

O'REILLY®

# Знакомство с PyTorch

ГЛУБОКОЕ ОБУЧЕНИЕ ПРИ ОБРАБОТКЕ  
ЕСТЕСТВЕННОГО ЯЗЫКА



 ПИТЕР®

Брайан Макмахан  
Делип Рао

---

# Natural Language Processing with PyTorch

*Build Intelligent Language Applications Using  
Deep Learning*

*Delip Rao and Brian McMahan*

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY®**

Брайан Макмахан  
Делип Рао

# Знакомство с PyTorch

---

ГЛУБОКОЕ ОБУЧЕНИЕ ПРИ ОБРАБОТКЕ  
ЕСТЕСТВЕННОГО ЯЗЫКА



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

2020

ББК 32.813  
УДК 004.8  
М15

### **Макмахан Брайан, Рао Делип**

**М15** Знакомство с PyTorch: глубокое обучение при обработке естественного языка. — СПб.: Питер, 2020. — 256 с.: ил. — (Серия «Бестселлеры O'Reilly»).  
ISBN 978-5-4461-1241-8

Обработка текстов на естественном языке (Natural Language Processing, NLP) — крайне важная задача в области искусственного интеллекта. Успешная реализация делает возможными такие продукты, как Alexa от Amazon и Google Translate. Эта книга поможет вам изучить PyTorch — библиотеку глубокого обучения для языка Python — один из ведущих инструментов для дата-сайентистов и разработчиков ПО, занимающихся NLP. Делип Рао и Брайан Макмахан введут вас в курс дел с NLP и алгоритмами глубокого обучения. И покажут, как PyTorch позволяет реализовать приложения, использующие анализ текста.

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

ББК 32.813  
УДК 004.8

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1491978238 англ.

© Authorized Russian translation of the English edition of Natural Language Processing with PyTorch (ISBN 9781491978238)

© 2019 Delip Rao and Brian McMahan.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same

ISBN 978-5-4461-1241-8

© Перевод на русский язык ООО Издательство «Питер», 2020

© Издание на русском языке, оформление ООО Издательство «Питер», 2020

© Серия «Бестселлеры O'Reilly», 2020

# Краткое содержание

Предисловие .....	13
Благодарности.....	16
От издательства .....	18
<b>Глава 1.</b> Введение .....	19
<b>Глава 2.</b> Краткое знакомство с NLP .....	47
<b>Глава 3.</b> Базовые компоненты нейронных сетей .....	58
<b>Глава 4.</b> Использование упреждающих сетей при NLP .....	100
<b>Глава 5.</b> Вложение слов и прочих типов.....	142
<b>Глава 6.</b> Моделирование последовательностей для обработки текстов на естественных языках .....	170
<b>Глава 7.</b> Продолжаем моделирование последовательностей для обработки текстов на естественных языках .....	184
<b>Глава 8.</b> Продвинутое моделирование последовательностей для обработки текстов на естественных языках .....	203
<b>Глава 9.</b> Классические методы и перспективные направления .....	236
Что читать дальше .....	249
Об авторах .....	251
Об иллюстрации на обложке .....	252

# Оглавление

Предисловие .....	13
Условные обозначения .....	14
Использование примеров кода .....	14
Благодарности.....	16
От издательства .....	18
<b>Глава 1. Введение .....</b>	<b>19</b>
Парадигма обучения с учителем .....	20
Кодирование наблюдаемых величин и целевых переменных .....	23
Унитарное представление.....	24
TF-представление .....	25
Представление TF-IDF .....	26
Кодирование целевых переменных .....	28
Графы вычислений .....	29
Основы PyTorch .....	30
Установка PyTorch.....	31
Создание тензоров.....	31

---

Типы и размер тензоров .....	34
Операции над тензорами.....	35
Обращение по индексу, срезы и объединение .....	38
Тензоры и графы вычислений.....	41
Тензоры CUDA .....	42
Упражнения .....	44
Решения .....	45
Резюме.....	46
Библиография .....	46
<b>Глава 2. Краткое знакомство с NLP .....</b>	<b>47</b>
Корпусы текстов, токены и типы .....	48
Униграммы, биграммы, триграммы... n-граммы.....	51
Леммы и основы слов.....	52
Категоризация предложений и документов.....	53
Категоризация слов: маркирование частей речи.....	53
Категоризация отрезков текста: разбивка на порции и распознавание поименованных сущностей.....	53
Структура предложений .....	54
Смысл и семантика слов .....	55
Резюме.....	57
Библиография .....	57
<b>Глава 3. Базовые компоненты нейронных сетей .....</b>	<b>58</b>
Перцептрон: простейшая нейронная сеть.....	58
Функции активации .....	60
Сигма-функция .....	60
Гиперболический тангенс .....	61
ReLU.....	62

Многомерная логистическая функция .....	63
Функции потерь.....	64
Среднеквадратичная погрешность .....	64
Функции потерь на основе дискретной перекрестной энтропии .....	65
Функция потерь на основе бинарной перекрестной энтропии .....	67
Углубляемся в обучение с учителем.....	68
Формирование модельных данных .....	68
Собираем все вместе: градиентное машинное обучение с учителем .....	71
Вспомогательные понятия машинного обучения .....	72
Точная оценка качества модели: метрики оценки.....	73
Точная оценка качества модели: разбиение набора данных .....	73
Когда прекращать обучение .....	74
Поиск правильных значений гиперпараметров .....	75
Регуляризация .....	75
Пример: классификация тональностей обзоров ресторанов.....	76
Набор данных обзоров Yelp.....	77
Представление набора данных в PyTorch .....	79
Классы Vocabulary, Vectorizer и DataLoader.....	82
Классификатор-перцептрон.....	88
Процедура обучения .....	89
Оценка, вывод и просмотр.....	95
Резюме .....	98
Библиография .....	99
 <b>Глава 4. Использование упреждающих сетей при NLP .....</b>	 100
Многослойный перцептрон .....	101
Простой пример: XOR .....	103
Реализация многослойных перцептронов в PyTorch .....	105



---

Пример: классификация фамилий с помощью MLP .....	109
Набор данных фамилий.....	110
Классы Vocabulary, Vectorizer и DataLoader.....	111
Модель SurnameClassifier.....	113
Процедура обучения .....	114
Оценка модели и получение предсказаний .....	116
Регуляризация многослойных перцептронов: регуляризация весов и структурная регуляризация.....	118
Сверточные нейронные сети .....	120
Гиперпараметры CNN .....	121
Реализация сверточных нейронных сетей в PyTorch .....	126
Пример: классификация фамилий с помощью CNN .....	130
Класс SurnameDataset .....	131
Классы Vocabulary, Vectorizer и DataLoader.....	132
Заново реализуем класс SurnameClassifier с помощью сверточных нейронных сетей.....	133
Процедура обучения .....	134
Оценка эффективности модели и предсказание.....	135
Прочие вопросы CNN .....	136
Субдискретизация .....	136
Пакетная нормализация (BatchNorm).....	137
Связи типа «сеть в сети» (свертки $1 \times 1$ ).....	138
Остаточные связи/остаточный блок.....	139
Резюме.....	140
Библиография .....	140
<b>Глава 5. Вложение слов и прочих типов.....</b>	<b>142</b>
Зачем нужно обучение вложениям .....	143
Эффективность вложений.....	144

Подходы к обучению вложениям слов .....	145
Практическое применение предобученных вложений слов .....	146
Пример: обучение вложениям модели непрерывного мультимножества слов .....	152
Набор данных «Франкенштейн» .....	153
Классы Vocabulary, Vectorizer и DataLoader .....	155
Модель CBOWClassifier .....	156
Процедура обучения .....	157
Оценка модели и получение предсказаний .....	158
Пример: перенос обучения для классификации документов с применением предобученных вложений .....	158
Набор данных AG News .....	159
Классы Vocabulary, Vectorizer и DataLoader .....	160
Модель NewsClassifier .....	163
Процедура обучения .....	166
Оценка модели и получение предсказаний .....	167
Резюме .....	168
Библиография .....	169

**Глава 6. Моделирование последовательностей для обработки текстов  
на естественных языках .....**

Введение в рекуррентные нейронные сети .....	172
Реализация RNN Элмана .....	174
Пример: классификация национальной принадлежности фамилий с помощью символьного RNN .....	177
Класс SurnameDataset .....	177
Структуры данных для векторизации .....	178
Модель SurnameClassifier .....	179
Процедура обучения и результаты .....	182
Резюме .....	183
Библиография .....	183

<b>Глава 7. Продолжаем моделирование последовательностей для обработки текстов на естественных языках .....</b>	<b>184</b>
Проблемы «наивных» RNN (RNN Элмана).....	185
Пример: символьная RNN для генерации фамилий .....	188
Класс SurnameDataset .....	188
Структуры данных для векторизации .....	189
От RNN Элмана к GRU .....	192
Модель 1. Контекстно не обусловленная модель SurnameGenerationModel .....	192
Модель 2. Контекстно обусловленная модель SurnameGenerationModel .....	194
Процедура обучения и результаты .....	195
Полезные советы по обучению моделей последовательностей .....	200
Библиография .....	202
<b>Глава 8. Продвинутое моделирование последовательностей для обработки текстов на естественных языках .....</b>	<b>203</b>
Модели преобразования последовательностей в последовательности, модели типа «кодировщик-декодировщик» и контекстно обусловленная генерация .....	203
Захватываем больше информации из последовательности: двунаправленные рекуррентные модели .....	207
Захватываем больше информации из последовательности: внимание .....	209
Оценка эффективности моделей генерации последовательностей.....	213
Пример: нейронный машинный перевод .....	215
Набор данных для машинного перевода.....	216
Конвейер векторизации для NMT .....	217
Кодирование и декодирование в NMT-модели.....	221
Процедура обучения и результаты .....	232
Резюме .....	234
Библиография .....	235

<b>Глава 9. Классические методы и перспективные направления.....</b>	<b>236</b>
Какие темы мы уже изучили .....	236
Вечные вопросы NLP .....	237
Диалоговые и интерактивные системы.....	237
Дискурс.....	239
Извлечение информации и интеллектуальный анализ текста .....	240
Анализ и информационный поиск документов .....	240
Перспективные направления в NLP .....	240
Паттерны проектирования для промышленных NLP-систем .....	242
Библиография .....	247
Что читать дальше .....	249
Об авторах .....	251
Об иллюстрации на обложке .....	252

# Предисловие

Цель этой книги — рассказать новичкам о возможностях обработки естественного языка (Natural Language Processing, NLP) и глубокого обучения. Эти сферы сегодня стремительно развиваются, и данная книга посвящена им обеим, с упором на реализацию. При ее написании нам часто приходилось принимать сложные, а иногда и неприятные решения относительно того, какой материал не включать в книгу. Мы надеемся, что начинающие программисты смогут получить базовые знания и хотя бы бегло ознакомиться с имеющимися возможностями. Машинное обучение вообще и глубокое обучение в частности — эмпирические, а не теоретические дисциплины. Многочисленные комплексные примеры, включенные во все главы, приглашают вас проверить изложенный материал на практике.

Мы начинали работу над книгой с версии PyTorch 0.2. Примеры обновлялись при выходе каждого релиза PyTorch с 0.2 до 0.4. К моменту выхода данной книги из печати ожидается выход версии PyTorch 1.0<sup>1</sup>. Примеры кода в издании совместимы с версией PyTorch 0.4 и должны работать в неизменном виде с версией PyTorch 1.0<sup>2</sup>.

Небольшое примечание относительно стиля написания этой книги. Мы намеренно избегали в большинстве мест математических описаний не потому, что математическая сторона глубокого обучения очень сложна (это вовсе не так), а потому, что это во многих случаях отвлекало бы нас от основной цели книги — помочь читателю, который только приступает к изучению данных вопросов. Во многих случаях как в коде, так и в сопровождающем его тексте мы предпочитали развернутые описания кратким. Продвинутые пользователи и опытные программисты, вероятно, отметят немало способов сделать код более лаконичным, но мы решили излагать материал как можно подробнее, чтобы сделать его доступным для максимально широкой аудитории читателей.

---

<sup>1</sup> Речь об оригинальном издании. Текущая версия — 1.2.0. — *Примеч. пер.*

<sup>2</sup> См. <https://pytorch.org/2018/05/02/road-to-1.0.html>.

## Условные обозначения

В данной книге используются следующие типографские обозначения.

### *Курсив*

Отмечает новые термины и слова, на которые нужно обратить внимание.

### Рубленый шрифт

Им выделены веб-ссылки и адреса электронной почты.

### Моноширинный шрифт

Используется для примеров программ, а также внутри абзацев для ссылки на элементы программ, такие как переменные или имена функций, базы данных, переменные среды, операторы и ключевые слова. Им также выделены имена и расширения файлов.

### Жирный моноширинный шрифт

Обозначает команды или другой текст, который должен быть в точности набран пользователем.

### Моноширинный курсив

Отмечает текст, который необходимо заменить пользовательскими значениями или значениями, определяемыми контекстом.



---

Этот элемент обозначает общее примечание.

---

## Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания по адресу <https://github.com/joosthub/PyTorchNLPBook>.

Эта книга написана для того, чтобы помочь вам делать вашу работу. В целом, если к книге прилагается какой-либо пример кода, вы можете использовать его в ваших программах и документации. Обращаться к нам за разрешением нет необходимости, разве что вы копируете значительную часть кода. Например, написание программы, использующей несколько фрагментов кода из этой книги, не требует отдельного разрешения. Для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение, конечно, требуется. Ответ на вопрос путем цитирования этой книги и цитирования примеров кода не требует разрешения. Включение значительного количества кода примеров из этой книги в документацию к вашему продукту может потребовать разрешения.

Мы ценим, хотя и не требуем, ссылки на первоисточник. Ссылка на первоисточник включает название, автора, издательство и ISBN. Например: «Делип Рао, Брайан Макмахан. *Знакомство с PyTorch. Глубокое обучение при обработке естественного языка*. СПб.: Питер — 2020, 9785446112418».

