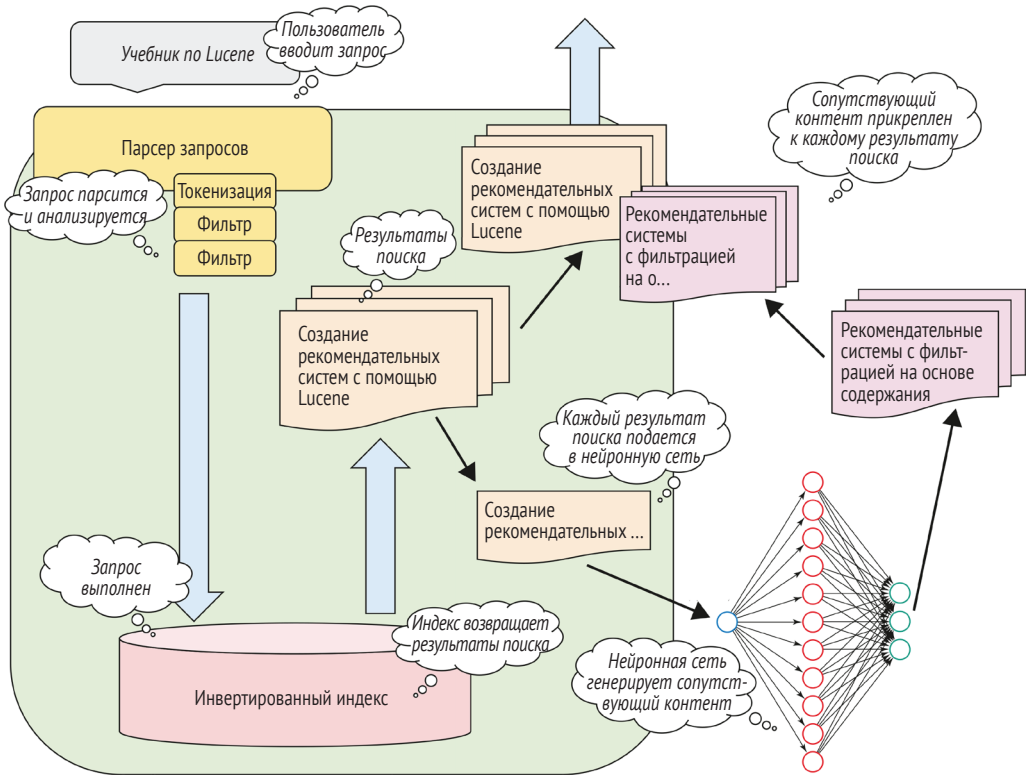


Рис. 6.4 ❖ Использование нейронных сетей для извлечения сопутствующего контента

6.3.2. Использование частых термов для поиска похожего контента

В предыдущей главе вы видели, как схема взвешивания TF-IDF для ранжирования опирается на частоты термов и документов, чтобы обеспечить меру важности документа. Логическое обоснование ранжирования с использованием TF-IDF заключается в том, что важность документа возрастает с локальной частотой и глобальной редкостью его термов по отношению к входному запросу. Основываясь на этих предположениях, можно определить алгоритм поиска документов, по-

хожих на входной документ, исключительно на основе поисковых возможностей поисковой системы.

Дампы из Википедии могут быть хорошей коллекцией для оценки эффективности алгоритма поиска сопутствующего контента. Каждая страница Википедии содержит контент и полезные метаданные (заголовок, категории, ссылки и даже некоторые ссылки на сопутствующий контент в разделе «Смотрите также»). Для индексирования дампов Википедии в Lucene можно использовать несколько доступных инструментов, например модуль `lucene-benchmark` (**http://mng.bz/A2Qo**). Предположим, вы проиндексировали каждую страницу Википедии с ее заголовком и текстом в два отдельных индекса Lucene. Учитывая результаты поиска, возвращаемые запросом, вы хотите получить пять наиболее похожих документов, которые будут показаны пользователю в качестве сопутствующего контента. Для этого вы выбираете каждый результат поиска, извлекаете наиболее важные термы из его содержимого (в данном случае из поля `text`) и выполняете еще один запрос, используя извлеченные термы (*см.* рис. 6.5). Первые пять итоговых документов можно использовать в качестве сопутствующего контента.

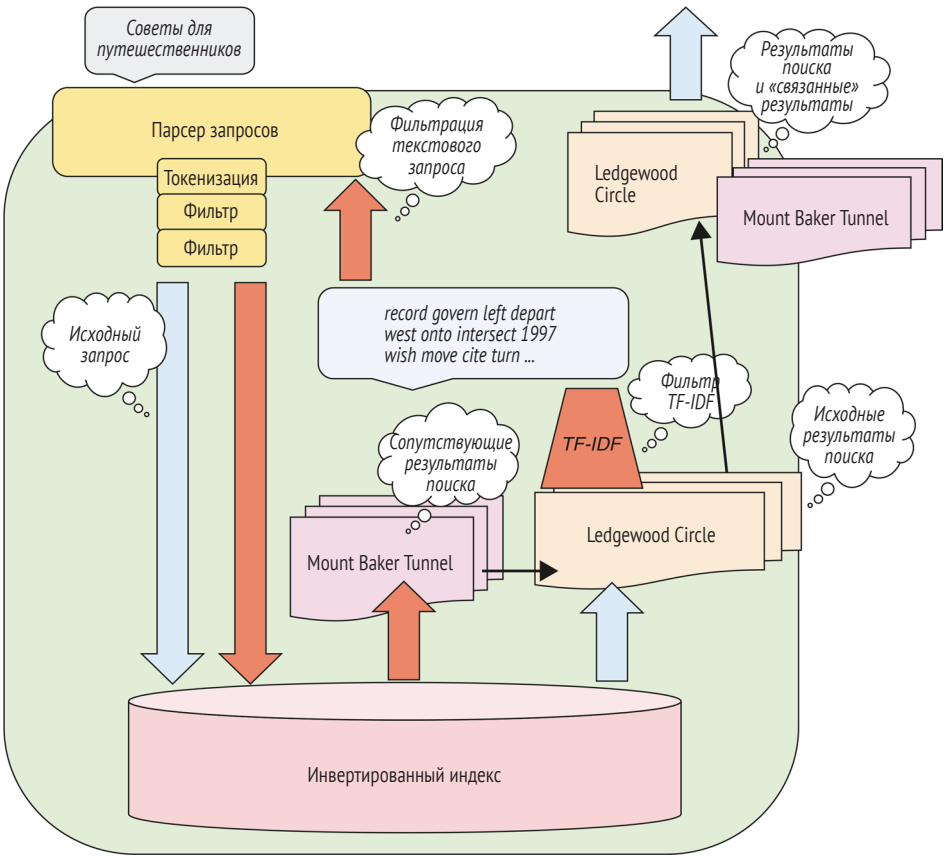


Рис. 6.5 ❖ Извлечение сопутствующего контента с использованием наиболее важных термов документа, взвешенных по схеме TF-IDF

Предположим, вы выполнили запрос «trip hints» и получили результат поиска, где речь идет о кольцевой развязке в Нью-Джерси под названием Ledgewood Circle. Вы берете все термы, содержащиеся на странице Википедии **https://en.wikipedia.org/wiki/Ledgewood_Circle**, и извлекаете те из них, которые имеют частоту как минимум 2 и частоту документа 5. Таким образом, вы получаете следующий список термов:

```
record govern left depart west onto intersect 1997 wish move cite turn
    township signal 10 lane travel westbound new eastbound us tree 46
    traffic ref
```

Затем используете эти термы в качестве запроса для получения документов, которые будут применяться в качестве сопутствующего контента, представленного конечному пользователю.

Lucene позволяет сделать это с помощью компонента `MoreLikeThis` (MLT, **http://mng.bz/ZZlR**), который может извлекать наиболее важные термы из документа и создавать объект `Query` для запуска через тот же `IndexSearcher`, который использовался для выполнения оригинального запроса.

Листинг 6.1 ❖ Поиск и получение соответствующего контента с помощью MLT

```
EnglishAnalyzer analyzer = new EnglishAnalyzer();
MoreLikeThis moreLikeThis = new MoreLikeThis(
    reader);
moreLikeThis.setAnalyzer(analyzer);

IndexSearcher searcher = new IndexSearcher(
    reader);

String fieldName = "text";
QueryParser parser = new QueryParser(fieldName,
    analyzer);
Query query = parser.parse("travel hints");

TopDocs hits = searcher.search(query, 10);

for (int i = 0; i < hits.scoreDocs.length; i++) {
  ScoreDoc scoreDoc = hits.scoreDocs[i];
  Document doc = searcher.doc(scoreDoc.doc);

  String title = doc.get("title");
  System.out.println(title + " : " +
    scoreDoc.score);

  String text = doc.get(fieldName);
  Query simQuery = moreLikeThis.like(fieldName,
    new StringReader(text));

  TopDocs related = searcher.search(simQuery, 5);
  for (ScoreDoc rd : related.scoreDocs) {
    Document document = reader.document(rd.doc);
    System.out.println("-> " + document.get(
        "title"));
  }
}
```

Создает экземпляр MLT

Создает IndexSearcher

Создает QueryParser

Определяет анализатор, который будет использоваться при поиске и извлечении термов из содержимого результатов поиска

Определяет анализатор, который будет использоваться MLT

Определяет, какое поле использовать при выполнении первого запроса и поиске сопутствующего контента через запрос, сгенерированный MLT

Парсит введенный пользователем запрос

Выполняет запрос и возвращает 10 лучших результатов поиска

Извлекает объект Document, связанный с текущим результатом поиска

Выводит название и оценку текущего документа

Извлекает содержимое поля text из текущего документа

Использует MLT для генерации запроса на базе содержимого найденного документа путем извлечения наиболее важных термов

Выполняет запрос, сгенерированный MLT

Выводит заголовок документа, который нашел объект Query, сгенерированный MLT

Для извлечения сопутствующего контента не требуется никакого машинного обучения: вы используете возможности поисковой системы для возврата связанных документов, содержащих наиболее важные термы, из результатов поиска. Вот пример результатов запроса «trip hints» и результата поиска «Ledgewood Circle»:

```
Ledgewood Circle : 7.880041
-> Ledgewood Circle
-> Mount Baker Tunnel
-> K-5 (Kansas highway)
-> Interstate 80 in Illinois
-> Modal dispersion
```

Первые три связанных документа (не считая документа «Ledgewood Circle») аналогичны оригинальному документу. Все они имеют отношение к чему-то, что связано с кольцевыми развязками, такими как туннель, шоссе или автомагистраль между штатами. Четвертый документ тем не менее не имеет абсолютно никакого отношения к теме: он касается волоконной оптики. Давайте подробно рассмотрим, откуда взялся этот результат. Для этого вы можете включить пояснение Lucene:

```
Query simQuery = moreLikeThis.like(fieldName, new StringReader(text));
TopDocs related = searcher.search(simQuery, 5);
for (ScoreDoc rd : related.scoreDocs) {
  Document document = reader.document(rd.doc);
  Explanation e = searcher.explain(simQuery, rd.doc);
  System.out.println(document.get("title") + " : " + e);
}
```

Получает объяснение для запроса MLT

Пояснение позволяет проверить, как соотносятся термы `signal`, `10`, `travel` и `new`:

```
Modal dispersion :
20.007288 = sum of:
  7.978141 = weight(text:signal in 1972) [BM25Similarity], result of:
    ...
  2.600343 = weight(text:10 in 1972) [BM25Similarity], result of:
    ...
  7.5186286 = weight(text:travel in 1972) [BM25Similarity], result of:
    ...
  1.9101752 = weight(text:new in 1972) [BM25Similarity], result of:
    ...
```

Проблема, связанная с этим подходом, состоит в том, что `MoreLikeThis` извлек наиболее важные термы в соответствии с взвешиванием TF-IDF, а оно, как вы видели в предыдущей главе, имеет проблему полагаться на частоты. Давайте посмотрим на эти важные термы, извлеченные из текста документа «Ledgewood Circle»: термы «record», «govern», «left», «depart», «west», «onto», «intersect», «1997», «wish», «move» и т. д., по-видимому, не предполагают, что документ имеет дело с кольцевой развязкой. Если вы попытаетесь прочитать их как предложение, то не сможете извлечь из этого особого смысла.

Пояснение по умолчанию использует Lucene `BM25Similarity`. В главе 5 вы видели, что можно использовать различные функции ранжирования и проверять, можно ли получить более качественные результаты. Если вы воспользуетесь `ClassicSimilarity` (модель векторного пространства с TF-IDF), то получите следующее:

```
Query simQuery = moreLikeThis.like(fieldName, new StringReader(text));
searcher.setSimilarity(new ClassicSimilarity());
TopDocs related = searcher.search(simQuery, 5);
for (ScoreDoc rd : related.scoreDocs) {
  Document document = reader.document(rd.doc);
  System.out.println(searcher.getSimilarity() +
    " -> " + document.get("title"));
}
```

Использует ClassicSimilarity вместо значения по умолчанию (только для поиска аналогичного контента)

Вот результаты:

```
ClassicSimilarity -> Ledgewood Circle
ClassicSimilarity -> Mount Baker Tunnel
ClassicSimilarity -> Cherry Tree
ClassicSimilarity -> K-5 (Kansas highway)
ClassicSimilarity -> Category:Speech processing
```

Они еще хуже: и «Cherry Tree», и «Speech Processing» совершенно не связаны с оригинальным документом «Ledgewood Circle». Давайте попробуем использовать сходство на базе языковой модели, `LMDirichletSimilarity`[1]:

```
Query simQuery = moreLikeThis.like(fieldName, new StringReader(text));
searcher.setSimilarity(
    new LMDirichletSimilarity());
TopDocs related = searcher.search(simQuery, 5);
for (ScoreDoc rd : related.scoreDocs) {
  Document document = reader.document(rd.doc);
  System.out.println(searcher.getSimilarity() +
    " -> " + document.get("title"));
}
```

Результаты приведены ниже:

```
LM Dirichlet(2000.000000) -> Ledgewood Circle
LM Dirichlet(2000.000000) -> Mount Baker Tunnel
LM Dirichlet(2000.000000) -> K-5 (Kansas highway)
LM Dirichlet(2000.000000) -> Interstate 80 in Illinois
LM Dirichlet(2000.000000) -> Creek Turnpike
```

Интересно, что все эти результаты выглядят хорошо – все они относятся к транспортной инфраструктуре, такой как шоссе или туннели.

Измерение качества сопутствующего контента с использованием категорий

В главе 5 вы узнали, как важно не проводить одиночные эксперименты. Хотя они позволяют детально понять, как работают поисковые модели в некоторых случаях, они не могут обеспечить общую оценку того, насколько хорошо такая модель работает с большим количеством данных. Поскольку на страницах Википедии есть категории, вы можете сначала оценить с их помощью верность сопутствующего контента. Если документы, найденные с помощью алгоритма сопутствующего контента (в данном случае `MoreLikeThis`), попадают в любую из категорий

[1] *См.*: *Chengxiang Zhai and John Lafferty*. A Study of Smoothing Methods for Language Models Applied to Ad Hoc Information Retrieval (**http://mng.bz/RGVZ**).

исходных документов, можно считать их актуальными. В реальной жизни вам может потребоваться выполнить эту оценку немного по-другому: например, вы также можете посчитать предлагаемый документ релевантным, если его категория является подкатегорией категории исходного документа. Это (и многое другое) можно сделать путем создания таксономии, извлекая ее из Википедии (**https://en.wikipedia.org/wiki/Help:Category**) или с помощью проекта DBpedia (коллективная попытка собрать структурированную информацию о контенте в Википедии; **http://wiki.dbpedia.org**). Но ради экспериментов, описанных в этой главе, можно определить меру верности как сумму раз, когда часть сопутствующего контента использует одну или несколько категорий с исходным документом, деленную на количество полученных связанных документов.

Давайте воспользуемся страницей в Википедии, посвященной футболисту Радамелю Фалькао, которая состоит из множества категорий (год рождения: 1986, игроки AS Monaco FC и т. д.). Использование `BM25Similarity` для ранжирования сгенерированного MLT объекта `Query` дает пять лучших связанных документов с общей категорией в скобках (если таковая имеется):

```
Bacary Sagna (*Expatriate footballers in England*)
Steffen Hagen (*1986 births*)
Andrés Scotti (*Living people*)
Iyseden Christie (*Association football forwards*)
Pelé ()
```

Первые четыре результата имеют общую категорию со страницей Радамеля Фалькао в Википедии, а «Pelé» – нет. Таким образом, верность равна 4 (количество результатов, у которых общая со страницей Радамеля Фалькао категория), деленная на 5 (количество возвращенных похожих результатов), или 0,8.

Чтобы оценить этот алгоритм, можно сгенерировать несколько случайных запросов и измерить определенную среднюю верность по возвращенному сопутствующему контенту. Давайте сгенерируем 100 запросов, используя слова, которые существуют в индексе (чтобы убедиться, что возвращается хотя бы один результат поиска), а затем извлечем 10 наиболее похожих документов, используя векторы абзацев и косинусное сходство. Для каждого из этих связанных документов проверьте, отображается ли одна из его категорий также в результатах поиска.

Листинг 6.2 ❖ Получение сопутствующего контента и расчет точности

```
int topN = 10;
String[] originalCategories = doc
    .getValues("category");
Query simQuery = moreLikeThis.like(fieldName,
    new StringReader(s));
for (Similarity similarity : similarities) {
  searcher.setSimilarity(similarity);
  TopDocs related = searcher.search(simQuery,
    topN);
  double acc = 0;
  for (ScoreDoc rd : related.scoreDocs) {
    if (rd.doc == scoreDoc.doc) {
```

Получает категории, связанные с исходной страницей Википедии, возвращенной запросом

Создает запрос по сопутствующему контенту с помощью MLT

Выполняет один и тот же запрос с несколькими реализациями Similarity, чтобы оценить, что работает лучше

Использует определенное сходство в IndexSearcher

Выполняет запрос по сопутствующему контенту

Инициализирует верность до нуля

Пропускает результат, если он равен исходному документу

```
      topN--;
      continue;
    }
    Document document = reader.document(rd.doc);
    String[] categories = document.getValues("category");
    if (categories != null && originalCategories != null) {
      if (find(categories, originalCategories)) {
        acc += 1d;
      }
    }
  }
  acc /= topN;
  System.out.println(similarity + " accuracy : " + acc);
}
```

Получает связанный документ

Если какая-либо категория сопутствующего контента содержится в исходном документе, повышает верность

Делит верность на количество возвращенных связанных документов

Соответствующий результат с `BM25Similiarity`, `ClassicSimilarity` и `LMDirichletSimilarity` выглядит следующим образом:

```
BM25(k1=1.2,b=0.75) accuracy : 0.2
ClassicSimilarity accuracy : 0.2
LM Dirichlet(2000.000000) accuracy : 0.1
```

Выполнение этих более 100 случайно сгенерированных запросов и соответствующие 10 лучших результатов дают следующие цифры:

```
BM25(k1=1.2,b=0.75) average accuracy : 0.09
ClassicSimilarity average accuracy : 0.07
LM Dirichlet(2000.000000) average accuracy : 0.07
```

Учитывая тот факт, что наилучшая возможная верность равна 1,0, это низкие значения. Лучший вариант находит связанный документ с соответствующей категорией только 9 % времени.

Хотя это субоптимальный результат, полезно рассмотреть его и поговорить о наличии информации о категориях в каждом документе. Во-первых, вы выбрали хороший показатель для измерения «приблизительности» сопутствующего контента, полученного с помощью этого подхода? Категории, прикрепленные к страницам Википедии, обычно хорошего качества, а категории страницы «Ledgewood Circle» – «Transportation in Morris County» и «Traffic circles in New Jersey». Такая категория, как «Traffic circles», также была бы уместной, но более общей. Таким образом, уровень детализации при выборе релевантных категорий, связанных с подобными статьями, может варьироваться и влиять на оценку верности, которую вы рассчитываете. Кроме того, нужно проанализировать, являются ли категории ключевыми словами, взятыми из текста. В случае с Википедией нет, но в целом это не всегда так. Можно подумать о том, чтобы расширить способ измерения верности, включив не только категории, к которым принадлежит документ, но и важные слова или понятия, упомянутые в тексте. Например, на странице «Ledgewood Circle» содержится раздел о противоречии, возникшем еще в 1990-х годах по поводу дерева, посаженного в центре транспортной развязки. Эта информация никак не представлена в категориях. Если вы хотите иметь возможность извлекать понятия, обсуждаемые на странице, можете добавить их в качестве дополнительных категорий (в этом случае это может быть общая категория «Споры»). Также можно

рассматривать это как маркировку каждого документа набором общих меток: это могут быть категории, понятия, упомянутые в тексте, важные слова и т. д. Суть состоит в том, что ваша мера верности так же хороша, как метки или категории, прикрепленные к документам. С другой стороны, способ создания и использования категорий может оказать значительное влияние на ваши оценки.

Во-вторых, правильно ли вы использовали метрику? Вы извлекли категории входного документа и сопутствующий контент, чтобы увидеть, принадлежала ли им какая-либо категория. На странице «Ledgewood Circle» нет категории «Traffic circle», но ее категорию «Traffic circles in New Jersey» можно рассматривать как подкатегорию более общей категории «Traffic circle». Распространив эти рассуждения на все категории в Википедии, можно представить себе дерево как то, что показано на рис. 6.6: узлы – это категории, и чем глубже узел, тем более конкретным и детализированным будет его категория.

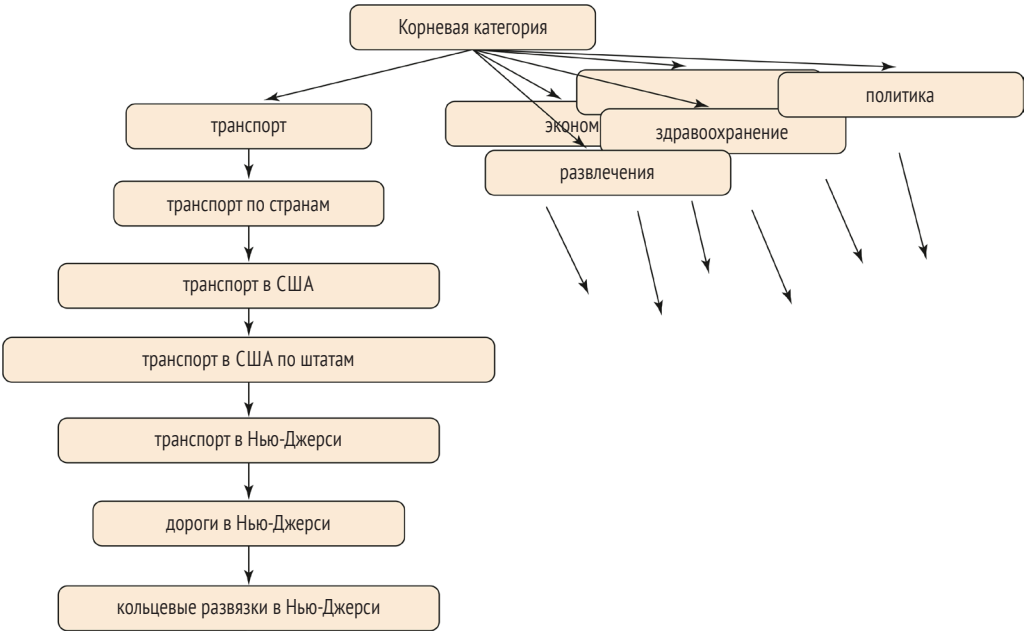


Рис. 6.6 ❖ Создание таксономии из категорий в Википедии

В этом эксперименте вы можете изменить правило сопоставления категорий с «по крайней мере, одна категория должна быть общей как для входа, так и для сопутствующего контента» на «по крайней мере, одна категория должна быть общей для входа и сопутствующего контента, или одна из категорий одного документа должна быть спецификацией другой категории в другом документе». Если вы знаете больше о том, какие отношения существуют между категориями (и метками в целом), то также можете использовать эту информацию. DBpedia можно использовать в качестве одного из таких источников информации об отношениях, существующих между страницами. Представьте, что алгоритм возвращает страницу «New Jersey», связанную с «Ledgewood Circle». Главное, что у них

общего, – это Леджвуд Сёркл, расположенный в штате Нью-Джерси, а именно в Роксбери. Если такая информация доступна, это отличная ссылка, по которой можно перемещаться, чтобы измерить релевантность существующего контента. Например, вы можете пометить как релевантные связанные документы, которые имеют какое-либо отношение к входному документу, или пометить документы как релевантные только в том случае, если они связаны каким-либо подмножеством существующих отношений.

Проект DBpedia записывает много таких отношений между страницами из Википедии. Можно рассматривать это как граф, узлами которого являются страницы; дуги – это отношения (с именем). На рис. 6.7 показаны отношения между Леджвуд Сёркл и Нью-Джерси с использованием RelFinder (**http://www.visualdataweb.org/relfinder.php**).

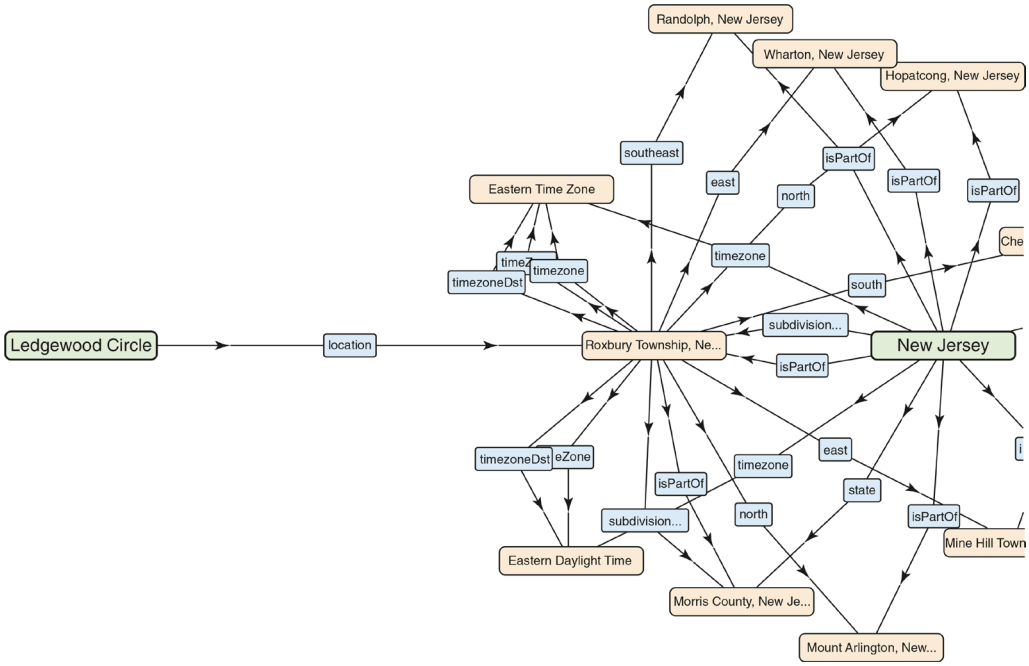


Рис. 6.7 ❖ Связи между страницами «Ledgewood Circle» и «New Jersey» в DBpedia

Наличие хорошей иерархической таксономии для категорий важно, когда вы используете их для измерения верности результатов из `MoreLikeThis` и других алгоритмов. С другой стороны, информация о категориях и их отношениях часто не доступна на практике; в таких случаях методы, основанные на неконтролируемом обучении, могут помочь вам определить, похожи ли два документа. Рассмотрим алгоритмы для изучения векторных представлений текста, таких как word2vec (для слов) или векторы абзаца (для последовательностей слов): когда вы наносите их на график, похожие слова или документы будут располагаться рядом друг с другом. В этом случае вы можете сгруппировать ближайшие векторы, чтобы

сформировать *кластеры* (есть несколько способов сделать это, но мы не будем их здесь рассматривать), и рассматривать связанные слова или документы как принадлежащие к одному кластеру. В следующем разделе мы рассмотрим один из наиболее простых вариантов использования векторных представлений документов: извлечение аналогичного контента.

6.3.3. Извлечение аналогичного контента с помощью векторов абзаца

Вектор абзаца изучает фиксированное (распределенное) векторное представление для каждой последовательности слов, вводимых в архитектуру нейронной сети. Вы можете подать в сеть документ целиком либо его части, например разделы статьи, абзацы или предложения. Решать вам. Например, если вы отправляете в сеть целые документы, можно попросить ее вернуть наиболее похожий документ, который она уже видела. Каждый принятый документ (и сгенерированный вектор) идентифицируется меткой.

Давайте вернемся к проблеме поиска сопутствующего контента для поисковой системы на страницах Википедии. В предыдущем разделе мы использовали `MoreLikeThis` для извлечения наиболее важных термов, а затем использовали их в качестве запроса для извлечения сопутствующего контента. К сожалению, показатели верности были низкими, прежде всего по следующим причинам:

- самые важные термы, извлеченные `MoreLikeThis`, были неплохими, но могли бы быть и лучше;
- если вы посмотрите на набор важных термов из документа, то можете не узнать, из каких они документов.

Давайте еще раз посмотрим на нашего друга на странице «Ledgewood Circle». Согласно MLT, наиболее важными термами являются:

```
record govern left depart west onto intersect 1997 wish move cite turn
    township signal 10 lane travel westbound new eastbound us tree 46
    traffic ref
```

Невозможно будет сказать, что эти термы взяты со страницы «Ledgewood Circle», поэтому нельзя ожидать очень точных предложений сопутствующего контента. В случае с векторными представлениями документов нет никакой явной информации, на которую можно смотреть (это общая проблема, существующая в глубоком обучении: непросто понять, что делают эти черные ящики). Нейронная сеть с вектором абзаца корректирует значения вектора каждого документа во время обучения, как описано в главе 5.

Давайте получим сопутствующий контент, найдя ближайшие векторы к вектору, представляющему входной документ, используя косинусное сходство. Для этого вы сначала выполняете введенный пользователем запрос, например «Ledgewood Circle», который возвращает результаты поиска. Для каждого такого результата вы извлекаете его векторное представление и смотрите на его ближайших соседей в пространстве векторных представлений. Это похоже на навигацию по графику или карте, на которой все документы представлены в соответствии с их семантическим сходством. Вы идете к точке, которая обозначает «Ledgewood Circle», находите ближайшие точки и видите, какие документы они представляют. Вы заметите, что соседи вектора «Ledgewood Circle» будут представлять докумен-

ты, относящиеся к дорожному движению и транспорту; если вы вместо этого выберете, например, векторы документов, относящихся к музыке, то увидите, что они будут расположены далеко от «Ledgewood Circle» и его соседей в пространстве представлений (*см.* рис. 6.8).

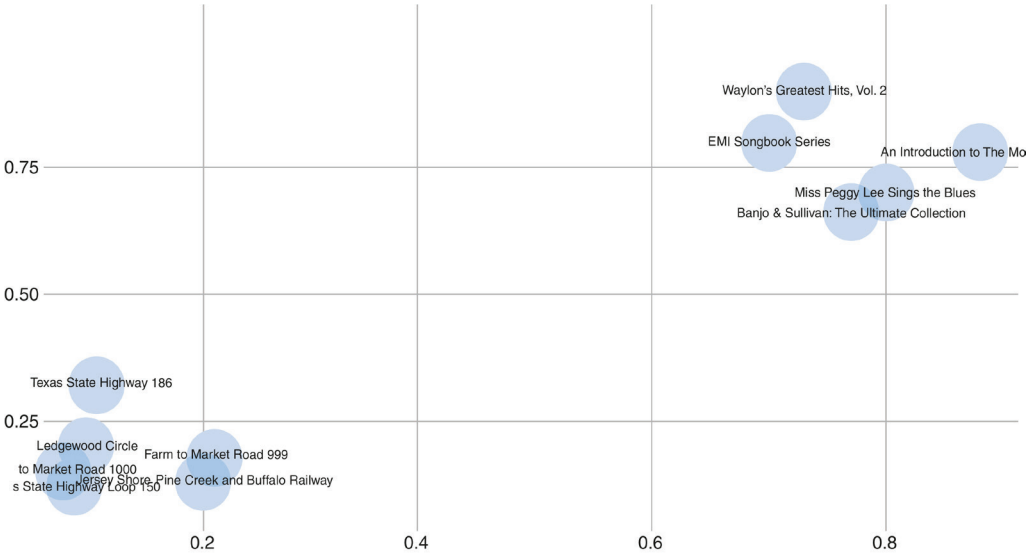


Рис. 6.8 ❖ Векторы абзаца для «Ledgewood Circle» и его соседей по сравнению с векторами абзаца, связанными с музыкой

Аналогично тому, что вы делаете для ранжирования, вы сначала передаете в сеть индексированные данные:

```
File dump = new File("/path/to/wikipedia-dump.xml");
WikipediaImport wikipediaImport = new WikipediaImport(dump,
    languageCode, true);
wikipediaImport.importWikipedia(writer, ft);
IndexReader reader = DirectoryReader.open(writer);
FieldValuesLabelAwareIterator iterator = new
    FieldValuesLabelAwareIterator(reader, fieldName);
ParagraphVectors paragraphVectors = new ParagraphVectors.Builder()
  .iterate(iterator)
  .build();
paragraphVectors.fit();
```

Как только это будет сделано, вы можете использовать встроенный в DL4J метод `nearestLabels`, чтобы найти векторы документов, наиболее близкие к вектору «Ledgewood Circle». Внутренне этот метод использует косинусное сходство, чтобы измерить, насколько близки два вектора:

```
TopDocs hits = searcher.search(query, 10);  <-- Выполняет исходный запрос
for (int i = 0; i < hits.scoreDocs.length; i++) {
  ScoreDoc scoreDoc = hits.scoreDocs[i];
  Document doc = searcher.doc(scoreDoc.doc);
```

```
  String label = "doc_" + scoreDoc.doc;
  INDArray labelVector = paragraphVectors
    .getLookupTable().vector(label);
  Collection<String> docIds = paragraphVectors
    .nearestLabels(labelVector, topN);
  for (String docId : docIds) {
    int docId = Integer.parseInt(docId.substring(4));
    Document document = reader.document(docId);
    System.out.println(document.get("title"));
  }
}
```

Для каждого результата создается метка

Получает векторное представление документа для результата поиска

Находит метки ближайших к вектору результатов поиска векторов

Для каждого ближайшего вектора парсит его метку и получает соответствующий документ Lucene

Результаты приведены ниже:

```
Texas State Highway 186
Texas State Highway Loop 150
Farm to Market Road 1000
Jersey Shore, Pine Creek and Buffalo Railway
Farm to Market Road 999
```

Посмотрите на этот простой пример. Результаты выглядят лучше тех, что дает MLT. Результатов не по теме нет: все они относятся к транспорту (тогда как MLT вернул страницу «Модальная дисперсия», которая относится к оптике).

Для подтверждения своих положительных ощущений, можно сделать то же, что вы уже делали, чтобы измерить эффективность `MoreLikeThis`, рассчитав среднюю верность этого метода. Чтобы провести справедливое сравнение, используйте тот же подход для проверки, присутствуют ли какие-либо категории результатов поиска (например, «Ledgewood Circle») в категориях сопутствующего контента. При примененении тех же случайно сгенерированных запросов, что и при оценке MLT, векторы абзацев дают следующую цифру:

```
paragraph vectors average accuracy : 0.37
```

Лучшая средняя верность для MLT составила 0,09; 0,37 – намного лучше.

Поиск похожих документов с близкой семантикой – одно из ключевых преимуществ использования векторных представлений документов. Вот почему они так полезны при обработке естественного языка и поиске. Как вы уже видели, они могут использоваться по-разному, в том числе для ранжирования и поиска аналогичного контента. Хотя векторы абзацев – не единственный способ, с помощью которого можно изучать векторные представления документов. Вы использовали усредненные векторные представления слов в главе 5, но специалисты продолжают работать над улучшенными и более продвинутыми способами извлечения представлений слов и документов.

6.3.4. Извлечение аналогичного контента с помощью векторов из моделей «кодер–декодер»

В главах 3 и 4 вы познакомились с архитектурой глубокой нейронной сети под названием модель *кодер–декодер* (или модель *seq2seq*). Возможно, вы помните, что эта модель состоит из LSTM-сети-кодера и LSTM-сети-декодера. Кодер преобразует входную последовательность слов в плотный вектор фиксированной длины в качестве выхода, который является входом для декодера, превращающего его обрат-

но в последовательность слов в качестве окончательного результата (см. рис. 6.9). Вы использовали такую архитектуру, чтобы производить представления альтернативных запросов и помочь пользователям ввести запрос. В этом случае вы, напротив, хотите использовать выход сети-кодера, так называемый *вектор мысли*.

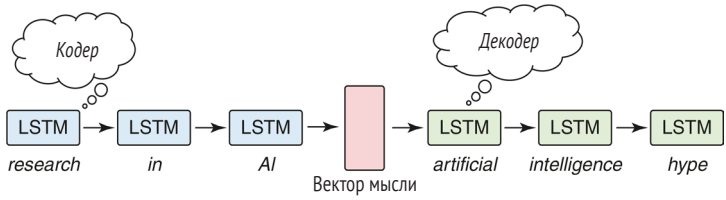


Рис. 6.9 ❖ Модель кодер–декодер

Причина, по которой он называется вектором мысли, заключается в том, что он представляет собой сжатое представление входной текстовой последовательности, которое при правильном декодировании генерирует желаемую выходную последовательность. Модели Seq2seq, как вы увидите в следующей главе, также используются для машинного перевода; они могут преобразовать предложение на языке ввода в переведенную последовательность. Вам нужно извлечь такие векторы мышления для входных последовательностей (документы, предложения и т. д.) и использовать их так же, как вы использовали векторы абзацев для измерения сходства между документами.

Во-первых, нужно подключиться к этапу обучения, чтобы вы могли «сохранять» векторные представления, поскольку они генерируются по одному шагу за раз. Вы помещаете их в `WeightLookupTable`. Это сущность, ответственная за хранение векторов слов в word2vec и векторов абзацев в объектах `ParagraphVectors`. С DL4J вы можете подключиться к обучающей фазе с помощью `TrainingListener`, который фиксирует прямой проход, когда вектор мысли генерируется LSTM-сетью-кодером. Вы извлекаете входной вектор и преобразуете его обратно в последовательность, извлекая слова по одному из исходного корпуса. Затем извлекаете вектор мысли и помещаете последовательность с вектором мысли в `WeightLookupTable`.