Если вам кажется, что использование вами примеров кода выходит за рамки правомерного применения или данного выше разрешения, не стесняйтесь, связывайтесь с нами по адресу [permissions@oreilly.com](mailto:permissions@oreilly.com).

У этой книги есть своя веб-страница со списком ошибок, примерами и другой дополнительной информацией. Она находится по адресу <http://bit.ly/nlprocbk>.

# Благодарности

Эта книга прошла через несколько стадий «эволюции», и каждая ее новая версия отличалась от предыдущей. В создании каждой из версий участвовали разные люди (и даже использовались различные фреймворки глубокого обучения).

Авторы хотели бы поблагодарить Гоку Мохандаса (Goku Mohandas) за его участие в создании книги. Гоку посвятил этому проекту немало сил, прежде чем вынужден был покинуть его по связанным со своей профессиональной деятельностью причинам. Мы скучаем по несравненному энтузиазму Гоку в отношении PyTorch и его оптимизму. Ждем от него в будущем замечательных свершений!

Данная книга не была бы столь аккуратной с технической точки зрения, если бы не благожелательные и вместе с тем первоклассные отзывы наших рецензентов Лилину Таню (Liling Tan) и Дебасишу Гошу (Debasish Gosh). Лилин помог нам советами по разработке программных продуктов с ультрасовременным NLP, а Дебасиш дал очень ценные отзывы с точки зрения разработчиков. Мы также благодарны за поддержку Альфредо Канциани (Alfredo Canziani), Сумису Чинтале (Soumith Chintala) и многим другим замечательным людям с форумов разработчиков PyTorch. Мы также почерпнули немало полезной информации из ежедневных обсуждений на тему NLP в сообществе #nlproc в Twitter. Многими из идей книги мы обязаны этому сообществу, в не меньшей мере, чем нашему личному опыту.

С нашей стороны было бы упущением не выразить благодарность редактору Джефу Блейлу (Jeff Bleiel) за его поддержку. Без его указаний эта книга никогда бы не увидела свет. Правки Боба Расселла (Bob Russell) и поддержка Нэн Барбер (Nan Barber) на этапе написания превратили нашу рукопись из чернового наброска в пригодную для печати книгу. Мы также хотели бы поблагодарить Шэннон Катт (Shannon Cutt) за ее поддержку в самом начале создания книги.

Большинство тем в книге основаны на материале двухдневных курсов обучения по NLP, которые мы проводили на конференциях по ИИ и Strata от издательства



O'Reilly. Мы хотели бы поблагодарить Бена Лорику (Ben Loriga), Джейсона Пердью (Jason Perdue) и Софию Демартини (Sophia DeMartini) за работу с нами на этих курсах.

Делип рад, что ему достался такой соавтор, как Брайан Макмахан (Brian McMahon). Брайан изо всех сил старался помочь в создании книги. Было очень приятно делить с ним радости и огорчения в процессе написания! Делип также хотел бы поблагодарить Бена Лорику из издательства O'Reilly, настоявшего в свое время на написании книги по NLP.

Брайан хотел бы поблагодарить Сару Мануэль (Sara Manuel) за ее бесконечную поддержку. Он также очень благодарен Делипу Рао (Delip Rao) — без его безграничного упорства и выдержки эта книга не появилась бы.

# От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

# 1

## Введение

Такие общеизвестные бренды, как Echo (Alexa), Siri и Google Translate, объединяет по крайней мере одно: это все программные продукты, производные от приложений для *обработки написанных на естественном языке текстов* (Natural Language Processing, NLP). Именно NLP — одна из двух главных тем данной книги. Термин *NLP* относится к решению практических задач с помощью приемов *понимания* текстов, включающих применение статистических методов (с использованием возможностей лингвистики или без). Это «*понимание*» текстов достигается главным образом за счет их преобразования в пригодные для вычислений *представления* (representations) в виде дискретных или непрерывных комбинаторных структур, таких как векторы/тензоры, графы и деревья.

Обучение подходящих для конкретной задачи представлений на основе данных (в данном случае текста) — предмет *машинного обучения* (machine learning). Машинное обучение (МО) применяется для анализа текстовых данных уже более трех десятилетий, но в последние десять<sup>1</sup> лет набор методов машинного обучения, известный под названием *глубокого обучения* (deep learning), особенно эволюционировал и доказал свою эффективность для различных задач искусственного интеллекта (ИИ) в сферах NLP, распознавания речи и машинного зрения. Глубокое обучение — вторая тема нашей книги.



---

В конце каждой главы приводится список литературы.

---

Проще говоря, глубокое обучение дает возможность эффективного обучения представлений на основе данных с помощью абстракции под названием «*граф*

---

<sup>1</sup> Хотя у нейронных сетей и NLP в целом долгая и богатая история, первое использование глубокого обучения в современном виде для NLP часто приписывают Коллоберту и Вестону (Collobert и Weston, 2008).

вычислений» (computational graph) и методов численной оптимизации. Успех глубокого обучения и графов вычислений был столь велик, что такие крупнейшие информационно-технологические компании, как Google, Facebook и Amazon, обнародовали свои реализации фреймворков для вычислений на графах и основанных на них библиотек, чтобы привлечь внимание исследователей и разработчиков. В этой книге мы используем для реализации алгоритмов глубокого обучения *PyTorch* — основанный на языке Python фреймворк для вычислений на графах, популярность которого непрерывно растет. В данной главе мы расскажем, что такое графы вычислений и почему мы выбрали в качестве фреймворка именно PyTorch.

Сферы машинного и глубокого обучения очень обширны. В этой главе и в большей части этой книги мы в основном будем иметь дело с так называемым *машинным обучением с учителем* (supervised learning), то есть с обучением с маркированными обучающими выборками. Если большинство этих терминов вам пока не знакомы — вы попали туда, куда надо! В этой, а также в дальнейших главах не только раскрывается, но и во всех подробностях исследуется смысл данных терминов. Если же вы уже частично знакомы с терминологией и слышали упомянутые здесь понятия — вам все равно стоит продолжить чтение главы по двум причинам: чтобы правильно понимать терминологию оставшейся части книги и заполнить пробелы в знаниях, необходимых для понимания следующих глав.

Задачи этой главы:

- ❑ выработать четкое понимание парадигмы машинного обучения с учителем, разобраться в терминологии и выработать концептуальную базу для дискуссии о задачах машинного обучения в последующих главах;
- ❑ научиться кодировать входные данные для задач машинного обучения;
- ❑ разобраться, что такое графы вычислений;
- ❑ освоить основы PyTorch.

Приступим!

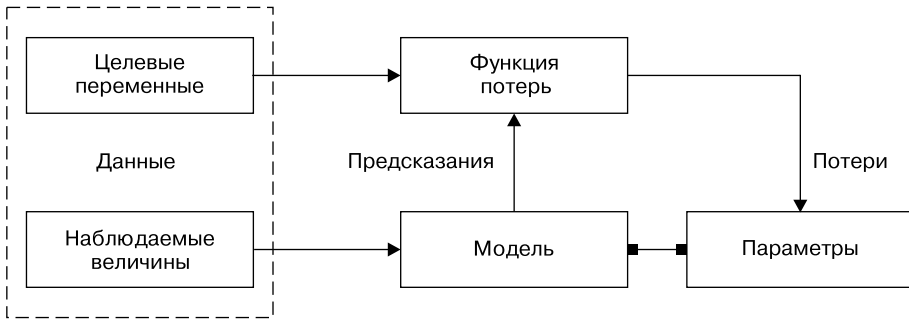
## Парадигма обучения с учителем

*Машинное обучение с учителем* (supervised learning) используется в тех случаях, когда для *наблюдаемых величин* (observations) доступны эталонные (контрольные) значения предсказываемых *целевых переменных* (targets). Например, при классификации документов целевой переменной является дискретная метка<sup>1</sup>, а наблюдаемой величиной — сам документ. В сфере машинного перевода наблюдаемой величи-

---

<sup>1</sup> Дискретной (или же категорийной) называется переменная, которая может принимать одно из фиксированного набора значений, например {ИСТИНА, ЛОЖЬ}, {ГЛАГОЛ, СУЩЕСТВИТЕЛЬНОЕ, НАРЕЧИЕ...} и т. д.

ной является фраза на одном языке, а целевой переменной — фраза на другом. Разобравшись с этой терминологией, мы можем проиллюстрировать парадигму обучения с учителем на рис. 1.1.



**Рис. 1.1.** Парадигма обучения с учителем — концептуальная база для обучения на маркированных входных данных

Разобьем парадигму обучения с учителем, показанную на рис. 1.1, на шесть основных составляющих.

- ❑ *Наблюдаемые величины* (наблюдения, observations) — сущности, поведение которых мы хотим предсказать. Наблюдаемые величины обозначаются  $x$ . Иногда мы будем называть их *входными данными* (inputs).
- ❑ *Целевые переменные* (targets) — соответствующие наблюдаемым величинам метки. Обычно именно их мы и предсказываем. Следуя общепринятой нотации в машинном/глубоком обучении, мы будем их обозначать  $y$ . Иногда эти метки называют *контрольными значениями* (ground truth).
- ❑ *Модель* представляет собой математическое выражение или функцию, принимающую на входе наблюдаемую величину  $x$  и предсказывающую значение целевой метки.
- ❑ *Параметры* — иногда называются также *весами* и служат для параметризации модели. Их обычно обозначают  $w$  (от англ. *weights*) или  $\hat{w}$ .
- ❑ *Предсказания*, называемые также *оценками* (estimates), представляют собой значения целевых переменных, предсказанные моделью по наблюдаемым величинам. Для их обозначения мы будем использовать «шляпку». Так, предсказание целевой переменной  $y$  обозначается  $\hat{y}$ .
- ❑ *Функция потерь* (loss function) — это функция, служащая мерой отклонения предсказания от целевой переменной для наблюдений из обучающей последовательности. Функция потерь ставит целевой переменной и ее предсказанию в соответствие скалярное вещественное значение, называемое *потерями* (loss). Чем меньше значение потерь, тем лучше модель предсказывает целевую переменную. Мы будем обозначать функцию потерь  $L$ .

Хотя математическая формализация не обязательна для успешного моделирования в сфере NLP/глубокого обучения или для написания данной книги, мы формально опишем парадигму обучения с учителем и познакомим новичков в этой области со стандартной терминологией, чтобы им были понятны обозначения и стиль написания научных статей, с которыми они могут встретиться в arXiv.

Рассмотрим набор данных  $D = \{x_i, y_i\}_{i=1}^n$ , в котором количество выборок равно  $n$ . Нам нужно на основе этого набора данных обучить функцию (модель)  $f$ , параметризованную весами  $w$ . Иначе говоря, делается предположение о структуре модели  $f$ , а при заданной структуре полученные в результате обучения значения весов  $w$  полностью характеризуют модель. Для входных данных  $X$  модель предсказывает значение  $\hat{y}$  целевой переменной:

$$\hat{y} = f(X, w).$$

В обучении с учителем в случае обучающих выборок нам известно истинное значение целевой переменной для наблюдаемой величины. Функция потерь в данном случае будет равна  $L(y, \hat{y})$ . Таким образом, обучение с учителем превращается в поиск оптимальных значений параметров/весов  $w$ , при которых достигается минимум совокупных потерь для всех  $n$  выборок.

### ОБУЧЕНИЕ С ПОМОЩЬЮ СТОХАСТИЧЕСКОГО ГРАДИЕНТНОГО СПУСКА

Задача машинного обучения с учителем состоит в поиске значений параметров, минимизирующих функцию потерь для данного набора данных. Другими словами, это эквивалентно поиску корней уравнения. Как известно, *градиентный спуск* (gradient descent) — распространенный метод поиска корней уравнения. Напомним, что в обычном методе градиентного спуска выбираются какие-либо начальные значения для корней (параметров), после чего они обновляются в цикле до тех пор, пока вычисленное значение целевой функции (функции потерь) не окажется ниже заданного порогового значения (критерий сходимости). Для больших наборов данных реализация обычного градиентного спуска — задача почти неразрешимая вследствие ограничений памяти и очень медленно работающая из-за вычислительных издержек. Вместо этого обычно используется аппроксимация градиентного спуска, называемая *стохастическим градиентным спуском* (stochastic gradient descent, SGD). При этом случайным образом выбирается точка данных (или подмножество точек данных), и для заданного подмножества вычисляется градиент. В случае отдельной точки данных такой подход называется *чистым SGD*, а в случае подмножества (из более чем одной) точек данных — *мини-пакетным SGD*. Эпитеты «чистый» и «мини-пакетный» обычно опускают, если используемый вариант метода понятен из контекста. На практике чистый SGD применяется редко из-за его очень медленной сходимости вследствие шума.

Существует множество вариантов общего алгоритма SGD, нацеленных на ускорение сходимости. В следующих главах мы рассмотрим некоторые из них, а также поговорим об использовании градиентов при обновлении значений параметров. Этот процесс итеративного обновления значений параметров называется методом *обратного распространения ошибки* (backpropagation). Каждый шаг (так называемая эпоха) алгоритма обратного распространения ошибки состоит из *прямого прохода* (forward pass) и *обратного прохода* (backward pass). При прямом проходе выполняется вычисление наблюдаемых величин при текущих значениях параметров и рассчитывается функция потерь. На обратном шаге значения параметров обновляются на основе градиента потерь.

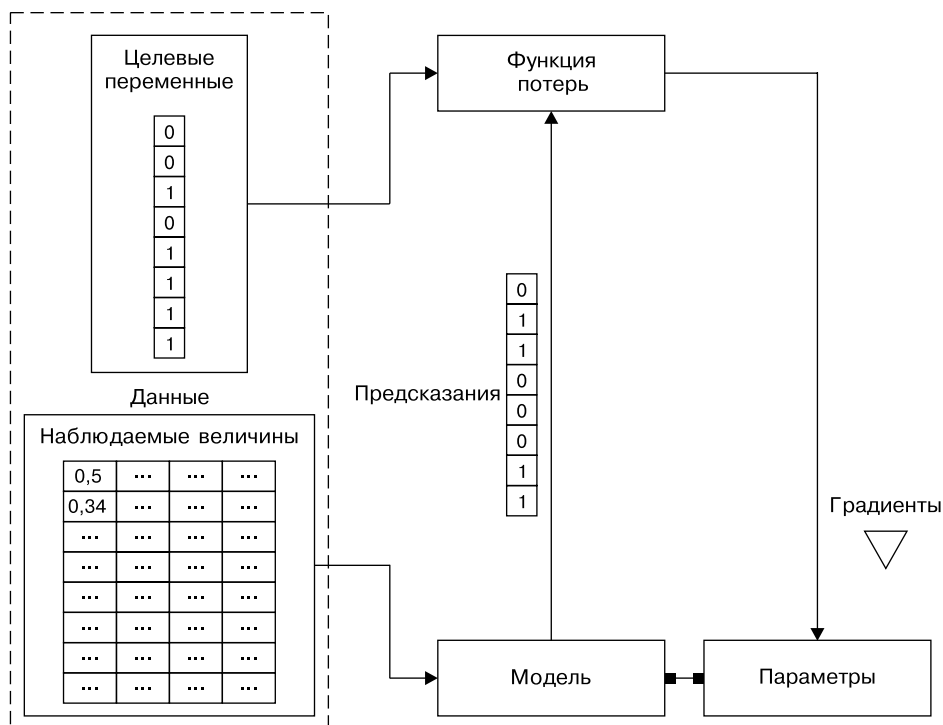
Обратите внимание, что все вышеизложенное относится отнюдь не только к глубокому обучению или нейронным сетям<sup>1</sup>. Стрелки на рис. 1.1 указывают направление «движения» данных при обучении системы. Мы еще вернемся к обучению и понятию «движения» данных в разделе «Графы вычислений» далее, но сначала взглянем на возможные способы численного представления наших входных данных и целевых переменных в задачах NLP для обучения моделей и предсказания целевых переменных.

## Кодирование наблюдаемых величин и целевых переменных

Чтобы использовать наблюдаемые величины (текст) в алгоритмах машинного обучения, необходимо представить их в числовом виде. Наглядная иллюстрация этого процесса приведена на рис. 1.2.

Использование числового вектора — простой способ представления текста. Существует множество способов выполнения подобного отображения/представления. На самом деле значительная часть данной книги посвящена тому, как путем обучения получить для задачи подобное представление на основе имеющихся данных. Однако мы начнем с нескольких простых эвристических представлений, в основе которых лежат подсчеты слов в тексте. Несмотря на простоту, они могут оказаться очень полезными в качестве отправного пункта для более многообещающего обучения представлением. Все эти представления на основе подсчетов начинаются с вектора фиксированной размерности.

<sup>1</sup> Глубокое обучение отличается от традиционных нейронных сетей, обсуждавшихся в литературе до 2006 года, тем, что относится к постоянно расширяющемуся набору методик обеспечения надежности за счет добавления дополнительных уровней сети. Мы расскажем, почему это так важно, в главах 3 и 4.



**Рис. 1.2.** Кодирование наблюдаемых величин и целевых переменных: целевые переменные и наблюдаемые величины с рис. 1.1 представлены здесь численно, в виде векторов или тензоров. Все это в совокупности называется кодированием входных данных

## Унитарное представление

*Унитарное представление* (one-hot representation), как понятно из названия, начинается с вектора нулей, в котором соответствующий элемент вектора устанавливается равным 1, если слово присутствует в предложении или документе. Рассмотрим следующие два предложения:

Time flies like an arrow.  
Fruit flies like a banana.

Разбиение этих предложений (взятых в нижнем регистре) на лексемы без учета знаков препинания даст нам словарь из восьми слов: {time, fruit, flies, like, a, an, arrow, banana}. Таким образом, каждое слово можно представить с помощью восьмимерного унитарного вектора. В этой книге мы будем обозначать унитарное представление лексемы/слова  $w$  следующим образом:  $1_w$ .

Свернутое унитарное представление фразы, предложения или документа представляет собой просто логическое ИЛИ унитарных представлений составляю-



щих их слов. С помощью показанного на рис. 1.3 кодирования можно получить унитарное представление фразы *like a banana* в виде матрицы  $3 \times 8$ , где столбцы представляют собой восьмимерные унитарные векторы. Часто встречается также свернутое (бинарное) кодирование, в котором текст/фраза представляется в виде вектора длиной, равной длине словаря, в котором нули и единицы указывают на отсутствие или наличие слова. Бинарное кодирование для фразы *like a banana* выглядит следующим образом:  $[0, 0, 0, 1, 1, 0, 0, 1]$ .

	time	fruit	flies	like	a	an	arrow	banana
1 <sub>time</sub>	1	0	0	0	0	0	0	0
1 <sub>fruit</sub>	0	1	0	0	0	0	0	0
1 <sub>flies</sub>	0	0	1	0	0	0	0	0
1 <sub>like</sub>	0	0	0	1	0	0	0	0
1 <sub>a</sub>	0	0	0	0	1	0	0	0
1 <sub>an</sub>	0	0	0	0	0	1	0	0
1 <sub>arrow</sub>	0	0	0	0	0	0	1	0
1 <sub>banana</sub>	0	0	0	0	0	0	0	1

**Рис. 1.3.** Унитарное представление для кодирования фраз Time flies like an arrow и Fruit flies like a banana



Если при чтении вы ужаснулись от того, что мы смешали два различных значения (смысла) слова flies — поздравляем, вы проникательный читатель! Естественный язык полон неоднозначностей, но благодаря чрезвычайно упрощающим допущениям все же можно создавать полезные решения. Существуют возможности создания представлений с учетом смысла, но не будем забегать вперед.

Хотя мы редко будем использовать в книге для входных данных что-либо, кроме унитарного представления, далее мы познакомим вас с представлениями «частотность терма» (Term-Frequency, TF) и «частотность терма — обратная частотность документа» (Term-Frequency-Inverse-Document-Frequency, TF-IDF), поскольку они очень популярны в NLP. У этих представлений очень давняя история в области информационного поиска (information retrieval, IR), они и сейчас активно применяются в промышленных системах NLP.

## TF-представление

TF-представление фразы, предложения или документа — это просто сумма унитарных представлений составляющих его слов. Возвращаясь к нашему примеру, при вышеприведенном унитарном представлении TF-представление предложения *Fruit flies like a banana* будет выглядеть так:  $[1, 2, 2, 1, 1, 0, 0, 0]$ . Отметим, что

каждый из элементов здесь представляет собой количество вхождений соответствующего слова в предложение (корпус). TF-представление слова  $w$  мы будем обозначать  $TF(w)$ .

**Пример 1.1.** Генерация свернутого унитарного или бинарного представления с помощью библиотеки scikit-learn (рис. 1.4)

```
from sklearn.feature_extraction.text import CountVectorizer
import seaborn as sns
```

```
corpus = ['Time flies flies like an arrow.',
          'Fruit flies like a banana.']
vocab=['an', 'arrow', 'banana', 'flies', 'fruit', 'like', 'time']
one_hot_vectorizer = CountVectorizer(binary=True)
one_hot = one_hot_vectorizer.fit_transform(corpus).toarray()
sns.heatmap(one_hot, annot=True,
            cbar=False, xticklabels=vocab,
            yticklabels=['Предложение 1',
                        'Предложение 2'])
```



**Рис. 1.4.** Свернутое унитарное представление, сгенерированное в примере 1.1

## Представление TF-IDF

Рассмотрим набор патентных документов. Вероятно, в большинстве из них встречаются такие слова, как *claim*, *system*, *method*, *procedure* и т. д., зачастую по несколько раз. В TF-представлении веса слов пропорциональны частоте, с которой они встречаются в документе. Однако такие распространенные слова, как *claim*, не вносят никакого вклада в наше понимание конкретного патента. Проще говоря, когда редкое слово (например, «тетрафторэтилен») попадает не так часто, но, вероятно, ясно свидетельствует о сущности патентного документа, то имеет смысл присвоить ему больший вес в представлении. Эвристический algo-

ритм учета обратной частотности документа (Inverse-Document-Frequency, IDF) именно так и работает.

IDF-представление специально снижает вес распространенных лексем и повышает вес редких в векторном представлении. Значение  $IDF(w)$  лексемы  $w$  учитывает корпус документов и равно:

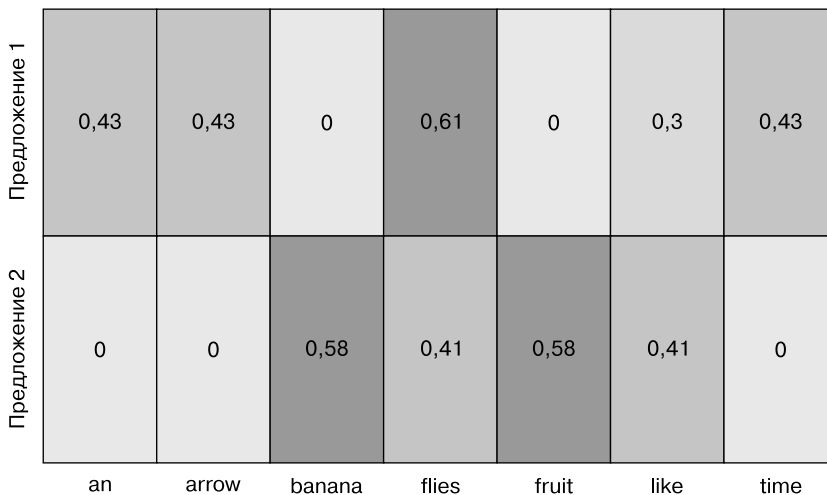
$$IDF(w) = \log \frac{N}{n_w},$$

где  $n_w$  равно количеству документов, содержащих слово  $w$ , а  $N$  — общее количество документов. Показатель TF-IDF равен просто  $TF(w) \cdot IDF(w)$ . Во-первых, обратите внимание, что для очень распространенного слова, встречающегося во всех документах (то есть  $n_w = N$ ),  $IDF(w)$  равно 0, а следовательно, и показатель TF-IDF равен 0, так что вес этого термина полностью аннулируется. Во-вторых, если терм встречается очень редко, скажем только в одном документе, то показатель IDF примет максимальное возможное значение,  $\log N$ . Пример 1.2 демонстрирует генерацию TF-IDF-представления для списка предложений на английском языке с помощью библиотеки scikit-learn.

**Пример 1.2.** Генерация TF-IDF-представления с помощью библиотеки scikit-learn (рис. 1.5)

```
from sklearn.feature_extraction.text import TfidfVectorizer
import seaborn as sns

vocab=['an','arrow','banana','flies','fruit','like','time']
tfidf_vectorizer = TfidfVectorizer()
tfidf = tfidf_vectorizer.fit_transform(corpus).toarray()
sns.heatmap(tfidf, annot=True, cbar=False, xticklabels=vocab,
            yticklabels= ['Предложение 1', 'Предложение 2'])
```



**Рис. 1.5.** TF-IDF-представление, сгенерированное в примере 1.2

В глубоком обучении редко встречается кодирование входных данных с помощью эвристических представлений вроде TF-IDF, поскольку цель состоит именно в получении представления. Зачастую начинают с унитарного представления с целочисленными индексами и специального слоя «поиска вложений» (embedding lookup), чтобы сформировать входные данные для нейронной сети. В следующих главах мы рассмотрим несколько примеров этого.

## Кодирование целевых переменных

Как отмечалось в разделе «Парадигма обучения с учителем» этой главы, конкретный характер целевой переменной зависит от решаемой задачи NLP. Например, в случаях машинного перевода, автоматического реферирования и формирования ответов на вопросы целевая переменная также представляет собой текст и кодируется с помощью подходов, аналогичных вышеописанному унитарному представлению.

Во множестве задач NLP используются дискретные метки в том случае, когда модель должна предсказывать одно значение из фиксированного набора. Они часто кодируются путем присвоения каждой метке уникального индекса, но такое простое представление становится проблематичным при наличии слишком большого количества выходных меток. В качестве примера можно привести задачу *языкового моделирования* (language modeling), цель которой — предсказать следующее слово по ранее наблюдавшимся словам. Пространство меток представляет собой весь словарный запас языка, который с легкостью может составлять несколько сотен тысяч слов, считая специальные символы, названия и т. д. Мы вернемся к этой задаче в следующих главах и посмотрим, как ее можно решить.

Некоторые задачи NLP требуют предсказания числового значения на основе заданного текста. Например, оценить удобочитаемость эссе на английском языке. По отрывку из отзыва о ресторане предсказать его числовой рейтинг с точностью до первого знака после запятой. По твитам какого-либо пользователя предсказать его возрастную группу. Существует несколько способов кодирования числовых целевых переменных, в том числе вполне приемлемый подход, при котором целевые переменные распределяются по дискретным корзинам — например, 0–18, 19–25, 25–30 и т. д., — как в простой задаче классификации<sup>1</sup>. Распределение по таким корзинам может быть однородным или неоднородным, определяемым данными. Хотя подробное обсуждение этого вопроса выходит за рамки книги, мы обращаем на него ваше внимание, поскольку кодирование целевых переменных в подобных случаях разительно влияет на производительность. Рекомендуем вам почитать книгу Догерти и др. (1995), а также изучить приведенные там источники литературы.

<sup>1</sup> «Порядковая» классификация представляет собой задачу многоклассовой классификации с частичной упорядоченностью меток. В нашем примере с возрастом категория 0–18 предшествует категории 19–25 и т. д.