Листинг 6.3 ❖ Извлечение векторов мысли

```
public class ThoughtVectorsListener implements TrainingListener {
  @Override
  public void onForwardPass(Model model,
        Map<String, INDArray> activations) {
    INDArray input = activations.get(
        inputLayerName);
    INDArray thoughtVector = activations.get(
        thoughtVectorLayerName);
    for (int i = 0; i < input.size(0); i++) {
      for (int j = 0; j < input.size(1); j++) {
        int size = input.size(2);
        String[] words = new String[size];
        for (int s = 0; s < size; s++) {
          words[s] = revDict.get(input.getDouble(i, j, s));
        }
```

Берет сетевой вход (последовательность слов, преобразованная в векторы) из входного слоя

Извлекает вектор мысли из слоя вектора мысли

Перестраивает последовательность по одному слову за раз из входного вектора

```
        String sequence = Joiner.on(' ')
            .join(words);
        lookupTable.putVector(sequence, thoughtVector
            .tensorAlongDimension(i, j));
        }
      }
    }
  }
}
```

Объединяет слова в последовательность (в виде строки)

Записывает вектор мысли, ассоциированный с входной текстовой последовательностью

Используя эти векторы, вы можете достичь того же уровня верности, что и векторы абзацев; разница заключается в том, что вы можете решать, как влиять на них. Эти векторы мысли генерируются как промежуточный продукт LSTM-сетей кодера и декодера.

Вы можете решить, что включать во вход кодера и что включать в выход декодера, на этапе обучения. Если вы поместите документы, принадлежащие к той же категории, по краям сети, сгенерированные векторы мысли научатся выводить документы, категории которых совпадают. Следовательно, вы можете добиться гораздо более высокой верности.

Если вы возьмете кодер–декодер LSTM, определенный в главах 3 и 4, и обучите его, используя документы, относящиеся к одной и той же категории, то получите среднюю верность 0,77. Это намного выше, даже если сравнивать с векторами абзаца!

Резюме

- Модели векторов абзаца предоставляют распределенные представления для предложений и документов с настраиваемой степенью деления (предложение, абзац или документ).
- Функции ранжирования, основанные на векторах абзацев, могут быть более эффективными, чем старые статистические модели и модели, которые основаны на векторных представлениях слов, поскольку они фиксируют семантику на уровне предложения или документа.
- Векторы абзацев также могут использоваться для эффективного извлечения сопутствующего контента на основе семантики документа, для оформления результатов поиска.
- Векторы мысли можно извлекать из моделей seq2seq для получения сопутствующего контента на основе семантики документа, чтобы украсить результаты поиска.

Часть III

ШАГ ЗА ПРЕДЕЛЫ

В первой части книги вы получили базовое представление о том, что такое поисковые системы и глубокие нейронные сети, как они работают и как могут работать вместе для создания более интеллектуальных поисковых систем. Во второй части мы рассмотрели технические подробности основных приложений на базе нейронных сетей для поисковых систем, в основном с использованием рекуррентных нейронных сетей и векторных представлений слов/документов, чтобы предоставить пользователям более релевантные результаты. В этой части книги мы рассмотрим более сложные темы и задачи, расширив варианты применения нейронных сетей до двух новых областей: поиск текста на нескольких языках с помощью машинного перевода (глава 7) и поиск изображений с использованием сверточных нейронных сетей (глава 8). Наконец, в главе 9 мы рассмотрим то, что имеет наибольшее значение в производственных сценариях: производительность, будь то обычная скорость при обучении и прогнозировании, или верность результатов. Вы увидите пример того, как настроить модель нейронной сети для достижения подходящей верности за разумное время обучения. Кроме того, мы рассмотрим, как работать с непрерывными потоками данных для поиска на базе нейронных сетей.

Глава 7

Поиск по языкам

О чем идет речь в этой главе:

- поиск информации на нескольких языках;
- статистический машинный перевод;
- модели seq2seq для машинного перевода;
- векторные представления слов для машинного перевода;
- сравнение эффективности методов машинного перевода для поиска.

В этой главе мы сосредоточимся на расширении ваших возможностей обслуживания пользователей, которые говорят, читают и пишут запросы на языках, отличных от языка, на котором написаны документы. В частности, вы увидите, как использовать машинный перевод для создания поисковой системы, которая может автоматически переводить запросы, чтобы их можно было использовать для поиска и доставки контента на нескольких языках. Мы потратим некоторое время на изучение того, как такая возможность перевода может быть полезна в различных контекстах, от обычных поисковых запросов в интернете до более конкретных случаев, когда важно не пропустить результаты поиска из-за языкового барьера. Преимущество возможности автоматического перевода запросов заключается в том, что ваши поисковые системы получают возможность охватить больше пользователей, не требуя от вас хранить по нескольку копий каждого текстового документа на разных языках.

7.1. Обслуживание пользователей, говорящих на нескольких языках

Многие из сценариев, представленных в предыдущих главах, фокусировались на вертикальных поисковых системах или поисковых системах, которые характерны для часто небольшой, четко определенной области, такой как поисковая система для обзоров фильмов. В этой главе, где рассматривается проблема получения полезной информации для пользователей, говорящих на разных языках, нет лучшего варианта, чем поиск в интернете или поиск по данным из любой точки мира. Мы ежедневно используем веб-поиск в поисковых системах, таких как Google Search, Bing и Baidu. Хотя большая часть онлайн-контента написана на языках, на которых говорит огромное количество людей (например, на английском), попрежнему существует большое число пользователей, которым необходимо получить информацию и которые надеются найти ее, используя свой родной язык.

Вы можете спросить, в чем смысл этой дискуссии. Если у вас есть страница в Википедии, написанная на итальянском языке, она, безусловно, будет проиндексирована, например, с помощью Google, и вы сможете выполнять поиск, написав запрос в поисковой строке Google на итальянском языке, как показано на рис. 7.1.

Рис. 7.1 ❖ Поиск по запросу «rete neurale», италоязычного аналога сочетания «нейронная сеть»

Хотя это и реалистично, особенно при поиске по темам, связанным с технологиями, часто целесообразнее писать запросы на английском языке. Это связано с тем, что объем информации, доступной на английском языке, особенно по техническим темам, нередко намного превышает объем, написанный на других языках. Пользователь, чей родной язык – итальянский (или датский, или китайский и т. д.), пишет запрос на английском языке, чтобы максимизировать вероятность получения как можно большего количества релевантных результатов. Эти результаты будут включать в себя только те документы, что написаны на английском. Дело в том, что результаты поиска, написанные на английском, не всегда так полезны для пользователей, как результаты на их родном языке. Позвольте мне объяснить. Я покажу, что можно сделать для запроса, написанного на английском языке пользователем, чей родной язык – итальянский. Как видно на рис. 7.2, запрос, написанный на английском языке, также дал результат поиска на итальянском языке, показанный справа. В подобных случаях, когда запрос выполняется зарегистрированным пользователем, поисковая система может искать родной язык пользователя и включать результаты на этом языке в дополнение к результатам, которые соответствуют исходному запросу (в данном случае на английском).

Какую пользу это приносит пользователю? Представьте, что вы читаете свою любимую книгу на своем родном языке, а не на языке, который вы изучали в школе. Даже если вы сможете понять содержание иноязычной версии книги, на это может потребоваться дополнительное время и усилия, и вы можете пропустить некоторые тонкие или особенно сложные части. То же самое относится и к документам в сети. Например, статья в Википедии об «искусственной нейронной сети» существует на многих языках, благодаря чему эту тему легче понять большему количеству пользователей. Представьте себе поисковую систему, которая не только показывает англоязычную статью (которая соответствует запросу, написанному на английском языке), но также выделяет статью, написанную на родном языке

пользователя, который ввел запрос. Эта поисковая система лучше удовлетворяет потребности большего количества пользователей.

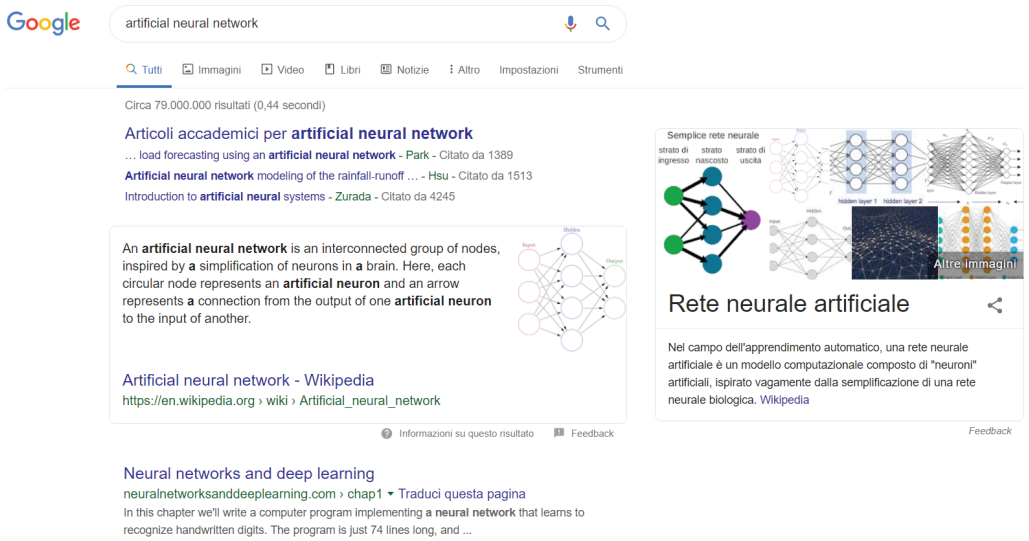


Рис. 7.2 ❖ Поиск по запросу «искусственная нейронная сеть» и получение результатов на итальянском и английском языках

Вы можете оборудовать свою поисковую систему таким образом, чтобы она возвращала результаты обоих типов, включив в нее средства *машинного перевода*. С помощью машинного перевода программа может переводить предложение с языка ввода в соответствующую версию на целевом языке. В оставшейся части этой главы вы узнаете, как использовать средства машинного перевода для выполнения перевода текста во время запроса, что приводит к повышению качества полноты и точности для запросов в поисковых системах на нескольких языках.

7.1.1. Перевод документов в сравнении с переводом запросов

Представьте себе, что вам необходимо создать поисковую систему, обладающую возможностями, аналогичными тем, что были кратко описаны в предыдущем разделе, для некоммерческой организации, которая помогает беженцам по всему миру административными и юридическими услугами. Поисковая система для такой организации поможет беженцам найти соответствующую документацию, например чтобы заполнить запрос на предоставление убежища. Каждая страна в мире, вероятно, требует, чтобы были заполнены и подписаны разные документы и формы; требования могут также варьироваться в зависимости от страны, из которой приехал заявитель. Пользователи такой платформы могут говорить на своем родном языке, но не на языке своей страны пребывания. Поэтому если беженцы из Исландии ищут убежище в Бразилии, им нужно будет получить документы, которые могут быть написаны на португальском языке. Если пользователи не знают португальского, откуда им знать, что включить в поисковый запрос для информации, которую они ищут?

Независимо от ситуации можно предположить, что пользователи хотят иметь возможность получать контент на своем родном языке, когда это возможно. Есть два простых способа сделать это с помощью машинного перевода:

- используйте программы машинного перевода для перевода запросов, чтобы найти совпадения на нескольких языках;
- создавайте контент на одном языке и используйте программы машинного перевода для создания переведенных копий документов, чтобы запросы могли соответствовать переведенным версиям.

Эти варианты не являются взаимоисключающими: вы можете использовать один из них или оба. Что подходит лучше всего, зависит от варианта использования.

Рассмотрим отзывы клиентов на таких сайтах, как Amazon и Airbnb. Подобные отзывы часто пишутся на родном языке, поэтому для облегчения использования результатов поиска может быть полезно перевести эти отзывы, когда они дойдут до пользователя.

Еще один хороший пример для перевода результатов поиска – это системы ответов на вопросы. При ответе на вопросы используется информационно-поисковая система, где пользователь указывает свое намерение в форме вопроса, написанного на естественном языке (например, «Кто был избран президентом США в 2009 году?»). Система выдает ответ: фрагмент (желательно информативного) текста, связанный с вопросом (например, «Барак Обама»).

С другой стороны, для веб-поиска, как обсуждалось в предыдущем разделе, может быть полезно перевести запрос, чтобы получить результаты на разных языках, потому что это дает больше возможностей конечным пользователям. Как только это будет сделано, вам нужно принять важное решение касательно ранжирования: как ранжировать результаты, полученные из переведенного запроса?

В случае когда беженец ищет на исландском языке документы, написанные на португальском, если пользователь вводит запрос «pólitísk hæli» (исландская версия фразы «политическое убежище»), запрос переводится на португальский («asilo politico»). В таких случаях используются результаты как исходных, так и переведенных запросов. В конкретном случае с пользователем, который ищет убежище, документы, возвращенные из переведенного запроса, важнее, потому что именно они должны быть заполнены пользователем и переданы в местные органы власти.

В веб-поиске это не всегда так. Давайте вернемся к странице в Википедии об «искусственных нейронных сетях». Англоязычная версия страницы содержит гораздо больше информации, чем итальянская версия. В зависимости от различных факторов, таких как интересы пользователя и предпочтительные темы, поисковая система может решить ранжировать переведенную страницу ниже, чем оригинал, поскольку она менее информативна. Если специалист, занимающийся глубоким обучением, ищет в сети «искусственные нейронные сети», итальянская версия страницы об «искусственных нейронных сетях» для них не будет полезна, поскольку объем информации тут меньше, по сравнению с исходной страницей на английском языке. Если же пользователь является новичком в этом вопросе, чтение страницы на его родном языке, вероятно, поможет ему понять тему. Хотя многое зависит от варианта использования, если вы решите использовать машинный перевод в поисковой системе, было бы неплохо ранжировать дополнительные результаты так же, как и «нормальные» результаты, или выше.

Оставшаяся часть данной главы посвящена переводу запросов, а не переводу документов; при переводе коротких или длинных фрагментов текста принципы одинаковы. С другой стороны, с технической точки зрения, работа с очень коротким текстом (таким как поисковый запрос) или очень длинным текстом (например, длинная статья) обычно сложнее, чем работа с отдельными предложениями.

7.1.2. Поиск по нескольким языкам

Давайте кратко рассмотрим, как включить машинный перевод в поисковую систему для перевода пользовательских запросов. В поиске по сети машинный перевод обычно выполняется в поисковой системе; пользователю об этом ничего не сообщается. В других упомянутых случаях применения пользователи могут указать желаемый язык для результатов поиска; искатель убежища знает подходящий язык для юридических документов, которые ему необходимы, но эта информация может быть недоступна поисковой системе.

В дальнейшем я предполагаю, что у вас есть набор средств машинного перевода, которые могут выполнять перевод с языка пользовательского запроса на другие языки, и что ваша поисковая система содержит документы на множестве разных языков – это распространенная настройка для поиска информации на разных языках в сети. Инструменты для выполнения машинного перевода могут быть реализованы множеством разных способов; по мере прохождения данной главы вы увидите несколько разных методов машинного перевода. Обычно с помощью таких средств можно переводить текст с исходного языка на *целевой*. Представьте, что у вас есть запрос, написанный на исландском языке, как упоминалось ранее, и у вас есть три модели, которые могут переводить с исландского на английский, с английского на исландский и с итальянского на английский соответственно. Поисковая система должна быть в состоянии выбрать правильный инструмент для перевода запроса. Если вы выберете инструмент перевода с итальянского на английский, перевод может быть невозможен или, что еще хуже, модель может дать плохой перевод. Это может привести к получению нежелательных результатов, что, конечно же, плохо.

Даже когда неподходящая модель не дает перевода, используются ресурсы памяти и ЦП, а следовательно, эта попытка может негативно повлиять на производительность, не давая полезного результата.

Для смягчения таких проблем рекомендуется размещать *детектор языка* поверх моделей машинного перевода. Детектор языка получает входной текст и выводит язык входной последовательности. Можно рассматривать его как классификатор текстов, выходными классами которого являются языковые коды (en, it, ic, pt и т. д.). Используя детектор языка, который предоставляет язык пользовательских запросов, вы можете выбрать подходящую модель машинного перевода для перевода запроса. Выходной текст из всех моделей машинного перевода будет отправлен поисковой системе в качестве дополнительного запроса вместе с исходным запросом; это как использовать логический оператор OR между исходной и переведенной версиями запроса (например, «pólitísk hæli OR political asylum»). На рис. 7.3 показан пример потока для использования машинного перевода во время запроса.

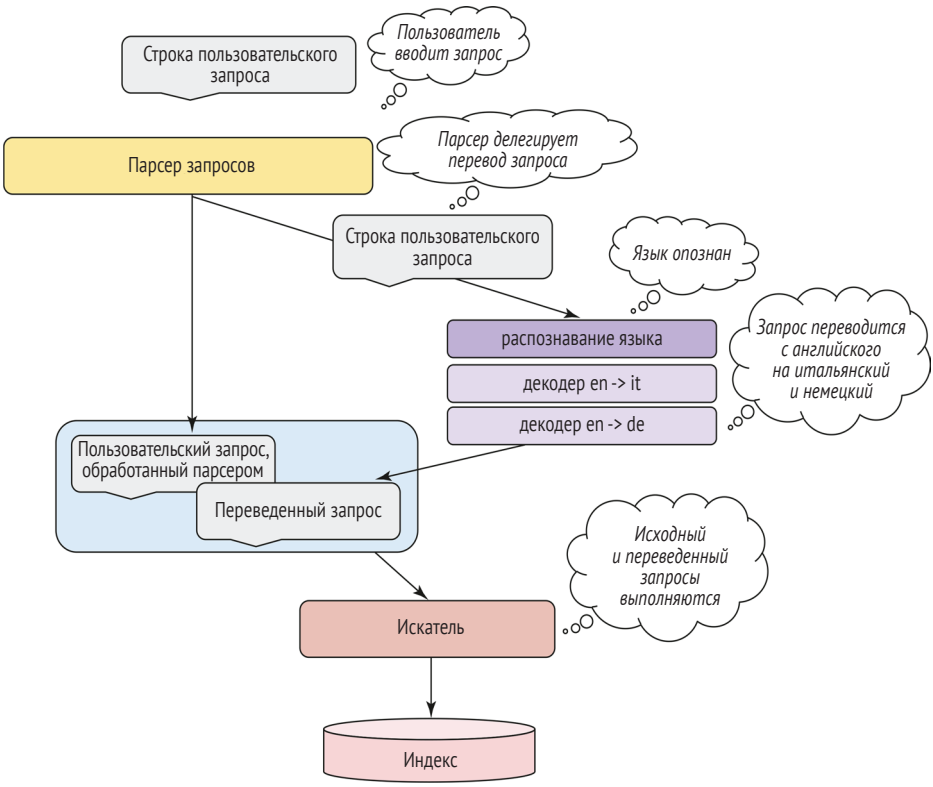


Рис. 7.3 ❖ Поток перевода запроса

Давайте посмотрим, как можно реализовать межъязыковой поиск поверх Apache Lucene. Пока что мы оставим часть, касающуюся машинного перевода, немного абстрактной. В следующих разделах мы рассмотрим различные типы моделей машинного перевода и изучим преимущества и недостатки каждой из них. В частности, мы сосредоточимся на том, почему большинство исследований и отраслей перешло от *статистического машинного перевода* (на основе статистического анализа распределения вероятностей для слов и фраз) на машинный перевод на базе нейронных сетей.

7.1.3. Запросы на нескольких языках поверх Lucene

Давайте продолжим с примером, где идет речь о просителях убежища. Предположим, я итальянский беженец, находящийся в Соединенных Штатах, и мне нужно заполнить некие юридические документы. Я набираю запрос на итальянском, ищу документы для въезда в США. Вот что должна делать поисковая система:

```
> q: documenti per entrare negli Stati Uniti   <-- Входной запрос
> detected language 'ita' for query            <-- Вывод,где определяется язык
> found 1 translation
> t: documents to enter in the US              <-- Переведенный запрос
```

```
> 'documenti per entrare negli ...' parsed as:
 '(text:documenti text:per text:entrare text:negli text:Stati text:Uniti)'
 OR
 '(text:documents text:to text:enter text:in text:the text:US)'
```

Расширенный запрос, содержащий как исходные, так и переведенные запросы, разделенные логическим оператором OR

Как вы, наверное, догадываетесь, «волшебство» происходит во время парсинга введенного пользователем запроса. Вот упрощенная последовательность операций, выполняемых анализатором запросов:

1) парсер запросов читает входной запрос;
2) парсер передает входной запрос детектору языка;
3) детектор языка определяет язык входного запроса;
4) парсер выбирает модели машинного перевода, которые могут переводить идентифицированный язык на другие языки;
5) каждая выбранная модель переводит входной запрос на другой язык;
6) парсер объединяет входной и переведенный текст в операторах запроса OR.

Мы расширим `QueryParser`, основной метод которого `#parse` преобразует строку в объект `Query`.

Листинг 7.1 ❖ Создание BooleanQuery, содержащего исходный запрос

```
@Override
public Query parse(String query) throws ParseException {

  BooleanQuery.Builder builder = new BooleanQuery
      .Builder();
  builder.add(new BooleanClause(super.parse(query),
      BooleanClause.Occur.SHOULD));
  ...
}
```

Создает булев запрос в Lucene

Анализирует исходный пользовательский запрос и добавляет его к булеву запросу в качестве оператора OR

Затем язык входного запроса извлекается детектором языка. (Есть много разных способов, с помощью которых это можно сделать; сейчас мы не будем на этом фокусироваться.) Мы будем использовать инструмент `LanguageDetector` из проекта Apache OpenNLP (**http://opennlp.apache.org**).

Листинг 7.2 ❖ Определение языка запроса

```
Language language = languageDetector.
  predictLanguage(query);
String languageCode = language.getLang();
```

Выполняет определение языка

Получает код языка (en, it и т. д.)

Здесь предполагается, что вы уже загрузили модели для выполнения машинного перевода, например в `Map`, ключом которой является код языка (`en` для английского языка, `it` для итальянского языка и т. д.) и значением которой является коллекция `TranslatorTools`. На данный момент не имеет значения, как реализован `TranslatorTool`; мы сосредоточимся на этом в следующих разделах.

Листинг 7.3 ❖ Выбор правильных средств TranslatorTools

```
private Map<String,Collection<TranslatorTool>> perLanguageTools;

@Override
public Query parse(String query) throws ParseException {
  ...
  Collection<TranslatorTool> tools =
      perLanguageTools.get(languageString);
  ...
}
```

Получает инструменты, которые могут выполнять перевод с обнаруженного языка на другие языки

Теперь, когда у вас загружены средства машинного перевода, вы можете использовать их для создания дополнительных логических операторов, которые будут добавлены в окончательный запрос.

Листинг 7.4 ❖ Перевод запроса и создание запроса с переведенным текстом

```
for (TranslatorTool tt : tools) {
  Collection<Translation> translations = tt.
      translate(query);
  for (Translation translation : translations) {
    String translationString = translation.
      getTranslationString();
    builder.add(new BooleanClause(super.parse(
      translationString), BooleanClause.Occur.SHOULD));
  }
}
return builder.build();
```

Переводит входной запрос

Перебирает все возможные переводы входного запроса

Получает текст перевода (каждый перевод состоит из текста и его оценки, представляющей качество перевода)

Анализирует переведенный запрос и добавляет его к булеву запросу для возврата

Завершает процесс создания булева запроса

После этого у вас все настроено с помощью анализатора запросов, который позволяет создавать запросы на нескольких языках. Недостающая часть реализует интерфейс `TranslatorTool` наилучшим образом. Для этого мы кратко рассмотрим различные способы решения задачи, связанной с машинным переводом. Сначала рассмотрим статистический машинный перевод, а затем перейдем к методам на базе нейронных сетей; это поможет вам понять основные проблемы перевода текста и то, как использование моделей на базе нейронных сетей обычно обеспечивает более качественный машинный перевод.

7.2. Статистический машинный перевод

Статистический машинный перевод использует статистические подходы, чтобы спрогнозировать, какое целевое слово или предложение является наиболее вероятным переводом входного слова или предложения. Например, программа для статистического машинного перевода должна быть в состоянии ответить на вопрос: «Каков наиболее вероятный перевод на английский язык слова “hombre”?» Для этого нужно обучить статистическую модель, используя *параллельный корпус*. Параллельный корпус представляет собой набор текстовых фрагментов (документы, предложения или даже слова), где каждый фрагмент контента представлен в двух версиях: исходный язык (например, испанский) и целевой язык (например, английский). Вот пример:

```
s: a man with a suitcase
t: un hombre con una maleta
```

Статистическая модель – это модель, которая может рассчитать вероятность исходного и целевого фрагментов текста. Правильно обученная статистическая модель для машинного перевода ответит на вопрос о наиболее вероятном переводе фрагмента текста, предоставив перевод наряду с его вероятностью:

```
hombre -> man (0.333)
```

Вероятность фрагмента переведенного текста поможет вам решить, можно ли считать перевод хорошим и, следовательно, следует ли использовать его в поиске. Модель статистического машинного перевода оценивает вероятность множества возможных переводов и возвращает только перевод с самой высокой вероятностью. Если вы попросите модель вывести все вероятности для запроса «hombre», то увидите высокие вероятности для хороших переводов и низкие вероятности для несвязанных переводов, как в этом выводе:

```
man     (0.333)
husband (0.238)
love    (0.123)
...
woman   (0.003)
truck   (0.001)
...
```

Под капотом модель вычисляет вероятность каждого возможного перевода и записывает вариант перевода с наилучшей вероятностью. Псевдокод такого алгоритма выглядит так[1]:

```
f = 'hombre'
for (each e in target language)
    p(e|f) = (p(f|e) * p(e)) / p(f)
    if (p(e|f) > pe~)
      e~ = e
      pe~ = p(e|f)
e~ = best translation, the one with highest probability
pe~ = the probability of the best translation
```

Рассчитывает вероятность текущего целевого слова с учетом исходного слова «hombre»

Если вероятность выше текущей наибольшей вероятности, вы получаете новый наиболее подходящий перевод

Записывает наиболее подходящий перевод

Записывает вероятность наиболее подходящего перевода

Алгоритм не сложен; единственная недостающая часть – как вычислить вероятности, такие как p (*e*) и p (*f*|*e*). В теории информации и статистике p (*f* |*e*) является условной вероятностью *e*, заданной *f*. В целом можно рассматривать это как вероятность того, что событие *e* произойдет как следствие события *f*. В таком случае «события» – это фрагменты текста! Не углубляясь в статистику, можно рассматривать вероятности слова, полагаясь на подсчет частот слов. Например, p(man) будет равно числу раз, когда слово *man* появляется в параллельном корпусе. Точно так же вы можете предположить, что p (hombre|man) равно числу раз, которое слово *man* встречается в предложении на целевом языке, которое спарено с предложе-

[1] См. также теорему Байеса: **https://en.wikipedia.org/wiki/Bayes%27_theorem**.

нием на испанском языке, содержащим слово *hombre*. Давайте рассмотрим приведенные ниже три параллельных предложения: два из них содержат слово *man* на исходном языке и слово *hombre* в целевом предложении; другое содержит слово *man* в исходном предложении, но не hombre в целевом:

```
s: a man with a suitcase
t: un hombre con una maleta

s: a man with a ball
t: un hombre con una pelota

s: a working man
t: un senor trabajando
```

В этом случае p (hombre | man) равно 2. В другом примере в параллельных предложениях p (senor | man) равно 1, потому что третье параллельное предложение содержит слово *man* в исходном предложении и *senor* в целевом предложении. В итоге слово *hombre* переводится как *man*, потому что среди множества возможных альтернатив *man* – слово, которое чаще всего используется, когда в предложении на испанском языке содержится слово *hombre*.

Вы познакомились с основами статистического машинного перевода. Вы также узнаете о трудностях, которые делают эту задачу более сложной, чем может показаться из данного введения; о них важно знать, потому что машинный перевод на базе нейронных сетей в меньшей степени подвержен таким проблемам, что частично обосновывает причины текущего перехода со статистического машинного перевода на нейронный машинный перевод.

7.2.1. Выравнивание

В предыдущем разделе вы узнали, что можно создать статистическую модель для перевода текста. Этот перевод происходит путем оценки вероятностей на основе частоты слов. На практике, однако, существуют и другие факторы. Например, совпадение двух слов *f* и *e* в двух исходных и целевых предложениях не означает, что одно является переводом другого. В ранее упомянутых предложениях слова *a* и *hombre* встречаются чаще, чем *hombre* и *man*:

```
s: a man with a suitcase
t: un hombre con una maleta

s: a man with a ball
t: un hombre con una pelota

s: a working man
t: un senor trabajando
```

Итак, p(hombre | a) = 3 и p(hombre | man) = 2. Означает ли это, что *a* – это английский эквивалент слова *hombre*? Конечно, нет! Эта информация важна при принятии решения, какой перевод слова *hombre* является верным: *a* или *man*.

Но переведенные слова не всегда идеально выровнены. Рассмотрим третье параллельное предложение: в этом контексте правильный перевод слова *man* – это *senor*. Но слово *man* находится на третьей позиции в исходном предложении, тогда как *senor* находится на второй позиции в целевом предложении:

```
s: a working man
t: un senor trabajando
```

Задача работы со словами, размещенными в разных позициях в исходных и целевых предложениях, называется *выравниванием слов*, и она играет важную роль в эффективности статистического машинного перевода. Модели статистического машинного перевода обычно определяют *функцию выравнивания,* которая отображает, например, целевое слово на испанском языке в позиции *i* в исходное слово на английском в позиции *j*. Отображение для предложения преобразует позиции в соответствии с индексами 1 → 1, 2 → 3, 3 → 2:

```
s: a working man        ←—— «a» и «un» находятся в одной позиции
   ↓       ↙
t: un senor trabajando ←—— «man» и «senor» – на одну позицию друг от друга
```

Еще один пример, где выравнивание слов играет важную роль, – это когда нет однозначного соответствия между словами на разных языках. Это особенно справедливо для языков, которые не происходят от того же корневого языка. Возьмем другой пример англо-испанского параллельного предложения:

```
s: I live in the USA
t: vivo en Estados Unidos
```

Здесь два особых случая:

- слова *I live* переведены на испанский одним словом *vivo*;
- слово *USA* переведено как два слова *Estados Unidos*.

Функция выравнивания слов должна также позаботиться о таких случаях:

```
s: I live in the USA
    ↘↙          ↙↘
t: vivo en Estados Unidos
```

7.2.2. Перевод на основе фраз

До сих пор мы обсуждали, как переводить отдельные слова. Но, как и во многих других областях обработки естественного языка, перевод одного слова затруднен без знания контекста. Перевод на основе фраз направлен на уменьшение количества ошибок из-за недостатка информации при переводе отдельных слов. Как правило, выполнение такого перевода требует больше данных для обучения хорошей статистической модели, но она может лучше обрабатывать более длинные предложения, и часто она более точна, чем статистические модели на базе слов. Все, что вы узнали о моделях статического машинного перевода на основе слов, применимо к моделям на основе фраз; единственное отличие состоит в том, что единицы перевода – это не слова, а фразы.

Когда модель на базе фраз получает входной текст, она разбивает его на фразы. Каждая фраза переводится независимо, а затем переводы для каждой фразы переупорядочиваются с использованием функции выравнивания фраз. До того, как нейронные модели для машинного перевода пришли к успеху, фразовые (и иерархические) модели статического машинного перевода были стандартом де-факто машинного перевода и использовались во многих инструментах, например в Google Translate.

7.3. Работа с параллельными корпусами

Как вы, вероятно, понимаете, одним из наиболее важных аспектов машинного обучения является наличие большого количества качественных данных. Модели машинного перевода обычно обучаются на параллельных корпусах: это (текстовые) наборы данных, предоставляемые на двух языках, поэтому слова, предложения и т. д., написанные на исходном языке, можно сопоставить со словами, предложениями и т. д. на целевом языке.

Очень полезным ресурсом для тех, кто интересуется машинным переводом, является Open Parallel Corpus (OPUS, **http://opus.nlpl.eu**). Он предоставляет множество параллельных ресурсов; вы можете выбрать исходный и целевой языки, и вам будет показан список параллельных корпусов в разных форматах. Каждый параллельный корпус обычно предоставляется в разных форматах XML или выделенных форматах машинного перевода, как, например, в проекте Moses (**www.statmt.org/moses**). Иногда также доступны словари перевода с частотами слов.

В этом контексте давайте настроим небольшой инструмент для парсинга формата Translation Memory eXchange (TMX) (**https://en.wikipedia.org/wiki/Translation_Memory_eXchange**). Хотя спецификация TMX не нова, многие существующие параллельные корпуса доступны в формате TMX в проекте OPUS, поэтому полезно иметь возможность работать с TMX при обучении своей первой модели нейронного машинного перевода.

Формат TMX-файла использует по одному XML-узлу `tu` на каждое параллельное предложение. У каждого узла `tu` есть два дочерних элемента `tuv`: один для исходного предложения и один для целевого. И у каждого из этих узлов есть узел `seg`, содержащий фактический текст.

Вот пример TMX-файла перевода с английского на итальянский:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tmx version="1.4">
<header creationdate="Wed Jul 30 13:12:22 2014"
          srclang="en"
          adminlang="en"
          o-tmf="unknown"
          segtype="sentence"
          creationtool="Uplug"
          creationtoolversion="unknown"
          datatype="PlainText" />
  <body>
    ...
    <tu>
      <tuv xml:lang="en">
        <seg>
            It contained a bookcase: I soon possessed myself of a volume.
        </seg>
      </tuv>
      <tuv xml:lang="it">
        <seg>
            Vi era una biblioteca e io m'impossessai di un libro.
        </seg>
      </tuv>
```

```
    </tu>
    ...
  </body>
</tmx>
```

В конце концов, вы заинтересованы в получении содержимого XML-узлов tuv и seg. Вы хотите собрать параллельные предложения, где можно получить исходный и целевой тексты. Для этого сначала создадим класс ParallelSentence.

Листинг 7.5 ❖ Класс для параллельных предложений

```
public class ParallelSentence {

  private final String source;
  private final String target;

  public ParallelSentence(String source, String target) {
    this.source = source;
    this.target = target;
  }

  public String getSource() {
    return source;
  }

  public String getTarget() {
    return target;
  }
}
```

Теперь давайте создадим класс TMXParser для извлечения коллекции параллельных предложений из TMX-файлов.

Листинг 7.6 ❖ Парсинг и перебор параллельного корпуса

```
TMXParser tmxParser = new TMXParser(Paths.get("/path/to/it-en-file.tmx")
    .toFile(), "it", "en");
Collection<ParallelSentence> parse = tmxParser.parse();
for (ParallelSentence ps : parse) {
  String source = ps.getSource();
  String target = ps.getTarget();
  ...
}
```

TMXParser изучит все узлы tu, tuv и seg и создаст коллекцию:

```
  public TMXParser(final File tmxFile, String
    sourceCode, String targetCode) {
    ...
  }
```

Создает парсер для файла TMX с указанием исходного и целевого языков

```
  public Collection<ParallelSentence> parse() throws IOException,
        XMLStreamException {
    try (final InputStream stream = new
          FileInputStream(tmxFile)) {
      final XMLEventReader reader = factory
          .createXMLEventReader(stream);
      while (reader.hasNext()) {
```

Читает файл

Создает XMLEventReader: служебный класс, который генерирует события каждый раз во время чтения XML-элементов

Перебирает каждое XML- событие (узлы, атрибуты и т. д.)