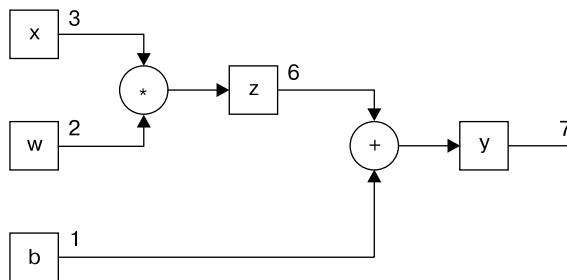
## Графы вычислений

Рисунок 1.1 характеризует парадигму обучения с учителем как архитектуру движения данных. Она состоит из модели (математического выражения), преобразующей входные данные для получения предсказаний, и функции потерь (еще одного выражения), которое генерирует сигнал обратной связи, предназначенный для подстройки параметров модели. Для реализации подобного движения данных удобно использовать такую структуру, как граф вычислений<sup>1</sup>. Формально граф вычислений — это абстракция для моделирования математических выражений. В контексте глубокого обучения реализации графов вычислений, такие как Theano, TensorFlow и PyTorch, обеспечивают также автоматическое дифференцирование, необходимое для получения градиентов параметров при обучении в рамках парадигмы машинного обучения с учителем. Мы обсудим это подробнее в следующем разделе — «Основы PyTorch».

*Вывод* (inference), или предсказание, представляет собой просто вычисление выражения (движение вперед по графу вычислений). Посмотрим, как граф вычислений моделирует выражения. Рассмотрим выражение:

$$y = wx + b.$$

Его можно разложить на два подвыражения:  $z = wx$  и  $y = z + b$ . После этого исходное выражение можно представить в виде ориентированного ациклического графа (directed acyclic graph, DAG), в котором вершинами являются математические операции, например умножение и сложение. Входные данные для операций — входящие в вершины ребра, а результаты операций — исходящие ребра. Граф вычислений для выражения  $y = wx + b$  приведен на рис. 1.6. В следующем разделе мы увидим, как с помощью PyTorch можно без труда создавать графы вычислений и вычислять градиенты без каких-либо дополнительных вспомогательных действий.



**Рис. 1.6.** Представление выражения  $y = wx + b$  с помощью графа вычислений

<sup>1</sup> Сеппо Линнаймаа (Seppo Linnainmaa) (<http://bit.ly/2Rnmdao>) впервые упомянул автоматическое дифференцирование на графах вычислений в своей магистерской дипломной работе в 1970 году! Различные варианты этой идеи легли в основу современных фреймворков глубокого обучения: Theano, TensorFlow и PyTorch.

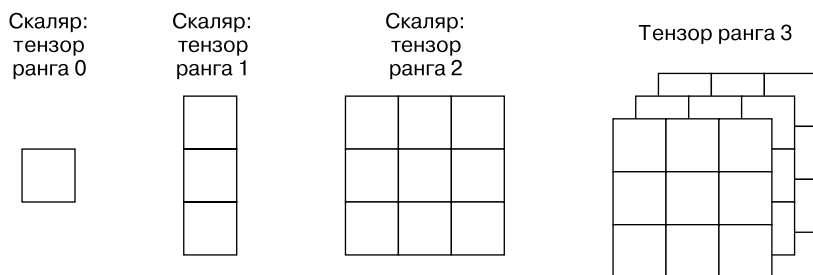
## Основы PyTorch

В этой книге PyTorch будет широко использоваться для реализации моделей глубокого обучения. PyTorch — фреймворк глубокого обучения с открытым исходным кодом, поддерживаемый силами сообщества разработчиков. В отличие от Theano, Caffe и TensorFlow, PyTorch реализует метод автоматического дифференцирования на основе так называемой ленты (tape) (<http://bit.ly/2Jrntq1>), с помощью которого можно описывать и выполнять графы вычислений динамически, что очень удобно для отладки и конструирования сложных моделей с минимальными усилиями.

### ДИНАМИЧЕСКИЕ И СТАТИЧЕСКИЕ ГРАФЫ ВЫЧИСЛЕНИЙ

Такие статические фреймворки, как Theano, Caffe и TensorFlow, требуют описания и компиляции графа вычислений перед его выполнением. Хотя это приводит к очень высокой производительности реализаций (что полезно в промышленной эксплуатации), но может доставить немало хлопот при исследовательской работе и во время разработки. Современные фреймворки, такие как Chainer, DyNet и PyTorch, реализуют динамические графы вычислений, не требующие компиляции моделей перед каждым выполнением, что обеспечивает возможность более гибкого, императивного стиля разработки. Динамические графы вычислений особенно удобны при моделировании задач NLP, в которых различные входные данные могут привести к разным структурам графа.

PyTorch — оптимизированная библиотека для работы с тензорами, включающая набор пакетов для глубокого обучения. Основное понятие этой библиотеки — *тензор* (tensor), математический объект для хранения многомерных данных. Тензор ранга 0 — просто число, или *скаляр*. Тензор ранга 1 — массив чисел (то есть *вектор*). Аналогично тензор ранга 2 представляет собой массив векторов (*матрицу*). Следовательно, тензор можно рассматривать как  $n$ -мерный массив скалярных значений, как показано на рис. 1.7.



**Рис. 1.7.** Тензоры как обобщение многомерных массивов

В этом разделе мы начнем знакомить вас с различными операциями PyTorch, включая:

- ❑ создание тензоров;
- ❑ операции с тензорами;
- ❑ обращение по индексам, срезы и объединение тензоров;
- ❑ вычисление градиентов с помощью тензоров;
- ❑ использование GPU с помощью тензоров CUDA.

Мы рекомендуем вам подготовить блокнот Python 3.5+, установить PyTorch, как описано далее, и следить за кодом примеров<sup>1</sup>. Мы также рекомендуем вам выполнить приведенные в этой главе упражнения.

## Установка PyTorch

Первый шаг — установка PyTorch на своей машине. Для этого нужно выбрать на сайте [pytorch.org](https://pytorch.org) параметры, соответствующие вашей системе. Выберите операционную систему, систему управления пакетами (мы рекомендуем Conda или Pip), затем используемую версию Python (рекомендуем 3.5 или выше). В результате этого будет сгенерирована команда, с помощью которой вы сможете установить нужную вам конфигурацию PyTorch. На момент написания данной книги команда установки для среды Conda выглядела следующим образом:

```
conda install pytorch torchvision -c pytorch
```



Если у вас в системе есть графический процессор (GPU) с поддержкой CUDA, рекомендуем выбрать также соответствующую версию CUDA. Дополнительную информацию вы найдете в инструкциях по установке на сайте [pytorch.org](https://pytorch.org).

## Создание тензоров

Во-первых, опишем вспомогательную функцию, `describe(x)`, для вывода различных характеристик тензора `x`, например типа тензора, его размерности и содержимого:

Input[0]

```
def describe(x):
    print("Type: {}".format(x.type()))
    print("Shape/size: {}".format(x.shape))
    print("Values: \n{}".format(x))
```

<sup>1</sup> Код для этого раздела можно найти в папке `/chapters/chapter_1/PyTorch_Basics.ipynb` репозитория GitHub для данной книги.

С помощью пакета `torch` из PyTorch можно создавать тензоры множеством способов. Один из них состоит в том, чтобы задать для тензора случайные значения, просто указав нужные размерности, как показано в примере 1.3.

**Пример 1.3.** Создание тензора в PyTorch с помощью конструктора класса `torch.Tensor`

Input[0]

```
import torch
describe(torch.Tensor(2, 3))
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 3.2018e-05,  4.5747e-41,  2.5058e+25],
        [ 3.0813e-41,  4.4842e-44,  0.0000e+00]])
```

Можно также создать тензор, инициализируя его случайными значениями, полученными из равномерного распределения по интервалу  $[0, 1)$  или стандартного нормального распределения<sup>1</sup>, как показано в примере 1.4. Тензоры со случайными начальными значениями, скажем взятыми из равномерного распределения, очень важны, как мы увидим в главах 3 и 4.

**Пример 1.4.** Создание тензора, инициализированного случайными значениями

Input[0]

```
import torch
describe(torch.rand(2, 3)) # случайное равномерное распределение
describe(torch.randn(2, 3)) # случайное нормальное распределение
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.0242,  0.6630,  0.9787],
        [ 0.1037,  0.3920,  0.6084]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ -0.1330, -2.9222, -1.3649],
        [ 2.3648,  1.1561,  1.5042]])
```

Можно также создавать тензоры, заполненные одним и тем же скалярным значением. Для создания тензора из нулей или единиц предусмотрены встроенные

<sup>1</sup> Стандартное нормальное распределение — это нормальное распределение с математическим ожиданием, равным 0, и дисперсией, равной 1.



функции, а для заполнения конкретными значениями можно воспользоваться методом `fill_()`. Знак подчеркивания (`_`) в названии методов PyTorch означает, что операция выполняется «на месте», то есть модифицирует содержимое без создания нового объекта, как показано в примере 1.5.

**Пример 1.5.** Создание заполненного значениями тензора

Input[0]

```
import torch
describe(torch.zeros(2, 3))
x = torch.ones(2, 3)
describe(x)
x.fill_(5)
describe(x)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 5.,  5.,  5.],
        [ 5.,  5.,  5.]])
```

В примере 1.6 показано, как создать тензор декларативным образом, с помощью списков языка Python.

**Пример 1.6.** Создание и инициализация тензора значениями из списков

Input[0]

```
x = torch.Tensor([[1, 2, 3],
                  [4, 5, 6]])
describe(x)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])
```

Значения можно брать как из списка, как в предыдущем примере, так и из массива NumPy. И конечно, тензор PyTorch всегда можно преобразовать в массив NumPy. Обратите внимание, что тип тензора — `DoubleTensor`, а не используемый по умолчанию `FloatTensor` (см. следующий раздел). Этот тип соответствует типу данных случайной матрицы NumPy — `float64`, — как показано в примере 1.7.

**Пример 1.7.** Создание и инициализация тензора значениями из массива NumPy

Input[0]

```
import torch
import numpy as np
np.random.rand(2, 3)
describe(torch.from_numpy(np.random.rand(2, 3)))
```

Output[0]

```
Type: torch.DoubleTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.8360,  0.8836,  0.0545],
        [ 0.6928,  0.2333,  0.7984]], dtype=torch.float64)
```

Возможность преобразования данных из массивов NumPy в тензоры PyTorch и наоборот играет важную роль при работе с унаследованными библиотеками, в которых используются числовые значения в формате NumPy.

## Типы и размер тензоров

У каждого тензора есть тип и размер. Тип тензора по умолчанию при использовании конструктора `torch.Tensor` — `torch.FloatTensor`. Однако существует возможность преобразовать тензор в другой тип (`float`, `long`, `double` и т. д.), указав его при инициализации или воспользовавшись потом одним из методов приведения типов. Существует два способа указания типа при инициализации: непосредственно вызвать конструктор конкретного типа тензора, например `FloatTensor` или `LongTensor`, или воспользоваться специальным методом `torch.tensor()`, добавив параметр `dtype` (пример 1.8).

**Пример 1.8.** Свойства тензоров

Input[0]

```
x = torch.FloatTensor([1, 2, 3],
                      [4, 5, 6])
describe(x)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])
```

Input[1] `x = x.long()  
describe(x)`

Output[1] `Type: torch.LongTensor  
Shape/size: torch.Size([2, 3])  
Values:  
tensor([[ 1, 2, 3],  
 [ 4, 5, 6]])`

Input[2] `x = torch.tensor([[1, 2, 3],  
 [4, 5, 6]], dtype=torch.int64)  
describe(x)`

Output[2] `Type: torch.LongTensor  
Shape/size: torch.Size([2, 3])  
Values:  
tensor([[ 1, 2, 3],  
 [ 4, 5, 6]])`

Input[3] `x = x.float()  
describe(x)`

Output[3] `Type: torch.FloatTensor  
Shape/size: torch.Size([2, 3])  
Values:  
tensor([[ 1., 2., 3.],  
 [ 4., 5., 6.]])`

Для получения информации о размерах измерений объекта-тензора используются его свойства `shape` и метод `size()`. Оба способа обращения к этим данным практически синонимичны. Проверка формы тензоров — незаменимый инструмент при отладке кода PyTorch.

## Операции над тензорами

После создания тензоров можно работать с ними так же, как с обычными типами языков программирования, с помощью операторов `+`, `-`, `*`, `/`. Вместо этих операторов можно использовать также соответствующие им функции, например `.add()`, как показано в примере 1.9.

**Пример 1.9.** Операции над тензорами: сложение

Input[0] `import torch  
x = torch.randn(2, 3)  
describe(x)`

Output[0]	Type: torch.FloatTensor Shape/size: torch.Size([2, 3]) Values: tensor([[ 0.0461,  0.4024, -1.0115], [ 0.2167, -0.6123,  0.5036]])
Input[1]	describe(torch.add(x, x))
Output[1]	Type: torch.FloatTensor Shape/size: torch.Size([2, 3]) Values: tensor([[ 0.0923,  0.8048, -2.0231], [ 0.4335, -1.2245,  1.0072]])
Input[2]	describe(x + x)
Output[2]	Type: torch.FloatTensor Shape/size: torch.Size([2, 3]) Values: tensor([[ 0.0923,  0.8048, -2.0231], [ 0.4335, -1.2245,  1.0072]])

Существуют также операции, которые можно применять к отдельным измерениям тензора. Как вы уже, наверное, заметили, в двумерных тензорах строки — это измерение 0, а столбцы — измерение 1 (пример 1.10).

**Пример 1.10.** Операции над отдельными измерениями тензоров<sup>1</sup>

Input[0]	import torch x = torch.arange(6) describe(x)
----------	--

<sup>1</sup> Операция `arange` служит для создания одномерных тензоров заданного размера со значениями, представляющими собой арифметическую прогрессию с указанным шагом (по умолчанию 1). Отметим также, что в зависимости от глобальных настроек типом по умолчанию возвращаемых `arange` значений может быть `LongTensor`. По этой причине некоторые из дальнейших примеров могут не работать в исходном виде и может понадобиться, например, указать в `arange` параметр `dtype` (см. <https://pytorch.org/docs/stable/torch.html?highlight=arange#torch.arange>). Операция `view` возвращает новый тензор с теми же данными, но другой формы (см. <https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view>). Операция `sum` возвращает свертку тензора по заданному измерению (см. <https://pytorch.org/docs/stable/torch.html#torch.sum>). Операция `transpose` транспонирует тензор, меняя местами два указанных измерения (см. <https://pytorch.org/docs/stable/torch.html#torch.transpose>). — *Примеч. пер.*

```
Output[0] Type: torch.FloatTensor
Shape/size: torch.Size([6])
Values:
tensor([ 0.,  1.,  2.,  3.,  4.,  5.])
```

```
Input[1] x = x.view(2, 3)
describe(x)
```

```
Output[1] Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
```

```
Input[2] describe(torch.sum(x, dim=0))
```

```
Output[2] Type: torch.FloatTensor
Shape/size: torch.Size([3])
Values:
tensor([ 3.,  5.,  7.])
```

```
Input[3] describe(torch.sum(x, dim=1))
```

```
Output[3] Type: torch.FloatTensor
Shape/size: torch.Size([2])
Values:
tensor([ 3., 12.])
```

```
Input[4] describe(torch.transpose(x, 0, 1))
```

```
Output[4] Type: torch.FloatTensor
Shape/size: torch.Size([3, 2])
Values:
tensor([[ 0.,  3.],
        [ 1.,  4.],
        [ 2.,  5.]])
```

Часто приходится выполнять над тензорами более сложные операции, включающие разнообразные комбинации доступа по индексам, срезов, объединения и перестановок элементов. Как и в NumPy и других библиотеках численного анализа, в PyTorch есть встроенные функции для упрощения подобных операций над тензорами.

## Обращение по индексу, срезы и объединение

Если вы используете NumPy, то способ обращения по индексам и выполнения срезов PyTorch, показанный в примере 1.11, должен быть хорошо вам знаком.

**Пример 1.11.** Выполнение срезов и обращение по индексу в тензоре

Input[0]

```
import torch
x = torch.arange(6).view(2, 3)
describe(x)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
```

Input[1]

```
describe(x[:1, :2])
```

Output[1]

```
Type: torch.FloatTensor
Shape/size: torch.Size([1, 2])
Values:
tensor([[ 0.,  1.]])
```

Input[2]

```
describe(x[0, 1])
```

Output[2]

```
Type: torch.FloatTensor
Shape/size: torch.Size([])
Values:
1.0
```

Пример 1.12 демонстрирует, что в PyTorch есть также функции для сложных операций доступа по индексам и выполнения срезов на тот случай, если вам потребуется эффективно обращаться к несмежным участкам тензора.

**Пример 1.12.** Сложный доступ по индексам: обращение по индексам к несмежным участкам тензора

Input[0]

```
indices = torch.LongTensor([0, 2])
describe(torch.index_select(x, dim=1, index=indices))
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 0.,  2.],
        [ 3.,  5.]])
```

```
Input[1] indices = torch.LongTensor([0, 0])
describe(torch.index_select(x, dim=0, index=indices))
```

```
Output[1] Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 0.,  1.,  2.]])
```

```
Input[2] row_indices = torch.arange(2).long()
col_indices = torch.LongTensor([0, 1])
describe(x[row_indices, col_indices])
```

```
Output[2] Type: torch.FloatTensor
Shape/size: torch.Size([2])
Values:
tensor([ 0.,  4.])
```

Обратите внимание, что тип индексов — `LongTensor`; таковы требования к доступу по индексу при использовании функций PyTorch. Можно также выполнять объединение тензоров с помощью встроенных функций конкатенации (пример 1.13), указывая тензоры и нужные измерения.

### Пример 1.13. Конкатенация тензоров

```
Input[0] import torch
x = torch.arange(6).view(2,3)
describe(x)
```

```
Output[0] Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
```

```
Input[1] describe(torch.cat([x, x], dim=0))
```

```
Output[1] Type: torch.FloatTensor
Shape/size: torch.Size([4, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
```

```
Input[2] describe(torch.cat([x, x], dim=1))
```

```
Output[2] Type: torch.FloatTensor
Shape/size: torch.Size([2, 6])
Values:
tensor([[ 0.,  1.,  2.,  0.,  1.,  2.],
        [ 3.,  4.,  5.,  3.,  4.,  5.]])
```

```
Input[3] describe(torch.stack([x, x]))
```

```
Output[3] Type: torch.FloatTensor
Shape/size: torch.Size([2, 2, 3])
Values:
tensor([[[ 0.,  1.,  2.],
          [ 3.,  4.,  5.]],
        [[ 0.,  1.,  2.],
          [ 3.,  4.,  5.]])
```

В PyTorch также реализованы высокопроизводительные операции линейной алгебры: умножение, вычисление обратного элемента и следа тензора (второго ранга. — *Примеч. пер.*), как показано в примере 1.14.

**Пример 1.14.** Операции линейной алгебры над тензорами

```
Input[0] import torch
x1 = torch.arange(6).view(2, 3)
describe(x1)
```

```
Output[0] Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
```

```
Input[1] x2 = torch.ones(3, 2)
x2[:, 1] += 1
describe(x2)
```

```
Output[1] Type: torch.FloatTensor
Shape/size: torch.Size([3, 2])
Values:
tensor([[ 1.,  2.],
        [ 1.,  2.],
        [ 1.,  2.]])
```

```
Input[2] describe(torch.mm(x1, x2))
```



Output[2]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 3.,  6.],
        [12., 24.]])
```

До сих пор мы говорили о способах создания и работы с константными объектами-тензорами PyTorch. Как и переменные языков программирования (например, Python), инкапсулирующие элементы данных и несущие об этих данных дополнительную информацию (в частности, адрес ячейки памяти, в которой они хранятся), тензоры PyTorch берут на себя все вспомогательные операции, необходимые для построения графов вычислений для машинного обучения. Для этого достаточно установить в момент создания экземпляра тензора соответствующий булев флаг.

## Тензоры и графы вычислений

Класс `tensor` библиотеки PyTorch инкапсулирует данные (сам тензор) и целый ряд операций, таких как алгебраические операции, доступ по индексам и операции изменения формы. Однако, как показано в примере 1.15, установка для тензора флага `requires_grad`, равного `true` делает возможным отслеживание градиента тензора, а заодно и функцию вычисления градиента — оба необходимы для машинного обучения на основе градиентного спуска, описанного в разделе «Парадигма обучения с учителем» на с. 20.

**Пример 1.15.** Создание тензоров для вспомогательных операций работы с градиентами

Input[0]

```
import torch
x = torch.ones(2, 2, requires_grad=True)
describe(x)
print(x.grad is None)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
True
```

Input[1]

```
y = (x + 2) * (x + 5) + 3
describe(y)
print(x.grad is None)
```

Output[1]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 2])
Values:
tensor([[ 21.,  21.],
        [ 21.,  21.]])
True
```

Input[2]

```
z = y.mean()
describe(z)
z.backward()
print(x.grad is None)
```

Output[2]

```
Type: torch.FloatTensor
Shape/size: torch.Size([])
Values:
21.0
False
```

Когда тензор создается с параметром `requires_grad=True`, тем самым от PyTorch требуется хранить вспомогательную информацию, необходимую для вычисления градиентов. Во-первых, PyTorch будет отслеживать значения, получаемые при прямом проходе. Затем, в конце вычислений, для определения обратного шага используется одно скалярное значение. Обратный шаг запускается путем вызова метода `backward()` для тензора, полученного в результате вычисления функции потерь. При обратном шаге вычисляется значение градиента для объекта-тензора, участвовавшего в прямом шаге.

В целом градиент представляет собой значение, отражающее угол наклона выходного значения функции по отношению к ее входному значению. В сфере графов вычислений градиенты существуют для всех параметров модели, их можно считать вкладом каждого параметра в сигнал рассогласования. В PyTorch получить доступ к градиентам узлов графа вычислений можно с помощью переменной экземпляра `.grad`. Она используется при оптимизации для обновления значений параметров.

## Тензоры CUDA

До сих пор наши тензоры располагались в памяти CPU. При выполнении операций линейной алгебры имеет смысл воспользоваться GPU, если в вашей системе он есть. Для применения GPU необходимо сначала выделить под тензор место в памяти GPU. Доступ к GPU производится с помощью специализированного API — CUDA. API CUDA был создан компанией NVIDIA, его можно использо-

вать только для GPU производства NVIDIA<sup>1</sup>. В PyTorch есть объекты-тензоры для CUDA, неотличимые на практике от обычных CPU-тензоров, за исключением внутреннего механизма выделения памяти.

PyTorch обеспечивает простоту создания этих CUDA-тензоров, поддерживая перемещение тензоров с CPU на GPU с сохранением их базового типа. В PyTorch рекомендуется применять *аппаратно независимый подход* и писать код, который бы работал как на CPU, так и на GPU. В примере 1.16 мы сначала проверяем, доступен ли GPU, с помощью функции `torch.cuda.is_available()` и извлекаем название устройства, вызвав метод `torch.device()`. Далее мы создаем все будущие тензоры и перемещаем их на нужное устройство с помощью метода `.to(device)`.

**Пример 1.16.** Создание CUDA-тензоров

Input[0]	<pre>import torch print (torch.cuda.is_available())</pre>
Output[0]	True
Input[1]	<pre># предпочтительный метод: аппаратно-независимое # создание экземпляров тензоров device = torch.device("cuda" if torch.cuda.is_available() else "cpu") print (device)</pre>
Output[1]	cuda
Input[2]	<pre>x = torch.rand(3, 3).to(device) describe(x)</pre>
Output[2]	<pre>Type: torch.cuda.FloatTensor Shape/size: torch.Size([3, 3]) Values: tensor([[ 0.9149,  0.3993,  0.1100],         [ 0.2541,  0.4333,  0.4451],         [ 0.4966,  0.7865,  0.6604]], device='cuda:0')</pre>

Чтобы работать с CUDA- и не-CUDA-объектами, необходимо гарантировать, что они располагаются на одном устройстве. В противном случае попытка вычислений завершится ошибкой, как показано в примере 1.17. Подобная ситуация возникает,

<sup>1</sup> За прошедшее с момента написания книги время ситуация изменилась к лучшему. См. <https://github.com/pytorch/pytorch/issues/488> и <https://github.com/pytorch/pytorch/pull/6625>. — *Примеч. пер.*

например, при работе с метриками мониторинга вычислений, не включенными в граф вычислений. При работе с двумя объектами-тензорами убедитесь, что они располагаются на одном устройстве.

**Пример 1.17.** Совместное использование CUDA- и CPU-тензоров

Input[0]	<pre>y = torch.rand(3, 3) x + y</pre>
Output[0]	<pre>----- RuntimeError                                Traceback (most recent call last)       1 y = torch.rand(3, 3) ----&gt; 2 x + y  RuntimeError: Expected object of type torch.cuda.FloatTensor but found type torch.FloatTensor for argument #3 'other'</pre>
Input[1]	<pre>cpu_device = torch.device("cpu") y = y.to(cpu_device) x = x.to(cpu_device) x + y</pre>
Output[1]	<pre>tensor([[ 0.7159,  1.0685,  1.3509],         [ 0.3912,  0.2838,  1.3202],         [ 0.2967,  0.0420,  0.6559]])</pre>

Помните, что перемещение данных из/в GPU — процесс весьма ресурсозатратный. Следовательно, обычно выполняется множество допускающих распараллеливание вычислений на GPU, а обратно в CPU перемещается только итоговый результат. Благодаря этому можно полностью задействовать возможности имеющихся GPU. Если у вас в системе есть несколько видимых для CUDA устройств (то есть несколько GPU), разумнее всего будет воспользоваться при выполнении программы переменной среды `CUDA_VISIBLE_DEVICES`, как показано здесь:

```
CUDA_VISIBLE_DEVICES=0,1,2,3 python main.py
```

Мы не станем обсуждать в этой книге параллелизм и обучение на нескольких GPU, но они очень важны для экспериментов по масштабированию (scaling) и иногда даже при обучении больших моделей. Рекомендуем вам заглянуть в документацию PyTorch и на дискуссионные форумы за дополнительной информацией по этому вопросу.

## Упражнения

Решение задач — лучший способ разобраться в чем-либо. Вот несколько разминочных упражнений. Большинство из этих задач требует обращения к официальной документации и поиска подходящих функций.

1. Создайте двумерный тензор, после чего добавьте в него новое измерение размером 1 перед измерением 0.
2. Удалите дополнительное измерение, которое вы только что добавили в предыдущий тензор.
3. Создайте случайный тензор формой  $5 \times 3$  в интервале  $[3, 7)$ .
4. Создайте тензор со значениями, взятыми из нормального распределения (математическое ожидание = 0, стандартное отклонение = 1).
5. Извлеките индексы всех ненулевых элементов из тензора `torch.Tensor([1, 1, 1, 0, 1])`.
6. Создайте случайный тензор размером  $(3, 1)$ , а затем горизонтально разместите в ряд четыре его копии.
7. Верните пакетное произведение двух трехмерных матриц (`a=torch.rand(3,4,5)`, `b=torch.rand(3,5,4)`).
8. Верните пакетное произведение трехмерной и двумерной матриц (`a=torch.rand(3,4,5)`, `b=torch.rand(5,4)`).

## Решения

1. `a = torch.rand(3, 3)`  
`a.unsqueeze(0)`
2. `a.squeeze(0)`
3. `3 + torch.rand(5, 3) * (7 - 3)`
4. `a = torch.rand(3, 3)`  
`a.normal_()`
5. `a = torch.Tensor([1, 1, 1, 0, 1])`  
`torch.nonzero(a)`
6. `a = torch.rand(3, 1)`  
`a.expand(3, 4)`
7. `a = torch.rand(3, 4, 5)`  
`b = torch.rand(3, 5, 4)`  
`torch.bmm(a, b)`
8. `a = torch.rand(3, 4, 5)`  
`b = torch.rand(5, 4)`  
`torch.bmm(a, b.unsqueeze(0).expand(a.size(0), *b.size()))`

## Резюме

В этой главе мы познакомили вас с основными темами книги — обработкой текстов на естественных языках (NLP) и глубоким обучением — и рассмотрели все детали парадигмы обучения с учителем. Теперь вы уже должны хорошо разбираться в таких понятиях, как наблюдаемые величины, целевые переменные, модели, параметры, предсказания, функция потерь, представления, обучение и вывод (или по крайней мере знать об их существовании). Мы также продемонстрировали вам, как кодировать входные данные (наблюдаемые величины и целевые переменные) для задач обучения с помощью унитарного кодирования. Кроме того, вы посмотрели на представления на основе подсчетов, такие как TF и TF-IDF. Мы начали наше знакомство с PyTorch с графов вычислений, затем сравнили статические графы вычислений с динамическими и обсудили операции PyTorch для работы с тензорами. В главе 2 вас ждет обзор традиционного NLP. Если вы новичок в тематике данной книги, то эти две главы станут для вас необходимым фундаментом для оставшихся глав.