```
      final XMLEvent event = reader.nextEvent();
      if (event.isStartElement() && event.asStartElement().getName()
          .getLocalPart().equals("tu")) {   <-- Перехватывает узлы tu
        parse(reader);   <-- Парсит узлы tu и считывает содержащиеся параллельные предложения
      }
    }
  }
  return parallelSentenceCollection;
}
```

Мы не будем слишком углубляться в код для извлечения параллельных предложений, потому что парсинг XML здесь не является основной целью. Для полноты картины приведем важную часть метода `parseEvent`:

```
if (event.isEndElement() && event.asEndElement()
    .getName().getLocalPart().equals("tu")) {   <-- Закрытие элемента tu. Параллельное предложение готово
  if (source != null && target != null) {
    ParallelSentence sentence = new ParallelSentence(source, target);
    parallelSentenceCollection.add(sentence);
  }
  return;
}
if (event.isStartElement()) {
  final StartElement element = event.asStartElement();
  final String elementName = element.getName().getLocalPart();
  switch (elementName) {
    case, "tuv":   <-- Читает код языка из элемента tuv
      Iterator attributes = element.getAttributes();
      while(attributes.hasNext()) {
        Attribute next = (Attribute) attributes.next();
        code = next.getValue();
      }
      break;
    case "seg":   <-- Читает текст из элемента seg
      if (sourceCode.equals(code)) {
        source = reader.getElementText();
      } else if (targetCode.equals(code)) {
        target = reader.getElementText();
      }
      break;
  }
}
```

Используя сгенерированные параллельные предложения, вы можете обучать модель машинного перевода – либо статистическую, как описано в предыдущем разделе, либо нейронную, которую вы увидите далее.

7.4. Нейронный машинный перевод

Располагая всеми этими знаниями о статистическом машинном переводе и параллельных корпусах, вы теперь готовы узнать, почему и как используются нейронные сети в контексте машинного перевода, применительно к поиску. Представьте, что вы инженер, задача которого состоит в создании поисковой системы

для некоммерческой организации, которая помогает беженцам со всего мира собирать информацию о необходимых юридических документах для каждой страны. Вам нужны модели машинного перевода для максимально возможного количества языковых пар (например, с испанского на английский, с суахили на английский, с английского на испанский и т. д.). Обучающие статистические модели, основанные на оценке явной вероятности, как обсуждаемые ранее модели на базе слов или фраз, отнимали бы много времени из-за большого количества ручной работы, которую обычно требует такой подход. Например, выравнивание слов потребует большого количества работы для каждой из языковых пар.

Когда появились первые модели нейронного машинного перевода, одной из их самых интригующих особенностей было то, что они не требовали особой настройки. Когда Илья Суцкевер представил работу, которую он и его соавторы проделали, работая с архитектурой кодер–декодер для нейронного машинного перевода[1], он заявил: «Мы используем минимальные инновации для достижения максимальных результатов»[2]. Это оказалось одним из лучших качеств данного типа модели.

В этом подходе используется сеть с глубокой долгой краткосрочной памятью (LSTM), выход которой представляет собой большой вектор, *вектор мысли*, упомянутый в главе 3, а затем передает последовательность (и вектор мысли) в другой LSTM-декодер, который генерирует переведенную последовательность. Со временем были предложены различные «ароматы» моделей нейронного машинного перевода, но основная идея использования сети кодер–декодер была важной вехой: это была первая модель, полностью основанная на нейронных сетях, которая превзошла модели статистического машинного перевода.

Эти модели являются гибкими для отображения последовательностей в последовательности в разных областях, а не только для машинного перевода. Например, вы использовали модели кодировщика–декодера seq2seq для выполнения расширения запросов в главе 3, а векторы мысли – для извлечения сопутствующего контента в главе 6. Теперь мы несколько более подробно рассмотрим, как работают такие модели и как последовательности поступают туда и выходят из них.

7.4.1. Модели кодер–декодер

На высоком уровне кодер LSTM считывает и кодирует последовательность исходного текста в вектор фиксированной длины, вектор мысли. Затем декодер выводит переведенную версию исходного предложения из кодированного вектора. Система кодер–декодер обучена максимизировать вероятность правильного перевода, учитывая исходное предложение. Таким образом, в некоторой степени эти модели, как и многие другие модели, основанные на глубоком обучении, являются статистической моделью! Разница по отношению к «традиционному» статистическому машинному переводу заключается в том, что модели нейронного машинного перевода учатся максимизировать правильность сгенерирован-

---

[1] *Ilya Sutskever, Oriol Vinyals, and Quoc V. Le*. Sequence to Sequence Learning with Neural Networks. 2014. September 10 (**https://arxiv.org/abs/1409.3215**).

[2] NIPS: Oral Session 4 – Ilya Sutskever // Microsoft Research. 2016. August 18 (**https://www.youtube.com/watch?v=-uyXE7dY5H0**).

ного перевода через нейронные сети, и делают они это многоцелевым способом. Например, нет необходимости в специальных инструментах для выравнивания слов; сети кодер–декодер нужна только огромная коллекция пар типа «исходное предложение – целевое». Ключевые особенности моделей кодер–декодер:

- их легко настроить и понять – модель интуитивно понятна;
- они могут обрабатывать входные и выходные последовательности переменной длины;
- они производят векторные представления входных последовательностей, которые можно использовать по-разному;
- их можно использовать для задач отображения seq2seq в различных областях;
- они являются многоцелевым инструментом, как только что было объяснено.

Давайте разберем схему, представленную на рис. 7.4, чтобы лучше понять, что находится в каждой части модели и как эти части работают вместе. Кодер состоит из рекуррентной нейронной сети, обычно LSTM или другой альтернативы, такой как управляемые рекуррентные блоки[1], на которых мы здесь не будем останавливаться подробно. Помните, что основное различие между сетью прямого распространения и рекуррентной нейронной сетью заключается в том, что последняя имеет рекуррентные слои, которые позволяют легко работать с неограниченными входными последовательностями при сохранении размера входного слоя фиксированным. Кодер RNN обычно глубокий, поэтому у него несколько скрытых рекуррентных слоев. Так же, как вы видели, когда мы знакомились с рекуррентными сетями в главе 3, можно добавить больше скрытых слоев, когда качество перевода низкое, даже если предоставлено большое количество обучающих данных. В общем от двух до пяти рекуррентных слоев вполне достаточно для обучающих наборов при порядке величины в десятки гигабайт. Выход сети кодера представляет собой вектор мысли, который соответствует последнему временному шагу последнего скрытого слоя сети. Например, если у кодера четыре скрытых слоя, последний временной шаг четвертого слоя будет представлять вектор мысли.

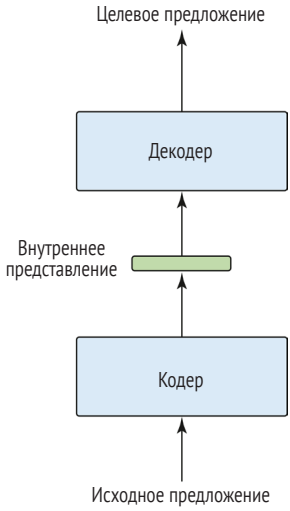


Рис. 7.4 ❖ Модель кодер–декодер

Для простоты давайте рассмотрим перевод предложения из четырех слов, написанного италоязычным пользователем, который ищет информацию о въезде в Великобританию с итальянской ID-картой. Исходное предложение может выглядеть примерно так: «carta id per gb». В кодер подается одно слово предложения на каждом временном шаге. После четырех временных шагов в кодер были введены все четыре слова во входном предложении, как показано на рис. 7.5.

---

[1] *См.* известную статью: *Kyunghyun Cho et al.* Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. 2014. 3 июня (**https://arxiv.org/abs/1406.1078**).

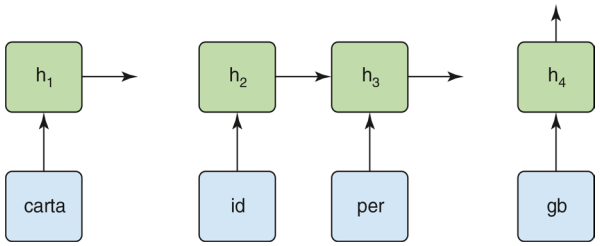


Рис. 7.5 ❖ Сеть-кодер с четырьмя скрытыми рекуррентными слоями

ПРИМЕЧАНИЕ На практике входная последовательность часто меняется на противоположную, поскольку оказывается, что таким образом нейронная сеть обычно дает лучшие результаты.

Когда вы узнали о word2vec в главе 2, то увидели, что слова часто преобразуются в векторы с унитарным кодированием, чтобы их можно было использовать в нейронной сети. Векторные представления слов были результатом работы алгоритма word2vec. Кодер делает нечто подобное, используя *слой векторного представления*. Вы преобразуете входные слова в векторы с унитарным кодированием, и входной слой сети имеет размер, равный размеру словаря слов в наборе исходных предложений. Помните, что вектор с унитарным кодированием для определенного слова, например *gb*, представляет собой вектор с 1 для индекса вектора, назначенного этому слову, и 0 во всех остальных позициях. Перед рекуррентным уровнем вектор с унитарным кодированием преобразуется в векторное представление слов слоя с меньшим размером, чем у входного слоя. Этот слой является слоем векторного представления, а его выход – векторное представление слова, подобное тому, что получено с помощью word2vec.

Если присмотреться к уровням кодера, вы увидите стек, аналогичный тому, что показан на рис. 7.6. Этот входной слой состоит из 10 нейронов, а это означает, что исходный язык содержит только 10 слов; в действительности входной слой может содержать десятки тысяч нейронов. Слой векторного представления уменьшает размер входного слова и генерирует вектор, значения которого – не просто 0 и 1, а реальные значения. Выходной вектор этого слоя затем передается в рекуррентные слои.

После обработки последнего слова во входной последовательности специальный токен (например, `<EOS>`: конец предложения) передается в сеть, чтобы сигнализировать о том, что ввод завершен и должно начаться декодирование. Это облегчает обработку входных последовательностей переменной длины, потому что декодирование не начнется, пока не будет получен токен `<EOS>`.

Декодирование зеркально отражает кодирование. Единственное отличие состоит в том, что декодер (см. рис. 7.7) получает и вектор фиксированной длины, и по одному исходному слову на каждом временном шаге.

Слой векторного представления не используется в декодере. Значения вероятности в выходном слое сети декодера используются для выборки слова из словаря на каждом временном шаге. Давайте теперь посмотрим на кодер–декодер LSTM с использованием DL4J в действии.

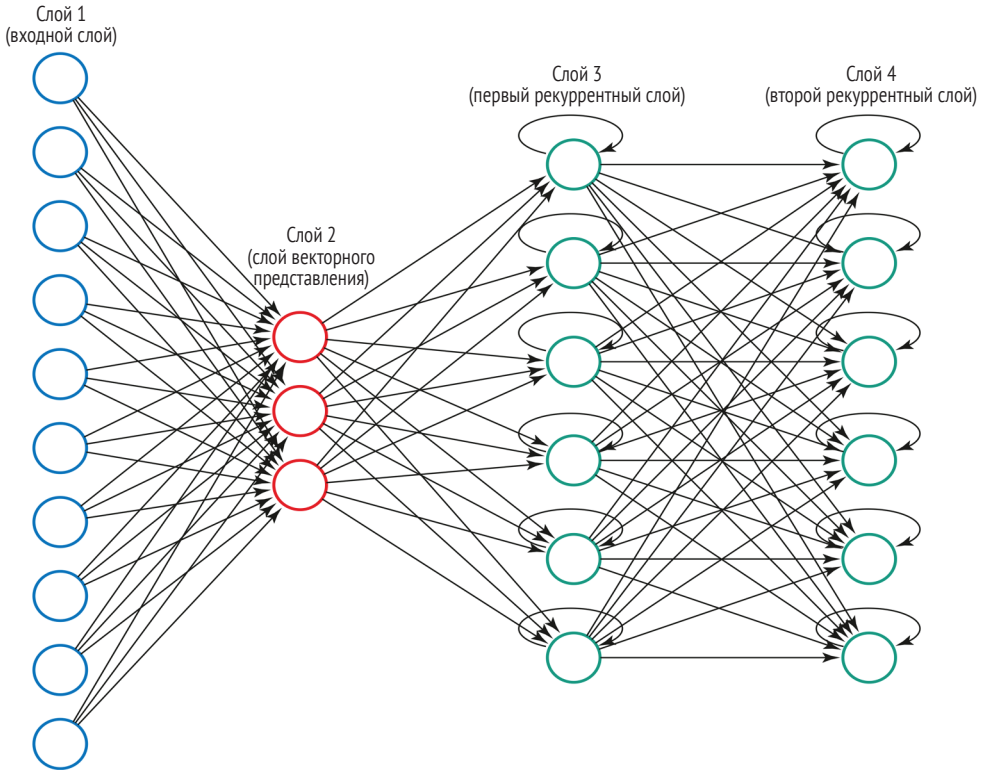


Рис. 7.6 ❖ Слои сети-кодера (до второго скрытого рекуррентного слоя) со словарем из 10 слов

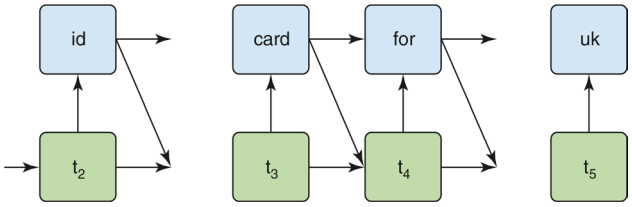


Рис. 7.7 ❖ Сеть-декодер с четырьмя скрытыми рекуррентными слоями

7.4.2. Модель «кодер–декодер» для машинного перевода в DL4J

DL4J позволяет вам объявлять архитектуру вашей нейронной сети с помощью *вычислительного графа*. Это распространенная парадигма в рамках глубокого обучения; аналогичные шаблоны используются и в других популярных инструментах глубокого обучения, таких как TensorFlow, Keras и др. С помощью вычислительного графа для нейронной сети вы можете объявить, какие слои существуют и как они связаны друг с другом.

Давайте рассмотрим слои сети кодера, определенные в предыдущем разделе. У вас есть входной слой, слой векторного представления и два рекуррентных

(LSTM) слоя (показаны так, как их представляет пользовательский интерфейс DL4J на рис. 7.8). Вычислительный граф кодера выглядит следующим образом:

```
ComputationGraphConfiguration.GraphBuilder graphBuilder =
        builder.graphBuilder()
    ...
    .addInputs("inputLine", ...)
    .setInputTypes(InputType.
        recurrent(dict.size()), ...)
    .addLayer("embeddingEncoder",
        new EmbeddingLayer.Builder()
            .nIn(dict.size())
            .nOut(EMBEDDING_WIDTH)
            .build(),
        "inputLine"
    .addLayer("encoder",
        new GravesLSTM.Builder()
            .nIn(EMBEDDING_WIDTH)
            .nOut(HIDDEN_LAYER_WIDTH)
            .activation(Activation.TANH)
            .build(),
        "embeddingEncoder")
    .addLayer("encoder2",
        new GravesLSTM.Builder()
            .nIn(HIDDEN_LAYER_WIDTH)
            .nOut(HIDDEN_LAYER_WIDTH)
            .activation(Activation.TANH)
            .build(),
        "encoder");
    ...
```

Определяет входной тип для рекуррентной сети

Создает слой векторного представления

Слой векторного представления ожидает количество входов, равное размеру словаря слов

Ширина выходного вектора

Вход слоя векторного представления

Добавляет первый слой кодера

Первый слой кодера является LSTM-слоем

Использует функцию tanh в слоях LSTM

Слой кодера берет входные данные из слоя embeddingEncoder

Добавляет второй уровень кодера (еще один слой LSTM)

Слой encoder2 берет входные данные из слоя кодера

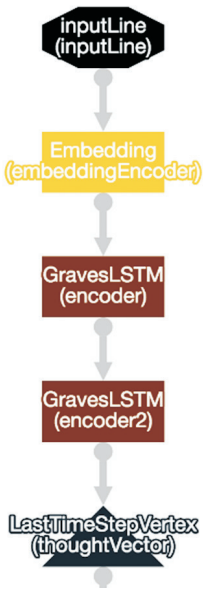


Рис. 7.8 ❖ Слои кодера

Декодер содержит два LSTM-слоя и выходной слой (*см.* рис. 7.9). Переведенные слова выбираются из выходных значений, сгенерированных функцией softmax на выходном слое:

```
...
.addLayer("decoder",
    new GravesLSTM.Builder()
        .nIn(dict.size() + HIDDEN_LAYER_WIDTH)
        .nOut(HIDDEN_LAYER_WIDTH)
        .activation(Activation.TANH)
        .build(),
    "merge")
.addLayer("decoder2",
    new GravesLSTM.Builder()
        .nIn(HIDDEN_LAYER_WIDTH)
        .nOut(HIDDEN_LAYER_WIDTH)
        .activation(Activation.TANH)
        .build(),
    "decoder")
.addLayer("output",
    new RnnOutputLayer.Builder()
        .nIn(HIDDEN_LAYER_WIDTH)
        .nOut(dict.size())
        .activation(Activation.SOFTMAX)
        .lossFunction(LossFunctions.
            LossFunction.MCXENT)
        .build(),
    "decoder2")
.setOutputs("output");
```

Рекуррентные слои декодера также основаны на LSTM

Нормальный выходной слой рекуррентной сети

Выход – это распределение вероятностей, генерируемое функцией активации softmax

Используемая функция потерь – это мультиклассовая перекрестная энтропия

Рис. 7.9 ❖ Слои декодера

В этот момент вы, наверное, можете подумать, что на этом все, но вам все еще не хватает клея, который соединяет кодер с декодером. Он состоит из:

- слоя вектора мысли, который захватывает распределенное представление исходного слова, используемого декодером для генерации правильного переведенного слова;
- бокового входа, используемого декодером для отслеживания генерируемых им слов.

Граф будет выглядеть немного сложнее, чем вы можете ожидать, потому что декодирование нейронной сети использует и вектор мысли, и выходы, которые он генерирует, на каждом временном шаге. Сеть-декодер начинает генерировать переведенные слова, как только получает специальное слово (например, *go*) на выделенном входе. На этом временном шаге декодер выбирает значение из вектора мысли, сгенерированное кодером, и это специальное слово и генерирует свое первое декодированное слово. На следующем временном шаге он использует только что сгенерированное декодированное слово в качестве нового входа вместе со значением вектора мысли, чтобы сгенерировать последующее слово – и так далее, пока он не сгенерирует специальное слово (такое как `EOS`), которое останавливает декодирование.

Таким образом, слой вектора мысли подается на последний временной шаг конечного рекуррентного (LSTM) слоя сети кодера и используется в качестве входа для декодера вместе со словом на каждом временном шаге декодирования, как показано на рис. 7.10. Полная модель выглядит, как показано на рис. 7.11.

Соединения между кодером и декодером, показанные на рис. 7.10, реализуются с помощью приведенного ниже кода:

```
.addVertex("thoughtVector", new LastTimeStepVertex(
    "inputLine"), "encoder2")

.addVertex("dup", new DuplicateToTimeSeriesVertex(
    "decoderInput"), "thoughtVector")

.addVertex("merge", new MergeVertex(), "decoderInput"
    , "dup")
```

Только последний временной шаг выхода кодера записывается в вектор мысли

Создает новый вход временного ряда для декодера, инициализированный с помощью значений из вектора мысли

Подготавливает декодер для приема объединенных входов от вектора мысли и входа со стороны декодера

Построив этот вычислительный граф, вы готовы приступить к обучению сети с помощью параллельного корпуса. Для этого вы создаете процессор `ParallelCorpusProcessor`, который обрабатывает параллельный корпус: например, в форме TMX-файла, загруженного из проекта OPUS. Этот процессор извлекает исходные и целевые предложения и создает словарь слов. Затем он будет использоваться для предоставления входных и выходных последовательностей, необходимых для обучения модели кодер–декодер:

```
File tmxFile = new File("/path/to/file.tmx");
ParallelCorpusProcessor corpusProcessor = new
    ParallelCorpusProcessor(tmxFile, "it", "en");
corpusProcessor.process();
Map<String, Double> dictionary =
    corpusProcessor.getDict();
Collection<ParallelSentence> sentences =
    corpusProcessor.getSentences();
```

TMX-файл, где содержится параллельный корпус

Парсит TMX-файл и извлекает исходные и целевые предложения на основе языковых кодов (например, «it» для источника, «en» для цели)

Обрабатывает корпус

Извлекает словарь корпуса

Получает параллельные предложения

Теперь словарь используется для настройки сети: размер словаря определяет количество входных данных (для векторов с унитарным кодированием). В данном случае словарь – это карта, ключами которой являются слова, а значение – это число, используемое для идентификации каждого слова при подаче его в слой векторного представления. Предложения и словарь необходимы для создания итератора

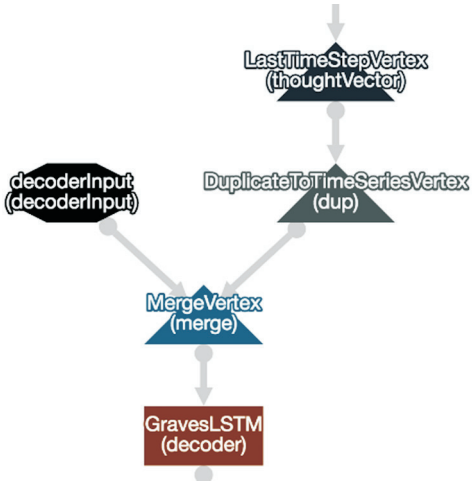


Рис. 7.10 ❖ Связи между кодером и декодером

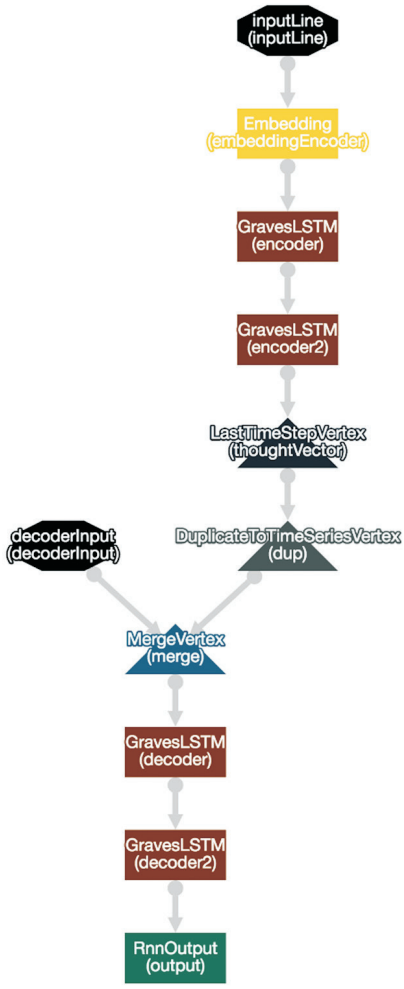


Рис. 7.11 ❖ Модель кодер–декодер с двумя LSTM-слоями на каждой стороне

для параллельных предложений. Затем `DataSetIterator` для параллельного корпуса используется для обучения сети в разные эпохи (*эпоха* обучения – это полный цикл обучения по всем имеющимся обучающим примерам из обучающего набора):

```
ComputationalGraph graph = createGraph(dictionary.
    getSize());                                          ← Создает сеть, используя вычислительный граф

ParallelCorpusIterator parallelCorpusIterator = new
    ParallelCorpusIterator(corpusProcessor);             ← Создает итератор для параллельного корпуса
for (int epoch = 0; epoch < EPOCHS; epoch++) {
  while (parallelCorpusIterator.hasNext()) {             ← Перебирает корпус
    MultiDataSet multiDataSet = parallelCorpusIterator
```

```
      .next();          ←— Извлекает пакет входных и выходных последовательностей
    graph.fit(multiDataSet);  ←— Обучает сеть по текущему пакету
  }
}
```

Теперь сеть начинает учиться генерировать англоязычные последовательности из последовательностей на итальянском. На рис. 7.12 показано снижение ошибки сети.

Рис. 7.12 ❖ Обучение сети кодер–декодер

Перевод, выполняемый сетью, состоит из прямой передачи всех слов во входной последовательности по сетям кодера и декодера. В кодере реализован API-интерфейс `TranslatorTool`, а метод `output` выполняет прямую передачу по нейронной сети. Это дает переведенную версию исходного предложения:

```
@Override
  public Collection<Translation> translate(String text) {
  double score = 0d;
  String string = Joiner.on(' ').join(output(text, score));
  Translation translation = new Translation(string, score);
  return Collections.singletonList(translation);
}
```

Метод `output` преобразует текстовую последовательность в вектор и затем передает ее по сетям кодера и декодера. Текстовый вектор подается в сеть, используя индексы слов, сгенерированные `ParallelCorpusProcessor`. Таким образом, вы преобразуете строку (`String`) в `List<Double>`, который представляет собой упорядоченный список индексов слов, соответствующих каждому токену в исходной последовательности:

```
Collection<String> tokens = corpusProcessor.tokenizeLine(text);
List<Double> rowIn = corpusProcessor.wordsToIndexes(tokens);
```

Теперь вы подготовите фактические векторы, которые будут использоваться в качестве входа для кодера (вектор `input`) и декодера (вектор `decode`), и выполните отдельные прямые передачи для кодера и декодера. Передача для кодера выглядит так:

```
net.rnnClearPreviousState();
Collections.reverse(rowIn);
Double[] array = rowIn.toArray(new Double[0]);
INDArray input = Nd4j.create(ArrayUtils.toPrimitive(array),
    new int[] {1, 1, rowIn.size()});
int size = corpusProcessor.getDict().size();
double[] decodeArr = new double[size];
decodeArr[2] = 1;
INDArray decode = Nd4j.create(decodeArr, new int[] {1, size, 1});
net.feedForward(new INDArray[] {input, decode}, false, false);
```

Передача для декодера немного сложнее, поскольку она предполагает использование вектора мысли, сгенерированного передачей для кодера, *и* векторов токенов исходной последовательности.

Таким образом, на каждом временном шаге декодер выполняет перевод, учитывая вектор мысли и вектор токена исходного предложения:

```
Collection<String> result = new LinkedList<>();
GravesLSTM decoder = (GravesLSTM) net.getLayer("decoder");
Layer output = net.getLayer("output");
GraphVertex mergeVertex = net.getVertex("merge");
INDArray thoughtVector = mergeVertex.getInputs()[1];
for (int row = 0; row < rowIn.size(); row++) {
  mergeVertex.setInputs(decode, thoughtVector);
  INDArray merged = mergeVertex.doForward(false);
  INDArray activateDec = decoder.rnnTimeStep(merged);
  INDArray out = output.activate(activateDec, false);
  double idx = sampleFrom(output);
  result.add(corpusProcessor.getRevDict().get(idx));
  double[] newDecodeArr = new double[size];
  newDecodeArr[idx] = 1;
  decode = Nd4j.create(newDecodeArr, new int[] {1, size, 1});
}
return result;
```

Наконец, все готово и можно приступить к переводу запросов с использованием сети кодер–декодер. (На практике вы выполняете этап обучения вне рабочего процесса поиска.) После завершения обучения модель сохраняется на диск, а затем загружается парсером запросов, определенным в начале этой главы:

```
ComputationGraph net ...
File networkFile = new File("/path/to/file2save");
ModelSerializer.writeModel(net, networkFile, true);
```

Парсер запросов создается с использованием сети кодер–декодер для предложений на итальянском (и детектора языка):

```
File modelFile = new File("/path/to/file2save");
ComputationGraph net = ModelSerializer.restoreComputationGraph(modelFile);
net.init();
TranslatorTool mtNetwork = new MTNetwork(modelFile);

Map<String, Collection<TranslatorTool>> mappings = new HashMap<>();
mappings.put("ita", Collections.singleton(mtNetwork));
LanguageDetector languageDetector = new LanguageDetectorME(new
    LanguageDetectorModel(new FileInputStream("/path/to/langdetect.bin")));
MTQueryParser MTQueryParser = new MTQueryParser("text",
    new StandardAnalyzer(), languageDetector, mappings);
```

Внутренняя регистрация парсера запросов расскажет вам, как он переводит входящие запросы. Предположим, пользователь из Италии хочет знать, действительно ли его удостоверение личности в Великобритании. Его запрос, написанный на итальянском языке, переводится на английский с использованием сети кодер–декодер:

```
> q: validità della carta d'identità in UK
> detected language 'ita' for query 'validità della carta d'identità in UK'
> found 1 translation
> t: identity card validity in the UK
> 'validità della carta d'identità in UK' was parsed as:
 '(text:validità text:della text:carta text:identità text:in text:UK)'
 OR
 '(text:identity text:card text:validity text:in text:the text:UK)'
```

На этом все. Многие продукционные системы машинного перевода используют такие модели или их расширения. Одним из ключевых преимуществ применения нейронного машинного перевода является то, что обычно он дает точные переводы, учитывая достаточное количество обучающих данных, но такие модели могут потребовать значительных вычислительных ресурсов при обучении. В следующем разделе мы рассмотрим еще один подход к реализации программ машинного перевода, где используются векторные представления слов и документов (word2vec, векторы абзаца и т. д.). По сравнению с моделями, подобными той, что была реализована в этом разделе, вряд ли удастся достичь того же уровня верности, однако вычислительных ресурсов потребуется гораздо меньше и, следовательно, это может стать хорошим компромиссом.

7.5. Векторные представления слов и документов для нескольких языков

В предыдущих главах использовались векторные представления слов (плотные векторы, представляющие семантику слов), в частности модель word2vec, как для генерации синонимов, чтобы обогатить текст документов, подлежащий индексации, так и для определения функции ранжирования, которая лучше отражает релевантность результатов поиска. В главе 6 вы увидели алгоритм вектора абзаца, который изучает плотные векторы последовательностей текста (целые документы или их части, такие как абзацы или предложения), и использовали его, чтобы рекомендовать похожий контент и создать еще одну (более мощную) функцию

ранжирования. Теперь вы увидите, как каждый из этих алгоритмов нейронных сетей применяется для перевода текста.

7.5.1. Монолингвальные векторные представления с использованием линейной проекции

Один из ключевых аспектов векторов слов, генерируемых моделью word2vec, заключается в том, что когда такие векторы изображаются в виде точек в векторном пространстве, слова с аналогичными значениями располагаются близко друг к другу. Вскоре после публикации статьи, которая познакомила читателей с word2vec, те же исследователи задались вопросом, что произойдет с векторными представлениями слов, если они будут получены из тех же данных, но будут переведены. Будет ли какая-либо связь между векторами слов для фрагмента текста на английском языке и того же текста, написанного на испанском? Они обнаружили, что существуют значительные геометрические сходства в отношениях между одними и теми же словами на разных языках. Например, распределение чисел и животных на английском и испанском языках аналогично, если нанести их соответствующие векторы слов на график, как видно на рис. 7.13.

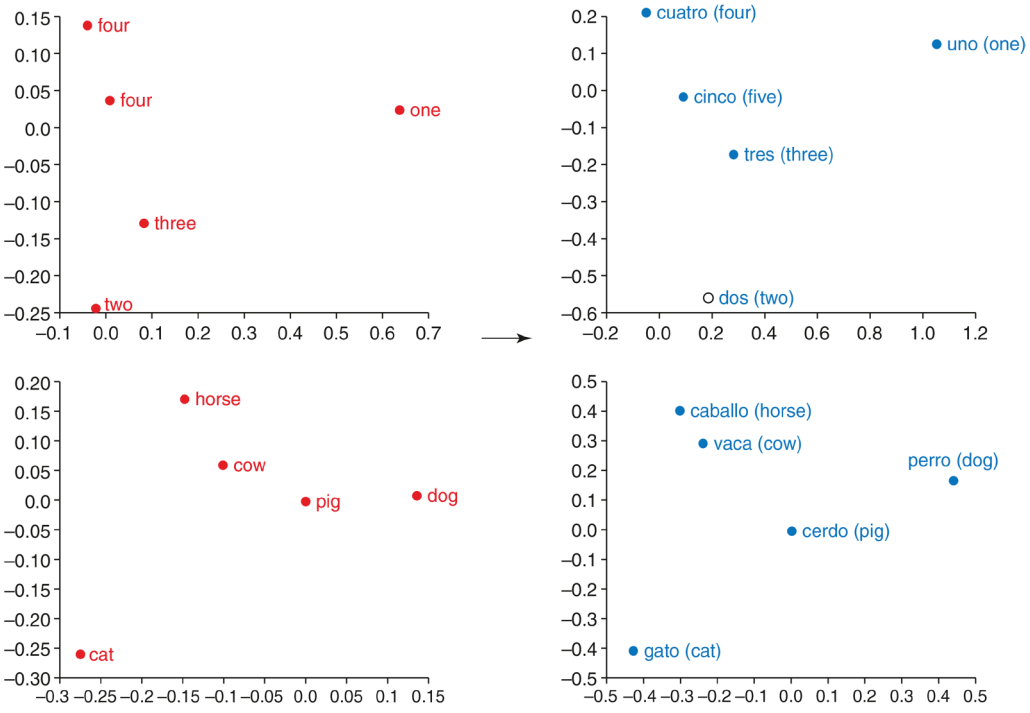


Рис. 7.13 ❖ Векторные представления английских и испанских слов из статьи Миколова и др. «Использование сходств между языками для машинного перевода»

Эти визуальные и геометрические сходства позволили предположить, что функция, которая может преобразовать вектор слова из англоязычного простран-

ства векторных представлений в вектор слова из испаноязычного пространства, была бы хорошим кандидатом для перевода слов. Такая функция называется *линейной проекцией*, потому что достаточно умножить исходный вектор (для слова на английском) на определенный *вектор перевода*, чтобы спроецировать исходное слово в целевое (на испанском). Предположим, у вас есть небольшой вектор (<0,1, 0,2>) для английского слова *cat* из модели word2vec англоязычного текста (на практике этого никогда не произойдет; реальные измерения векторных представлений слов обычно порядка сотен или тысяч). Можете изучить матрицу преобразования, которая будет аппроксимировать исходный вектор слова *cat* в соответствующем векторе <0,07, 0,22> слова *gato* в испаноязычном пространстве. Матрица преобразования умножает его веса на входной вектор и выводит спроецированный вектор.

Чтобы понять, как это выглядит на практике, давайте настроим это в DL4J, используя тот же англо-итальянский параллельный корпус, который применяется для кодера–декодера. Вы получите параллельный корпус и создадите две независимые модели word2vec, одну для исходного языка и другую для целевого.

Листинг 7.7 ❖ Создание двух независимых моделей word2vec

```
Collection<ParallelSentence> parallelSentences = new
    TMXParser(tmxFile, source, target).parse();

Collection<String> sources = new LinkedList<>();
Collection<String> targets = new LinkedList<>();
for (ParallelSentence sentence : parallelSentences) {
    sources.add(sentence.getSource());
    targets.add(sentence.getTarget());
}

int layerSize = 100;
Word2Vec sourceWord2Vec = new Word2Vec.Builder()
        .iterate(new CollectionSentenceIterator(sources))
        .tokenizerFactory(new DefaultTokenizerFactory())
        .layerSize(layerSize)
        .build();
sourceWord2Vec.fit();
Word2Vec targetWord2vec = new Word2Vec.Builder()
        .iterate(new CollectionSentenceIterator(targets))
        .tokenizerFactory(new DefaultTokenizerFactory())
        .layerSize(layerSize)
        .build();
targetWord2vec.fit();
```

Парсит файл параллельного корпуса

Создает две отдельные коллекции для исходного и целевого предложений

Обучает две модели word2vec: одну из исходных предложений и другую из целевых

Размеры векторных представлений, равные размеру скрытого слоя модели word2vec, должны быть одинаковыми для обеих моделей

Обучает две модели word2vec: одну из исходных предложений и другую из целевых

Размеры векторных представлений, равные размеру скрытого слоя модели word2vec, должны быть одинаковыми для обеих моделей

В этом случае вам также понадобится дополнительная информация о переводе слов, а не только необработанный исходный и целевой текст. Вы должны быть в состоянии сказать, какое итальянское слово является переводом каждого английского слова в параллельном корпусе. Эту информацию можно получить либо из словаря (содержащего такую информацию, как *cat* = *gato*), либо из корпуса с выравниванием по границе слова, где для каждого параллельного предложения доступна позиционная информация об исходных и целевых словах. На портале OPUS легко найти файлы словарей с переводом по одному слову на строку:

```
...
Transferring trasferimento
Transformation Trasformazione
Transient transitori
...
```

Можно выполнить парсинг словаря с помощью приведенной ниже строки кода:

```
List<String> strings = FileUtils.readLines(dictionaryFile,
    Charset.forName("utf-8"));
int dictionaryLength = strings.size() - 1;
```

К этому моменту вы изучили векторные представления слов как для англоязычных, так и для италоязычных предложений. Следующим шагом является построение матрицы перевода. Для этого вам нужно поместить векторные представления слов для английского и итальянского языков в две отдельные матрицы. Каждая матрица содержит строку для каждого слова, а каждая строка состоит из векторного представления относительно данного слова. Из этих матриц вы будете изучать матрицу проекции.

Листинг 7.8 ❖ Размещение векторных представлений из каждой модели word2vec в отдельной матрице

```
INDArray sourceVectors = Nd4j.zeros(dictionaryLength, layerSize);
INDArray targetVectors = Nd4j.zeros(dictionaryLength, layerSize);
int count = 0;
for (String line : strings) {
    String[] pair = line.split(" );
    String sourceWord = pair[0];
    String targetWord = pair[1];
    if (sourceWord2Vec.hasWord(sourceWord) &&
            targetWord2Vec.hasWord(targetWord)) {
        sourceVectors.putRow(count, sourceWord2Vec
            .getWordVectorMatrix(sourceWord));
        targetVectors.putRow(count, targetWord2Vec
            .getWordVectorMatrix(targetWord));
        count++;
    }
}
```

При наличии двух матриц матрицу проекции можно изучать, используя различные методы. Цель состоит в том, чтобы минимизировать расстояние между каждым вектором целевого слова и соответствующим вектором исходного слова, умноженным на матрицу преобразования. В этом примере используется алгоритм линейной регрессии под названием *нормальное уравнение*. Мы пропустим подробности; ключевым моментом является то, что этот подход находит комбинацию значений в матрице проекции, которая даст лучшие результаты перевода.

Листинг 7.9 ❖ Находим матрицу проекции

```
INDArray pseudoInverseSourceMatrix = InvertMatrix.pinvert(
    sourceVectors, false);
INDArray projectionMatrix = pseudoInverseSourceMatrix.mmul(
    targetVectors).transpose();
```

Инвертирует матрицу исходных векторов

Рассчитывает матрицу перевода

На этом этап обучения завершается. Все это теперь инкапсулировано в `TranslatorTool` под названием `LinearProjectionMTEmbeddings`. Шаги обучения могут выполняться либо в конструкторе, либо в специальном методе (например, `LinearProjectionMTEmbeddings # train`).

С этого момента вы можете использовать две модели word2vec в сочетании с матрицей проекции для перевода слов. Для каждого исходного слова вы проверяете, что у вас есть его векторное представление, а затем умножаете этот вектор на матрицу проекции. Такой вектор-кандидат представляет собой аппроксимацию вектора целевого слова. Наконец, вы ищете ближайшего соседа вектора-кандидата в целевом пространстве векторного представления: слово, ассоциированное с результирующим вектором, – это искомый перевод.

Листинг 7.10 ❖ Декодирование исходного слова в целевое

```
public List<Translation> decodeWord(int n, String sourceWord) {
    if (sourceWord2Vec.hasWord(sourceWord)) {
        INDArray sourceWordVector = sourceWord2Vec
            .getWordVectorMatrix(sourceWord);
        INDArray targetVector = sourceWordVector
            .mmul(projectionMatrix.transpose());
        Collection<String> strings = targetWord2Vec
            .wordsNearest(targetVector, n);
        List<Translation> translations = new ArrayList<>(strings.size());
        for (String s : strings) {
            Translation t = new Translation(s,
                targetWord2Vec.similarity(s,
                sourceWord));
            translations.add(t);
            log.info("added translation {} for {}", t, sourceWord);
        }
        return translations;
    } else {
        return Collections.emptyList();
    }
}
```

Проверяет, есть ли у исходной модели word2vec вектор исходного слова

Получает векторное представление слова

Умножает исходный вектор на матрицу проекции

Находит ближайшего соседа вектора-кандидата

Добавляет переводы к окончательному результату, включая оценку, основанную на расстоянии между исходным и целевым словами

Можно выполнить пословный перевод для более длинных текстовых последовательностей, извлекая токены из входной текстовой последовательности и применяя метод `decodeWord` к каждому исходному слову.

Листинг 7.11 ❖ Перевод текста с использованием LinearProjectionMTEmbeddings

```
public Collection<Translation> translate(String text) {
    StringBuilder stringBuilder = new StringBuilder();
    double score = 0;
    List<String> tokens = tokenizerFactory.create(
        text).getTokens();
    for (String t : tokens) {
        if (stringBuilder.length() > 0) {
            stringBuilder.append(' ');
        }
        List<Translation> translations = decodeWord(
            1, t);
```

Разбивает входной текст на токены (слова)

Переводит одно слово за раз и получает ровно один перевод