## Библиография

1. Официальная документация по API PyTorch: <http://bit.ly/2RjBxVw>.
2. *Dougherty J., Kohavi R., Sahami M.* Supervised and Unsupervised Discretization of Continuous Features // Proceedings of the 12th International Conference on Machine Learning, 1995.
3. *Collobert R., Weston J.* A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning // Proceedings of the 25th International Conference on Machine Learning, 2008.

# 2 Краткое знакомство с NLP

Обработка написанных на естественном языке текстов (natural language processing, NLP), с которой мы познакомились в предыдущей главе, и *вычислительная лингвистика* (computational linguistics, CL) — две сферы вычислительных методов изучения естественных языков. NLP занимается разработкой методов решения связанных с языками практических задач, таких как извлечением информации, автоматическим распознаванием речи, машинным переводом, анализом тональности высказываний, формированием ответов на вопросы и автоматическим реферированием. CL, с другой стороны, применяет вычислительные методы для постижения свойств естественных языков. Как мы понимаем язык? Как мы создаем языки? Как мы учим языки? Как различные языки связаны друг с другом?

В литературе методы часто переходят из одной сферы в другую, из CL в NLP и наоборот. Полученные в CL знания о языках могут применяться для формирования априорных распределений в NLP, а методы статистики и машинного обучения из NLP могут использоваться для поиска ответов на актуальные для CL вопросы. Фактически некоторые из этих вопросов разрослись до размеров отдельных областей знания, например таких, как фонология, морфология, синтаксис, семантика и прагматика.

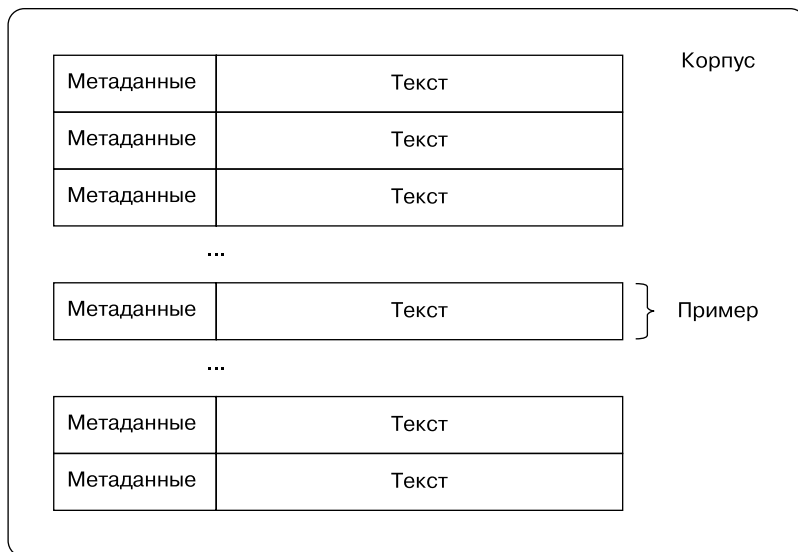
Эта книга посвящена только NLP, но мы будем регулярно, по мере необходимости, заимствовать идеи из CL. Прежде чем с головой нырнуть в методы нейронных сетей — основную тему оставшихся глав — стоит вспомнить некоторые обычные понятия и методы NLP. Этому и посвящена глава 2.

Если у вас уже есть базовые знания о NLP, то можете пропустить эту главу, хотя лучше просмотреть ее из соображений ностальгии и чтобы согласовать терминологию на будущее.

## Корпусы текстов, токены и типы

Все методы NLP, классические и современные, начинаются с набора текстовых данных, называемого также *корпусом* (corpus). Корпус обычно состоит из неформатированного текста (в кодировках ASCII или UTF-8) и соответствующих тексту метаданных. Неформатированный текст состоит из символов (байтов), но обычно бывает удобно группировать эти символы в непрерывные единицы данных — *токены*, или *лексемы*<sup>1</sup> (tokens). В английском языке токены соответствуют последовательностям слов и чисел, разделяемых пробелами или знаками пунктуации.

Метаданные могут представлять собой произвольные вспомогательные элементы информации, относящиеся к этому тексту, например метки, идентификаторы и метки даты/времени. В терминологии машинного обучения текст вместе с метаданными называется *примером* (*экземпляром*) (instance) или *точкой данных* (data point). Корпус (рис. 2.1) — набор примеров — называется также *набором данных* (dataset). С учетом ориентации данной книги на машинное обучение мы будем попеременно использовать в ней термины «корпус» и «набор данных».



**Рис. 2.1.** Корпус: отправной пункт задач NLP

<sup>1</sup> В простых случаях понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор, оператор», «часть речи» и т. п.).



Процесс разбиения текста на токены называется *токенизацией* (tokenization). Например, во фразе на языке эсперанто “*Maria frapis la verda sorĉistino.*”<sup>1</sup> содержится шесть токенов. Токенизация может оказаться намного сложнее обычного разбиения текста по символам, не являющимся алфавитно-цифровыми, как показано на рис. 2.2. Для таких агглютинативных языков, как турецкий, разбиения по пробелам и знакам препинания может оказаться недостаточно и могут понадобиться специализированные методики. Как вы увидите в главах 4 и 6, в некоторых видах нейронных сетей можно полностью обойти проблемы с токенизацией за счет представления текста в виде потока байтов; для агглютинативных языков это очень важно.

Turkish	English
kork(-mak)	(to) fear
korku	fear
korkusuz	fearless
korkusuzlaş (-mak)	(to) become fearless
korkusuzlaşmış	One who has become fearless
korkusuzlaştır(-mak)	(to) make one fearless
korkusuzlaştırıl(-mak)	(to) be made fearless
korkusuzlaştırılmış	One who has been made fearless
korkusuzlaştırılabil(-mek)	(to) be able to be made fearless
korkusuzlaştırılabilecek	One who will be able to be made fearless
korkusuzlaştırabileceklerimiz	Ones who we can make fearless
korkusuzlaştırabileceklerimizden	From the ones who we can make fearless
korkusuzlaştırabileceklerimizdenmiş	I gather that one is one of those we can make fearless
korkusuzlaştırabileceklerimizdenmişçesine	As if that one is one of those we can make fearless
korkusuzlaştırabileceklerimizdenmişçesineyken	when it seems like that one is one of those we can make fearless

**Рис. 2.2.** Сложность токенизации в таких языках, как турецкий, быстро растет

<sup>1</sup> На английском эта фраза звучит как *Mary slapped the green witch* («Мэри шлепнула зеленую ведьму»). Мы будем постоянно использовать это предложение в качестве примера в данной главе. Не подумайте, что мы пропагандируем насилие, этот пример — наш реверанс в сторону самого известного современного учебника по искусственному интеллекту Рассела и Норвига (Russell, Norvig, 2016), в котором он также постоянно используется в качестве примера.

Наконец, рассмотрим следующий твит.



Токенизация твитов требует сохранения хештегов и дескрипторов имен пользователей, а также трэктовки смайликов, таких как :-), и URL в качестве одного элемента. Следует ли рассматривать хештег #MakeAMovieCold как один токен или четыре? Большинство научных статей не уделяет особого внимания подобным вопросам, да и вообще, правила токенизации зачастую выбираются произвольно, хотя на практике выбор влияет на точность больше, чем можно предположить. Большинство пакетов NLP с открытым исходным кодом учитывают сложность работы по предварительной обработке данных и предоставляют неплохие инструменты для токенизации. Пример 2.1 демонстрирует листинги из двух часто применяемых пакетов обработки текстов — NLTK (<http://www.nltk.org/>) и spaCy<sup>1</sup> (<https://spacy.io/>).

### Пример 2.1. Токенизация текста

Input[0]	<pre>import spacy nlp = spacy.load('en') text = "Mary, don't slap the green witch" print([str(token) for token &gt;in nlp(text.lower())])</pre>
Output[0]	<pre>['mary', ',', ' ', 'do', 'n't', 'slap', 'the', 'green', 'witch', '.']</pre>
Input[1]	<pre>from nltk.tokenize import TweetTokenizer tweet=u"Snow White and the Seven Degrees       #MakeAMovieCold@midnight:-)" tokenizer = TweetTokenizer() print(tokenizer.tokenize(tweet.lower()))</pre>
Output[1]	<pre>['snow', 'white', 'and', 'the', 'seven', 'degrees',  '#makeamoviecold', '@midnight', ':-)']</pre>

Уникальные токены из корпуса называются *типами*. Множество всех типов в корпусе называется его *словарем* (vocabulary) или *лексиконом* (lexicon). Слова делятся на *значимые* (content words) и *стоп-слова* (stopwords). Такие стоп-слова,

<sup>1</sup> Для их использования понадобится не только установить соответствующие пакеты, но и отдельно загрузить модель en для spaCy: `python -m spacy download en`. — *Примеч. пер.*

как артикли и предлоги, служат в основном грамматическим целям, например используются в качестве заполнителей между значимыми словами.

### ПРОЕКТИРОВАНИЕ ПРИЗНАКОВ

Процесс лингвистического анализа языка и применения полученных значений к решению NLP-задач называется *проектированием признаков* (feature engineering). Мы постараемся как можно меньше касаться в книге этого вопроса ради удобства читателя и лучшей переносимости моделей между разными языками. Но при создании и развертывании реальных, работающих в промышленной эксплуатации систем проектирование признаков незаменимо, хотя в последнее время часто утверждают обратное. Общее введение в проектирование признаков вы можете найти в книге Чжэна и Казари (Zheng, Casari, 2016).

## Униграммы, биграммы, триграммы... $n$ -граммы

$N$ -граммы — последовательности фиксированной длины ( $n$ ) непрерывно следующих друг за другом токенов, встречающихся в тексте. Биграмма состоит из двух токенов, униграмма — из одного. Генерация  $n$ -грамм на основе текста — достаточно простая задача (пример 2.2), и пакеты NLTK и spaCy предоставляют удобные методы для этой цели.

**Пример 2.2.** Генерация  $n$ -грамм на основе текста

Input[0]

```
def n_grams(text, n):
    """
    Принимает на входе токены или текст, возвращает список  $n$ -грамм
    """
    return [text[i:i+n] for i in range(len(text)-n+1)]

cleaned = ['mary', ',', 'n't', 'slap', 'green', 'witch', '.']
print(n_grams(cleaned, 3))
```

Output[0]

```
[['mary', ',', 'n't'],
 [' ', 'n't', 'slap'],
 ['n't', 'slap', 'green'],
 ['slap', 'green', 'witch'],
 ['green', 'witch', '.']]
```

В некоторых случаях, когда полезная информация содержится и в подсловах, может оказаться удобно сгенерировать  $n$ -граммы из символов. Например, суффикс «-ол» в слове «метанол» указывает, что речь идет о спирте; если задача включает классификацию органических соединений, то очевидно, что информация о захваченных  $n$ -граммами подсловах может оказаться полезной. В подобных

случаях можно переиспользовать тот же код, но уже рассматривая каждый из символов как токен<sup>1</sup>.

## Леммы и основы слов

*Леммы* (lemmas) — корневые формы слов. Рассмотрим слово *fly*. Склоняя его, можно получить множество различных слов — *flow, flew, flies, flown, flowing* и т. д., — и для всех этих, казалось бы, различных слов *fly* является леммой. Иногда бывает полезно свернуть токены до соответствующих лемм ради понижения размерности векторного представления. Такая свертка называется *лемматизацией* (lemmatization), посмотреть на нее в действии вы можете в примере 2.3.

**Пример 2.3.** Лемматизация: свертка слов до корневых форм<sup>2</sup>

Input[0]

```
import spacy
nlp = spacy.load('en')
doc = nlp(u"he was running late")
for token in doc:
    print('{} --> {}'.format(token, token.lemma_))
```

Output[0]

```
he --> he
was --> be
running --> run
late --> late
```

В spaCy, например, для извлечения лемм применяется встроенный словарь под названием WordNet, но лемматизацию можно рассматривать как задачу машинного обучения, для решения которой требуется понимание морфологии языка.

*Стемминг*, или *выделение основы* (stemming) слова, — лемматизация «для бедных»<sup>3</sup>. Она включает получение общих форм (*основ*) слов путем использования самодельных правил для обрезания окончаний. В пакетах с открытым исходным кодом часто реализуются популярные стеммеры, включая стеммер Портера и SnowBall. Поиск нужных API spaCy/NLTK для стемминга оставим в качестве упражнения читателю.

<sup>1</sup> В главах 4 и 6 мы рассмотрим модели глубокого обучения, неявно схватывающие, причем достаточно эффективно, такую подструктуру.

<sup>2</sup> Не удивляйтесь, если первая строка полученных вами результатов будет отличаться от приведенной здесь и выглядеть так: he --> -PRON-.

С недавних пор spaCy свертывает все местоимения в специальный токен -PRON- (от англ. pronoun — «местоимение»). См. <https://spacy.io/api/annotation/#lemmatization>. — *Примеч. пер.*

<sup>3</sup> Рассмотрим разницу между выделением основы и лемматизацией на примере слова «гуси». В результате лемматизации получается «гусь», а после стемминга — «гус».

## Категоризация предложений и документов

Категоризация или классификации документов — одно из самых первых приложений NLP. Описанные в главе 1 представления TF и TF-IDF можно непосредственно применять для классификации и категоризации длинных фрагментов текста, таких как документы или предложения. Задачи вроде присвоения меток тем, предсказания тональностей обзоров, фильтрации спама в электронной почте, идентификации языков и сортировки электронной почты можно рассматривать как задачи классификации документов с учителем (очень удобны варианты с частичным обучением, когда используется лишь небольшой маркированный набор данных, но эта тема выходит за рамки данной книги).

## Категоризация слов: маркирование частей речи

Концепцию маркирования можно распространить с документов на отдельные слова или токены. Распространенный пример категоризации слов — маркирование частей речи (part-of-speech, POS), показанное в примере 2.4.

**Пример 2.4.** Маркирование частей речи

Input[0]

```
import spacy
nlp = spacy.load('en')
doc = nlp(u"Mary slapped the green witch.")
for token in doc:
    print('{ } - {}'.format(token, token.pos_))
```

Output[0]

```
Mary - PROP
slapped - VERB
the - DET
green - ADJ
witch - NOUN
. - PUNCT
```

## Категоризация отрезков текста: разбивка на порции и распознавание поименованных сущностей

Довольно часто бывает нужно маркировать отрезок текста, то есть непрерывную границу мультитокена (multitoken boundary). Например, рассмотрим предложение *Mary slapped the green witch*. Пусть необходимо выделить в нем именные (noun phrases, NP) и глагольные группы (verb phrases, VP):

[NP Mary] [VP slapped] [the green witch].

Это называется *разбивкой на порции* (chunking) или *поверхностным синтаксическим разбором* (shallow parsing). Задача поверхностного синтаксического разбора состоит в извлечении более высокоуровневых единиц, состоящих из грамматических «атомов», таких как существительные, глаголы, прилагательные и т. д. При отсутствии данных для обучения модели с целью поверхностного синтаксического разбора можно написать регулярное выражение для сопоставления с тегами частей речи для аппроксимации поверхностного синтаксического разбора. К счастью, для английского и большинства других распространенных языков уже существуют подобные данные и заранее обученные модели. В примере 2.5 показан поверхностный синтаксический разбор с помощью spaCy.

**Пример 2.5.** Разбивка на порции по именным группам

Input[0]

```
import spacy
nlp = spacy.load('en')
doc = nlp(u"Mary slapped the green witch.")
for chunk in doc.noun_chunks:
    print ('{} - {}'.format(chunk, chunk.label_))
```

Output[0]

```
Mary - NP
the green witch - NP
```

Еще один удобный тип отрезков текста: *поименованная сущность* (named entity). Поименованная сущность — символьное значение, упоминающее понятие реального мира, например человека, местоположение, организацию, название лекарства и т. д. Вот пример:

John **PERSON** was born in Chicken **GPE**, Alaska **GPE**, and studies at Cranberry Lemon University **ORG**.

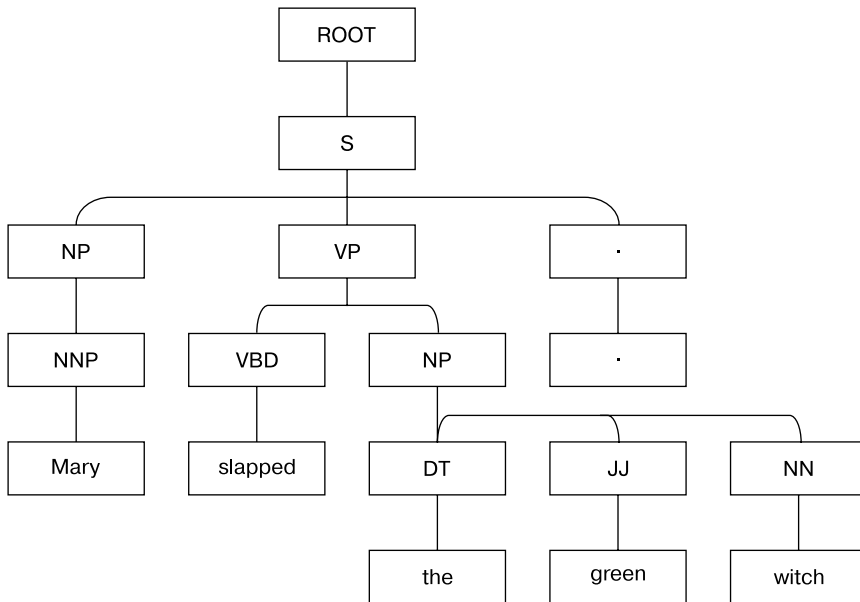
## Структура предложений

При поверхностном синтаксическом разборе в тексте распознаются фразовые единицы, а задача просто синтаксического разбора состоит в распознавании взаимосвязей между ними. Вам, наверное, приходилось в школе рисовать схемы предложений, подобные показанной<sup>1</sup> на рис. 2.3.

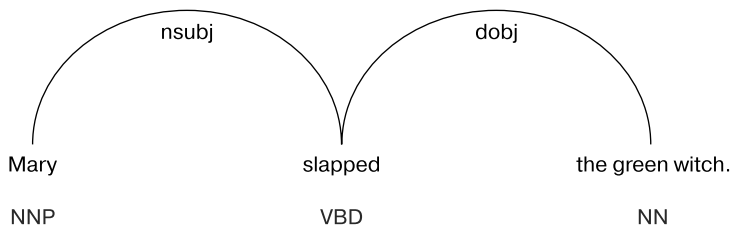
Деревья синтаксического разбора демонстрируют связи различных грамматических единиц в предложении. Дерево синтаксического разбора на рис. 2.3 иллюстрирует так называемый *разбор на составляющие* (constituent parse). Еще один, возможно,

<sup>1</sup> Полный список обозначений можно найти по адресу <https://spacy.io/api/annotation/#pos-tagging>. — *Примеч. пер.*

даже более удобный способ демонстрации взаимосвязей — применение синтаксического разбора зависимостей (dependency parsing), показанного на рис. 2.4.



**Рис. 2.3.** Разбор на составляющие предложения Mary slapped the green witch



**Рис. 2.4.** Разбор зависимостей предложения Mary slapped the green witch

Больше о традиционном синтаксическом разборе вы можете узнать из литературы, указанной в разделе «Библиография» в конце главы.

## Смысл и семантика слов

У слов есть значение, и зачастую не одно. Различные значения слова называются его *смыслами* (senses). WordNet (<http://bit.ly/2CQNVYX>) — лексическая база данных английского языка, разработанная в Принстонском университете и выпущенная

вместе с сопутствующим программным обеспечением. Предназначена для каталогизации всех смыслов всех (ну, практически) слов английского языка вместе с другими лексическими связями<sup>1</sup>. Например, рассмотрим слово *plane*. На рис. 2.5 показаны различные его смыслы.

Word to search for:

Display Options:

Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations  
 Display options for sense: (gloss) "an example sentence"

**Noun**

- S: (n) airplane, aeroplane, **plane** (an aircraft that has a fixed wing and is powered by propellers or jets) *"the flight was delayed due to trouble with the airplane"*
- S: (n) **plane**, sheet ((mathematics) an unbounded two-dimensional shape) *"we will refer to the plane of the graph as the X-Y plane"; "any line joining two points on a plane lies wholly on that plane"*
- S: (n) **plane** (a level of existence or development) *"he lived on a worldly plane"*
- S: (n) **plane**, planer, planing machine (a power tool for smoothing or shaping wood)
- S: (n) **plane**, carpenter's plane, woodworking plane (a carpenter's hand tool with an adjustable blade for smoothing or shaping wood) *"the cabinetmaker used a plane for the finish work"*

**Verb**

- S: (v) **plane**, shave (cut or remove with or as if with a plane) *"The machine shaved off fine layers from the piece of wood"*
- S: (v) **plane**, skim (travel on the surface of water)
- S: (v) **plane** (make even or smooth, with or as with a carpenter's plane) *"plane the top of the door"*

**Adjective**

- S: (adj) flat, level, **plane** (having a surface without slope, tilt in which no part is higher or lower than another) *"a flat desk"; "acres of level farmland"; "a plane surface"; "skirts sewn with fine flat seams"*

Рис. 2.5. Смыслы слова *plane* (любезно предоставлено WordNet)

Усилия, десятилетиями вкладываемые в такие проекты, как WordNet, заслуживают того, чтобы обратить на эти проекты внимание, несмотря на существование более современных подходов. В последующих главах книги вас ожидают примеры использования существующих лингвистических ресурсов в контексте нейронных сетей и методов глубокого обучения.

<sup>1</sup> Предпринимаются и попытки создания многоязычных версий WordNet. См., например, проект BabelNet (<http://babelnet.org/>).



Смыслы слов можно также выяснить из контекста — автоматическое обнаружение смыслов слов было самым первым методом частичного обучения, использовавшимся для NLP. И хотя мы не касались здесь этого вопроса, рекомендуем прочитать главу 17 книги Юравски и Мартина (Jurafsky, Martin, 2014) и главу 7 книги Мэннинга и Шютце (Manning, Schütze, 1999).

## Резюме

В этой главе мы рассмотрели основную терминологию и идеи NLP, которые могут пригодиться в следующих главах. Глава охватывает лишь малую толику возможностей NLP. Мы опустили некоторые важные аспекты, поскольку хотим посвятить большую часть книги использованию для NLP глубокого обучения. Впрочем, важно отдавать себе отчет в существовании огромного пласта научно-исследовательских работ по NLP, не использующих нейронных сетей и тем не менее оказывающих большое влияние на промышленность (то есть широко задействуемых при создании промышленных систем). Во многих случаях подходы с применением нейронных сетей следует считать вспомогательными, а не заменой традиционных методов. Опытные специалисты-практики часто берут все самое лучшее из обоих подходов для создания передовых систем. Чтобы узнать больше о традиционных подходах к NLP, рекомендуем обратиться к источникам литературы, приведенным в следующем разделе.

## Библиография

1. *Manning C. D., Schütze H.* Foundations of Statistical Natural Language Processing. MIT press, 1999.
2. *Bird S., Klein E., Loper E.* Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit. O'Reilly, 2009.
3. *Smith N. A.* Linguistic Structure Prediction. Morgan and Claypool, 2011.
4. *Jurafsky D., Martin J. H.* Speech and Language Processing, Vol. 3. Pearson, 2014.
5. *Russell S. J., Norvig P.* Artificial Intelligence: A Modern Approach. Pearson, 2016.
6. *Zheng A., Casari A.* Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists. O'Reilly, 2018.

# 3

## Базовые компоненты нейронных сетей

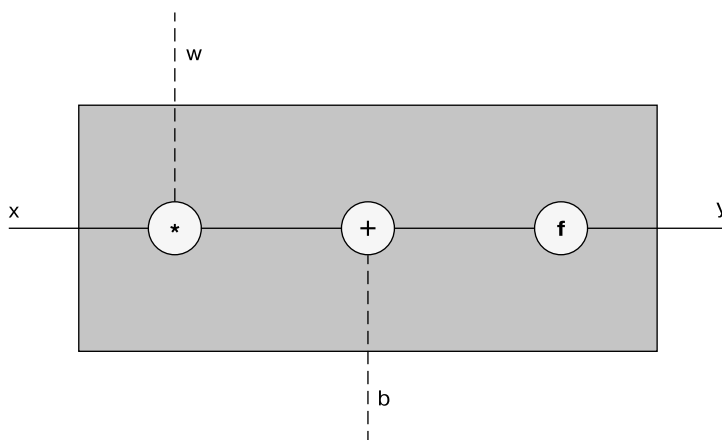
Эта глава закладывает фундамент для последующих, знакомя вас с основными концепциями создания нейронных сетей, такими как функции активации (activation functions), функции потерь, оптимизаторы и параметры машинного обучения с учителем. Мы начнем с перцептрона — моноблочной нейронной сети, чтобы продемонстрировать взаимосвязь различных понятий. Перцептроны служат стандартными блоками при создании более сложных нейронных сетей. Вы постоянно будете сталкиваться с подобным паттерном на протяжении книги — все обсуждаемые нами архитектуры и сети могут применяться либо сами по себе, либо в качестве элементов других сложных сетей. Этот блочный характер станет понятнее при обсуждении нами графов вычислений и других вопросов в книге.

### Перцептрон: простейшая нейронная сеть

Простейший блок нейронных сетей — *перцептрон* (perceptron). Изначально перцептрон был создан как очень отдаленная модель биологического нейрона. Как и у биологического нейрона, у него есть вход и выход, а также поток «сигналов», идущих от входа к выходу, как показано на рис. 3.1.

У каждого блока перцептрона есть вход ( $x$ ), выход ( $y$ ) и три «рычага управления»: веса ( $w$ ), смещение ( $b$ ) и функция активации ( $f$ ). Веса и смещение выясняют путем обучения на данных, функция активации подбирается исходя из интуитивных представлений создателя сети о ней и целевых выходных данных. Математически это можно выразить следующим образом:

$$y = f(wx + b).$$



**Рис. 3.1.** Граф вычислений для перцептрона с входом ( $x$ ) и выходом ( $y$ ). Параметры модели составляют веса ( $w$ ) и смещение ( $b$ )

Обычно у перцептрона несколько входных значений. Для представления такого общего случая можно воспользоваться векторами. То есть  $x$  и  $w$  — векторы, а произведение  $x$  и  $w$  в вышеприведенной формуле заменяется их скалярным произведением:

$$y = f(\mathbf{wx} + \mathbf{b}).$$

Функция активации, обозначенная здесь  $f$ , обычно нелинейная. График линейной функции представляет собой прямую. В этом примере  $\mathbf{wx} + \mathbf{b}$  — линейная функция. Так что, по существу, перцептрон — композиция линейной и нелинейной функций. Линейное выражение  $\mathbf{wx} + \mathbf{b}$  называют также *аффинным преобразованием* (affine transform).

В примере 3.1 представлена реализация перцептрона с помощью фреймворка PyTorch, которая принимает произвольное число входных данных, выполняет аффинное преобразование, применяет функцию активации и генерирует одно выходное значение.