```
        Translation translation = translations.get(0);
        score += translation.getScore();      ← Накапливает оценку перевода
        stringBuilder.append(translation);    ← Накапливает переведенные слова в StringBuilder
    }
    String string = stringBuilder.toString();
    Translation translation = new Translation(string,
        score / (double) tokens.size());      ← Создает полученный перевод
    log.info("{} translated into {}", text, translation);   с переведенным текстом и оценкой
    return Collections.singletonList(translation);
}
```

Наконец, вы готовы выполнить несколько тестовых переводов.

Листинг 7.12 ❖ Тестирование LinearProjectionMTEmbeddings

```
String[] ts = new String[]{"disease", "cure",
    "current", "latest", "day", "delivery", "destroy",
    "design", "enoxacine", "other", "validity",
    "other ingredients", "absorption profile",
    "container must not be refilled"};   ← Тестирует входные слова и предложения
File tmxFile = new File("en-it_emea.tmx");   ← Файл параллельного корпуса
File dictionaryFile = new File("en-it_emea.dic");   ← Файл параллельного словаря
LinearProjectionMTEmbeddings lpe = new
    LinearProjectionMTEmbeddings(tmxFile,
    dictionaryFile, "en", "it");   ← Обучает модели и проекционную матрицу
                                     для LinearProjectionMTEmbeddings
for (String t : ts) {
  Collection<TranslatorTool.Translation> translations =
      linearProjectionMTEmbeddings.transalate(t);   ← Возвращает лучший перевод
  System.out.println(t + " -> " + translations);      для каждого входного текста
}
```

Вы можете ожидать хороших результатов, особенно когда речь идет о переводе отдельных слов. При таком подходе каждый перевод выполняется изолированно, без использования окружающих слов, поэтому есть возможности для улучшения. В приведенном ниже выводе я вручную добавил тег верности к каждому переводу (в угловых скобках), чтобы помочь читателям, не владеющим итальянским:

```
disease -> malattia <PERFECT>
cure -> curativa <AVERAGE>
current -> stanti <BAD>
day -> giorno <PERFECT>
destroy -> distruggere <PERFECT>
design -> disegno <PERFECT>
enoxacine -> tioridazina <BAD>
other -> altri <PERFECT>
validity -> affinare <BAD>
other ingredients -> altri eccipienti <PERFECT>
absorption profile -> assorbimento profilo <GOOD>
container must not be refilled -> sterile deve non essere usarla <BAD>
```

Результаты хорошие, хотя и не идеальные; вы ожидаете, что должным образом обученная модель кодер–декодер будет работать лучше, но количество требуемых временных и вычислительных ресурсов, как правило, намного ниже при

использовании линейно проецируемых векторных представлений, что люди, работающие с системами с ограниченными ресурсами, могут захотеть воспользоваться компромиссом. Кроме того, модели word2vec можно повторно использовать в других контекстах. Например, можно использовать эти проецируемые векторные представления для машинного перевода, чтобы сделать поиск более эффективным, а также можно использовать модели word2vec при ранжировании или расширении синонимов. Вы можете выбрать, какую модель word2vec применять во время поиска, используя детектор языка, подобный тому, который вы применяли для расширения запроса.

Резюме

- Машинный перевод может быть полезен в контексте поиска для улучшения взаимодействия с пользователями, которые говорят на разных языках.
- Статистические модели могут обеспечить хорошую верность перевода, но количество парных настроек, необходимых для каждого языка, нетривиально.
- Нейронные модели машинного перевода предоставляют способы научиться переводить последовательности текста на разные языки менее четким, но более мощным способом.

Глава 8

Поиск изображений на основе контента

О чем идет речь в этой главе:
- поиск изображений на базе контента;
- работа со сверточными нейронными сетями;
- использование запроса по образцу для поиска похожих изображений.

Традиционно большинство пользователей использует поисковые системы, создавая текстовые запросы и потребляя (читая) текстовые результаты. По этой причине большая часть данной книги посвящена тому, чтобы показать вам, как нейронные сети могут помочь пользователям осуществлять поиск по текстовым документам. Вы увидели, как:
- использовать word2vec для генерации синонимов из данных, введенных в поисковую систему, что облегчает пользователям поиск документов, которые они могут пропустить в противном случае;
- расширять поисковые запросы изнутри через рекуррентные нейронные сети, предоставляя поисковой системе возможность выражать запрос несколькими способами, не прося пользователя написать их все;
- ранжировать результаты текстового поиска, используя векторные представления слов и документов, таким образом предоставляя более релевантные результаты поиска конечным пользователям;
- переводить текстовые запросы с помощью модели seq2seq, чтобы улучшить работу поисковой системы с текстом, написанным на нескольких языках, и лучше обслуживать пользователей, говорящих на разных языках.

Но пользователи все чаще ожидают, что поисковые системы будут «умнее» и смогут обрабатывать больше, нежели просто текстовые запросы. Пользователи хотят, чтобы поисковые системы выполняли поиск в сети с помощью голоса, как при использовании встроенного в смартфон микрофона, и возвращали не только текстовые документы, но и соответствующие изображения, видео и другие форматы. В дополнение к веб-поиску для других типов поисковых систем становится нормой индексировать изображения и видео, а также текст. Например, сайт газеты состоит не только из текстовых статей: на домашней странице любой газеты, помимо текста, вы найдете мультимедийный контент. Поэтому поисковая система этих сайтов должна индексировать изображения и видео, а также текст.

В течение некоторого времени базы данных индексируют изображения, используя *метаданные*: записанную информацию об изображении, такую как его заголовок или описание содержимого, которое прикреплено к изображению. Традиционные методы поиска информации, а также новые подходы, описанные в этой книге, используют теги метаданных, чтобы помочь пользователям находить изображения, которые они ищут. Но создание и ввод описаний и тегов вручную для каждого изображения, которое необходимо проиндексировать, утомительно, отнимает много времени и подвержено субъективным ошибкам – ведь кушетка для одного индексатора может быть софой для другого. Было бы неплохо, если бы вы могли индексировать изображения и делать их доступными для поиска, как они есть, без какого-либо ручного вмешательства?

В этой главе мы рассмотрим, как это сделать: оснастить поисковую систему поиском изображений, который позволяет пользователям осуществлять поиск по изображениям на основе их содержимого, а не на основе текстовых описаний их содержимого. Для создания такого вида поиска изображений мы будем использовать сверточные нейронные сети, которые представляют собой особый тип глубокой нейронной сети.

Поисковая система изображений работает путем индексации признаков изображения. Когда мы говорим о машинном обучении, *признак* представляет собой семантически релевантные данные, которые мы хотим получить, чтобы решить конкретную задачу. Говоря более конкретно, при работе с изображениями признак изображения может быть представлен конкретными точками или областями изображения (например, высококонтрастные области, формы, края и т. д.). Я начну с того, что коснусь традиционных способов извлечения важной семантики из изображений, потому что мы можем использовать эти методы в качестве руководства для решения проблем извлечения признаков из изображений. Это ключевой шаг, поскольку извлеченные признаки затем можно использовать для сравнения изображений, создания запросов и ответов на них, а также для выполнения других задач, которые должна выполнять поисковая система.

Затем я покажу вам другой и более подходящий способ извлечения признаков изображения с помощью глубоких нейронных сетей, который требует меньше ручной работы и не требует ручного извлечения. Наконец, мы рассмотрим, как включить извлеченные признаки в поисковую систему, а также учесть производительность, основанную на времени и пространстве, что необходимо для управления данным типом поиска изображений.

> **ПРИМЕЧАНИЕ** В этой главе для простоты обсуждаются изображения, а не видео. Видео, по сути, представляет собой последовательность изображений с прикрепленными звуковыми битами, поэтому вы, безусловно, можете применять подходы, описанные в этой главе, к поиску видео, а также к поиску изображений.

8.1. Содержимое изображения и поиск

Еще в главе 1 я кратко познакомил вас с одним из самых многообещающих аспектов глубокого обучения: обучением представлениям. *Обучение представлениям* – это задача получения входных данных (например, изображений) и автоматического извлечения признаков, что облегчает программе решение конкретной проблемы (например, распознавать, какие объекты изображены на рисунке, на-

сколько похожи два изображения и т. д.). Хорошее представление определенного изображения должно быть выразительным, то есть в идеале оно должно предоставлять информацию о различных аспектах изображения (содержащиеся объекты, свет, экспозиция и т. д.), а также упрощать сравнение отдельных аспектов (например, вам нужно определить, есть ли на двух изображениях бабочка, сравнивая такие выученные представления). На высоком уровне изучение представления изображения с использованием глубокой нейронной сети обычно следует простой схеме, показанной на рис. 8.1, где пиксели преобразуются в границы, границы – в формы, а формы – в объекты.

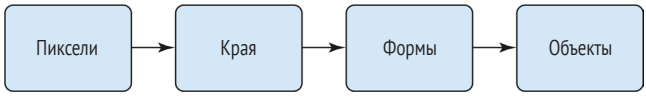


Рис. 8.1 ❖ Постепенное изучение абстракций изображений

Давайте рассмотрим изображение, хранящееся на жестком диске компьютера, и посмотрим, что такое двоичное представление говорит нам о его содержимом. Можете ли вы быстро открыть файл изображения как текстовый файл и сразу же распознать, что показано на изображении? Ответ – нет. Если вы посмотрите на необработанное содержимое (например, с помощью команды Linux `cat`) файла изображения, на котором изображена, скажем, бабочка, то не увидите ничего, что могло бы рассказать вам о его содержимом:

```
$ cat butterfly.jpg
????m,ExifII*
            ???(2?;??i?h%??*?1HH2018:07:01
     08:37:38&??6??>"?'??0?2???0230?F?Z??
n?v?
~? ?
??|?
?)2?*4?5.*5?9??59?0100??p????)??)??)?????0?1?
```

Файл изображения показывает бабочку, если открыть его с помощью соответствующей программы: вы можете использовать инструменты для «просмотра» изображений, но компьютер не может автоматически распознать, что содержит изображение, или сказать вам, что это изображение пожилой дамы, дикое животное на природе или что бы то ни было еще. Представление двоичного содержимого изображения не подходит для того, чтобы сказать вам, что на нем изображена бабочка.

Однако глубокое обучение *может* помочь вам изучить представление, которое при правильном использовании может рассказать вам больше о содержимом изображения. В этом случае глубокая нейронная сеть может сказать вам, что на изображении изображена бабочка. Алгоритм глубокого обучения обычно делает это, изучая все больше и больше информации на каждом глубоком слое. Например, на первых слоях он изучает границы, на последующих слоях – формы, а на последних слоях – объекты (например, бабочку или ее часть), чтобы иметь возможность сказать, что содержит изображение. Кроме того, эта информация из всех слоев часто кодируется в плотном векторном представлении каждого изображения. Да-

лее в этой главе мы рассмотрим краткий обзор данного процесса, и вы, наконец, познакомитесь с глубокими нейронными сетями, которые могут изучать представления изображений.

Если вы когда-либо пытались создать открытку, используя изображение (конечно же, бесплатное), доступное в интернете, то у вас могли возникнуть проблемы при поиске изображений, относящихся к определенной теме. Допустим, например, что вы купили масштабную модель автомобиля для своего племянника или племянницы и хотите напечатать открытку автомобиля, которую можно использовать в качестве карточки, на которой можно написать что-либо и отправить ее ему или ей. Итак, вы переходите в поисковую систему изображений – возможно, Google Images или Adobe Stock – и вводите что-то вроде «спортивная машина» в поле запроса. В этом процессе важно понимать, что пользователи ищут изображения, которые содержат определенный объект или определенный признак. Например, вам может понадобиться «красный спортивный автомобиль» или «винтажный спортивный автомобиль». Поисковые системы изображений часто используют механизм, называемый *запросом по образцу*, когда вы загружаете или берете изображение, которое будет использоваться в качестве входного запроса. Затем поисковая система возвращает изображения, аналогичные тому, которое является входным.

Давайте на минутку остановим наш текущий запрос и посмотрим, как работает запрос по образцу. Мы начнем с размышлений о том, как создаются изображения: как цифровая камера или графическое приложение создает и сохраняет изображение. Сделайте снимок с помощью камеры и где-нибудь сохраните файл, содержащий двоичные данные (0 и 1). Можно рассматривать это изображение, хранящееся на компьютере, как сетку определенной ширины и высоты, где каждая ячейка в сетке называется *пикселем*, а у каждого пикселя есть определенный цвет. Цветной пиксель может быть представлен по-разному, и для описания цветов используется несколько цветовых моделей. Для простоты мы выберем наиболее распространенную схему, *RGB* (красный, зеленый, синий), в которой каждый цвет состоит из смеси красного, зеленого и синего. Каждый из этих трех цветов имеет диапазон значений от 0 до 255, указывающий на количество красного, зеленого и синего, которые будут использоваться в каждой комбинации (не только *один* красный). Каждое такое значение затем может быть представлено 8 двоичными значениями (2 ^ 8 = 256), и, таким образом, оно содержит все возможные диапазоны от 0 до 255. Итак, RGB-изображение имеет сетку, пиксели которой состоят из двоичных значений, которые обозначают их цвета[1]: *красный* цвет – R: 255, G: 0, B: 0; *синий* – это R: 0, G: 0, B: 255 и т. д.

Учитывая это, давайте разморозим наш запрос. Как сопоставить запрос «спортивный автомобиль», если изображения представляют собой просто серию битов? В следующих разделах вы увидите несколько разных способов сопоставления запросов и изображений, а также изучите методы обнаружения конкретного спортивного автомобиля, который вам нужен.

---

[1] Хотя на практике изображения могут иметь множество различных форматов и цветовых схем, основная проблема заключается в том, что изображения обычно хранятся в виде простых двоичных файлов, необязательно с метаданными, которые обычно ничего не говорят об их содержимом.

8.2. Взгляд назад: поиск изображений на базе текста

Пользователи, естественно, склонны думать об изображениях с точки зрения того, какие объекты они содержат (например, спортивные автомобили), а не их RGB-значений. Но формы и цвета лучше подходят для определения потребности в информации относительно того, что они ищут, будь то красный спортивный автомобиль, спортивный автомобиль «Формулы-1» или какой-то другой тип автомобиля.

Менее умный, но распространенный подход к решению проблемы сопоставления текстовых запросов с двоичными изображениями заключается в добавлении метаданных к изображениям во время индексации. Вы индексируете изображения, но у каждого есть соответствующий текстовый заголовок или описание. Это позволяет выполнять обычный поиск с помощью текстового запроса; поиск вернет изображения, у которых есть прикрепленный к ним текст метаданных, соответствующий запросу. Концептуально это не сильно отличается от обычного полнотекстового поиска, за исключением того, что результаты поиска представляют собой изображения, а не заголовки документов или выдержки из них.

Используя запрос «спортивный автомобиль», предположим, что есть четыре изображения, которые могут соответствовать этому запросу. Во время индексирования вы можете использовать как данные изображения, так и небольшую подпись, описывающую каждое изображение; *см.* рис. 8.2. Данные изображения используются для возврата фактического содержимого изображения конечному пользователю (в списке результатов поиска), а текстовое описание изображения индексируется, чтобы соответствовать запросам и изображениям (как вы увидите в следующем разделе, на рис. 8.3).

Рис. 8.2 ❖ Обозначенные вручную изображения спортивных автомобилей

Если вы ищете «спортивный автомобиль», поисковая система вернет все изображения, показанные на рис. 8.2. Если вы ищете «черный спортивный ав-

томобиль», только два из них появятся в списке результатов (напомним, что использование двойных кавычек в запросе приводит к совпадению по всей фразе «черный спортивный автомобиль», а не отдельным словам «черный», «спортивный» и «автомобиль»).

Рис. 8.3 ❖ Несколько изображений, подпадающих под описание «высокий человек»

Данный подход можно реализовать в Lucene простым способом. Вы сохраняете двоичное изображение как есть, но индексируете введенное вручную описание изображения (описание не будет возвращено с результатами поиска):

```
byte[] bytes = ...                                  ← Получает содержимое изображения в виде байта []
String description = ...                            ← Записывает описание изображения в виде строки
Document doc = new Document();                      ← Добавляет двоичное содержимое
                                                      изображения в качестве поля stored
doc.add(new StoredField("binary", bytes));

doc.add(new TextField("description", description, Field.Store.NO));   ← Добавляет описание изображения
                                                                        в виде поля text
writer.addDocument(doc);                            ← Индексирует документ изображения

writer.commit();                                    ← Фиксирует изменения индекса
```

Во время поиска можно использовать простой текстовый запрос:

```
DirectoryReader reader = DirectoryReader
    .open(writer);                                  ← Открывает IndexReader для индекса, содержащего изображения

IndexSearcher searcher = new
    IndexSearcher(reader);                          ← Создает IndexSearcher для выполнения запроса

TopDocs topDocs = searcher.search(new PhraseQuery(  ← Выполняет запрос «черный спортивный
    "description", "black", "sports", "car"), 3);    автомобиль» в поле заголовка
for (ScoreDoc sd : topDocs.scoreDocs) {
    Document document = reader.document(sd.doc);    ← Выбирает каждый совпадающий документ

    IndexableField binary = document.getField(
        "binary");                                  ← Извлекает поле «binary»

    BytesRef imageBinary = binary.binaryValue();    ← Извлекает фактическое изображение
    ...                                               в виде двоичного файла и что-то с ним
}                                                     делает
```

Этот подход может работать для небольшого количества изображений. Но очень часто речь идет о данных размером порядка миллионов или миллиардов документов. Даже у небольшого интернет-магазина, который делает открытки, вероятно, будут сотни или тысячи изображений. Во многих случаях невозможно попросить людей взять на себя (не очень приятную) задачу просмотра каждого изображения и придумать хорошее описание. А иногда такой текст не подходит для всех случаев поиска. (В продукционных системах нередко возникают проблемы типа «Почему запрос «черный спортивный автомобиль» не возвращает черный спортивный флуоресцентный автомобиль? Пожалуйста, измените описание, чтобы он мог соответствовать такому запросу».) Поводя итог, скажем, что этот подход не масштабируется, и он так же хорош, как и качество описаний: плохие описания приводят к нерелевантным результатам поиска.

8.3. Понять изображения

Как я уже говорил, изображение можно описать различными способами, и наиболее распространенным является указание людей, предметов, животных и других узнаваемых объектов, которые в нем содержатся: например, «Это изображение человека». Кроме того, можно упомянуть описательные подробности, такие как «На этом изображении показан высокий мужчина». Однако, как видно по рис. 8.3, подобные краткие описания могут быть двусмысленными. Неоднозначность проистекает из того простого факта, что один объект или сущность можно описать множеством разных способов.

Все три изображения на этом рисунке, безусловно, соответствуют описанию «высокий мужчина», которое может использоваться в качестве текстового запроса. Изображение в середине, однако, отличается от других. Да, это изображение высокого мужчины, но это также изображение игрока баскетбольной команды Houston Rockets. Поэтому другие фразы, в том числе «баскетболист», «игрок houston rockets» и «баскетболист в футболке с номером 35», также описывают это изображение. Человек, которому поручено писать короткие метатеги, не может учитывать все возможные способы описания изображения.

В том же духе описание типа «баскетболист в футболке с номером 35» идеально подошло бы не только к изображению в центре на рис. 8.3, но и к изображениям на рис. 8.4, где показаны совершенно разные игроки и команды. В этом случае пользователь может искать один вид изображения и получать совершенно иной, даже если оба изображения имеют один и тот же описательный метатег и будут отображаться в результатах поиска.

Такие простые примеры учат вас, что текст чрезвычайно подвержен несоответствиям, потому что одну сущность (человек, животное, объект и т. д.) можно описать множеством разных способов. Из-за этого качество результатов поиска зависит от того:

- как пользователь определяет запросы;
- как написаны документы.

Вы уже видели такие проблемы в контексте поиска – это одна из причин, почему мы используем синонимы, расширение запросов и т. д. Поисковая система должна быть достаточно умной, чтобы иметь возможность улучшать запросы и индексированные документы.

Рис. 8.4 ❖ Изображения, подпадающие под описание «баскетболист в футболке с номером 35»

Напротив, изображения, визуально говоря, обычно меньше подвержены такой двусмысленности. Давайте возьмем первое изображение, описанное как «высокий мужчина», и представим, что вы нашли изображения, которые визуально похожи на него, как на рис. 8.5.

Рис. 8.5 ❖ Визуально похожие изображения

Входное изображение позволяет лучше определить, что на нем, независимо от различных способов его текстового описания. В то же время легко сказать, *не* является ли изображение похожим на входное: например, изображение баскетболиста на рис. 8.3 явно отличается от изображений на рис. 8.5 с точки зрения цвета и типа одежды, которую носит мужчина.

Использование образцов изображений вместо текста в качестве входных запросов (также называемых *выполнением запросов по образцу*) очень распространено на платформах поиска изображений, где системы пытаются извлечь семантическую информацию из изображений для точного поиска, вместо того чтобы текстовые метаданные описывали каждое изображение. Пользователи выражают свои намерения с помощью визуального запроса. Как и в случае с текстовыми запросами, качество запроса влияет на релевантность результатов. На секунду поразмышляем с точки зрения текста. Это может помочь: запрос «красная машина» может возвращать результаты в диапазоне от игрушечного автомобиля до гоночной машины «Формулы-1», если он красный. Если запрос – «красный спортивный автомобиль» или «красный гоночный автомобиль “Формулы-1”», то диапазон возможных релевантных результатов будет менее широким и менее рас-

плывчатым. То же самое относится к визуальным запросам и результатам поиска: чем точнее запрос (визуальное описание необходимой информации), тем лучше результаты поиска. Что касается изображений, то разница состоит не в способности пользователя написать «хороший» запрос. Напротив, более значимым является алгоритм, отвечающий за извлечение информации для индексации и поиск изображений.

Например, захват объектов и их признаков (цвета, света, формы и т. д.) в изображении является одной из проблем в этой области.

Теперь у вас есть все, что нужно, чтобы приступить к рассмотрению алгоритмов для извлечения информации из изображений для представления их таким образом, чтобы можно было выполнять запросы, которые возвращают значимые результаты.

8.3.1. Представления изображений

На данный момент самая большая проблема заключается в том, как описать изображения так, чтобы можно было найти похожие. В этом примере вам нужно создать открытку для подарка. Было бы здорово, если бы вы смогли сфотографировать подарок с помощью камеры и использовать его в качестве запроса к поисковой системе изображений. Таким образом, на открытке будет красивая картинка, которая как-то подсказывает, что находится внутри подарочной коробки, когда вы дарите ее получателю.

Хотя изображения состоят из пикселей, невозможно выполнить сравнение простых пикселей. Пиксельные значения сами по себе не дают достаточной информации о том, что находится на изображении. Одна из проблем состоит в том, что пиксель обозначает только очень маленькую часть изображения и не дает информации о его контексте. Красный пиксель может быть частью красного яблока или красного автомобиля: невозможно определить, к чему он относится, просто взглянув на пиксели. Даже если пиксели сами по себе дают полезную глобальную информацию об изображении, большое высококачественное изображение в наши дни может содержать миллионы пикселей, поэтому если вы будете выполнять сравнение пиксель за пикселем, это не будет эффективным в вычислительном отношении. Кроме того, даже два снимка одного и того же объекта, снятых одной и той же камерой в одинаковых условиях (свет, экспозиция и т. д.), но сделанных с двух несколько разных углов, вероятно, будут создавать очень разные двоичные изображения попиксельно.

В данном случае вам нужно сфотографировать свой подарок – спортивную модель красного цвета, пока вы находитесь дома, не заботясь об условиях освещения и точном угле съемки. Например, вы можете сделать снимок, подобно тому, что показан на рис. 8.6. И вы хотите, чтобы поисковик вернул красивое фото с реальным красным спортивным автомобилем, желательно той же модели, что и на рис. 8.7.

Для преодоления проблемы пикселей, предоставляющих плохую информацию, наиболее широко используемый метод создания изображений с возможностью поиска состоит в том, чтобы извлекать из них *визуальные признаки* и индексировать их вместо «просто» пикселей. Эти визуальные признаки обещают предоставить информацию, которую можно использовать для поиска содержимого изображения. Признаки обычно представлены наборами чисел или векторов.

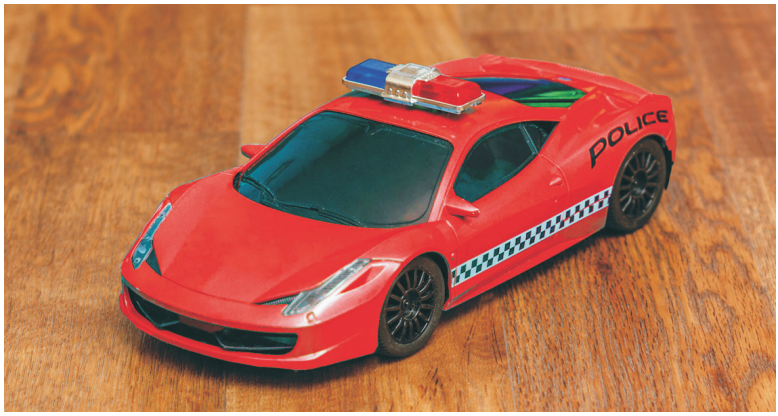


Рис. 8.6 ❖ Красный игрушечный спортивный автомобиль, который вы хотите подарить

Рис. 8.7 ❖ Фотография красного спортивного автомобиля, подобная той, которую вы хотите найти в поисковой системе, на основе фото игрушки

ПРИМЕЧАНИЕ В следующем разделе вы увидите, что это означает, когда мы рассмотрим несколько методов извлечения признаков. Понимание того, как извлечение признаков работает с методами, основанными на ненейронных сетях, полезно, чтобы установить основу для разновидности семантики, которую они могут передать, и того, насколько они различны (и менее читабельны) по отношению к признакам, извлеченным методами глубокого обучения. Как вы узнаете позже в этой главе, объем работ, затрачиваемых на методы глубокого обучения, намного меньше, чем требуется для проектирования точного алгоритма извлечения признаков. Также важным является тот факт, что на момент написания этих строк извлечение признаков на базе глубокого обучения превосходило любой алгоритм извлечения, созданный вручную.

Поисковая система должна уметь работать с такими признаками, чтобы находить похожие изображения, когда речь идет о запросе по образцу. Она извлечет признаки из изображений во время индексации и из изображения для примера запроса во время запроса. Поэтому извлечение признаков важно для понимания того, что показано на изображении; но еще один важный аспект заключается в том, как эффективно сравнить признаки разных изображений. Методы индексирования признаков будут влиять на объем дискового пространства, необходимого для хранения таких инвертированных индексов; быстрые алгоритмы поиска признаков необходимы для эффективного получения изображений во время поиска.

Визуальные признаки могут быть разных типов:

- они могут относиться к *глобальным признакам*, таким как цвета, используемые в изображении, идентифицированные текстуры либо глобальные или средние значения RGB и других цветовых моделей (CMYK, HSV и т. д.);
- они могут относиться к *локальным признакам* (извлеченные из частей изображения), таким как границы, углы или другие интересные ключевые точки в ячейках изображения (например, в таких методах, как масштабно-инвариантная трансформация признаков, ускоренные робастные признаки, разность гауссианов и т. д., которые будут обсуждаться позже в этой главе);
- их можно изучать от начала до конца как семантические абстракции, близкие к процессу человеческого познания, благодаря использованию глубоких нейронных сетей.

Первые два типа часто упоминаются как *вручную построенные* признаки, потому что соответствующие алгоритмы были разработаны и настроены для целей, основанных на эвристике. Многие модели глубокого обучения для представления изображений снабжают сетевые слои пикселями изображений (входные данные) и учатся классифицировать изображения (выходные классы сети); во время обучения нейронная сеть *изучает* признаки автоматически – это третий тип признаков.

Давайте теперь рассмотрим методы извлечения локальных и глобальных вручную построенных признаков. Затем мы сосредоточимся на изучении признаков на базе глубокого обучения для изображений.

8.3.2. Извлечение признаков

Многие камеры позволяют просматривать снимок, как только он сделан. Некоторые также предоставляют информацию о количестве цветов, содержащихся в изображении для каждого из трех RGB-каналов (красный, зеленый, синий). Давайте возьмем в качестве примера изображение бабочки, показанное на рис. 8.8. Камера, используемая для съемки, отображает цветовую гистограмму, как показано на рис. 8.9.

Цветовая гистограмма представляет собой распределение возможных значений трех цветовых каналов (например, от 0 до 255) среди пикселей. Например, если определенный пиксель имеет значение красного канала 4, а другой пиксель имеет такое же значение, цветовая гистограмма красного канала этого изображения будет иметь размер 2 для значения 4 (два пикселя имеют значение красного канала 4). Этот процесс, применяемый ко всем каналам и пикселям в определенном изображении, создает три красных, зеленых и синих графика, как показано на рис. 8.9. Цветовая гистограмма является примером очень простого, интуитивно понятного глобального признака, который можно использовать для описания изображения. Далее мы рассмотрим экстракторы глобальных и локальных признаков.

Рис. 8.8 ❖ Фотография бабочки

Рис. 8.9 ❖ Цветовая гистограмма фотографии бабочки

Глобальные признаки

Вместо того чтобы индексировать изображения вручную, помечая их заголовками или описаниями, можно индексировать двоичные изображения, сопровождаемые их извлеченными признаками, как показано на рис. 8.10. Для этого можно использовать библиотеку с открытым исходным кодом Lucene Image Retrieval (LIRE, лицензированную по лицензии GNU GPL 2) для извлечения цветовой гистограммы из изображения. LIRE предоставляет множество полезных инструментов

для работы с изображениями, которые удобны для Lucene. (На момент написания этих строк она еще не поддерживает методы на основе глубокого обучения для извлечения признаков изображения.) Вот пример:

```
File file = new File(imgPath);                                    // Файл изображения
SimpleColorHistogram simpleColorHistogram = new SimpleColorHistogram();  // Создает объект цветовой гистограммы

BufferedImage bufferedImage = ImageIO.read(file);                 // Читает изображение из файла

simpleColorHistogram.extract(bufferedImage);                      // Извлекает цветную гистограмму из изображения

double[] features = simpleColorHistogram.getFeatureVector();      // Извлекает векторы признаков цветовой
                                                                  // гистограммы в виде двойного массива
```

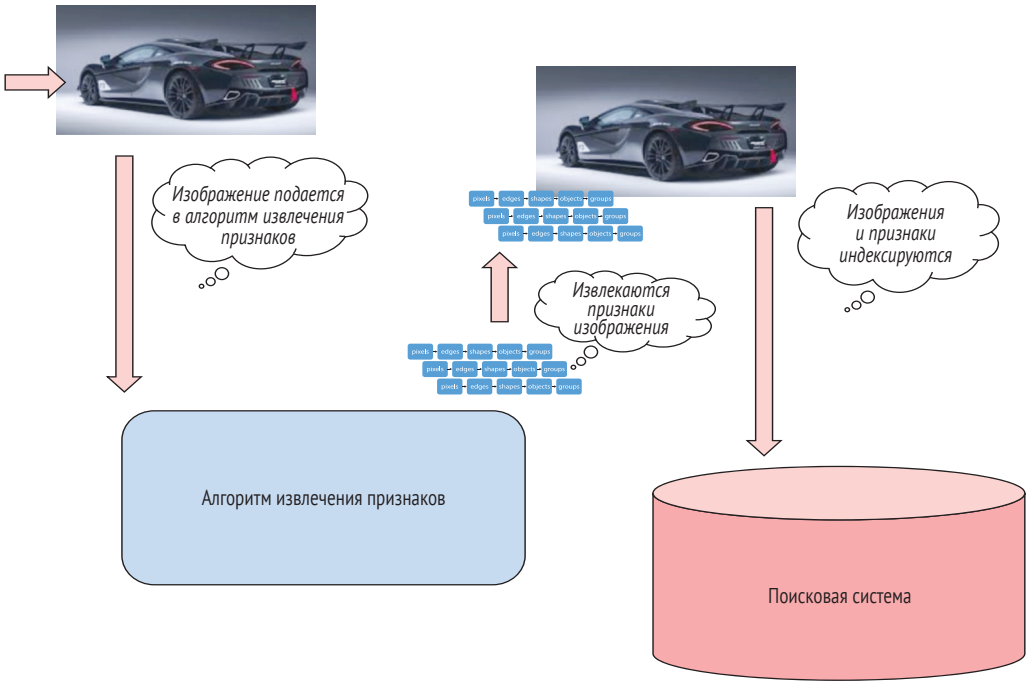


Рис. 8.10 ❖ Индексирование изображений с их признаками

Такое глобальное представление изображений имеет преимущество в том, что оно интерпретируется человеком и, как правило, эффективно с точки зрения производительности. Но если вы на мгновение подумаете о том, что представление изображения с использованием цветовой гистограммы связано с распределением цветов по изображению (независимо от положения), нетрудно понять, что два разных изображения с одним и тем же объектом (например, бабочка) могут иметь очень разные цветовые распределения. Рассмотрим изображение бабочки, показанное ранее, на рис. 8.8, и еще одно изображение бабочки, показанное на рис. 8.11.

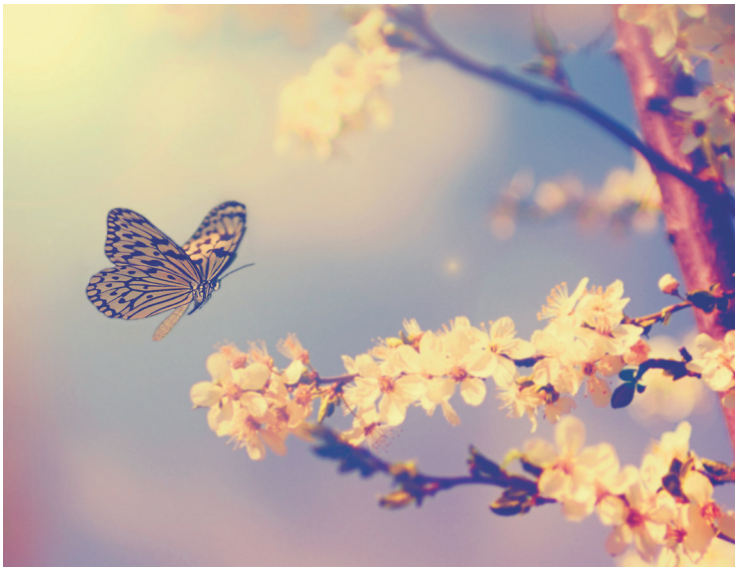


Рис. 8.11 ❖ Еще одна фотография бабочки

Хотя бабочка является основным объектом на обоих изображениях, они имеют разные цветовые схемы: как видно в электронных версиях этой книги, на рис. 8.8 основные цвета – желтый и зеленый; тогда как на рис. 8.11 основными цветами являются красный, синий и желтый. Сравнения изображений на основе гистограмм в основном основаны на цветовых распределениях. Гистограммы изображений сильно отличаются (см. рис. 8.12), поэтому поисковая система не будет считать эти изображения похожими. Помните, что в этот момент вы еще не выполняете поиск – вы анализируете гистограмму и пытаетесь понять, какую информацию она может вам дать.

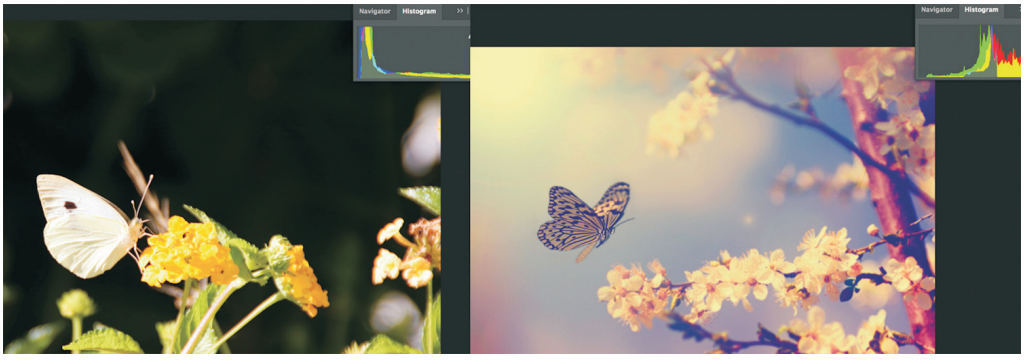


Рис. 8.12 ❖ Сравнение гистограмм двух фотографий бабочек

Схема цветовой гистограммы – это лишь один из многих возможных способов извлечения глобальных признаков, но в целом они испытывают проблему, заключающуюся в том, что детали изображения трудно захватить. Например, первое

изображение бабочки содержит не просто бабочку: там также есть цветы и листья. Такие сущности не фиксируются цветовой гистограммой; грубо говоря, подобные гистограммы говорят вам: «Есть определенное количество светло-зеленого, количество желтого, немного белого, немного черного и т. д.». Одна из ситуаций, когда глобальные признаки могут хорошо работать, – это обнаружение дублированного изображения, когда ищется изображение, очень похожее (если не точно такое же) на то, что под рукой.

Одна деталь, которая очень поможет, – это выделение фоновых областей фотографии из центрального изображения. Нам бы хотелось, чтобы представления двух изображений бабочек каким-то образом понимали, что области, содержащие бабочку, более важны, чем фоновые участки.

Локальные признаки

В отличие от глобальных признаков, локальные признаки могут более точно фиксировать детали частей изображения. Поэтому если вы хотите, чтобы программа обнаруживала потенциально интересные объекты (например, бабочку) на изображении, распространенный подход – это начать с разделения этого изображения на более мелкие ячейки, а затем искать в этих ячейках соответствующие формы или объекты. Давайте посмотрим, как это работает, на рис. 8.13, используя ту же фотографию бабочки, но теперь разделенную на более мелкие ячейки.

Рис. 8.13 ❖ Разбиение изображения на более мелкие ячейки

После того как изображение разбито на более мелкие части (например, квадраты), задача извлечения локальных признаков будет состоять из двух этапов:

1) найти интересные точки (а не объекты);
2) кодировать интересные точки относительно локальной области в дескриптор, который можно использовать позже для сопоставления интересных областей.

Но что означает слово *интересный* в данном контексте? Вы ищете точки, которые ограничивают или центрируют области изображения, содержащие объекты. Конечная цель по-прежнему состоит в том, чтобы найти объекты и представить их с помощью сопоставимых признаков. Учитывая два изображения, на которых есть бабочки, вам нужны признаки, которые несут эту информацию в обоих изображениях. Каждое изображение обычно представляется в виде вектора признаков – ряда признаков, поэтому если вы вычисляете расстояние (например, косинусное расстояние) между векторами признаков изображения, изображения, содержащие одинаковые или похожие объекты, должны быть близко (иметь маленькое расстояние).

Типичные виды локальных признаков включают в себя понятные человеку визуальные признаки, такие как границы и углы. Но на практике используются такие методы извлечения локальных признаков, как масштабно-инвариантная трансформация признаков (SIFT) и ускоренные робустные признаки (SURF).

Масштабно-инвариантная трансформация признаков

Поиск ребер – относительно простая задача, которую можно решить с помощью математических инструментов, таких как преобразования Фурье, Лапласа или Габора; алгоритмы SIFT и SURF более сложные, но вместе с тем и более мощные. Например, с помощью SIFT можно распознать важные области на изображении, чтобы объект и повернутая версия одного и того же объекта создавали одинаковые или похожие локальные признаки. Это означает, что с помощью признаков на базе SIFT изображения, которые содержат одинаковые повернутые объекты, можно распознавать как похожие.

Мы не будем подробно рассматривать SIFT, потому что она не является основной темой этой книги; но если говорить вкратце, она использует *фильтр* под названием *лапласиан гауссиана* для распознавания интересных точек на изображении. Можно рассматривать фильтр как маску, применяемую к изображению. Фильтр лапласиан гауссиана создает изображение, на котором выделены края и другие ключевые точки, а большинство других точек больше не видно. Фильтр применяется к предварительно обработанной версии изображения, поэтому результирующее изображение представляется в масштабно-инвариантном виде. После применения такого фильтра интересные точки становятся неизменными по ориентации путем записи ориентации каждой точки, поэтому при каждом сравнении с другими точками компонент ориентации включается в каждую операцию вычисления или сравнения. Наконец, все найденные локальные признаки кодируются в одном сопоставимом векторе дескриптора/признака.

Локальные признаки являются представлениями частей изображения. Отдельное изображение ассоциировано с несколькими локальными признаками. Но вам нужно одно представление изображения, чтобы:

- окончательное представление изображения содержало информацию обо всех интересных локальных точках;
- можно было выполнить эффективное сравнение во время запроса (один вектор признаков против множества векторов).

Для этого необходимо объединить локальные признаки в одно представление (вектор признаков). Обычный подход заключается в агрегировании локальных признаков с использованием модели «мешок визуальных слов» (BOVW). Вы, возможно, помните модель мешка слов, о которой шла речь ранее: в такой модели документ представлен в виде вектора, размер которого равен числу слов во всех существующих документах. Каждая позиция в векторе связана с определенным словом: если значение равно 1 (или любому значению больше нуля: например, рассчитывается с использованием меры TF-IDF), то соответствующий документ содержит это слово; в противном случае значение равно 0.

Вспомните примеры представлений «мешок слов» для документов в главе 5, приведенных в табл. 8.1.

Таблица 8.1. Мешок слов

Термы	bernhard	bio	Dive	hypothesis	in	influence	into	life	mathematical	riemann
Документ1	1.28	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.28
Документ2	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0

В модели BOVW каждое значение вектора больше нуля, если изображение имеет локальный признак, соответствующий этой позиции. Таким образом, вместо слова «bernhard» или «bio» в случае с текстом у модели BOVW будет «локальный-признак1», «локальный-признак2» и т. д. Каждое изображение представлено по тому же принципу, но с использованием кластеризованных локальных признаков вместо слов; *см.* табл. 8.2.

Таблица 8.2

Признаки	локальный-признак1	локальный-признак2	локальный-признак3	локальный-признак4	локальный-признак5
Изображение1	0.3	0.0	0.0	0.4	0.0
Изображение2	0.5	0.7	0.0	0.8	1.0

Используя экстракторы локальных признаков, такие как SIFT, каждое изображение поставляется с несколькими дескрипторами, которые могут различаться в зависимости от качества изображения, его размера и других факторов.

Модель BOVW включает в себя дополнительный этап предварительной обработки для определения фиксированного числа локальных признаков. Предположим, что для набора данных изображений SIFT извлекает локальные признаки для каждого изображения, но у некоторых – десятки признаков, а у других – сотни. Чтобы создать общий словарь локальных признаков, все локальные признаки собираются вместе, и к ним применяется алгоритм кластеризации, такой как алгоритм k-средних, чтобы извлечь *n* центроидов. Центроиды – это слова для модели BOVW.

Если вы посмотрите на ясное небо темной ночью, то увидите много разных звезд. Каждую звезду можно рассматривать как точку кластера: локальный признак. Теперь представьте, что самые яркие звезды на небе имеют больше звезд рядом с собой (на самом деле яркость звезды зависит от расстояния, размера, возраста, радиоактивности и других факторов). В этих условиях самые яркие звез-

ды – это центроиды кластера; можно использовать их для обозначения всех точек с некоторой аппроксимацией. Таким образом, вместо миллиардов звезд (локальных признаков) вы рассматриваете только десятки или сотни звезд: круговые центроиды. Вот что делают алгоритмы кластеризации (*см.* рис. 8.14).

Рис. 8.14 ❖ Звезды, кластеры и центроиды

Теперь можно использовать LIRE для создания векторов признаков изображения с использованием модели BOVW. Сперва вы извлекаете локальные признаки с помощью SIFT и генерируете словарь визуальных слов, используя алгоритм кластеризации, например алгоритм k-средних:

```
for (String imgPath : imgPaths) {
    File file = new File(imgPath);
    SiftExtractor siftExtractor = new
        SiftExtractor();

    BufferedImage bufferedImage = ImageIO
        .read(file);

    siftExtractor.extract(bufferedImage);

    List<LocalFeature> localFeatures = siftExtractor
        .getFeatures();

    for (LocalFeature lf : localFeatures) {
      kMeans.addFeature(lf.getFeatureVector());
    }
}
for (int k = 0; k < 15; k++) {
    kMeans.clusteringStep();
}
Cluster[] clusters = kMeans.getClusters();
```

Перебирает все изображения

Создает экстрактор локальных признаков на основе алгоритма SIFT

Читает содержимое изображения

Выполняет алгоритм SIFT для заданного изображения

Извлекает все локальные признаки

Добавляет все признаки SIFT текущего изображения в качестве точек для кластеризации

Выполняет кластеризацию, используя метод k-средних, для предварительно определенного количества шагов

Извлекает сгенерированные кластеры

Этот код вычисляет все визуальные слова как фиксированное количество кластеров. При наличии визуального словаря локальные признаки каждого изображения сравниваются с центроидами кластера для вычисления окончательного значения каждого визуального слова. Эта задача выполняется моделью BOVW, которая вычисляет евклидово расстояние между признаками SIFT и центроидами кластеров:

```
for (String imgPath : imgPaths) {
    File file = new File(imgPath);
    SiftExtractor siftExtractor = new SiftExtractor();
    BufferedImage bufferedImage = ImageIO.read(file);
    siftExtractor.extract(bufferedImage);
    List<LocalFeature> localFeatures = siftExtractor
        .getFeatures();
    BOVW bovw = new BOVW();
    bovw.createVectorRepresentation(localFeatures
        , clusters);
    double[] featureVector = bovw
        .getVectorRepresentation();
}
```

Снова перебирает все изображения

Снова извлекает локальные признаки SIFT. Признаки SIFT могут быть временно кешированы для каждого изображения на карте, чтобы не вычислять их дважды

Создает экземпляр BOVW

Вычисляет одно векторное представление текущего изображения с учетом локальных признаков SIFT и центроидов

Извлекает векторы признаков

Этот код дает одно представление для каждого изображения, которое вы можете использовать при поиске изображений.

В данных примерах глобальное извлечение признаков использует простой экстрактор цветовой гистограммы, а при извлечении локальных признаков используется SIFT в сочетании с BOVW. Это лишь некоторые из алгоритмов, которые можно использовать для явного извлечения признаков. Например, для глобального извлечения признаков в качестве альтернативы применяется подход размытого цвета, который является немного более гибким. Для извлечения локальных признаков SURF (упомянутый ранее) – это разновидность SIFT, который является более надежным и, как правило, более подходящим с точки зрения скорости.

Основными преимуществами экстрактора признаков цветовой гистограммы являются его простота и интуитивность; основное преимущество SIFT, SURF и других экстракторов локальных признаков заключается в том, что они хорошо работают при идентификации объектов на меньших участках изображения с использованием масштабной и ротационной инвариантности. На практике продукционная система нуждается в подходе, который дает лучшие гарантии с точки зрения верности, скорости, проектирования и технического обслуживания, необходимых для работы всей системы. Если у вас есть вектор признака фиксированного размера, представляющий изображение, стратегии индексации и поиска имеют наибольшее значение с точки зрения скорости, в чем вы убедитесь позже в этой главе. Что касается проектирования, технического обслуживания и верности, экстракторы глобальных и локальных признаков, которые мы обсуждали до сих пор, были перегружены архитектурами глубокого обучения. Суть состоит в том, что признаки не извлекаются вручную, а изучаются с помощью глубокой нейронной сети.

В следующем разделе вы увидите, как это делает извлечение признаков простым всеобъемлющем учебным процессом, от пикселей до векторов признаков. Такие генерируемые с помощью глубокого обучения признаки также обычно лучше с точки зрения семантического понимания визуальных объектов.

8.4. Глубокое обучение для представления изображений

До сих пор в этой главе мы извлекали признаки из изображений. Изучение представлений данных – то, что сделало глубокое обучение настолько успешным в последние годы. Компьютерное зрение было первой областью, где глубокое обучение превзошло предыдущие современные подходы; в компьютерном зрении перед компьютерами стоит задача распознавать объекты на изображениях или видео. Это можно использовать в различных приложениях, от сканирования сетчатки до выявления нарушений правил вождения (таких как идентификация транспортных средств, которые совершают обгон, где это запрещено), оптического распознавания символов и т. д. Успех этой технологии побуждает исследователей и инженеров в области глубокого обучения работать над усложняющимися задачами, такими как, например, автомобили без водителя.

Некоторые известные результаты глубокого обучения, примененные к изображениям, включают в себя LeNet (**http://yann.lecun.com/exdb/lenet**), нейронную сеть, которая может распознавать рукописные и печатные цифры; и AlexNet (**http://mng.bz/6j4y**), нейронную сеть, которая может распознавать объекты на изображении. AlexNet особенно интересна для поиска изображений, потому что смогла классифицировать (назначить категорию) определенное изображение среди 1000 различных очень мелкозернистых категорий. Например, она может различать очень похожих собак разных пород, как показано на рис. 8.15.

Энтлебухер Аппенцеллер

Рис. 8.15 ❖ Изображение собак, классифицированных сетью AlexNet

LeNet и AlexNet используют специальный вид (искусственной) нейронной сети прямого распространения под названием *сверточная нейронная сеть*. В последние годы эти сети применяются не только при работе с изображениями и видео, но также когда речь идет о звуке и тексте; они очень гибкие и могут использоваться для самых разных задач.

В начале этой главы я упоминал, что можно использовать глубокое обучение для поиска все более абстрактных структур в изображениях. Исследователи обнаружили, что это то, что делают сверточные нейронные сети во время фазы обучения. По мере увеличения количества слоев слои, расположенные ближе к входу, изучают необработанные признаки, такие как границы и углы, в то время как слои, расположенные ближе к концу глубокой нейронной сети, изучают признаки, которые обозначают формы и объекты. Далее вы изучите архитектуру сверточных нейронных сетей, узнаете, как их обучить и настроить и, наконец, как извлечь признаки для поиска изображений (как показано на рис. 8.16).

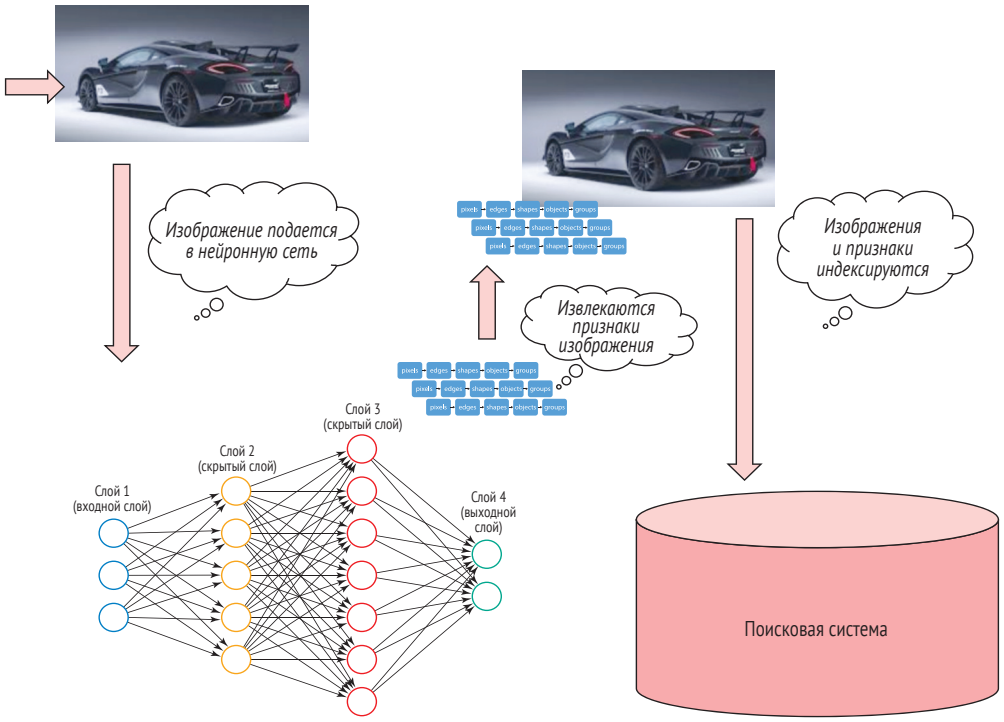


Рис. 8.16 ❖ Индексирование изображений с их признаками, извлеченных нейронной сетью

8.4.1. Сверточные нейронные сети

Несмотря на названия, связь между искусственными нейронными сетями и тем, как работает человеческий мозг, не очевидна. Наиболее распространенные архитектуры нейронных сетей являются фиксированными: часто нейроны являются полносвязными, тогда как нейроны в мозге не имеют таких фиксированных (и простых) структур. Изначально сверточные нейронные сети были вдохновлены тем, как работает зрительная кора головного мозга человека: выделенные клетки заботятся об определенных частях изображения, передавая информацию другим клеткам, которые обрабатывают информацию в потоке, подобном тому, который вы увидите в сверточной сети. Принципиальная разница того, как сверточные

нейронные сети работают относительно других типов нейронных сетей, заключается в том, что они не обрабатывают входные данные *плоских сигналов* (например, плотные векторы с унитарным кодированием).

Когда мы рассматривали создание цветовой гистограммы изображения, я упомянул, что изображения обычно представляются с использованием RGB: один пиксель описывается тремя различными значениями для красного, зеленого и синего каналов. Если распространить это на все изображение со множеством разных пикселей, у вас будет представление изображения шириной X и высотой Y, состоящее из трех разных матриц для каждого из трех компонентов RGB, каждый из которых содержит Y строк и X столбцов. Например, изображение размером 3×3 пикселя будет иметь 3 матрицы с 9 значениями в каждой. Код RGB R: 31, G: 39 и B: 201 будет генерировать цвет, показанный на рис. 8.17.

Если вы представите такое значение, помещенное в первый элемент второй строки изображения 3×3, матрицы RGB могут выглядеть так, как показано в табл. 8.3–8.5 (жирным шрифтом обозначены значения, которые обозначают пиксель на рис. 8.17).

Таблица 8.3. Красный канал

0	4	0
31	8	3
1	12	39

Таблица 8.4. Синий канал

10	40	31
39	0	0
87	101	18

Таблица 8.5. Зеленый канал

37	46	1
201	8	53
0	0	10

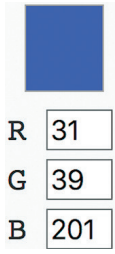


Рис. 8.17 ❖ Пример пиксельного значения RGB

Вместо одной матрицы слов или векторов символов нейронная сеть должна обрабатывать три матрицы для каждого входного изображения, по одной на цветовой канал. Это создает серьезные проблемы с производительностью, когда вы обрабатываете изображения с помощью обычных полносвязных нейронных сетей прямого распространения. Очень маленьким изображениям размером 100×100 потребуется 100 * 100 * 3 = 30 000 обучающихся весов только для первого слоя. Для изображения среднего размера (1024×768) первому слою потребуется более 2 000 000 параметров (1024 * 768 * 3 = 2 359 296)!

Сверточные нейронные сети решают проблему обработки путем обучения на больших входах, применяя легковесную конструкцию в слоях и соединениях нейронов. Меньшее количество подключений означает меньшее количество весов, которые будет изучать сеть. А меньшее количество весов делает обучение менее сложным с вычислительной точки зрения, а также быстрее. Не все нейроны в этом типе слоя всегда связаны с нейронами в предыдущем слое; такие нейроны име-

ют *рецептивное поле* определенного настраиваемого размера, которое определяет локальную область входных матриц, к которым они подключены. Поэтому некоторые нейроны не связаны со всей входной областью и, следовательно, не имеют прикрепленного веса. Такие слои называются *сверточными* и являются основным строительным блоком сверточных нейронных сетей (вместе со слоями подвыборки; *см*. рис. 8.18).

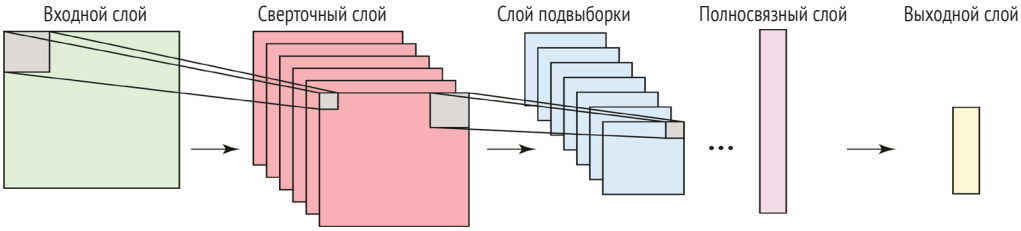


Рис. 8.18 ❖ Строительные блоки (и поток) сверточных нейронных сетей

Когда я кратко рассказал об экстракторе SIFT, то упомянул фильтр лапласиан гауссиана, который идентифицирует интересные точки на изображении. Сверточные слои несут ту же ответственность; но, в отличие от фиксированного фильтра лапласиана гауссиана, сверточные фильтры *изучаются* на этапе обучения сети, чтобы лучше адаптироваться к изображениям в обучающем наборе (*см.* рис. 8.19).

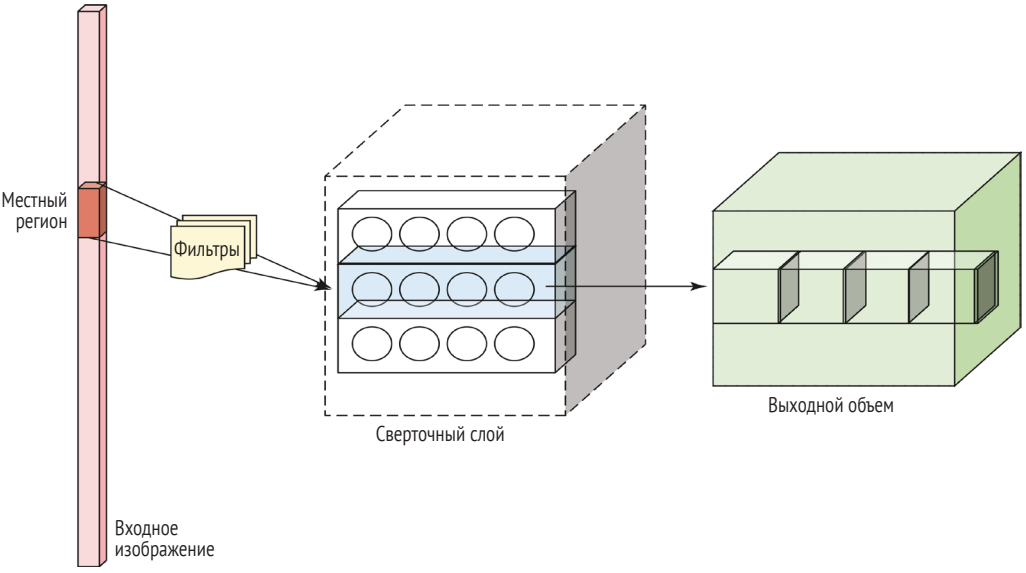


Рис. 8.19 ❖ Сверточный слой

Сверточные слои имеют настраиваемую глубину (4 на рис. 8.19), ряд фильтров и некоторые другие параметры конфигурации. Фильтры слоя содержат параметры (веса), которые изучаются сетью посредством обратного распространения ошибки во время обучения. Можно рассматривать каждый фильтр как маленькое

окно для целого изображения, изменяющего входные пиксели, которые «видит» в данный момент; фильтр скользит по всему изображению, поэтому он применяется ко всем входным значениям. Эта скользящая фильтрация является операцией свертки, которая и дает имя данному типу слоя (и сети).

У фильтра 5×5 – 25 весов, поэтому он видит 25 пикселей за раз. Говоря математически, фильтр вычисляет произведение точек между 25 значениями пикселей и 25 весами фильтра. Предположим, что сверточный слой получает входное изображение 100×100×3 (также называемое *входным объемом*, потому что у него 3 измерения). Если слой имеет 10 фильтров, выход имеет объем в 100×100×10 значений. 10 сгенерированных матриц 100×100 (по одной на каждый фильтр) называются *картами активации*.

Когда фильтр скользит по входным значениям, он перемещается по одному значению/пикселю за раз. Но иногда фильтр может сдвигать два или три значения за раз (например, по оси ширины), чтобы уменьшить количество сгенерированных выходов. Этот параметр для размера перемещения обычно называется `шагом`. Скольжение по одному значению за раз – это `шаг = 1`, скольжение по двум значениям означает `шаг = 2` и т. д.

Сверточные сети также уменьшают вычислительную нагрузку при обучении с большими входными объемами, применяя способ контроля количества изучаемых весов. Представьте, что все нейроны на рис. 8.19 имеют определенную глубину (например, `глубина = 2`). Тогда у них будут одинаковые веса. Этот метод называется *копированием параметров*.

В конце концов, основные различия между сверточными слоями и обычными полносвязными слоями нейронной сети заключаются в том, что сверточные нейроны связаны только с локальной областью входа, а некоторые нейроны в сверточном слое копируют параметры.

Слои подвыборки

Ответственность слоя подвыборки заключается в субдискретизации входного объема: он уменьшает входной размер при попытке сохранить наиболее важную информацию. Преимущество состоит в уменьшении вычислительной сложности и количества параметров, которые необходимо изучить для последовательных слоев (например, других сверточных слоев). Слои подвыборки не ассоциированы с весами, которые нужно изучать; они смотрят на части входного объема и извлекают одно или несколько значений в зависимости от выбранной функции. Распространенные функции – это `max` и `average`.

Подобно сверточным слоям, слои подвыборки имеют настраиваемый размер рецептивного поля и `шаг`. Например, слой подвыборки с размером рецептивного поля 2 и `шагом` 2 с функцией `max` будет принимать четыре значения из входного объема и выводить максимальное значение из этих входных значений.

Обучение сверточных нейронных сетей

Вы узнали об основных строительных блоках сверточных сетей. Давайте сложим их вместе, чтобы создать реальную сеть, и посмотрим, как она обучается. Помните, что основной целью является извлечение векторов признаков, которые отражают понятие семантически похожих изображений.

Типичная архитектура сверточной нейронной сети обычно включает в себя, по меньшей мере, один (или несколько) сверточный слой в сопровождении:

- плотного, полносвязного слоя для хранения векторов признаков для изображений;
- выходного слоя, содержащего оценки классов для каждого из классов, которыми изображение может быть помечено.

Сверточная нейронная сеть обычно обучается с учителем с использованием обучающих примеров, чьи входные данные представляют собой изображение, а ожидаемые результаты представляют собой набор классов, к которым принадлежит изображение.

Давайте посмотрим на известный набор данных, который часто использовался в исследованиях компьютерного зрения. Набор данных CIFAR (**www.cs.toronto.edu/~kriz/cifar.html**) содержит тысячи изображений, помеченных 10 категориями (*см.* рис. 8.20). Изображения из набора данных CIFAR представляют собой цветные изображения, каждое из которых имеет размер 32×32 пикселя (очень маленький). Таким образом, первый слой будет получать входные данные значений 32 * 32 * 3 = 3072.

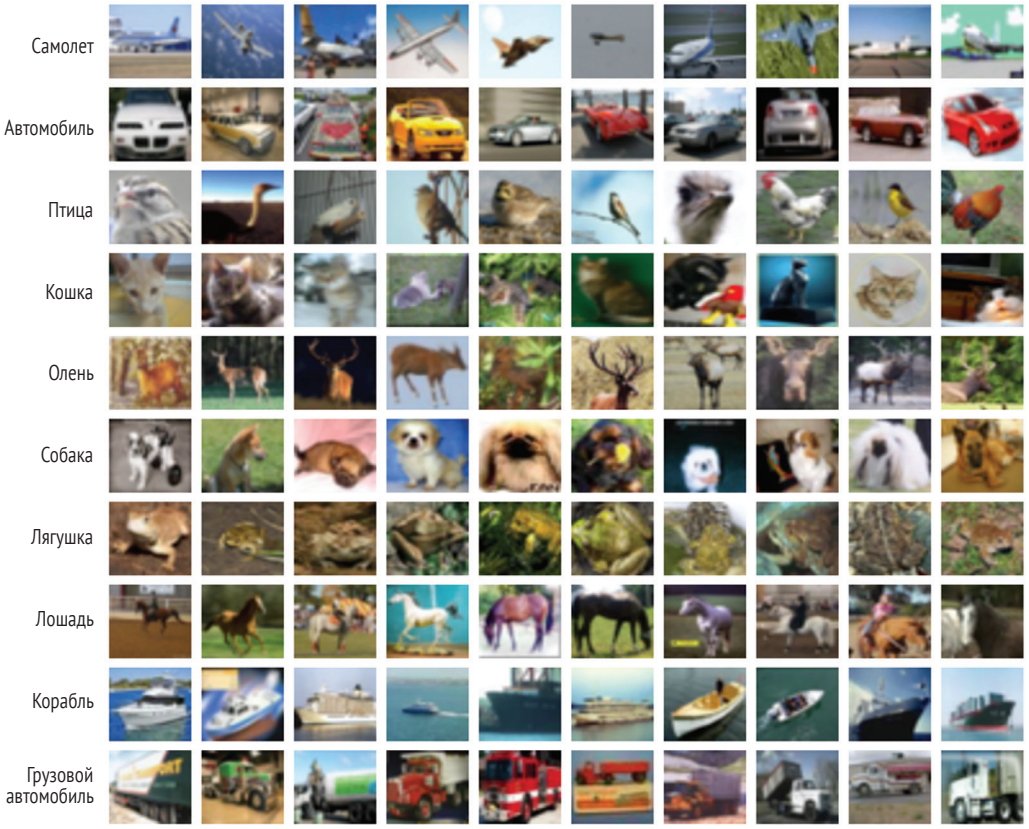


Рис. 8.20 ❖ Примеры из набора данных CIFAR

Давайте создадим простую сверточную сеть с двумя слоями свертки + подвыборки, одним плотным слоем и выходным слоем; *см.* рис. 8.21. Вы ожидаете, что сеть произведет оценку вероятности того, что входное изображение относится

к любой из 10 категорий; на рис. 8.22 показан пример результатов (изображение было сгенерировано с использованием демонстрации ConvNetJS CIFAR-10 по адресу **https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html**).

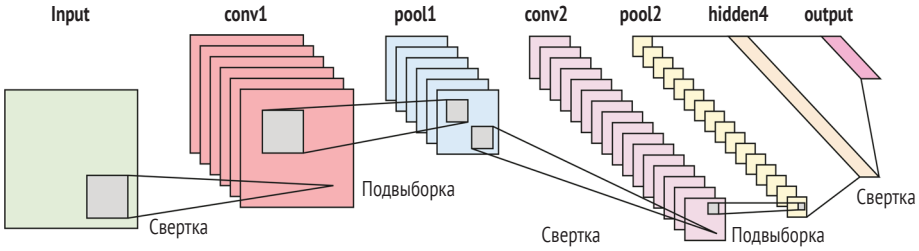


Рис. 8.21 ❖ Простая сверточная сеть с двумя слоями свертки + подвыборки

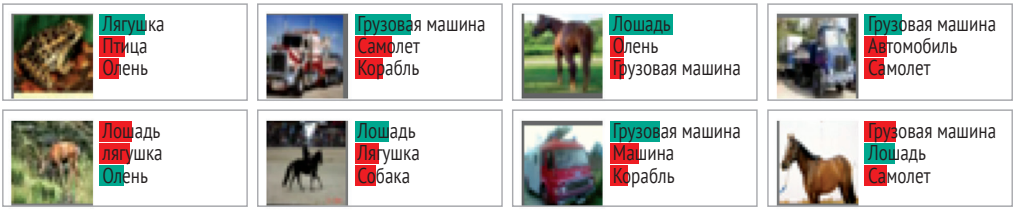


Рис. 8.22 ❖ Тестирование сверточной сети на наборе данных CIFAR

Как видно, во время обучения сверточной сети никакого инжиниринга признаков выполняться не должно; векторы признаков могут быть удалены из конечного плотного слоя, от начала до конца. Вам просто нужно большое количество картинок с метками!

Эта архитектура представляет собой простой пример сверточной нейронной сети. Многие вещи могут быть изменены в фундаментальном проекте и во многих гиперпараметрах. Например, оказалось, что добавление дополнительных сверточных слоев повышает верность. Размер рецептивного поля и глубина сверточных слоев или операций подвыборки (max, average и т. д.) – все это мощные аспекты, которые можно настроить для повышения верности.

Настройка сверточной нейронной сети в DL4J

Сверточную сеть из предыдущего раздела можно легко реализовать в Deeplearning4j. DL4J поставляется со вспомогательным классом для итерации и обучения по набору данных CIFAR, поэтому давайте используем его для обучения сверточной сети.

Листинг 8.1 ❖ Установка сверточной сети для CIFAR в DL4J

```
int height = 32;      <-- Высота входных изображений
int width = 32;       <-- Ширина входных изображений
int channels = 3;     <-- Количество каналов изображения, которые будут использоваться
int numSamples = 50000; <-- Количество обучающих примеров из набора данных CIFAR
```

```
int batchSize = 100;  <-- Размер мини-пакета

int epochs = 10;  <-- Количество эпох для обучения сети

MultiLayerNetwork model = getSimpleCifarCNN();  <-- Устанавливает сетевую архитектуру
CifarDataSetIterator dsi = new CifarDataSetIterator(
    batchSize, numSamples, new int[] {height, width,
    channels}, false, true);  <-- Создает итератор для набора данных CIFAR

for (int i = 0; i < epochs; ++i) {
  model.fit(dsi);  <-- Обучает сеть
}
cf.saveModel(model, "simpleCifarModel.json");  <-- Сохраняет модель для последующего использования
```

Архитектура модели определяется методом `getSimpleCifarCNN`, который показан далее и на рис. 8.23.

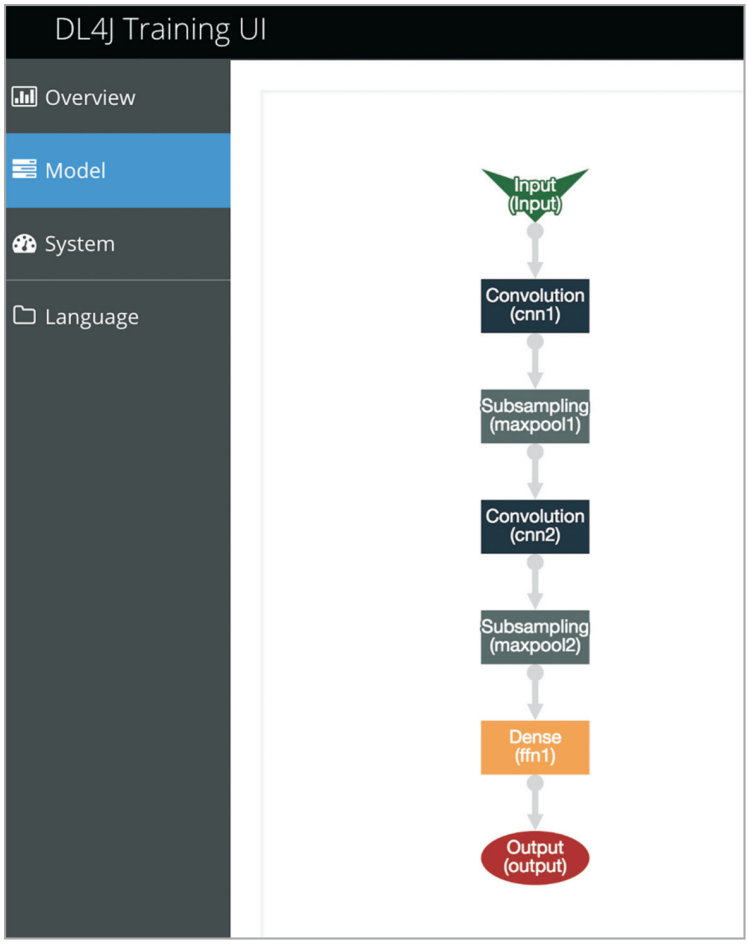


Рис. 8.23 ❖ Результирующая модель из интерфейса DL4J

Листинг 8.2 ❖ Настройка сверточной сети

```
public MultiLayerNetwork getSimpleCifarCNN() {
  MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
      .list()
      .layer(0, new ConvolutionLayer.Builder(
          new int[]{4, 4}, new int[]{1, 1},
          new int[]{0, 0}).name("cnn1")
          .convolutionMode(ConvolutionMode.Same) <------ Первый сверточный слой
          .nIn(3).nOut(64).weightInit(WeightInit.XAVIER_UNIFORM).activation(
            Activation.RELU)

      .layer(1, new SubsamplingLayer.Builder(
          PoolingType.MAX, new int[]{2,2})
          .name("maxpool1").build())  <------ Первый слой подвыборки

      .layer(2, new ConvolutionLayer.Builder(
          new int[]{4,4}, new int[] {1,1},
          new int[]{0,0}).name("cnn2")
          .convolutionMode(ConvolutionMode.Same) <------ Второй сверточный слой
          .nOut(96).weightInit(WeightInit.XAVIER_UNIFORM)
          .activation(Activation.RELU).build())

      .layer(3, new SubsamplingLayer.Builder(
          PoolingType.MAX, new int[]{2,2}).name(
          "maxpool2").build())  <------ Первый слой подвыборки

      .layer(4, new DenseLayer.Builder().name(
          "ffn1").nOut(1024).build()) <------ Плотный слой (из которого вы будете извлекать признаки)

      .layer(5, new OutputLayer.Builder(LossFunctions
          .LossFunction.NEGATIVELOGLIKELIHOOD) <------ Выходной слой

.name("output").nOut(numLabels).activation(Activation.SOFTMAX).build())
      .backprop(true).pretrain(false)
      .setInputType(InputType.convolutional(height, width, channels))
      .build();
  MultiLayerNetwork model = new MultiLayerNetwork(conf);
  model.init();
  return model;
}
```

Как только сеть закончила обучение, можно приступить к использованию сетевых выходов.

Вспомните цветовую гистограмму или модели BOVW – вы получили вектор признаков для каждого изображения. Сверточная сеть дает вам больше: плотный слой рядом с выходным слоем содержит векторы признаков, которые вы можете использовать для сравнения изображений, и у вас также есть обученная сеть, которую можно использовать для маркировки новых изображений.

После завершения обучения, если вы хотите проиндексировать векторы признаков, изученные для каждого изображения сверточной нейронной сетью, вы должны снова выполнить итерацию по набору данных изображений, выполнить вычисления прямого распространения для каждого изображения и извлечь векторы признаков, сгенерированные сетью.

Листинг 8.3 ❖ Извлечение векторов признаков

```
DataSetIterator iterator = ...   <-- Получает итератор для изображений, которые будут обработаны
while (iterator.hasNext()) {     <-- Перебирает набор данных

  DataSet batch = iterator.next(batchSize); <-- Перебирает каждый пакет
  for (int k = 0; k < batchSize; k++) {     <-- Перебирает каждое изображение из текущего пакета

    DataSet dataSet = batch.get(k);

    List<INDArray> activations = model.
        feedForward(dataSet.getFeatureMatrix(),
         false);  <-- Выполняет прямую передачу без обучения с текущим изображением (пиксели в качестве входа)

    INDArray imageRepresentation = activations
        .get(activations.size() - 2);  <-- Извлекает представление изображения, хранящееся в плотном слое перед конечным выходным слоем

    INDArray classification = activations.get(
        activations.size() - 1);  <-- Извлекает оценки классификации текущего изображения
    ...  <-- Обрабатывает (сохраняет) представление изображения
  }
}
```

Теперь вы готовы узнать, как эффективно индексировать и искать векторы признаков, извлеченные сверточной сетью (хотя в целом это относится к любому вектору признаков).

8.4.2. Поиск изображений

Давайте вернемся к примеру в начале этой главы: у вас есть фотография, сделанная камерой смартфона. Вы хотите найти профессиональную фотографию, которую можно использовать в качестве открытки к подарку. Вам нужно сделать следующее:

1) предоставить сверточной сети входное изображение;
2) извлечь сгенерированные векторы признаков;
3) использовать векторы признаков, чтобы сделать запрос для поиска похожих изображений в поисковой системе.

Вы видели, как выполнить первые два шага, в предыдущем разделе. В этом разделе вы узнаете, как эффективно выполнять запрос.

Очевидно, что для выполнения запроса нужно было бы сравнить векторы признаков входного изображения и векторы признаков всех изображений, хранящихся в поисковой системе. Представьте себе, что вы извлекаете векторы признаков из сверточной сети, такой, которую вы видели в предыдущем разделе, и наносите их на график: точки представляют собой похожие изображения, которые расположены близко друг к другу. Это та же нить рассуждений, которую мы использовали, когда имели дело с векторными представлениями слов и документов. Таким образом, вы можете вычислить расстояние между вектором признаков из входного изображения и векторами признаков всех других изображений и вернуть, например, 10 лучших изображений, которые имеют наименее удаленные векторы признаков. С вычислительной точки зрения этот подход не будет масштабироваться, поскольку время, затрачиваемое на выполнение запроса, растет линейно с увеличением количества изображений в поисковой системе. В реальной жизни такие алгоритмы поиска ближайших соседей часто аппроксимируются: они работают

лучше, но достигается это за счет верности. Такой приближенный алгоритм поиска ближайших соседей может не возвращать точные ближайшие элементы относительно входного изображения, но все же будет возвращать близких соседей гораздо быстрее.

В Lucene можно использовать (экспериментальный) класс `FloatPointNearestNeighbor`, который предоставляет приблизительную функцию ближайшего соседа, или реализовать приблизительный поиск ближайшего соседа с использованием локально-чувствительного хеширования. `FloatPointNearest-Neighbor` затратнее во время поиска, без дополнительной площади пространства в индексе; локально-чувствительное хеширование увеличивает размер индекса, потому что он требует от вас хранить больше, чем просто векторы признаков, но он быстрее во время поиска. Мы начнем с использования класса `Float-PointNearestNeighbor`, а затем посмотрим на локально-чувствительное хеширование.

Использование класса FloatPointNearestNeighbor

Чтобы использовать класс `FloatPointNearestNeighbor`, вам нужно извлечь векторы признаков сверточной сети и индексировать их в Lucene в виде точек. Последние версии Lucene имеют поддержку *n*-мерных точек (еще один способ увидеть вектор) на основе алгоритма *k*-мерного дерева (**https://en.wikipedia.org/wiki/K-d_tree**). Таким образом, вектор признаков, который вы извлекаете из сверточной сети, индексируется с использованием специального типа поля `FloatPoint`.

Листинг 8.4 ❖ Индексирование вектора признаков в виде точки

```
List<INDArray> activations = cnnModel.feedForward(currentImage, false);
INDArray imageRepresentation = activations
    .get(activations.size() - 2);
float[] aFloat = imageRepresentation.data()
    .asFloat();
doc.add(new FloatPoint("features", floats));
```

Получает вектор признаков, сгенерированный сверточной сетью

Преобразует его в массив с плавающей точкой

Индексирует вектор признаков как FloatPoint

К сожалению, начиная с Lucene 7 плавающие точки (`FloatPoint`) могут индексировать точки, размерность которых не превышает 8. Векторы признаков обычно намного больше: например, наша сверточная сеть для CIFAR генерирует векторы признаков, размерность которых равна 1024. Вам нужно уменьшить `float[]`, используемое для создания экземпляра `FloatPoint`, с 1024 значений до максимум 8.

Можно попытаться уменьшить количество размерностей в векторах, сохраняя при этом наиболее важную информацию; эта техника также называется *снижением размерности*. Существуют различные алгоритмы уменьшения размерности, и мы рассмотрим тот, который вы также можете использовать в других сценариях.

Распространенным алгоритмом уменьшения размерности является *метод главных компонент*. Как следует из названия, метод главных компонент определяет наиболее важные признаки из набора векторов объектов и отбрасывает остальные. Векторы признаков, извлеченные из сверточной сети, имеют по 1024 значения каждый. Вам нужно использовать метод главных компонент для объединения 1024 значений каждого вектора признаков в максимум 8 различных значений. С помощью этого метода вы преобразуете точку/вектор на графике с 1024 координатами в точку/вектор на графике с 8 координатами.

Интуитивно понятно, что алгоритм метода главных компонент просматривает значения каждого признака в каждом векторе, чтобы найти признаки, значения

которых отличаются больше всего (имеют наибольшую дисперсию). Такие признаки считаются наиболее важными. Метод главных компонент не отказывается от других; скорее, он создает из них новые признаки, чтобы избежать потери информации. Признаки с наибольшей дисперсией имеют больший вес при создании новых признаков. Этот метод объединит векторы признаков размером 1024, извлеченные из сверточной сети, в 8 новых признаков, чтобы вы могли индексировать каждый вектор признаков как точку Lucene.

Метод главных компонент можно реализовать несколькими способами; поскольку вы имеете дело с векторами, можно складывать их вместе в большую матрицу и использовать алгоритмы матричного разложения, такие как неотрицательное матричное разложение, усеченное сингулярное разложение и др. Мы не будем вдаваться в подробности относительно того, как работают такие алгоритмы, потому что это выходит за рамки данной книги. DL4J предоставляет инструменты для реализации метода главных компонент, поэтому мы будем использовать их.

В CIFAR имеется около 50 000 изображений с 1024 размерностями каждое, таким образом, у вас есть огромная матрица с 50 000 строк (количество векторов объектов) и 1024 столбцами (объект измерение векторов). Вам нужно уменьшить ее до матрицы 50 000×8.

Листинг 8.5 ❖ Построение матрицы векторов признаков изображения

```
CifarDataSetIterator iterator ...
INDArray weights = Nd4j.zeros(50000, 1024);  <-- Создает матрицу весов
while (iterator.hasNext()) {
  DataSet batch = iterator.next(batchSize);  <-- Перебирает весь набор данных (CIFAR)
  for (int k = 0; k < batchSize; k++) {
    DataSet dataSet = batch.get(k);
    List<INDArray> activations = model
        .feedForward(dataSet.getFeatureMatrix(),
        false);  <-- Использует model.feedForward
    INDArray imageRepresentation = activations
        .get(activations.size() - 2);  <-- Извлекает векторы признаков из плотного слоя
    float[] aFloat = imageRepresentation.data().asFloat();
    weights.putRow(idx, Nd4j.create(aFloat));  <-- Сохраняет векторы признаков в матрице весов
  }
}
```

Создав полную матрицу векторов признаков, можно запустить метод главных компонент и получить векторы, которые достаточно малы, чтобы индексировать их как плавающие точки в Lucene. Обратите внимание на то, что поскольку эта матрица настолько велика, у метода главных компонент может уйти некоторое время (например, несколько минут на современном ноутбуке), чтобы завершить работу.

Листинг 8.6 ❖ Сокращение векторных размерностей до 8

```
int d = 8;  <-- Размер целевого вектора
INDArray reduced = PCA.pca(weights, d, true);  <-- Выполняет метод главных компонент
                                                   для матрицы весов
```

Хотя это должно работать хорошо, можно сгенерировать менее крупные векторы признаков лучшего качества, позаимствовав технику сжатия векторных

представлений слов из статьи «Простое и эффективное снижение размерности векторных представлений слов»[1] и использовать ее и для векторов изображений тоже. Этот метод основан на комбинации метода главных компонент и алгоритма постобработки, чтобы выделить, какие свойства векторного представления «сильнее». Алгоритм постобработки для более сильных векторных представлений описан в статье Цзяци Му, Сума Бхата и Прамода Вишваната[2]. Можно реализовать постобработку в DL4J следующим образом.

Листинг 8.7 ❖ Постобработка для более сильных векторных представлений

```
private INDArray postProcess(INDArray weights, int d) {
  INDArray meanWeights = weights.sub(weights.meanNumber());
  INDArray pca = PCA.pca(meanWeights, d, true);
  for (int j = 0; j < weights.rows(); j++) {
    INDArray v = meanWeights.getRow(j);
    for (int s = 0; s < d; s++) {
      INDArray u = pca.getColumn(s);
      INDArray mul = u.mmul(v).transpose().mmul(u);
      v.subi(mul.transpose());
    }
  }
  return weights;
}
```

Удаляет средние значения из каждого векторного представления в матрице весов

Выполняет метод главных компонент для полученной матрицы весов

Подчеркивает конкретные значения каждого вектора

Вычитает значения главных компонент для каждого вектора

Возвращает модифицированную матрицу весов

Весь алгоритм этой модифицированной версии снижения размерности векторных представлений выполняет постобработку для матрицы весов, за которой следует метод главных компонент, а затем повторная обработка в сокращенной матрице весов.

Листинг 8.8 ❖ Снижение размерности с использованием постобработки векторных представлений

```
int d = 8;
INDArray x = postProcess(weights, d);

INDArray pcaX = PCA.pca(x, d, true);

INDArray reduced = postProcess(pcaX, d);
```

Постобработка исходных значений вектора признаков

Постобработка уменьшенных значений вектора признаков

Выполняет метод главных компонент для получения восьмимерных векторов признаков

Теперь вы можете перебрать матрицу весов и проиндексировать каждую строку как `FloatPoint` в Lucene.

Листинг 8.9 ❖ Индексирование векторов признаков

```
IndexWriter writer = new IndexWriter(directory, config);

for (int k = 0; k < reduced.rows(); k++) {
    Document doc = new Document();
    doc.add(new FloatPoint("features", reduced.getRow(k)
    .toFloatVector()));
    doc.add(new TextField("label", ..., Field.Store.YES));
```

Создает IndexWriter

Перебирает строки сокращенной матрицы весов

Индексирует вектор как FloatPoint

Индексирует метку, связанную с текущим вектором

[1] **https://arxiv.org/abs/1708.03629**.

[2] **https://arxiv.org/abs/1702.01417**.

```
    writer.addDocument(doc); ◄── Индексирует документ
}
writer.commit(); ◄──────────── Фиксирует изменения
```

Теперь, когда у вас есть изображения, проиндексированные с помощью их векторов признаков, вы можете выполнить запрос по образцу изображения и найти наиболее похожие изображения в поисковой системе. Таким образом, для запуска некоторых тестов вы получаете случайное проиндексированное изображение, извлекаете его векторы признаков и затем выполняете поиск с использованием класса `FloatPointNearestNeighbor`.

Листинг 8.10 ❖ Поиск ближайшего соседа

```
                    Получает документ, ассоциированный со случайно
                                  сгенерированным идентификатором
int rowId = random.nextInt(reader.numDocs());
Document document = reader.document(rowId); ◄──┐
TopFieldDocs docs = FloatPointNearestNeighbor.nearest(searcher,
    "features", 3, reduced.getRow(rowId).toFloatVector()); ◄──┐
ScoreDoc[] scoreDocs = docs.scoreDocs;                        │
System.out.println("query image of a : " + document.get("label"));
for (ScoreDoc sd : scoreDocs) { ◄──────── Перебирает результаты поиска
    System.out.println("-->" + sd.doc + " : " +
    reader.document(sd.doc).getField("label").stringValue());
}
```

Извлекает признаки входного изображения и выполняет поиск ближайшего соседа, возвращая три первых результата

Например, вы ожидаете, что ближайшие соседи изображений собак будут также помечены как собаки. Вот пример вывода:

```
query image of a : dog
--> 67 : dog
--> 644 : dog
--> 101 : cat

query image of a : automobile
--> 2 : automobile
--> 578 : automobile
--> 311 : truck

query image of a : deer
--> 124 : deer
--> 713 : dog
--> 838 : deer

query image of a : airplane
--> 412 : airplane
--> 370 : airplane
--> 239 : ship

query image of a : cat
--> 16 : cat
--> 854 : cat
--> 71 : cat
```

Мы начали с извлечения признаков и пришли к индексации и, наконец, поиску по изображениям. Я упомянул, что вы можете улучшить производительность

поиска, воспользовавшись алгоритмом под названием *локально-чувствительное хеширование*; вы познакомитесь с ним в следующем разделе. Там же рассматривается одна из возможных реализаций в Lucene.

8.4.3. Локально-чувствительное хеширование

Простейшая возможная реализация алгоритма *k*-ближайшего соседа проходит по всем существующим изображениям в поисковой системе и сравнивает вектор признаков входного изображения с каждым вектором признаков индексированного изображения, сохраняя только k-ближайшие из них. Это ближайшие соседи ввода – результаты поиска. Это то, что мы реализовали в предыдущем разделе.

Вспомните приведенный ранее пример со звездами и кластеризацией: если вы нанесете на график векторы признаков изображения и примените алгоритм кластеризации, то получите кластеры и центроиды. Каждое изображение принадлежит кластеру, и у каждого кластера есть центроид, который является центром кластера. Вместо того чтобы сравнивать векторы признаков входного изображения со всеми векторами из всех изображений, вы можете сравнить их только с векторами признаков центроидов.

Количество кластеров обычно намного меньше, чем количество точек (векторов), поэтому это повышает скорость сравнения. Найдя ближайший кластер, вы можете решить, нужно ли остановиться и сохранить все другие векторы, принадлежащие кластеру, в качестве ближайших соседей или нужно выполнить второй раунд поиска ближайших соседей по другим векторам признаков, принадлежащих ближайшему кластеру.

Эту основную идею можно реализовать несколькими способами. Конечно, вы можете запустить алгоритм кластеризации k-средних для векторов признаков и проиндексировать центроиды специальной меткой (например, добавив выделенное поле, которое есть только у центроидов), чтобы во время поиска выполнялся начальный запрос для извлечения центроидов. Когда центроиды доступны, одно или два выполнения алгоритма поиска точного или приблизительного ближайшего соседа (сначала для центроидов, а затем для ближайших точек кластера).

Одна из проблем заключается в том, что кластер необходимо поддерживать в актуальном состоянии; по мере индексации новых изображений кластер и, следовательно, центроиды могут меняться. Для этого может потребоваться запустить алгоритм кластеризации несколько раз. То же самое относится и к алгоритму снижения размерности, необходимому для индексации малых векторов.

Более облегченный, но прекрасный подход заключается в использовании хеш-функций и хеш-таблиц для поиска нечетких дубликатов. Хеш-функции – это всего лишь один из способов, с помощью которого детерминированные функции всегда могут преобразовать ввод в один и тот же вывод. (Невозможно восстановить входное значение из выходного.) Причина выбора хеш-функций для этой задачи заключается в том, что они очень хорошо подходят для обнаружения нечетких дубликатов. Когда два значения выдают одинаковый результат, они вызывают *коллизию хеш-функции*. Когда хеш-функция применяется к различным входным данным и вам нужно быстро получить их, их можно собрать в хеш-таблицу. Приятный момент касательно хеш-таблиц заключается в том, что вы можете получить элемент с помощью хеширования; вместо того чтобы искать его, просматривая все элементы, хеш-функция сообщает вам его местоположение в хеш-таблице.

С помощью локально-чувствительного хеширования векторы признаков входного изображения передаются через несколько различных хеш-функций, поэтому аналогичные элементы отображаются в одни и те же *сегменты* (хеш-таблицы). Внутренне цель локально-чувствительного хеширования состоит в том, чтобы максимизировать вероятность коллизии хеша для двух подобных элементов. Когда входное изображение подается в алгоритм локально-чувствительного хеширования, оно передает свои векторы признаков через несколько хеш-функций, и сегмент, в котором оказываются векторы признаков входного изображения, говорит, с какими изображениями необходимо сравнивать входное изображение. Эта операция выполняется так же быстро, как и функции хеширования, которые обычно бывают быстрыми. Кроме того, используя специальные типы хеш-функций, вы обычно можете отобразить аналогичные входные данные в те же сегменты.

В Lucene этот подход реализуется путем создания специального анализатора. Анализатор локально-чувствительного хеширования, который мы создадим, выполнит ряд шагов для получения значений хеш-функции или сегментов, которые хранятся в индексе в виде обычного текста. Поэтому хотя вы можете использовать поля `FloatPoint` для работы с векторами признаков в качестве точек в векторном пространстве, вы также можете использовать текстовые возможности Lucene, а хеш-значения, сгенерированные алгоритмом, будете сохранять как простые токены.

Алгоритм локально-чувствительного хеширования создаст хеши для частей вектора признаков, а также и для всего вектора. Это делается для максимизации вероятности совпадения. Сначала вы токенизируете вектор признаков и извлекаете каждый признак с его положением. Например, из вектора признаков <0,1, 0,2, 0,3, 0,4, 0,5> вы получите следующие токены: 0,1 (позиция 0), 0,2 (позиция 1), 0,3 (позиция 2), 0,4 (позиция 3), 0,5 (позиция 4). Вы можете включить позицию каждого токена в текст токена, чтобы хеш-функция, применяемая к тексту токена, рассчитывалась на основе позиции каждого токена. Весь вектор признаков также сохраняется.

Затем вы создадите n-граммы для каждого отдельного токена: вы создаете не хеши всего вектора или отдельных признаков, а, скорее всего, вектора и его частей. Например, биграмма вектора признаков <0,1, 0,2, 0,3, 0,4, 0,5> – 0,1_0,2,0,2_0,3, 0,3_0,4, 0,4_0,5.

Наконец, вы примените локально-чувствительное хеширование с помощью встроенного фильтра Lucene `MinHash`. Фильтр `MinHash` применяет к термам несколько хеш-функций, генерируя соответствующие значения.

Листинг 8.11 ❖ Класс LSHAnalyzer

```
public class LSHAnalyzer extends Analyzer {
...
  @Override
  protected TokenStreamComponents createComponents(String fieldName) {
    Tokenizer source = new FeatureVectorsTokenizer();   <-- Токенизирует вектор признаков
    TokenFilter featurePos = new FeaturePositionTokenFilter(source);   <-- Прикрепляет информацию о позиции к каждому токену
    ShingleFilter shingleFilter = new ShingleFilter
        (featurePos, min, max);   <-- Создает n-граммы признаков
        shingleFilter.setTokenSeparator(" ");
    shingleFilter.setOutputUnigrams(false);
    shingleFilter.setOutputUnigramsIfNoShingles(false);
    TokenStream filter = new MinHashFilter(shingleFilter, hashCount,
```

```
        bucketCount, hashSetSize, bucketCount > 1);
    return new TokenStreamComponents(source, filter);
  }
...
}
```

Применяет фильтр локально-чувствительного хеширования

Чтобы использовать локально-чувствительное хеширование, необходимо воспользоваться этим анализатором (как вы видели в других частях книги) как во время индексации, так и во время поиска по полю, в котором индексируются векторы признаков. Обратите внимание, что, используя локально-чувствительное хеширование, вам не нужно сокращать векторы признаков, как вы это делали в предыдущем разделе: их можно оставить такими, как есть (например, 1024 значения), и передать в `LSHAnalyzer`, который создает хеш-значения векторов признаков.

Как и прежде, вы настраиваете `LSHAnalyzer`, чтобы использовать его в поле, где будут храниться значения хеш-функции.

Листинг 8.12 ❖ Настройка LSHAnalyzer для поля «lsh»

```
Map<String, Analyzer> mappings = new HashMap<>();

mappings.put("lsh", new LSHAnalyzer());

Analyzer perFieldAnalyzer = new PerFieldAnalyzerWrapper(new
    WhitespaceAnalyzer(), mappings);

IndexWriterConfig config = new IndexWriterConfig(perFieldAnalyzer);

IndexWriter writer = new IndexWriter(directory, config);
```

Создает карту, содержащую анализаторы для каждого поля

Всякий раз, когда в документе есть поле с именем «lsh», используется LSHAnalyzer

Создает конфигурацию индексации с помощью определенных анализаторов

Создает анализатор для каждого поля

Создает IndexWriter для индексации документов Lucene

После настройки конфигурации индексирования можно приступить к индексированию векторов признаков. Предполагая, что вы извлекли векторы признаков для каждого изображения в матрице (например, с именем `weights`), где каждая строка имеет 1024 столбца (значения признаков), вы можете проиндексировать каждую строку в поле `lsh`, которое обрабатывается `LSHAnalyzer`.

Листинг 8.13 ❖ Индексация векторов признаков в поле lsh

```
int k = 0;
for (String sl : stringLabels) {
  Document doc = new Document();
  float[] fv = weights.getRow(k).toFloatVector();

  String fvString = toString(fv);

  doc.add(new TextField("label", sl, Field.Store.YES));

  doc.add(new TextField("lsh", fvString, Field.Store.YES));

  writer.addDocument(doc);
  k++;
}
writer.commit();
```

Перебирает изображения по их меткам (например, «собака», «олень», «машина» и т. д. для набора данных CIFAR)

Получает вектор признаков из матрицы weights

Преобразует вектор признаков float [] в строку

Индексирует метку текущего изображения

Индексирует вектор признаков текущего изображения в поле «lsh»

Индексирует документ

Сохраняет изменения на диске

Чтобы запросить похожие изображения с помощью локально-чувствительного хеширования, вы получаете вектор признаков искомого изображения, извлекаете его хеши токенов и выполняете простой текстовый запрос, используя эти хеши.

Листинг 8.14 ❖ Выполнение запроса с использованием LSHAnalyzer

```
String fvString = reader.document(docId).get("lsh");
Analyzer analyzer = new LSHAnalyzer();
Collection<String> tokens = getTokens(analyzer, "lsh", fvString);
BooleanQuery.Builder booleanQuery = new BooleanQuery.Builder();
for (String token : tokens) {
  booleanQuery.add(new ConstantScoreQuery(new TermQuery(new Term(
  fieldName, token))), BooleanClause.Occur.SHOULD);
}
Query lshQuery = booleanQuery.build();
TopDocs topDocs = searcher.search(lshQuery, 3);
```

Получает строку вектора признаков искомого изображения

Создает LSHAnalyzer

Получает хеши токенов вектора признаков с помощью LSHAnalyzer

Создает логический запрос

Для каждого хеша токена создается запрос терма (с постоянной оценкой)

Завершает создание запроса

Выполняет запрос и получает три лучших результата

С помощью локально-чувствительного хеширования, как правило, можно получить похожих кандидатов быстрее, чем если бы вы выполняли запрос с помощью поиска ближайшего соседа, за счет увеличения пространства индекса, занятого термами векторов признаков, созданными `LSHAnalyzer`. Преимущества локально-чувствительного хеширования в скорости особенно очевидны, когда число изображений в индексе очень велико. Кроме того, сокращение размерностей векторов признаков до небольшого значения (например, 8, как в предыдущем разделе) иногда может быть очень затратным с точки зрения вычислений; локально-чувствительное хеширование не требует предварительной обработки векторов признаков, поэтому оно может быть более подходящим вариантом, чем ближайший сосед в таких сценариях, независимо от времени запроса.

8.5. Работа с непомеченными изображениями

В этом разделе мы рассмотрим случай, когда у вас есть набор непомеченных изображений и вы не можете создать обучающий набор, где каждое изображение помечено соответствующими классами (например, олень, автомобиль, корабль, трек и т. д. в наборе данных CIFAR).

Это может быть ваш собственный набор изображений, который вы хотите найти. Как вы видели в предыдущих разделах, необходимо иметь векторное представление каждого изображения, чтобы искать его, основываясь на его содержимом. Но если на ваших изображениях нет меток, нельзя сгенерировать их векторы признаков, используя архитектуру сверточных нейронных сетей, описанную в предыдущих разделах.

Для преодоления этой проблемы вы будете использовать тип нейронной сети, задача которой состоит в том, чтобы научиться кодировать входные данные, как правило, с меньшей размерностью, по сравнению с исходной, и затем реконструировать ее. Такие нейронные сети, именуемые *автоэнкодерами*, обычно строят-

ся так, что одна часть сети кодирует входные данные в вектор (также известный как *скрытое представление*) с фиксированным размером, а затем этот вектор снова преобразуется в исходные входные данные, которые также используются в качестве целевого вывода. Такие автоэнкодеры можно использовать, например, для преобразования вектора изображения в восьмимерный вектор, чтобы его можно было индексировать как `FloatPoint`. Часть автоэнкодера, которая преобразует входные данные в другой вектор с желаемой размерностью (в нашем случае это может быть восемь), называется *кодером*. Часть сети, которая преобразует скрытое представление обратно в исходные данные, называется *декодером*. Чаще всего структуры кодера и декодера одинаковы и только отражены зеркально, как видно на примере автоэнкодера на рис. 8.24.

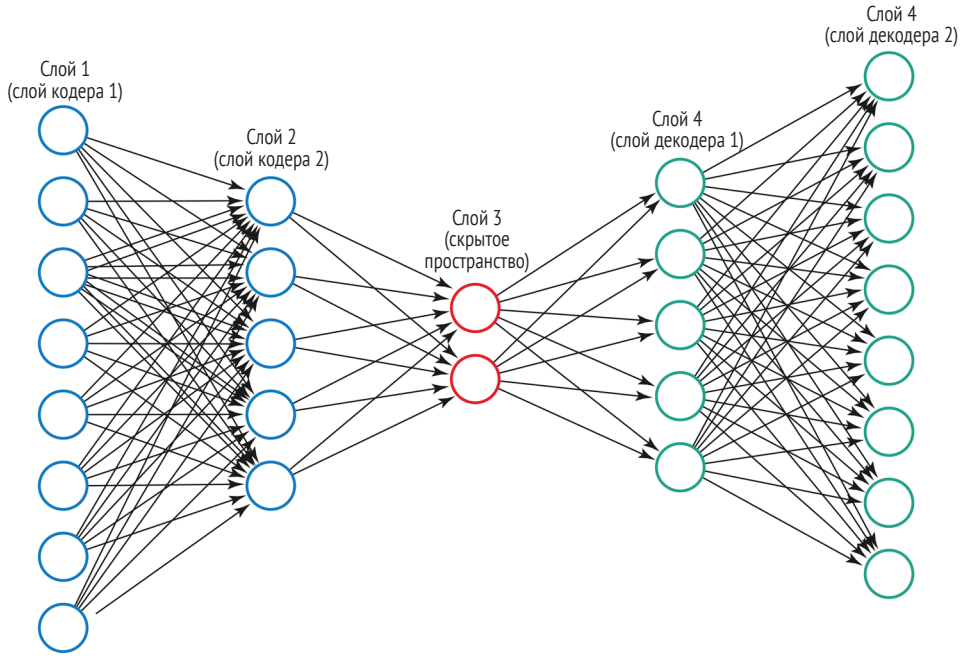


Рис. 8.24 ❖ Автокодировщик

Существует много «вариаций» автоэнкодеров. В случае генерации компактного скрытого представления больших векторов изображения вы будете использовать *вариационный автоэнкодер*. Вариационный автоэнкодер генерирует скрытые представления, которые следуют единичному нормальному распределению.

Чтобы проверить использование автоэнкодера с непомеченными данными, вы по-прежнему будете использовать набор данных CIFAR, но не будете использовать классы, прикрепленные к каждому изображению, для обучения сети. Вместо этого вы будете использовать их, чтобы оценить, насколько хороши результаты поиска после окончания обучения. Но важной частью этого подхода является то, что у вас будет возможность генерировать плотное векторное представление, например вектор признаков, для своих изображений, даже когда они не помечены.

Давайте создадим вариационный автоэнкодер в DL4J со скрытым представлением размером 8 и двумя скрытыми слоями для кодера и декодера. Первый скрытый слой будет иметь 256 нейронов, а второй – 128.

Листинг 8.15 ❖ Конфигурация вариационного автоэнкодера

```
int height = 32;
int width = 32;
int numSamples = 2000;

MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .list()
    .layer(0, new VariationalAutoencoder.Builder()
            .activation(Activation.SOFTSIGN)
            .encoderLayerSizes(256, 128)
            .decoderLayerSizes(256, 128)
            .pzxActivationFunction(Activation.IDENTITY)
            .reconstructionDistribution(
                new BernoulliReconstructionDistribution(
                    Activation.SIGMOID.getActivationFunction()))
            .numSamples(numSamples)
            .nIn(height * width)
            .nOut(8)
            .build())
    .pretrain(true).backprop(false).build();

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

Использует особый класс builder

Определяет размер каждого скрытого слоя для кодера

Определяет размер каждого скрытого слоя для декодера

Размер входных данных

Размер скрытого представления

Вы хотите использовать изображения CIFAR для обучения вариационного автоэнкодера; однако, как уже обсуждалось в предыдущих разделах, изображения состоят из нескольких каналов. В случае с CIFAR каждое изображение ассоциировано с тремя матрицами размером 32×32. Даже если вы используете один-единственный канал, автоэнкодер ожидает вектор, а не матрицу. Чтобы исправить это, нужно преобразовать матрицу 32×32 в вектор размером 1024, что можно сделать с помощью операции *изменения формы,* как показано в приведенном ниже коде. Для простоты мы предполагаем использование изображений CIFAR в оттенках серого, поэтому вместо трех каналов используется только один.

Листинг 8.16 ❖ Изменение формы изображений CIFAR для вариационного автоэнкодера

```
int channels = 1;
int batchSize = 128;
CifarDataSetIterator dsi = new CifarDataSetIterator(
    batchSize, numSamples, new int[] {height, width,
    channels}, preProcessCifar, true);

Collection<DataSet> reshapedData = new
    LinkedList<>();
while (dsi.hasNext()) {
  DataSet batch = dsi.next(batchSize);
  for (int k = 0; k < batchSize; k++) {
    DataSet current = batch.get(k);
    DataSet dataSet = current.reshape(1, height *
```

Читает набор данных CIFAR

Перебирает изображения

Сохраняет измененные данные в коллекции, которая будет использоваться для обучения вариационного автоэнкодера

```
        width);                                  Изменяет изображение с 32×32 на 1
    reshapedData.add(dataSet);                   Добавляет измененное изображение в коллекцию
  }
}
dsi.reset();
```

После того как изображения были изменены, вы можете подать их в вариационный автоэнкодер для обучения.

Листинг 8.17 ❖ Предварительное обучение вариационного автоэнкодера

```
int epochs = 3;
DataSetIterator trainingSet = new
    ListDataSetIterator<>(reshapedData);     Преобразует коллекцию в DL4J DataSetIterator
model.pretrain(trainingSet, epochs);         Обучает вариационный автоэнкодер
                                             для некоего количества эпох
```

Как только обучение закончится, вы можете, наконец, проиндексировать каждое скрытое представление изображения в индекс Lucene. Для упрощения оценки вы также можете проиндексировать метки каждого изображения в поисковую систему. Таким образом можно сравнить метки искомого изображения с метками результирующих изображений.

Листинг 8.18 ❖ Индексирование векторов изображений, извлеченных из вариационного автоэнкодера

```
VariationalAutoencoder vae = model.getLayer(0);   Получает вариационный автоэнкодер
trainingSet.reset();                              для извлечения векторов изображения
List<float[]> featureList = new LinkedList<>();
while (trainingSet.hasNext()) {                   Итерация по изображениям CIFAR
  DataSet batch = trainingSet.next(batchSize);
  for (int k = 0; k < batchSize; k++) {
    DataSet dataSet = batch.get(k);
    INDArray labels = dataSet.getLabels();
    String label = cifarLabels.get(labels.argMax(1)   Получает метку, прикрепленную
        .maxNumber().intValue());                     к текущему изображению

    INDArray latentSpaceValues = vae.activate(dataSet
        .getFeatures(), false, LayerWorkspaceMgr
        .noWorkspaces());                         Заставляет автоэнкодер выполнять
    float[] aFloat = latentSpaceValues.data().asFloat();   прямую передачу с текущим измененным
    Document doc = new Document();                изображением в качестве ввода
    doc.add(new FloatPoint("features", aFloat));
    doc.add(new TextField("label", label, Field.Store.YES));
    writer.addDocument(doc);                  Индексирует документ с вектором изображения и меткой
    featureList.add(aFloat);                  Сохраняет извлеченные признаки каждого изображения
  }                                           в список, чтобы вы могли использовать их позже
}                                             для выполнения запросов
writer.commit();
```

После того как все изображения были проиндексированы с их скрытым представлением и меткой, вы можете использовать `FloatPointNearestNeighbor` от Lucene

для выполнения поиска ближайшего соседа. Чтобы увидеть, являются ли результаты успешными, не просматривая каждый запрос и данные результирующего изображения, вы можете проверить, имеют ли запрос и каждое результирующее изображение одну и ту же метку.

Листинг 8.19 ❖ Опрос по изображению с использованием ближайшего соседа

```
DirectoryReader reader = DirectoryReader.open(writer);
IndexSearcher searcher = new IndexSearcher(reader);

Random r = new Random();
for (int counter = 0; counter < 10; counter++) {
    int idx = r.nextInt(reader.numDocs() - 1);
    Document document = reader.document(idx);
    TopFieldDocs docs = FloatPointNearestNeighbor
        .nearest(searcher, "features", 2, featureList
        .get(idx));
    ScoreDoc[] scoreDocs = docs.scoreDocs;
    System.out.println("query image of a : " +
        document.get("label"));
    for (ScoreDoc sd : scoreDocs) {
      System.out.println("-->" + sd.doc +" : " +
        reader.document(sd.doc).getField("label")
        .stringValue());
    }
    counter++;
}
```

Выбирает случайное число

Извлекает документ со случайным числом в качестве идентификатора документа

Выполняет поиск ближайшего соседа, используя вектор изображения, ассоциированный с идентификатором документа

Выводит метку искомого изображения

Выводит идентификатор документа полученного изображения и метку

Мы ожидаем, что искомое и результирующее изображения будут использовать одну и ту же метку в большинстве случаев. Это можно проверить в приведенном ниже выводе:

```
query image of a : automobile
-->277 : automobile
-->1253 : automobile
query image of a : airplane
-->5250 : airplane
-->1750 : ship
query image of a : deer
-->7315 : deer
-->1261 : bird
query image of a : automobile
-->9983 : automobile
-->4239 : automobile
query image of a : airplane
-->6838 : airplane
-->4999 : airplane
```

Как и ожидалось, в большинстве результатов вы видим общую метку. Обратите внимание, что вы также можете использовать метод локально-чувствительного хеширования, описанный в предыдущем разделе, вместо поиска ближайшего соседа.

Резюме

- Поиск по двоичному контенту, например по изображениям, требует изучения представления, которое может отображать визуальную семантику, которую можно сравнивать по изображениям.
- Традиционные методы извлечения признаков имеют ограничения и требуют значительной работы.
- Сверточные нейронные сети лежат в основе недавнего роста глубокого обучения, потому что они могут постепенно изучать абстракции представления изображений (края, формы и объекты) во время обучения сети.
- Сверточные нейронные сети можно использовать для извлечения векторов признаков из изображений, которые можно использовать для поиска похожих изображений.
- Методы локально-чувствительного хеширования можно использовать в качестве альтернативы подходу с использованием ближайшего соседа для поиска изображения на основе векторов признаков.
- Автоэнкодеры могут помочь извлекать векторы изображений, если ваши изображения не помечены.

Глава 9

Взглянем на производительность

О чем идет речь в этой главе:

- настройка моделей глубокого обучения в рабочем окружении;
- оптимизация производительности и развертывания;
- практические поисковые системы на базе нейронных сетей для работы с потоками данных.

Прочитав предыдущие восемь глав, вы, надеюсь, получили широкое понимание глубокого обучения и того, как оно может улучшить процесс поиска. На этом этапе вы должны быть готовы максимально использовать глубокое обучение при настройке успешных поисковых систем для своих пользователей. Однако по пути вы, возможно, задавались вопросом относительно того, как применить эти идеи в реальных условиях:

- как эти подходы применяются на практике?
- если добавить эти алгоритмы глубокого обучения, серьезно ли это повлияет на временные и пространственные ограничения ваших систем?
- насколько велико это влияние, и какие части или процессы (такие как поиск или индексирование) будут затронуты?

В данной главе мы рассмотрим эти практические проблемы, и я расскажу, о чем стоит подумать при применении глубокого обучения и нейронных сетей в своей поисковой системе. Мы посмотрим на биты производительности, когда поисковые системы и нейронные сети работают бок о бок, и я представлю несколько предложений на основе примеров для применения этих методов глубокого обучения на практике.

В предыдущих главах рассматривались различные проблемы поиска, которые может решить глубокое обучение. Если говорить о применении модели word2vec для расширения синонимов (глава 2) или рекуррентных нейронных сетей для расширения запросов (глава 3), вы, наверное, помните, что данные поступают в нейронные сети и из них, а также из поисковых систем и из них. Мы можем рассматривать поисковую систему и нейронную сеть как два отдельных компонента архитектуры программного обеспечения для решения реальных задач. Нейронная сеть должна быть обучена прогнозировать точные результаты. В то же время поисковая система должна принимать данные, чтобы пользователи могли их ис-

кать. Чтобы использовать глубокое обучение для получения более эффективных результатов поиска, нам нужна нейронная сеть, которая будет эффективной. Это несколько противоречивые требования, которые поднимают ряд логистических вопросов:

- должно ли обучение проходить до индексации?
- или сначала должно быть индексирование?
- можете ли объединить эти задачи подачи данных?
- как обрабатывать обновления данных?

Мы ответим на некоторые из этих вопросов, рассмотрев соображения, которые необходимо учитывать при запуске развертываний поисковых систем с использованием нейронных сетей на практике.

9.1. Производительность и перспективы глубокого обучения

Новые архитектуры глубокого обучения для решения все более сложных задач публикуются постоянно. Мы рассмотрели некоторые из них в этой книге: например, создание текста (главы 3 и 4), перевод текста с одного языка на другой (глава 7), классификация и представление изображений на основе их содержимого (глава 8) и т. д. Постоянно исследуются и публикуются не просто целые модели, но и новые типы функций активации, функции потерь, оптимизации алгоритма обратного распространения, схемы инициализации весов и многое другое.

Концепции глубокого обучения, представленные в этой книге, применимы к недавним, текущим и (надеюсь) новым архитектурам нейронных сетей. Если вы несете ответственность за инфраструктуру поисковой системы, то, вероятно, будете искать подходы, которые, как продемонстрировали исследователи, лучше всего для конкретной задачи (также известные как последние достижения). Например, возьмем машинный перевод или поиск изображений: на момент написания этих строк последние достижения в области машинного перевода представлены моделями seq2seq с акцентом на сети кодер–декодер[1]. Поэтому вам бы хотелось реализовать эти модели, и вы ожидаете, что они дадут хорошие результаты, подобные тем, о которых вы можете прочитать в соответствующих научных статьях. В этих случаях первая задача состоит в том, чтобы воспроизвести модель, описанную в статье, а затем заставить ее эффективно работать с вашими данными и инфраструктурой. Для этого:

- нейронная сеть должна обеспечивать точные результаты;
- нейронная сеть должна быстро предоставлять результаты;
- программное и аппаратное обеспечение должно быть адекватно вычислительной нагрузке с точки зрения времени и пространства (и помните: обучение стоит дорого).

В следующем разделе мы пройдем весь процесс реализации модели нейронной сети для решения конкретной задачи и посмотрим, какие общие шаги вам, возможно, придется предпринять на пути решения этих задач.

---

[1] *См.* недавнее исследование, в котором даже отказываются от рекуррентных нейронных сетей: *Ашиш Васвани и др.* Attention Is All You Need (**http://mng.bz/nQZK**).

9.1.1. От проектирования модели до производства

В главе 8 вы увидели сверточные нейронные сети, классифицирующие изображения, в действии. После завершения обучения вы использовали сеть для извлечения векторов признаков, которые должны быть проиндексированы и найдены поисковой системой. Но мы не учитывали верность классификаций нейронных сетей. Давайте теперь отследим показатели верности, времени обучения и прогнозирования на пути к созданию хорошей модели нейронной сети, которую можно использовать в сочетании с поисковой системой. Мы вернемся к набору данных CIFAR, который использовали в главе 8, и посмотрим, как постепенно настроить модель нейронной сети для повышения верности при сохранении разумного времени обучения; мы пройдем этот процесс шаг за шагом, как вы бы делали это в собственном проекте.

Индексирование обычно обходится дорого, когда используются реальные данные. CIFAR – это всего лишь несколько десятков тысяч изображений, но во многих развертываниях в реальном времени необходимо индексировать сотни тысяч, миллионы или миллиарды изображений или документов. Если вы проиндексировали 100 млн изображений с векторами признаков, вам не нужно повторять этот процесс, как вам может понадобиться, если векторы признаков не будут точно отражать содержимое изображения, и, следовательно, опыт взаимодействия будет не очень удачным. Поэтому обычно вы проводите несколько экспериментов и оценок перед индексацией векторов признаков.

Давайте начнем со сверточной нейронной сети, похожей на одну из первых архитектур на базе CNC, которая достигла хороших результатов при категоризации изображений: архитектуры LeNet (**http://yann.lecun.com/exdb/lenet**). Это простая сверточная сеть, аналогичная той, которую вы установили в главе 8, но у нее слегка другие параметры конфигурации глубины свертки, размера рецептивного поля, шага и размерности плотного слоя (см. рис. 9.1).

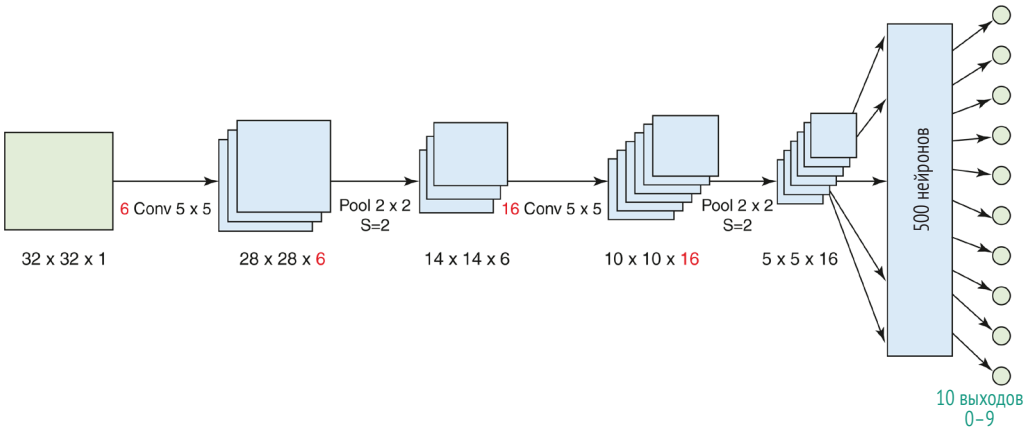


Рис. 9.1 ❖ Пример модели LeNet

Эта модель содержит две последовательности сверточных слоев, за которыми следует слой подвыборки с определением максимального значения и полносвязный слой. Фильтры имеют размер 5×5, глубина первого сверточного слоя равна

28, а глубина второго сверточного слоя – 10. Плотный слой имеет размер 500. Слои подвыборки с определением максимального значения имеют шаг, равный 2.

Листинг 9.1 ❖ Тип модели LeNet

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .list()
    .layer(0, new ConvolutionLayer.Builder(new int[]{5, 5}, new int[]{1, 1}
        , new int[]{0, 0}).convolutionMode(ConvolutionMode.Same)
        .nIn(3).nOut(28).activation(Activation.RELU).build())
    .layer(1, new SubsamplingLayer.Builder(PoolingType.MAX,
        new int[]{2,2}).build())
    .layer(2, new ConvolutionLayer.Builder(new int[]{5,5}, new int[] {1,1},
        new int[] {0,0}).convolutionMode(ConvolutionMode.Same)
        .nOut(10).activation(Activation.RELU).build())
    .layer(3, new SubsamplingLayer.Builder(PoolingType.MAX,
        new int[]{2,2}).build())
    .layer(4, new DenseLayer.Builder().nOut(500).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
        .NEGATIVELOGLIKELIHOOD)
        .nOut(numLabels).activation(Activation.SOFTMAX).build())
    .backprop(true)
    .pretrain(false)
    .setInputType(InputType.convolutional(height, width, channels))
    .build();
```

Это старая модель, поэтому не стоит ожидать, что она будет работать слишком хорошо, но можно начать с маленькой модели и посмотреть, насколько далеко она продвинется.

Сначала мы проведем обучение на более чем 2000 примеров из набора данных CIFAR, чтобы быстро узнать, насколько хороши параметры модели. Если модель начинает расходиться слишком рано, вы можете избежать загрузки огромных обучающих наборов, прежде чем обнаружите это.

Листинг 9.2 ❖ Обучение с использованием более чем 2000 образцов из CIFAR

```
int height = 32;
int width = 32;
int channels = 3;
int numSamples = 2000;
int batchSize = 100;
boolean preProcessCifar = false;
CifarDataSetIterator dsi = new CifarDataSetIterator(batchSize, numSamples,
    new int[] {height, width, channels}, preProcessCifar, true);

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
for (int i = 0; i < epochs; ++i) {
  model.fit(dsi);
}
```

Вы используете только 2000 случайных образцов из набора данных CIFAR

Оценка модели

Чтобы отслеживать, насколько хорошо нейронная сеть учится классифицировать изображения, вы будете следить за процессом обучения с помощью пользователь-

ского интерфейса DL4J. В лучшем случае вы бы увидели, что оценка постепенно снижается до 0, но в этом случае, как показано на рис. 9.2, она очень медленно уменьшается, не достигая точки, близкой к 0. Вспомните, что оценка является мерой величины ошибки, которую нейронная сеть фиксирует при попытке спрогнозировать классы для каждого входного изображения. Таким образом, при такой статистике вы не ожидаете, что она будет работать очень хорошо.

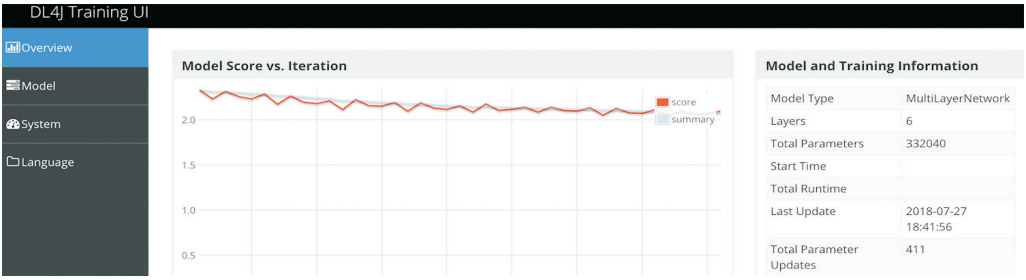


Рис. 9.2 ❖ Обучение LeNet

Чтобы оценить верность прогнозов модели машинного обучения, всегда рекомендуется отделять набор данных, используемых для обучения (обучающий набор), от набора данных, которые будут использоваться для проверки качества модели (набор тестов). Во время обучения может наблюдаться переобучение что, следовательно, может дать подходящие показатели верности в обучающем наборе, без возможности правильно обобщать слегка отличающиеся данные. Поэтому использование набора тестов помогает выяснить, насколько хорошо модель может работать с данными, на которых она ранее не обучалась.

Можно создать отдельный итератор для другого набора изображений и передать его инструментам DL4J для оценки.

Листинг 9.3 ❖ Оценка модели с помощью DL4J

```
CifarDataSetIterator cifarEvaluationData = new
    CifarDataSetIterator(batchSize, 1000, new int[] {
    height, width, channels}, preProcessCifar, false);  <-- Создает итератор набора тестов
Evaluation eval = new Evaluation(cifarEval
    .getLabels());  <-- Создает экземпляр инструмента оценки DL4J

while(cifarEvaluationData.hasNext()) {  <-- Перебирает набор тестовых данных

  DataSet testDS = cifarEvaluationData.next(
    batchSize);  <-- Выбирает следующий мини-пакет данных (в данном случае 100 примеров)

  INDArray output = model.output(testDS
    .getFeatureMatrix());  <-- Выполняет прогноз по текущему пакету

  eval.eval(testDS.getLabels(), output);  <-- Выполняет оценку, используя фактический вывод
}                                             и выходные метки CIFAR
System.out.println(eval.stats());  <-- Выводит статистику
```

Статистика оценки включает в себя такие показатели, как верность, точность, полнота, F-мера и матрица ошибок (F-мера – это показатель, значение которого находится в диапазоне от 0 до 1 и который учитывает точность и полноту):

```
=======================Evaluation Metrics=======================
 # of classes:    10
 Accuracy:        0.2310
 Precision:       0.2207
 Recall:          0.2255
 F1 Score:        0.2133
Precision, recall & F1: macro-averaged (equally weighted avg. of 10 classes)
========================Confusion Matrix========================
 0  1  2  3  4  5  6  7  8  9
\\------------------------------
31  9  4 10  2  3  6  3 26  9 | 0 = airplane
 6 19  0  7  6  6  4  0 16 25 | 1 = automobile
18  8  6 14  8  6 15  4 12  9 | 2 = bird
11 14  1 28 14  5  8  6  3 13 | 3 = cat
 8  5  3 14 15  5 15  7  5 13 | 4 = deer
 9  5  5 21 18  8  8  1  3  8 | 5 = dog
 8  9  7 12 21  4 29  7  5 10 | 6 = frog
11 11  8 13  8  4  6 10 11 20 | 7 = horse
18  6  1  9  4  1  2  2 47 16 | 8 = ship
12 12  2  8  6  3  2  3 23 38 | 9 = truck
```

В матрице ошибок видно, что в случае с классом `airplane` в первом ряду ему была правильно назначена 31 выборка, но примерно такое же количество прогнозов (26) было присвоено неверному классу `ship` для изображения `airplane`. В идеале матрица ошибок должна содержать высокие значения на правой диагонали и низкие значения во всех остальных местах.

Если вы измените значение `numSamples` на 5000 и снова проведете обучение и оценку, то получите более качественные результаты:

```
=======================Evaluation Metrics=======================
 # of classes:    10
 Accuracy:        0.3100
 Precision:       0.3017
 Recall:          0.3061
 F1 Score:        0.3010
Precision, recall & F1: macro-averaged (equally weighted avg. of 10 classes)
========================Confusion Matrix========================
 0  1  2  3  4  5  6  7  8  9
\\------------------------------
38  2  6  3  5  1  1  9 25 13 | 0 = airplane
 4 34  3  2  4  4  6  4 14 14 | 1 = automobile
15  4 12  7 15  9 16  8 10  4 | 2 = bird
 7  4  4 26 16 11 15 13  1  6 | 3 = cat
 4  2 10  9 24  7 13  5  8  8 | 4 = deer
 7  5  5 19  9 14 11  7  3  6 | 5 = dog
 3  8 10  9 22  5 40 12  1  2 | 6 = frog
 4  8  6 13 12  2  9 29  2 17 | 7 = horse
17  5  2  8  4  2  0  4 51 13 | 8 = ship
 7 13  3  4  4  3  2 10 21 42 | 9 = truck
```

F-мера выросла на 9 % (0,30 против 0,21), что является большим шагом вперед, но получение хороших результатов только в 30 % случаев не подходит для реальных условий эксплуатации.

Возможно, вы помните, что для обучения нейронной сети используется алгоритм обратного распространения ошибки (в конечном итоге с вариациями, в зависимости от конкретной архитектуры, например обратное распространение во времени для рекуррентных нейронных сетей). Алгоритм обратного распространения направлен на уменьшение ошибки прогнозирования, совершаемой сетью, путем корректировки весов таким образом, чтобы общий коэффициент ошибок уменьшался. В какой-то момент алгоритм найдет набор весов (например, весов, прикрепленных к соединениям между нейронами в разных слоях) с наименьшей возможной ошибкой, но на это может уйти много времени, в зависимости от особенностей данных, используемых для обучения:

- *разнообразие в обучающих примерах* – часть текста написана на официальном языке, а другая – на сленге. Или некоторые изображения – это снимки, сделанные в дневное время, а другие – ночью;
- *шум в обучающих примерах* – в некоторых текстах содержатся опечатки или грамматические ошибки. Или некоторые изображения имеют низкое качество либо содержат водяные знаки или другие типы шумов, которые усложняют процесс обучения.

Способность обучения нейронной сети сходиться к правильному набору весов также во многом зависит от параметров настройки, таких как *скорость обучения*: я уже упоминал об этом, но стоит повторить, что это фундаментальный аспект. Довольно высокая скорость обучения приведет к сбою, а очень низкая скорость обучения приведет к тому, что обучение займет слишком много времени, чтобы сойтись к нужному набору весов.

На рис. 9.3 показаны потери той же нейронной сети, но с разными скоростями обучения. Отчетливо видно, что обе скорости обучения сходятся с течением времени к одному и тому же набору весов. Обучение начинается в момент времени t0; давайте рассмотрим, что произойдет, если вы прекратите обучение в моменты времени t1 или t2. Если вы прекратите обучение после небольшого количества итераций (до момента времени t1), то исключите высокую скорость обучения, потому что она будет увеличивать потери, а не уменьшать. Если вы прекратите обучение в момент времени t2, то вместо этого откажетесь от низкой скорости обучения, потому что она сохранит тот же результат, что и высокая скорость обучения, или начнет увеличиваться. Поэтому было бы неплохо придумать несколько возможных архитектур с разумными настройками параметров и провести несколько экспериментов.

В реализациях DL4J `Updater` можно установить скорость обучения для своей нейронной сети.

Листинг 9.4 ❖ Установка скорости обучения

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .updater(new Sgd(0.01)) ◄——— Устанавливает скорость обучения на 0,01
    ...
    .build();
```

Дополнительные веса

Использование дополнительных весов для изучения может привести к тому, что обучение займет больше времени и ресурсов; распространенная ошибка – добавлять слои или максимально увеличивать их размер. Но добавление слоев мо-

жет помочь, когда в сети недостаточно сил для обучения, чтобы соответствовать множеству различных примеров обучения, например когда число весов намного меньше, чем количество примеров, и нейронная сеть испытывает трудности с конвергенцией к хорошему набору весов (возможно, оценка не опускается ниже определенного значения).

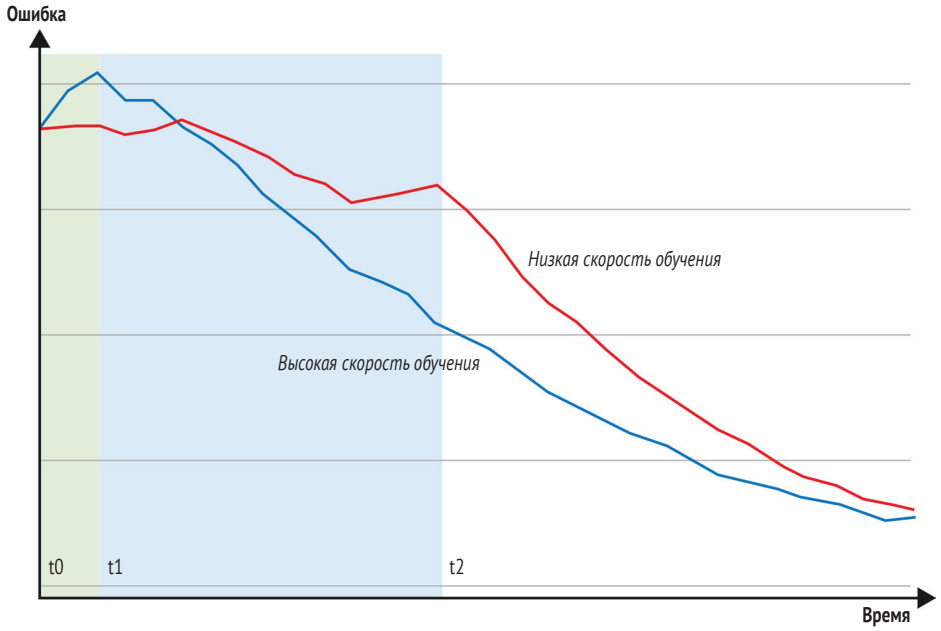


Рис. 9.3 ❖ График потерь той же нейронной сети, которая обучается с разными скоростями

Код, определенный в предыдущих разделах, обучил относительно легковесную сверточную сеть, используя 5000 примеров. Давайте посмотрим, что произойдет, если вы сделаете сверточные слои более глубокими (глубиной 96 и 256 соответственно). Время обучения для 5000 примеров увеличивается с 10 минут до 1 часа со следующей статистикой оценки:

```
========================Evaluation Metrics========================
 # of classes:    10
 Accuracy:        0.3011
 Precision:       0.3211
 Recall:          0.2843
 F1 Score:        0.3016
```

В этом случае не стоило добавлять сети больше мощности.

Работа с глубокими нейронными сетями в промышленной эксплуатации требует опыта, но это не волшебство. Количество изучаемых весов является важным фактором: количество точек данных в обучающем наборе всегда должно быть меньше количества весов. Возможные последствия несоблюдения этого правила – переобучение и трудности в схождении.

Давайте рассмотрим некоторые данные. У вас есть крошечные изображения размером 32×32 пикселей. Сверточные сети изучают признаки с течением времени с помощью сверточных слоев, выполняя субдискретизацию, используя слои подвыборки. Возможно, это помогло бы придать начальному сверточному слою еще несколько весов, но при этом у слоя подвыборки получилось бы значение шага 2 вместо 1. Вы ожидаете, что обучение сети приведет к лучшим результатам за меньшее время:

```
=======================Evaluation Metrics=======================
 # of classes:    10
 Accuracy:        0.3170
 Precision:       0.3069
 Recall:          0.3297
 F1 Score:        0.3316
```

Обучение заканчивается через 5 минут вместо 7 благодаря переходу на слой подвыборки, и качество результатов также улучшилось. Возможно, это не очень большое достижение, но это будет иметь реальное значение во время обучения по всему набору данных.

Обучение с большим количеством данных

До сих пор вы проводили эксперименты только с несколькими примерами из набора данных CIFAR. Чтобы лучше понять, насколько хорошо работает модель сверточной сети, нужно обучить ее, используя большее количество данных.

В CIFAR у вас есть свыше 50 000 изображений: вы должны разделить набор данных таким образом, чтобы большая его часть использовалась для обучения, но многие изображения были доступны для оценки.

Перед использованием полного набора данных важно отметить время, затраченное на обучение относительно доступного оборудования и требований производственного сценария. Первая итерация обучения для 10 эпох с 2000 изображений заняла 3 минуты на обычном ноутбуке; обучение 5000 изображений для 10 эпох заняло 7 минут. Это приемлемые показатели для экспериментов, где требуется быстрая обратная связь, но обучение с полным набором данных для нескольких эпох может занять часы – это время, которое было бы лучше использовать, если бы вы заранее знали, что изменить.

Теперь давайте запустим текущие настройки для 50 000 изображений для обучения и 10 000 для оценки. Вы ожидаете боле подходящих результатов оценки и более низкого балла в конце обучения:

```
=======================Evaluation Metrics=======================
 # of classes:    10
 Accuracy:        0.4263
 Precision:       0.4213
 Recall:          0.4263
 F1 Score:        0.4131
Precision, recall & F1: macro-averaged (equally weighted avg. of 10 classes)
========================Confusion Matrix========================
  0   1   2   3   4   5   6   7   8   9
\\----------------------------------------
459  60  39  40  14  24  41  49 191  83 | 0 = airplane
```

```
 29 592   3  30   3  12  47  34  50 200 | 1 = automobile
 92  50 123  81 165  89 229  97  46  28 | 2 = bird
 19  34  40 247  48 200 216 103  19  74 | 3 = cat
 44  21  58  83 284  60 263 128  33  26 | 4 = deer
 11  22  69 189  63 337 158 100  29  22 | 5 = dog
  3  26  20  90  66  32 661  52   9  41 | 6 = frog
 24  38  27  86  72  69 107 494  18  65 | 7 = horse
122  92  12  25   6  21  23  26 546 127 | 8 = ship
```

После обучения с использованием почти всего обучающего набора F-мера равна 0,41 спустя почти 3 часа (на обычном ноутбуке). Пока еще нельзя быть довольным верностью модели: она бы делала ошибки 59 % времени.

В этом случае полезно взглянуть на кривую потерь, показанную на рис. 9.4. Кривая уменьшается и могла продолжать делать это, если бы у вас было больше данных. К сожалению, в этом случае у вас нет такой возможности, только если вы не используете меньший набор тестов.

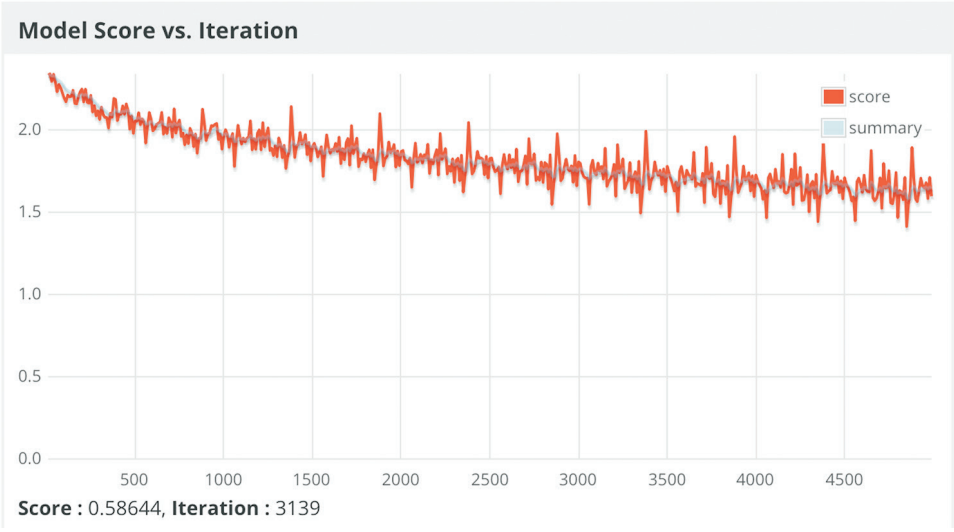


Рис. 9.4 ❖ Полный график потерь во время обучения сверточной сети

Регулируем размер пакета

Когда у вас есть такие кривые, можно посмотреть, используете ли вы неправильный размер для параметра `batch` (пакет). *Пакет* или *мини-пакет* – это набор обучающих примеров, которые собираются и передаются в нейронную сеть как единый пакетный вход. Например, вместо подачи по одному изображению за раз и, следовательно, за один входной объем (набор суммированных матриц) за один раз вы можете одновременно сжать несколько входных объемов. Обычно это имеет два последствия:

- обучение идет быстрее;
- вероятность переобучения меньше.

Если для параметра мини-пакета задано значение 1, вы увидите кривую, которая, особенно на первых итерациях, значительно увеличивается и уменьшается. С другой стороны, если у вас слишком большой мини-пакет, сеть может быть не в состоянии узнать о конкретных шаблонах и признаках, которые редко встречаются во входных данных.

Возможно, что плоская кривая потерь связана с размером пакета (в нашем случае 100), который слишком велик для этих данных. Чтобы понять, будет ли нечто подобное иметь значение, полезно провести быстрые тесты на небольших частях набора данных. Изменения настроек могут быть подтверждены позже с помощью обучения с использованием полного набора данных, если вы получите обнадеживающие результаты. Итак, давайте установим значение параметра `batch` на 48, выполним обучение на 5000 примерах и оценку для 1000 изображений. Вы ожидаете менее плавную кривую наряду с меньшими потерями и надеетесь на лучшие показатели верности:

```
========================Evaluation Metrics=======================
 # of classes:    10
 Accuracy:        0.3274
 Precision:       0.3403
 Recall:          0.3324
 F1 Score:        0.3218
```

Как видно из этих результатов и на рис. 9.5, уменьшение размера пакета помогло: потеря, близкая к минимальной, была достигнута гораздо быстрее, чем при размере пакета, установленном на 100. Но обучение заняло больше времени: 9 минут вместо предыдущих 7. Разницу в 2 минуты можно заметить в большем масштабе, но это приемлемо, если время обучения окупается со значительно лучшей F-меркой.

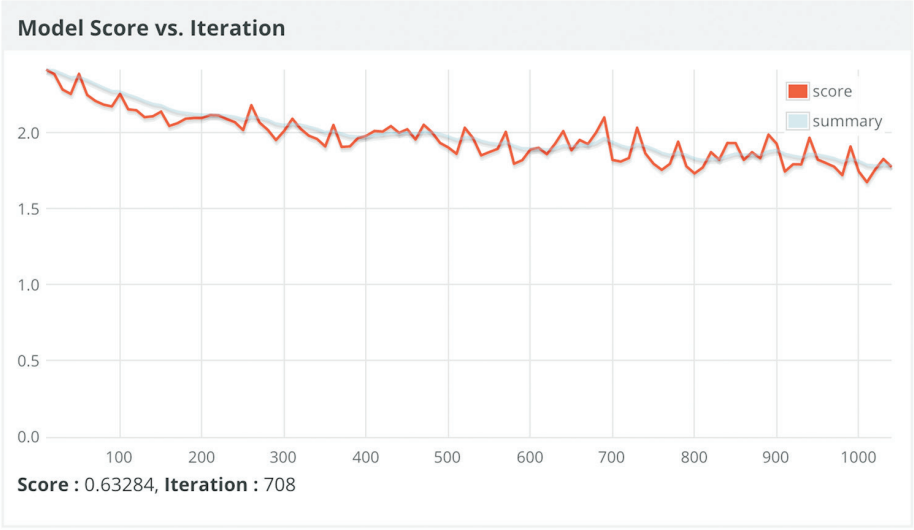


Рис. 9.5 ❖ Обучение с размером пакета 48

Показатель F-мерки улучшился с 0,30 до 0,32. Таким образом, уменьшение размера пакета кажется хорошей идеей, которую вам нужно доказать с помощью полного обучения. Мы не будем сравнивать F-мерку небольшого обучающего набора, подобного этому, с меркой, достигнутой при обучении с использованием свыше 50 000 изображений, потому что это будет несправедливо и может ввести нас в заблуждение (и расстроить наши усилия).

Но можно ли добиться большего успеха с еще меньшим размером пакета? Давайте установим размер пакета на 24 и посмотрим:

```
======================Evaluation Metrics======================
 # of classes:    10
 Accuracy:        0.3601
 Precision:       0.3512
 Recall:          0.3551
 F1 Score:        0.3340
```

Как видно по рис. 9.6, кривая намного острее, и потери близки к потерям, когда значение параметра `batch` было равно 48. F-мерка выше (0,33), но обучение заняло 13 минут вместо 9.

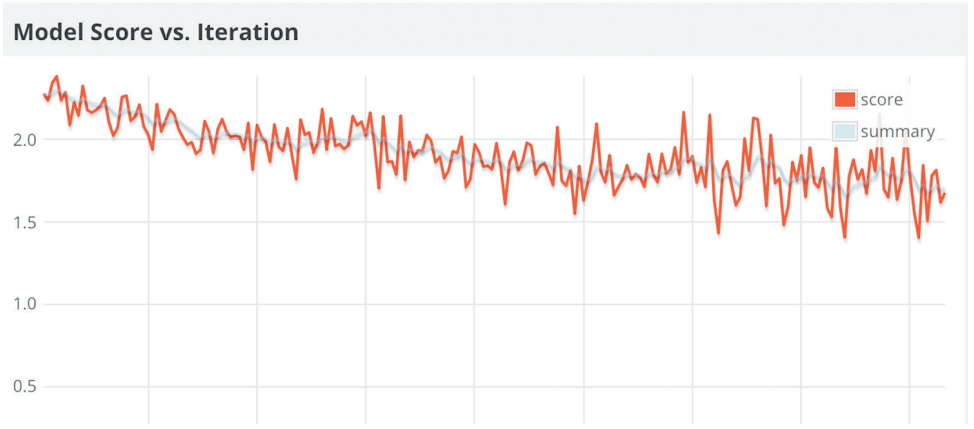


Рис. 9.6 ❖ Обучение с размером пакета 24

Оценка и итерация

На этом этапе вы должны принять решение: можете ли вы позволить себе более дорогостоящее обучение с точки зрения времени и вычислительных ресурсов (что может означать больше денег – например, если вы проводите обучение при реальной эксплуатации через облачные сервисы), чтобы получить более качественные показатели? Хорошей практикой является сохранение различных моделей, которые вы генерируете вместе с метриками оценки и временем обучения, чтобы иметь возможность обратиться к ним на более позднем этапе, когда вам нужно принять решение.

При меньшем размере пакета нейронная сеть должна быть способна лучше обрабатывать более разнообразные входные данные, но кривая становится более

четкой. Обучение последней модели с 50 000 примеров дало следующие результаты оценки после 5 часов обучения на ноутбуке:

```
=======================Evaluation Metrics=======================
 # of classes:    10
 Accuracy:        0.5301
 Precision:       0.5213
 Recall:          0.5095
 F1 Score:        0.5153
```

F-мерка с цифрой 0,41 улучшилась на 10 %, чтобы достичь неплохого показателя в 0,51. Но это все равно не то, что вы будете отправлять конечным пользователям. С таким числом, если пользователи будут искать изображения оленей, они могут получить только пять оленей – на оставшихся изображениях будут изображены кошки, собаки или даже грузовики и корабли!

Вы пытались использовать более глубокие сверточные слои, но это не помогло. Вы убедились, что верность увеличивается с количеством используемых данных. Размер пакета оказался важным параметром даже на этапе создания прототипа, чтобы добиться лучших результатов, но изменения в размере пакета влияют на время обучения.

Однако есть еще ряд факторов, которые необходимо учитывать:

- дополнительное количество эпох при обучении;
- проверьте инициализацию весов и смещений;
- посмотрите на варианты регуляризации;
- измените способ обновления весов нейронной сети во время обратного распространения (алгоритм обновления);
- определите, поможет ли добавление слоев в этом случае.

Давайте рассмотрим все эти варианты.

Эпохи

В настоящее время в примере используется 10 эпох, поэтому нейронная сеть видит одни и те же входные пакеты 10 раз. Обоснование заключается в том, что сеть должна иметь возможность получать правильные веса для этих данных с большей вероятностью, если она «видит» их несколько раз. Низкие числа, такие как 5, 10 и 30, распространены на этапе разработки, когда сеть проектируется, но можно изменить это значение при обучении вашей окончательной модели. Если вы увеличиваете количество эпох, но не видите значительного улучшения, вероятно, это все, на что способна сеть при текущей настройке этих данных; в этом случае нужно изменить что-то еще.

Изменение количества эпох с 10 на 20 в этом случае дает следующие результаты:

```
=======================Evaluation Metrics=======================
 # of classes:    10
 Accuracy:        0.3700
 Precision:       0.3710
 Recall:          0.3646
 F1 Score:        0.3565
```

Обучение заняло 28 минут; *см.* рис. 9.7.

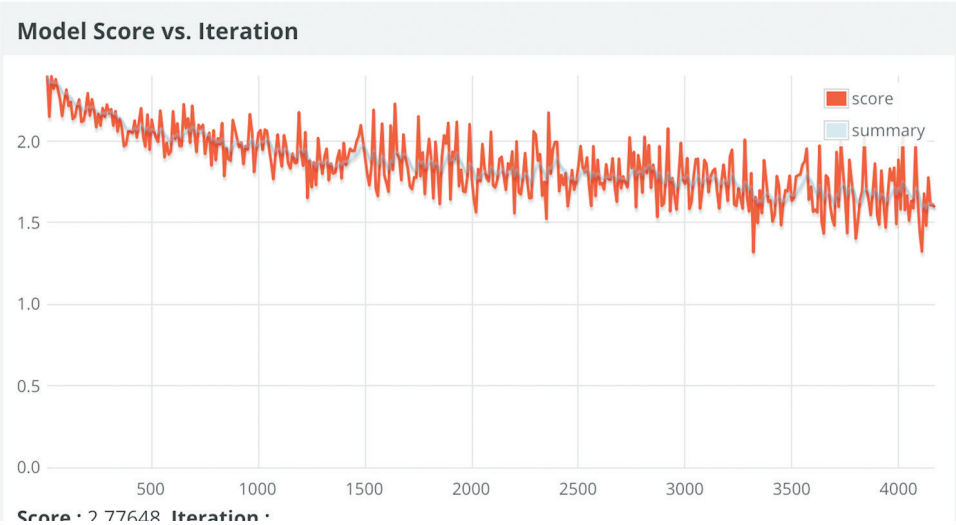


Рис. 9.7 ❖ Кривая потерь для 20 эпох

Инициализация весов

Представьте себе нейронную сеть до того, как она получит какие-либо входные данные. Все нейроны имеют функции активации и связи. Когда сеть начинает получать входные данные и распространяет ошибку вывода обратно, она начинает изменять веса, присоединенные к каждому соединению.

Удивительно эффективным изменением, которое вы можете сделать в своей нейронной сети, является способ инициализации этих весов. Многочисленные исследования показали, что инициализация весов оказывает существенное влияние на эффективность обучения[1].

Простые вещи, которые можно сделать для инициализации весов, – установить все их значения на ноль или случайные числа. Несколько глав назад вы видели, как алгоритм обучения (обратное распространение ошибки) приводит к изменению весов сети: можно визуально воспринимать это как перемещение точки на поверхности ошибки (см. рис. 9.8). Точка на такой поверхности представляет собой набор весов, а точка минимальной высоты – это точка, где веса заставляют сеть совершать наименьшую возможную ошибку.

Самая высокая точка на изображении обозначает набор весов с высокой ошибкой, точка в середине – набор весов со средней ошибкой, а самая низкая – это точка с наименьшей возможной ошибкой. Надеемся, что алгоритм обратного распространения заставит веса сети переместиться из исходного положения в точку, отмеченную внизу. Теперь поговорим об инициализации весов: она будет от-

[1] *См.*: *Ксавье Глоро и Джошуа Бенджио*. Understanding the Difficulty of Training Deep Feedforward Neural Networks // Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS). Chia Laguna Resort, Sardinia, Italy, 2010 (**http://mng.bz/vNZM**); *Кайминг Хэ и др.* Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (**https://arxiv.org/abs/1502.01852**).

вечать за настройку начальной точки алгоритма при поиске наилучшего набора весов. При инициализации весов, равной 0, веса сети могут находиться в белой точке в центре: это не плохо и не хорошо. При случайной инициализации вам может повезти, и вы поместите веса вблизи нижней точки (но это маловероятно) или установите веса где-то далеко оттуда, например в точке, отмеченной сверху. Исходная позиция влияет на способность обратного распространения когда-либо достичь нижней точки или может, по крайней мере, сделать процесс более продолжительным и более сложным. Таким образом, хорошая инициализация весов нейронной сети имеет решающее значение для успешного обучения.

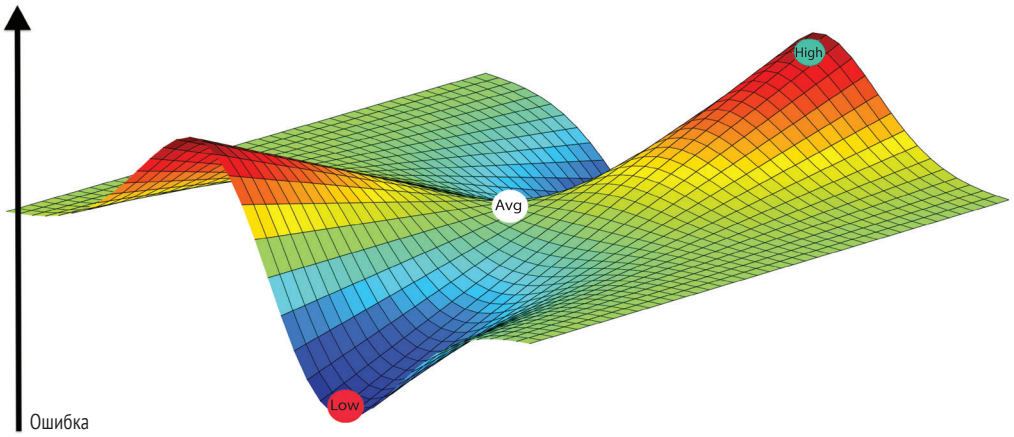


Рис. 9.8 ❖ Поверхность ошибки с некоторыми интересными точками

Подходящая, обычно используемая инициализация весов носит название *инициализация Ксавье*. По сути, она инициализирует веса нейронной сети, рисуя их из распределения с нулевым средним и определенной дисперсией для каждого нейрона. Начальный вес зависит от количества нейронов с исходящими соединениями к этому конкретному нейрону. Можно установить это в DL4J в определенном слое с помощью такого кода:

```
.layer(2, new ConvolutionLayer.Builder(new int[]{5,5}, new int[] {1,1}, new
     int[] {0,0})
  .convolutionMode(ConvolutionMode.Same)
  .nOut(10)
  .weightInit(WeightInit.XAVIER_UNIFORM)
  .activation(Activation.RELU)
```

Инициализирует веса данного слоя, используя распределение Ксавье

Регуляризация

Ранее, когда количество входных данных в одном пакете было уменьшено, мы заметили, что кривая потерь становится менее плавной. Это связано с тем, что при меньшем количестве пакетов алгоритм обучения более склонен к переобучению (*см.* рис. 9.9).

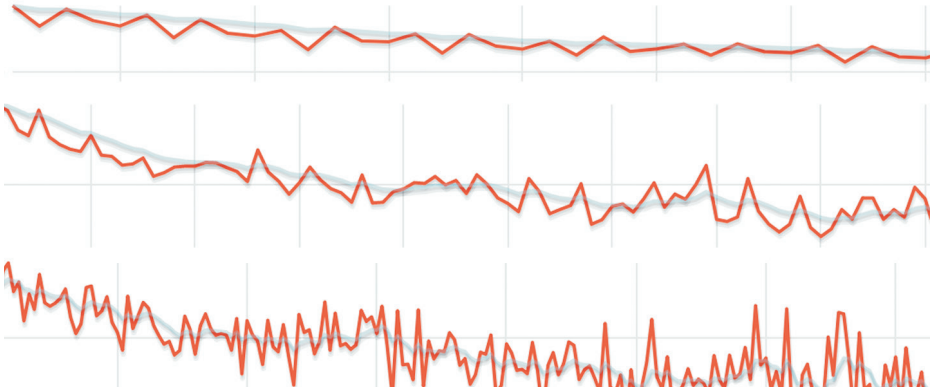


Рис. 9.9 ❖ Кривая потерь заостряется при более мелких размерах пакета

Часто полезно вводить методы регуляризации в свой алгоритм обучения нейронной сети. Это помогает благодаря небольшому размеру пакета, но в целом это хорошая практика. Количество используемых регуляризаций зависит от варианта использования:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .gradientNormalization(GradientNormalization.RenormalizeL2PerLayer)
    .l1(1.0e-4d).l2(5.0e-4d)
```

С учетом регуляризации и инициализации весов давайте проведем еще один этап обучения для 10 эпох на 5000 изображений. Вот окончательные результаты:

```
========================Evaluation Metrics========================
 # of classes:    10
 Accuracy:        0.4454
 Precision:       0.4602
 Recall:          0.4417
 F1 Score:        0.4438
```

Обучение заняло 16 минут, но, как видно по рис. 9.10, потери уменьшаются намного быстрее и достигают более низкого значения, чем при предыдущих настройках. Как и ожидалось, значение F-мерки высокое при относительно небольшом количестве обучающих примеров.

Заметив улучшения при использовании большего числа эпох, давайте увеличим его до 20, как делали это ранее:

```
========================Evaluation Metrics========================
 # of classes:    10
 Accuracy:        0.4435
 Precision:       0.4624
 Recall:          0.4395
 F1 Score:        0.4411
```

Хотя время обучения увеличивается до 19 минут, кривая выглядит более или менее похожей, и, что удивительно, F-мерка остается неизменной (*см.* рис. 9.11). Для этого есть несколько возможных причин: во-первых, вам может потребоваться больше данных.

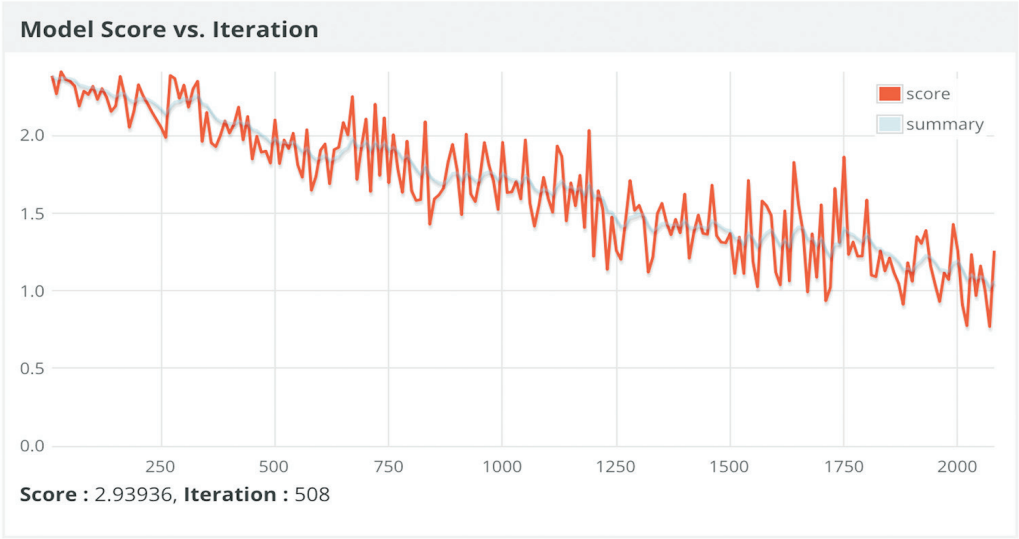


Рис. 9.10 ❖ Оптимальная настройка

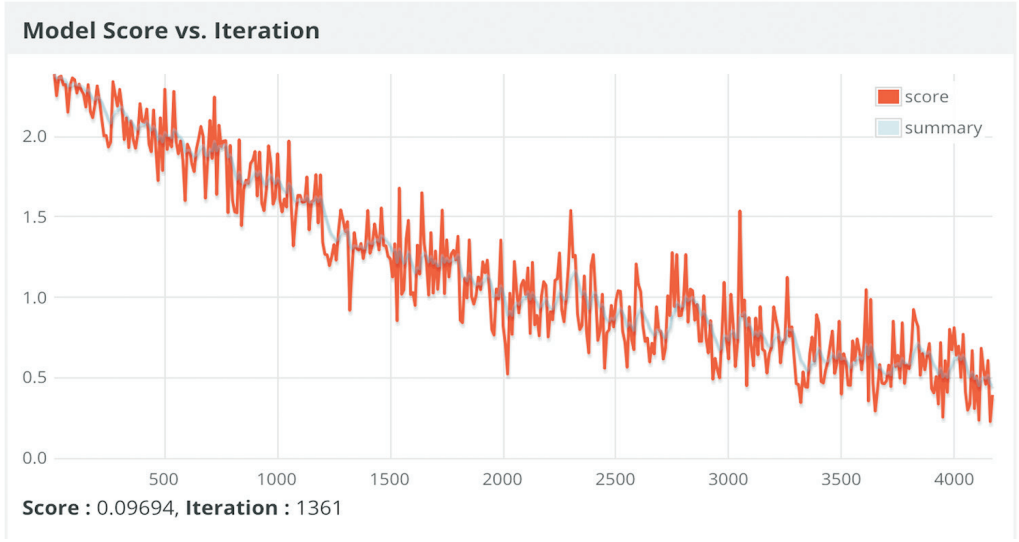


Рис. 9.11 ❖ Оптимальная настройка для 20 эпох

Давайте оценим верность последних настроек, используя весь набор данных из 50 000 изображений (*см.* рис. 9.12).

```
=======================Evaluation Metrics=======================
 # of classes:    10
 Accuracy:        0.5998
 Precision:       0.6213
 Recall:          0.5998
 F1 Score:        0.5933
```

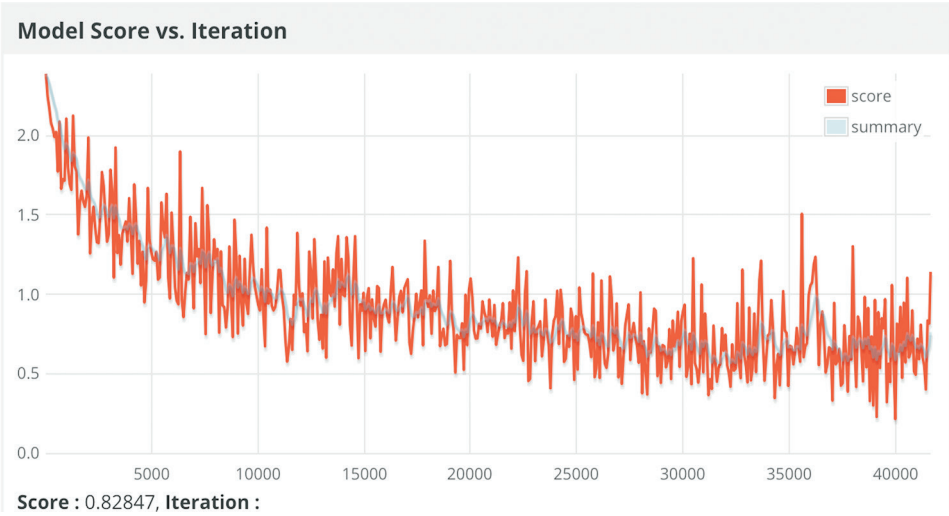


Рис. 9.12 ❖ Кривая потерь во время обучения для всего набора данных

Достичь хороших показателей не всегда легко, и может потребоваться несколько итераций только что описанного процесса. Глядя на недавние исследования, всегда полезно выяснить, существуют ли более подходящие решения для различных аспектов нейронных сетей. В некоторой степени настройка нейронной сети сродни искусству; опыт помогает, но знакомство с математикой и динамикой обучения является ключом к выработке эффективных моделей и настроек.

9.2. Индексы и нейроны работают вместе

Мы только что прошли сквозной процесс настройки и настройки глубокой нейронной сети для достижения наилучших результатов с точки зрения верности. Мы также кратко отметили время, необходимое для обучения всей сети. Используя такой набор, можно решить только половину проблемы. Цель состоит в том, чтобы применять модели глубокого обучения в контексте поиска для обеспечения конечным пользователям более значимых результатов поиска. Теперь вопрос заключается в том, как использовать и обновлять эти модели вместе с поисковыми системами.

Предположим на мгновение, что у вас есть предварительно обученная модель, которая идеально подходит для данных, подлежащих индексации. Вы индексируете текстовые документы и хотите использовать, например, предварительно обученную модель seq2seq для извлечения векторов мысли, которые будут использоваться поисковой системой в функции ранжирования. Простым решением будет создание конвейера индексации документов, в котором текст документа сначала отправляется в модель seq2seq, а затем соответствующий вектор мысли извлекается и индексируется вместе с текстом документа в поисковую систему. На рис. 9.13 видно, что действия и обязанности нейронной сети и поисковой системы сильно чередуются.

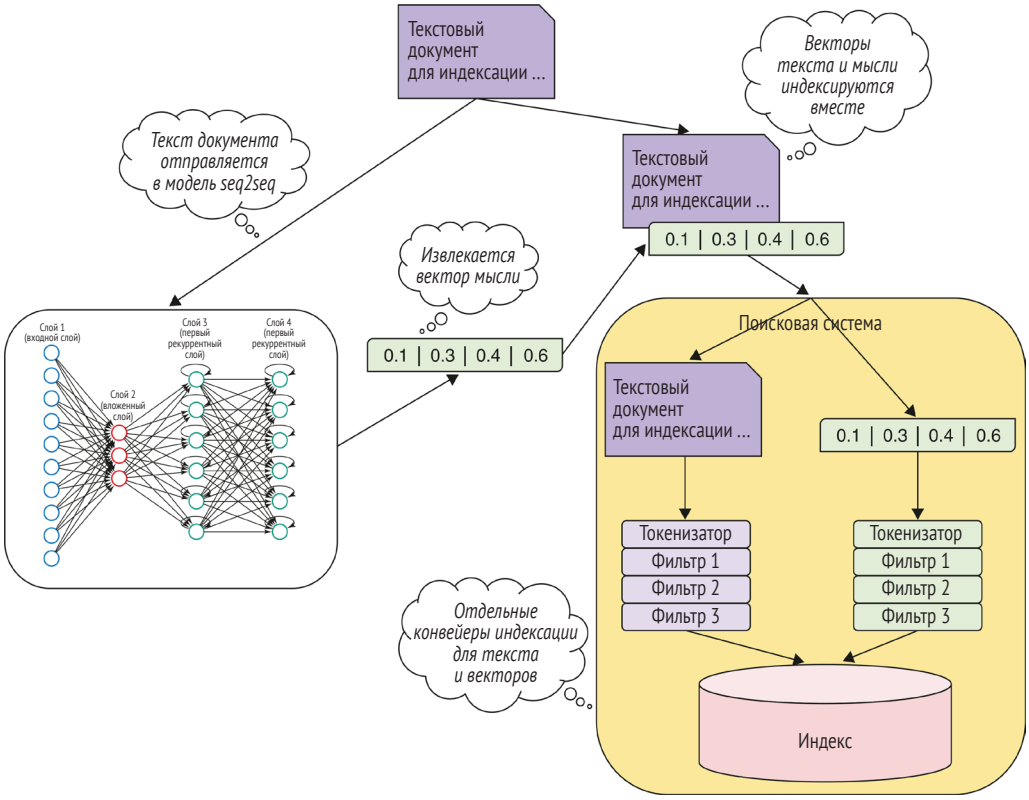


Рис. 9.13 ❖ Взаимодействия нейронной сети с поисковой системой во время индексации

Во время поиска модель seq2seq снова используется для извлечения векторов мысли из запроса (*см.* рис. 9.14). Затем функция ранжирования выполняет ранжирование с использованием векторов мысли запросов и документов (предварительно сохраненных в индексе).

Глядя на эти графики, можно подумать, что все кажется разумным. Но нейронная сеть может вводить издержки как для индексации, так и для поиска:

- *время прогнозирования нейронной сети* – сколько времени уходит у нейронной сети, чтобы извлечь векторы мысли для документов во время индексации? Сколько времени уходит у нейронной сети, чтобы извлечь векторы мысли для запросов во время поиска?
- *размер индекса поисковой системы* – сколько места занимают сгенерированные векторные представления в дополнение к пространству для хранения данных, используемому текстовыми документами?

В целом наиболее важным аспектом для производительности является фаза запроса/поиска. Нельзя ожидать, что пользователи будут ждать несколько секунд только потому, что ваша функция ранжирования возвращает более подходящие результаты. В большинстве случаев пользователи никогда не узнают, что скрывается за полем поиска, – они просто ожидают, что он будет быстрым и надежным и даст хорошие результаты.

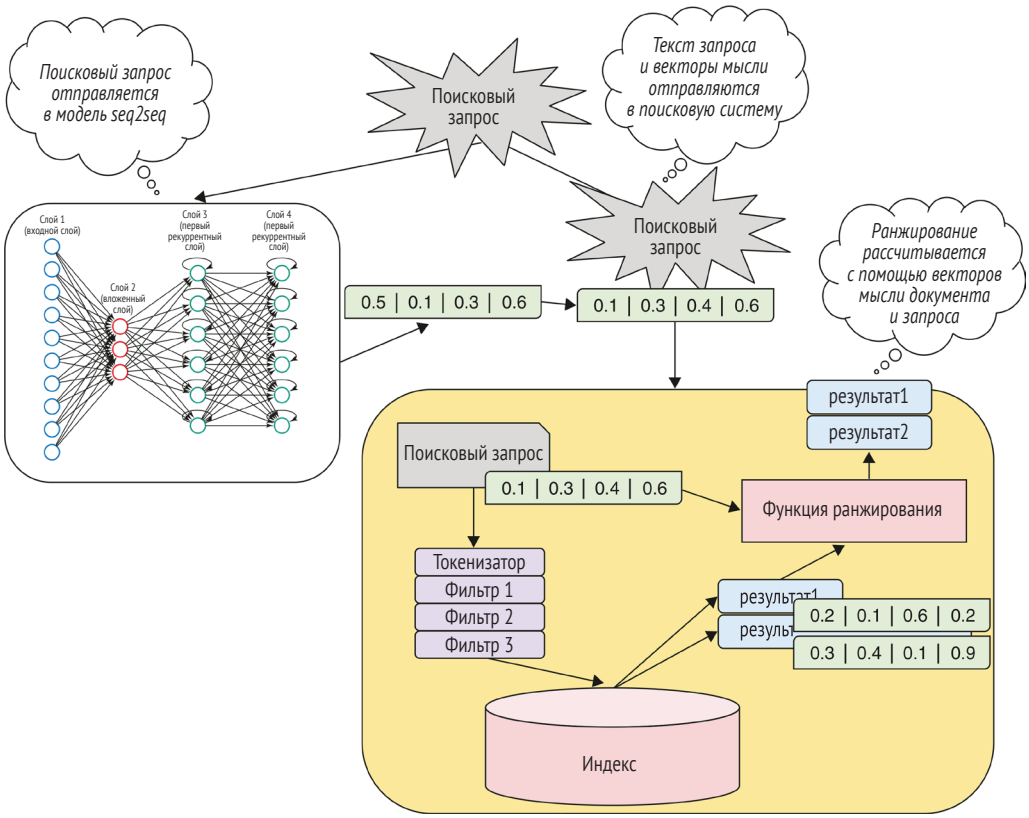


Рис. 9.14 ❖ Взаимодействия нейронной сети с поисковой системой во время поиска

В предыдущем разделе мы рассматривали верность результатов, обращая внимание на время обучения. Вам также необходимо отслеживать время, затрачиваемое сетью, чтобы вычислить полную прямую передачу от входа к слою, с которого вы получаете сетевой выход.

В случае сети кодер–декодер прямой передаче на стороне кодера сети необходимо только извлечь векторы мысли. Сторона декодера используется лишь в том случае, если вы также хотите использовать входной текст для обучения с применением целевого выхода (если он у вас есть).

Издержки при индексации также следует принять во внимание. В «статическом» сценарии, когда вы принимаете набор документов, даже если он очень большой, это может быть не важно, поскольку вы можете принять совокупные издержки в один или два часа, если это происходит только один раз. Но повторная индексация или одновременная индексация большого объема может быть проблематичной. *Повторная индексация* означает новую индексацию всего корпуса документов в поисковой системе с нуля. Обычно это делается из-за изменения в конфигурации конвейеров анализа текста или по причине добавления обработчика документов для извлечения дополнительных метаданных.

Например, возьмем простую поисковую систему на базе Lucene без возможности расширения запросов. Для использования модели word2vec для расширения синонимов во время индексации нужно взять все существующие документы и повторно проиндексировать их, чтобы результирующий инвертированный индекс также содержал слова/синонимы, извлеченные word2vec. Чем больше индекс, тем больше будет влияние реиндексации.

Еще один аспект-параллелизм: может ли нейронная сеть работать с параллельными входными данными? Это деталь реализации, которая может зависеть от конкретной технологии, используемой для реализации вашей модели, но ее необходимо учитывать как во время индексации (несколько параллельных процессов индексации), так и во время поиска (несколько пользователей одновременно выполняет поиск).

Векторные представления и плотные векторы в целом могут иметь большое количество размерностей. Их эффективное хранение – открытая проблема. В реальном мире выбор может быть ограничен возможностями используемой технологии поисковой системы. В Lucene, например, плотные векторы можно проиндексировать как:

- *двоичные файлы* – каждый вектор хранится в виде непроверенного двоичного файла, и вся обработка векторных представлений выполняется при извлечении двоичного файла;
- *n-мерные точки* – каждый вектор хранится в виде точки со множеством размерностей (по одному на каждую векторную размерность). Можно выполнить базовые геометрические запросы и запросы ближайшего соседа. На данный момент Lucene может индексировать до 8-мерных векторов, поэтому вам придется уменьшить векторы более высокой размерности (например, 100-мерные векторы слов) до не более чем 8-мерных векторов, чтобы проиндексировать их в Lucene (как мы это дели в главе 8 для векторов признаков изображения, используя метод главных компонент);
- *текст* – поначалу это может звучать странно, но при соответствующем дизайне векторы можно индексировать и искать как текстовые блоки[1].

Другие библиотеки, например Vespa (**http://vespa.ai**), и такие поисковые платформы, как Apache Solr (**https://lucene.apache.org/solr**) и Elasticsearch (**www.elastic.co/products/asticsearch**), могут предложить дополнительные или иные варианты.

9.3. Работа с потоками данных

Во всех примерах, приведенных в этой книге, используются статические наборы данных. Статические наборы данных отлично подходят для иллюстративных целей, поскольку благодаря им проще сосредоточиться на конкретном наборе данных. Кроме того, при создании поисковой системы обычно начинают с набора документов (текст и/или изображения), которые вы хотите проиндексировать. Но когда поисковая система вводится в эксплуатацию и ее начинают использовать, вероятно, потребуются новые документы.

---

[1] *См.*: *Ян Ригл и др*. Semantic Vector Encoding and Similarity Search Using Fulltext Search Engines (**https://arxiv.org/pdf/1706.00957.pdf**).

Рассмотрим приложение, позволяющее пользователям искать популярные посты в социальных сетях на разные темы. Можно начать с набора загруженных или купленных постов, но поскольку основное внимание уделяется популярным постам, вам нужно продолжать принимать данные по мере того, как тенденции со временем меняются. Подобное приложение может работать с новостями, а не с сообщениями в социальных сетях. Можно скачать корпус новостей, например NYT Annotated Corpus (**https://catalog.ldc.upenn.edu/LDC2008T19**), но каждый день приложение должно загружать много новых статей, чтобы пользователи могли их искать.

В наши дни принято использовать *потоковую архитектуру* для адресации входящих потоков данных. В потоковой архитектуре данные непрерывно поступают из одного или нескольких источников и преобразуются функциями, расположенными в конвейере. Данные могут быть преобразованы, агрегированы или отброшены в любое время. Наконец, они достигают *приемника*: финальной стадии конвейера, такой как постоянная система, например база данных или индекс поисковой системы.

В предыдущем примере потоковая архитектура может непрерывно поглощать посты из социальных сетей и индексировать их в поисковую систему. Еще одно приложение, работающее с индексированными данными, может считывать индекс и предоставлять функции поиска конечным пользователям. Но вы работаете с нейронными сетями, поэтому вам нужно обучить модели нейронных сетей, которые вы хотите использовать.

В качестве примера давайте создадим приложение, чтобы непрерывно находить наиболее релевантные посты для каждого набора предопределенных тем; *см.* рис. 9.15. Для этого вы будете использовать потоковую архитектуру для постоянного выполнения следующих действий:

- поглощать посты из социальных сетей (в данном случае Twitter);
- обучать различные модели нейронных сетей для извлечения векторных представлений документов;
- индексировать текст и векторные представления в Lucene;
- писать наиболее релевантные посты для каждой модели ранжирования и для каждой темы.

Наконец, вы быстро оцените, какая из различных моделей ранжирования (нейронная или нет) является более перспективной. Такое приложение можно использовать, например, на этапе предэксплуатации, чтобы помочь выбрать модель ранжирования, которая лучше всего подходит для производственного приложения.

Чтобы настроить потоковую архитектуру, давайте воспользуемся Apache Flink (**http://flink.apache.org**). Это распределенная платформа обработки данных. Потоковый конвейер Flink сделает следующее:

- будет передавать сообщения из социальной сети Twitter (**http://twitter.com**), которые содержат определенные ключевые слова;
- извлечет текст каждого твита, язык, пользователя и т. д.;
- извлечет векторные представления документов, используя две отдельные модели: векторы абзаца и усредненные векторные представления слов word2vec;

- проиндексирует каждый твит с его текстом, языком, пользователем и векторными представлениями документов в Lucene;
- выполнит предварительно определенные запросы ко всем индексированным данным, используя разные модели ранжирования (классические и нейронные);
- запишет результат в CSV-файл, который можно проанализировать на более позднем этапе, чтобы оценить качество результатов поиска.

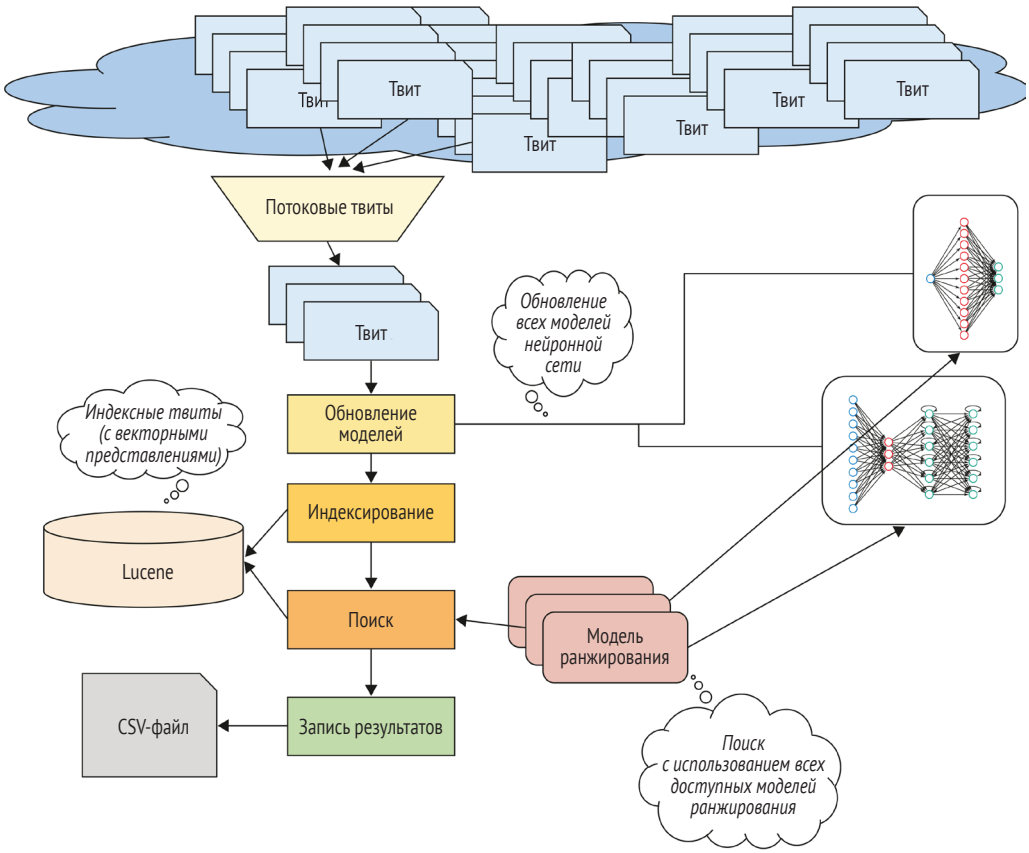


Рис. 9.15 ❖ Потоковое приложение для непрерывного обучения, индексации и поиска постов в социальных сетях

Выходной файл расскажет вам, как различные модели ранжирования отреагировали на изменение данных относительно набора фиксированных запросов для определенных тем. Это обеспечит ценную информацию о том, насколько хорошо модели ранжирования адаптируются к новым постам. Если модель ранжирования продолжает давать одни и те же результаты, несмотря на изменение данных, это, вероятно, не лучший вариант для приложения, которое стремится собирать актуальные данные.

Сперва давайте определим поток данных, поступающих из Twitter.

Листинг 9.5 ❖ Определение потока данных Twitter с помощью Flink

```
final StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();

Properties props = new Properties();
props.load(StreamingTweetIngestAndLearnApp.class.getResourceAsStream(
    "/twitter.properties"));
TwitterSource twitterSource = new TwitterSource(props);
String[] tags = {"neural search", "natural language processing", "lucene",
    "deep learning", "word embeddings", "manning"};
twitterSource.setCustomEndpointInitializer(new FilterEndpoint(tags));

DataStream<Tweet> twitterStream = env.addSource(twitterSource)
    .flatMap(new TweetJsonConverter());
```

Определяет среду выполнения Flink

Загружает набор удостоверений защиты для доступа к Twitter

Создает новый источник Flink для данных Twitter

Определяет темы, которые будут использоваться для получения постов из Twitter (будут приниматься только твиты, содержащие эти ключевые слова)

Создает поток для данных Twitter

Начинает с преобразования необработанного текста в формат JSON для твитов

Добавляет фильтр по темам в источник Twitter

Этот листинг выполняет необходимую настройку для начала приема твитов, которые содержат ключевые слова / темы «нейронный поиск», «обработка естественного языка», «lucene», «глубокое обучение», «векторные представления слов» и «manning».

Затем вы определите ряд функций для работы с твитами. Мы также сосредоточимся на деталях реализации, касающихся производительности. Например, имеет ли смысл выполнять предопределенные запросы каждый раз, когда приходит новый твит? Возможно, лучше сделать это, когда у вас есть больше данных (например, 20 твитов), которые могут повлиять на ранжирование. По этой причине вы определите функцию *окна количества*, которая будет передавать данные следующей функции только после получения 20 твитов. Кроме того, обновление модели нейронной сети всего одним образцом – как правило, плохая идея: использование более крупного обучающего пакета менее подвержено колебаниям ошибок при обучении (что приводит к более плавной кривой обучения).

Листинг 9.6 ❖ Управление потоковыми данными

```
Path outputPath = new Path("/path/to/data.csv");
OutputFormat<Tuple2<String, String>> format = new
    CsvOutputFormat<>(outputPath);

DataStreamSink<Tuple2<String, String>> tweetSearchStream =
    twitterStream
        .countWindowAll(batchSize)
        .apply(new ModelAndIndexUpdateFunction())
        .map(new MultiRetrieverFunction())
        .map(new ResultTransformer()).countWindowAll(1)
        .apply(new TupleEvictorFunction())
        .writeUsingOutputFormat(format);
env.execute();
```

Выходной CSV-файл

Определяет окно количества для потоковых данных

Обновляет модели, извлекает признаки и обновляет индекс

Выполняет предопределенные запросы

Преобразует вывод способом, подходящим для составления CSV-файла

`ModelAndIndexUpdateFunction` отвечает за обновление моделей нейронной сети и индексацию документов в Lucene. Теоретически можно разделить ее на множество крошечных функций; но для удобства чтения процессы потребления и поиска легче разделить только на две функции. Теоретически можно использовать столько нейронных моделей ранжирования, сколько захотите; в этом примере используются те, что были определены в главах 5 и 6, применяя word2vec и векторы абзаца соответственно, для влияния на ранжирование.

После приема каждого твита оба вектора абзаца и модели word2vec используются для генерации двух отдельных векторных представлений. Векторы индексируются вместе с текстом твита и используются классами `ParagraphVectorsSdentifity` и `WordEmbeddingsSimilarity` во время поиска.

Листинг 9.7 ❖ Функция для обновления модели и индексации

```
public class ModelAndIndexUpdateFunction implements AllWindowFunction<Tweet,
    Long, GlobalWindow> {

  @Override
  public void apply(GlobalWindow globalWindow, Iterable<Tweet> iterable,
    Collector<Long> collector) throws Exception {
    ParagraphVectors paragraphVectors = Utils.fetchVectors();
    CustomWriter writer = new CustomWriter();
    for (Tweet tweet : iterable) {          <-- Перебирает текущий пакет твитов
      Document document = new Document();
      document.add(new TextField("text", tweet.getText(),
        Field.Store.YES));                  <-- Создает документ Lucene для текста текущего твита

      INDArray paragraphVector =
        paragraphVectors.inferVector(tweet.getText());  <-- Выводит вектор абзаца и обновляет модель
      document.add(new BinaryDocValuesField(
        "pv", new BytesRef(paragraphVector.data().asBytes())));  <-- Индексирует вектор абзаца

      INDArray averageWordVectors =
        averageWordVectors(word2Vec.getTokenizerFactory()
        .create(tweet.getText()).getTokens(), word2Vec.lookupTable());
      document.add(new BinaryDocValuesField(
        "wv", new BytesRef(averageWordVectors.data().asBytes())));
      ...

      writer.addDocument(document);         <-- Индексирует документ
    }
    long commit = writer.commit();          <-- Фиксирует все твиты в Lucene

    writer.close();                         <-- Закрывает IndexWriter (освобождая ресурсы)

    collector.collect(commit);
  }
}
```

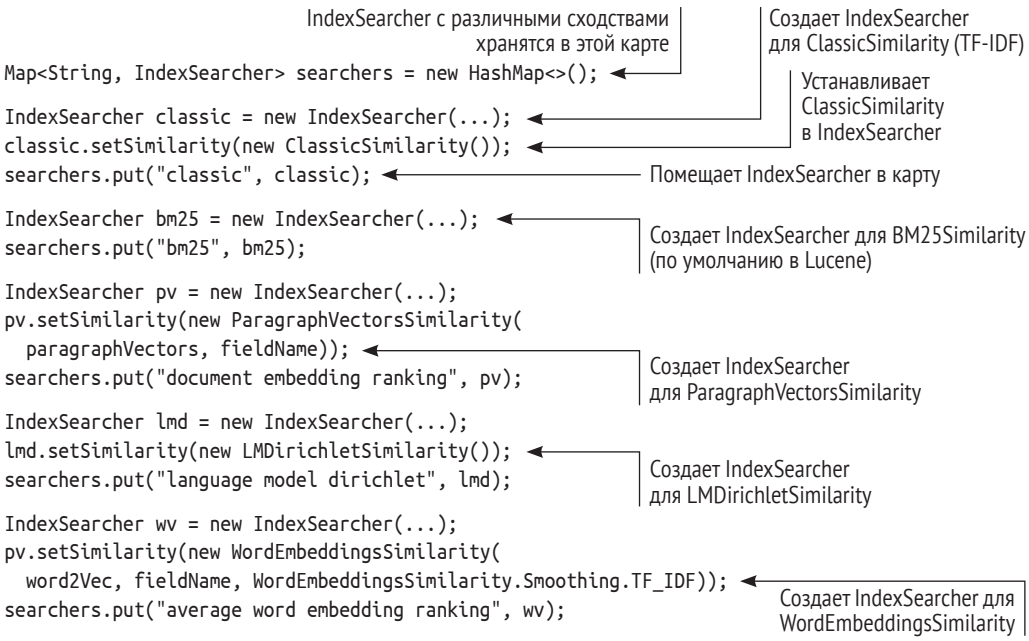
Выводит вектор документа из word2vec и обновляет модель

Индексирует усредненный вектор слов

Передает идентификатор фиксации в следующую функцию (можно использовать, чтобы отслеживать изменения индекса с течением времени)

`MultiRetrieverFunction` содержит некоторый базовый код поиска Lucene для выполнения фиксированных запросов (таких как «deep learning search») для всего индекса с различными функциями ранжирования. Во-первых, он устанавливает `IndexSearcher`, каждый из которых использует разное сходство.

Листинг 9.8 ❖ Установка IndexSearcher




```
Map<String, IndexSearcher> searchers = new HashMap<>();

IndexSearcher classic = new IndexSearcher(...);
classic.setSimilarity(new ClassicSimilarity());
searchers.put("classic", classic);

IndexSearcher bm25 = new IndexSearcher(...);
searchers.put("bm25", bm25);

IndexSearcher pv = new IndexSearcher(...);
pv.setSimilarity(new ParagraphVectorsSimilarity(
  paragraphVectors, fieldName));
searchers.put("document embedding ranking", pv);

IndexSearcher lmd = new IndexSearcher(...);
lmd.setSimilarity(new LMDirichletSimilarity());
searchers.put("language model dirichlet", lmd);

IndexSearcher wv = new IndexSearcher(...);
pv.setSimilarity(new WordEmbeddingsSimilarity(
  word2Vec, fieldName, WordEmbeddingsSimilarity.Smoothing.TF_IDF));
searchers.put("average word embedding ranking", wv);
```

Вы можете добавить столько моделей ранжирования, сколько захотите. Затем вы перебираете доступные `IndexSearcher` и выполняете один и тот же запрос для каждого из них. Наконец, результаты записываются в CSV-файл.

Вывод, агрегированный в CSV-файле во время выполнения `MultiRetrieverFunction`, содержит строку для каждой модели ранжирования. В каждой строке сначала указывается название модели (`classic`, `bm25`, `average wv ranking`, `paragraph vectors ranking` и т. д.), затем идет запятая и текст первого результата поиска, возвращаемого этой моделью ранжирования. Со временем вы получите огромный CSV-файл, содержащий результаты одного и тот же запроса для всех различных моделей ранжирования.

Давайте посмотрим на результаты двух последовательных выполнений (помечены вручную с помощью `<iteration-1>` и `<iteration-2>` для лучшей читаемости):

```
...
...<iteration-1>
...
classic,Amazing what neural networks can do.
// Computational Protein Design with Deep Learning Neural Networks
language model dirichlet,Amazing what neural networks can do.
// Computational Protein Design with Deep Learning Neural Networks
bm25,Amazing what neural networks can do.
// Computational Protein Design with Deep Learning Neural Networks
average wv ranking,Amazing what neural networks can do.
// Computational Protein Design with Deep Learning Neural Networks
paragraph vectors ranking,Amazing what neural networks can do.
// Computational Protein Design with Deep Learning Neural Networks
...
```

```
...<iteration-2>
...
classic,Amazing what neural networks can do.
// Computational Protein Design with Deep Learning Neural Networks
language model dirichlet,Amazing what neural networks can do.
// Computational Protein Design with Deep Learning Neural Networks
bm25,Amazing what neural networks can do.
// Computational Protein Design with Deep Learning Neural Networks
average wv ranking,The Connection Between Text Mining and Deep Learning
paragraph vectors ranking,All-optical machine learning using diffractive
deep neural networks:
...
```

Обратите внимание, что ненейронные модели ранжирования не меняли свой главный результат поиска, тогда как модели, основанные на векторных представлениях, сразу адаптировались к новым данным: это та возможность, для которой могут быть полезны модели нейронного ранжирования. Потоковая архитектура может не отставать от больших объемов данных, которые будут проиндексированы в поисковой системе, оценивать лучшие модели и тщательно определять, как нейронные сети и поисковые системы могут наилучшим образом работать вместе.

Резюме

- Обучать модели глубокого обучения не всегда просто; часто необходимы настройка и корректировка для реальных ситуаций.
- Поисковые системы и нейронные сети нередко представляют собой две разные системы, которые взаимодействуют как во время индексации, так и во время поиска. Очень важно следить за их производительностью, чтобы у пользователя было хорошее общее впечатление с точки зрения времени отклика.
- Реальные развертывания, такие как сценарий потоковой передачи, должны учитывать нагрузку и параллелизм, а также оценивать качество для достижения наилучшего возможного поискового решения.

Глядя вперед

Мы начали эту книгу с того, что задались вопросом, возможно ли использовать глубокие нейронные сети в качестве умных помощников, которые помогут предоставить наиболее подходящие инструменты поиска. В ходе прохождения глав мы затронули несколько аспектов распространенных поисковых систем, где у глубокого обучения есть значительный потенциал, чтобы помочь пользователям найти то, что они ищут.

Я надеюсь, что вы все больше и больше интересовались этой темой, когда мы рассматривали все более сложные аспекты и алгоритмы. Данная книга дала вам инструменты и практические советы, которые вы можете использовать немедленно; надеюсь, это также вдохновило вас посмотреть, что можно улучшить и какие проблемы остаются нерешенными, и захотеть погрузиться в этот процесс.

Пока я писал книгу, было опубликовано много новых статей о глубоком обучении, в том числе связанных с поиском. Новые функции активации были определены как полезные, и были предложены новые модели с многообещающими результатами. Я призываю вас не останавливаться на достигнутом и продолжать думать о том, что вам и вашим пользователям нужно и как добиться этого творческим путем.

Мы лишь коснулись области применения глубокого обучения для поиска информации. Вы узнали об основах нейронного поиска и теперь готовы учиться и делать больше самостоятельно. Развлекайтесь!

Предметный указатель

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**.

Томмазо Теофили

Глубокое обучение для поисковых систем

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Беликов Д. А.*
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. 25,84. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**