**Пример 3.1.** Реализация перцептрона с помощью PyTorch

```
import torch
import torch.nn as nn
class Perceptron(nn.Module):
    """ Перцептрон представляет собой один линейный слой """
    def __init__(self, input_dim):
        """
        Аргументы:
            input_dim (int): размер вектора входных признаков
        """
        super(Perceptron, self).__init__()
```

```

self.fc1 = nn.Linear(input_dim, 1)

def forward(self, x_in):
    """ Прямой проход перцептрона

    Аргументы:
        x_in (torch.Tensor): тензор входных данных
        x_in.shape должен быть равен (batch, num_features)
    Возвращает:
        Итоговый тензор. tensor.shape должен быть равен (batch,).
    """
    return torch.sigmoid(self.fc1(x_in)).squeeze()

```

В модуле `torch.nn` фреймворка PyTorch есть удобный класс `Linear`, берущий на себя всю «бухгалтерию» весов и смещений, а также выполнение нужного аффинного преобразования<sup>1</sup>. В подразделе «Углубляемся в обучение с учителем» на с. 68, вы прочитаете, как узнать значения весов  $w$  и смещения  $b$  путем обучения на данных. В предыдущем примере в качестве функции активации использовалась *сигма-функция* (sigmoid function). В следующем разделе мы обсудим некоторые распространенные функции активации, включая и эту.

## Функции активации

Функции активации — нелинейности, вводимые в нейронную сеть для того, чтобы уловить сложные взаимосвязи данных. В подразделе «Углубляемся в обучение с учителем» на с. 68 и в разделе «Многослойный перцептрон» на с. 101 вы узнаете, зачем в машинном обучении нужны нелинейности, но сначала рассмотрим несколько часто используемых функций активации<sup>2</sup>.

### Сигма-функция

Сигма-функция — одна из первых функций активации в истории нейронных сетей. Она «размазывает» полученное вещественное значение по диапазону от 0 до 1. Математически сигма-функция описывается следующим образом:

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

<sup>1</sup> Значения весов и смещений обрабатываются внутри класса `nn.Linear`. Если по какой-то маловероятной причине вам понадобится модель без смещения, можно явным образом указать `bias=False` в конструкторе `nn.Linear`.

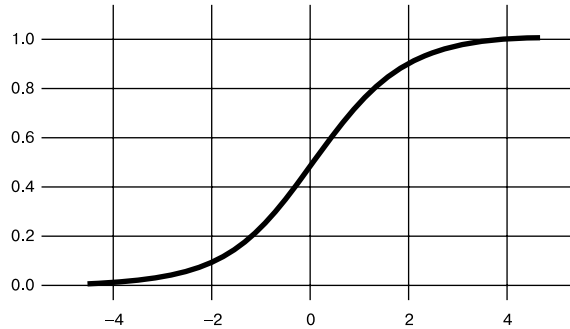
<sup>2</sup> Существует множество видов функций активации — в одном только фреймворке PyTorch насчитывается более 20 готовых. Разобравшись с этой главой, вы сможете обратиться к документации (<http://bit.ly/2SuIQLm>) и узнать о них больше.

Из этого выражения очевидно, что сигма-функция — гладкая, дифференцируемая функция. Пакет `torch` реализует сигма-функцию в виде `torch.sigmoid()`, как показано в примере 3.2.

**Пример 3.2.** Сигма-функция активации<sup>1</sup>

```
import torch
import matplotlib.pyplot as plt

x = torch.range(-5., 5., 0.1)
y = torch.sigmoid(x)
plt.plot(x.numpy(), y.numpy())
plt.show()
```



Как вы видите из графика, сигма-функция очень быстро насыщается (то есть выдает экстремальные значения на выходе) для большинства входных значений. Это может стать проблемой, поскольку градиент становится равен нулю или расходится до слишком большого значения с плавающей точкой. Эти феномены известны под названиями *проблема «исчезающего» градиента* (vanishing gradient problem) и *проблема «взрывного» роста градиента* (exploding gradient problem) соответственно. В результате сигма-блоки обычно можно увидеть в нейронных сетях только на выходе, где такое размазывание позволяет интерпретировать выходные данные как вероятности.

## Гиперболический тангенс

Функция активации `th`<sup>2</sup> — лишь незначительно отличающийся вариант сигма-функции. Это очевидно из выражения для `th`:

$$f(x) = \text{th } x = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

После небольших преобразований (которые мы оставляем читателю в качестве упражнения) можно убедиться, что `th` — просто линейное преобразование сигма-функции, как показано в примере 3.3. Это будет очевидно, если написать код PyTorch для вызова функции гиперболического тангенса `tanh()` и построить ее график. Обратите внимание, что гиперболический тангенс, как

<sup>1</sup> Напомним, что для отображения графиков в блокноте iPython необходимо сначала выполнить команду `%matplotlib inline`. — *Примеч. пер.*

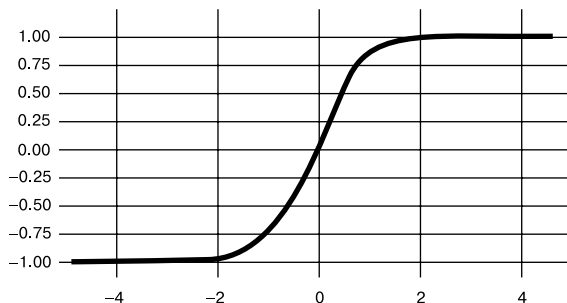
<sup>2</sup> В англоязычной литературе обозначается `tanh`. — *Примеч. пер.*

и сигма-функция, — сжимающая функция, но отображающая вещественные значения из  $(-\infty, +\infty)$  в диапазоне  $[-1, +1]$ .

**Пример 3.3.** Функция активации th

```
import torch
import matplotlib.pyplot as plt

x = torch.range(-5., 5., 0.1)
y = torch.tanh(x)
plt.plot(x.numpy(), y.numpy())
plt.show()
```



## ReLU

ReLU (произносится «рей-лу») расшифровывается как *выпрямленный линейный блок* (rectified linear unit). Вероятно, это важнейшая из функций активации. На самом деле можно даже сказать, что многие из последних новшеств глубокого обучения были бы невозможны без ReLU. Для столь фундаментальной концепции она появилась удивительно недавно. И форма ее также очень проста:

$$f(x) = \max(0, x).$$

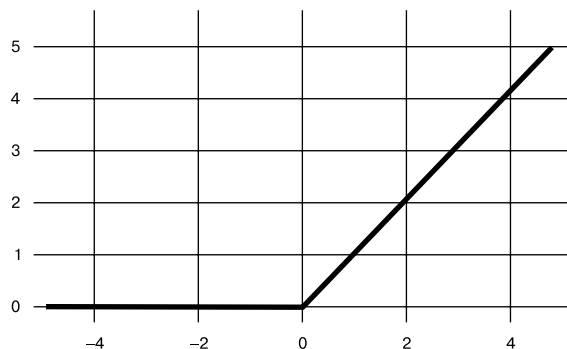
Таким образом, все, что делает блок ReLU, — обнуляет отрицательные значения, как показано в примере 3.4.

**Пример 3.4.** ReLU-активация

```
import torch
import matplotlib.pyplot as plt

relu = torch.nn.ReLU()
x = torch.range(-5., 5., 0.1)
y = relu(x)

plt.plot(x.numpy(), y.numpy())
plt.show()
```



Эффект обрезания ReLU, помогающий справиться с проблемой исчезающего градиента, может сам стать проблемой, когда с течением времени некоторые выходные

значения в сети просто становятся равны 0 и никогда более не восстанавливаются. Эта проблема носит название «умирающего ReLU». Для уменьшения этого эффекта были предложены функции активации под названием ReLU «с утечкой» (leaky ReLU) и параметрический ReLU (parametric ReLU, PReLU), где коэффициент утечки  $a$  — параметр, значение которого определяется в процессе обучения. Результат показан в примере 3.5.

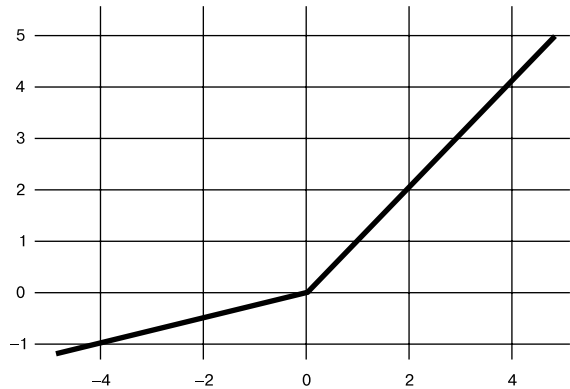
$$f(x) = \max(x, ax).$$

#### Пример 3.5. PReLU-активация

```
import torch
import matplotlib.pyplot as plt

prelu = torch.nn.PReLU(num_
parameters=1)
x = torch.range(-5., 5., 0.1)
y = prelu(x)

plt.plot(x.numpy(), y.numpy())
plt.show()
```



## Многомерная логистическая функция

Еще один вариант функции активации — многомерная логистическая функция (softmax). Подобно сигма-функции, многомерная логистическая функция сжимает результат каждого из блоков в диапазон от 0 до 1, как показано в примере 3.6. Впрочем, она также делит каждый из результатов на сумму всех результатов, в итоге получается дискретное распределение вероятности<sup>1</sup> по  $k$  возможным классам:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}.$$

Сумма вероятностей в полученном распределении равна 1. Это очень удобно для интерпретации результатов при классификации, так что данное преобразование обычно сочетается с целевой функцией вероятностного обучения, например

<sup>1</sup> Слова «распределение» и «вероятность» здесь следует воспринимать с толикой здорового скепсиса. Под «вероятностью» мы понимаем ограниченность значений на выходе отрезком  $[0, 1]$ , а под «распределением» — то, что сумма результатов равна 1.

с дискретной перекрестной энтропией, описанной в подразделе «Углубляемся в обучение с учителем» на с. 68.

**Пример 3.6.** Многомерная логистическая функция активации

Input[0]

```
import torch.nn as nn
import torch

softmax = nn.Softmax(dim=1)
x_input = torch.randn(1, 3)
y_output = softmax(x_input)
print(x_input)
print(y_output)
print(torch.sum(y_output, dim=1))
```

Output[0]

```
tensor([[ 0.5836, -1.3749, -1.1229]])
tensor([[ 0.7561,  0.1067,  0.1372]])
tensor([ 1.])
```

В этом разделе мы изучили четыре важные функции активации: сигма-функцию, гиперболический тангенс, ReLU и многомерную логистическую функцию. Это лишь четыре из множества возможных функций активации, применимых при создании нейронных сетей. По мере чтения данной книги вам станет понятно, где и какие функции активации следует применять, но общий совет: используйте то, что работало раньше.

## Функции потерь

В главе 1 мы рассмотрели общую архитектуру машинного обучения с учителем и поговорили о том, как функции потерь (целевые функции) помогают обучающему алгоритму в выборе правильных параметров на основе данных. Напомним, что функция потерь принимает на входе контрольные значения ( $y$ ) и предсказания ( $\hat{y}$ ) и генерирует вещественный показатель. Чем больше значение этого показателя, тем хуже предсказание, производимое данной моделью. PyTorch реализует в пакете nn больше функций потерь, чем мы можем здесь охватить, но несколько наиболее используемых мы все же рассмотрим.

## Среднеквадратичная погрешность

Для задач регрессии, в которых выходные значения сети ( $\hat{y}$ ) и целевая переменная ( $y$ ) представляют собой непрерывные значения, в качестве функции потерь часто применяется среднеквадратичная погрешность (mean squared error, MSE):

$$L_{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n (y - \hat{y})^2.$$



MSE — это просто среднее значение квадратов разностей предсказанных и целевых значений. Можно использовать для задач регрессии и несколько других функций потерь, например среднюю абсолютную погрешность (mean absolute error, MAE) и корень среднеквадратичной погрешности (root mean squared error, RMSE), но все они связаны с вычислением вещественного расстояния между выходным и целевым значениями. Пример 3.7 демонстрирует вариант реализации функции потерь MSE с помощью фреймворка PyTorch.

**Пример 3.7.** Функция потерь MSE

Input[0]

```
import torch
import torch.nn as nn

mse_loss = nn.MSELoss()
outputs = torch.randn(3, 5, requires_grad=True)
targets = torch.randn(3, 5)
loss = mse_loss(outputs, targets)
print(loss)
```

Output[0]

tensor(3.8618)

## Функции потерь на основе дискретной перекрестной энтропии

Дискретная перекрестная энтропия обычно используется при постановке задачи многоклассовой классификации, при которой выходные значения интерпретируются как предсказания вероятностей принадлежности определенным классам.

Целевая переменная ( $y$ ) представляет собой вектор из  $n$  элементов, соответствующий истинному полиномиальному распределению<sup>1</sup> по всем классам. Если правильно определен только один класс, то вектор будет унитарным. Выходные данные сети ( $\hat{y}$ ) также представляют собой вектор из  $n$  элементов, но соответствующий уже предсказанному сетью полиномиальному распределению. Дискретная перекрестная энтропия оценивает потери путем сравнения этих двух векторов ( $y, \hat{y}$ ):

$$L_{\text{перекрестная\_энтропия}}(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i).$$

Истоки понятия перекрестной энтропии и выражения для нее лежат в теории информации, но для наших целей удобно рассматривать их как метод вычисления разности двух распределений. Нам нужно, чтобы вероятность правильно определенного класса была близка к 1, а остальных — к 0.

<sup>1</sup> К вектору полиномиального распределения предъявляется два требования: сумма элементов вектора должна быть равна 1, а все элементы вектора — неотрицательны.

Для правильного применения функции `CrossEntropyLoss()` фреймворка PyTorch важно понимать взаимосвязь между выходными параметрами сети, способом вычисления функции потерь и вычислительными ограничениями, возникающими вследствие фактического представления чисел с плавающей точкой. А именно, нюансы взаимосвязи между выходными параметрами сети и функцией потерь определяют четыре элемента информации. Во-первых, существуют ограничения на размер чисел сверху и снизу. Во-вторых, если входной параметр экспоненциальной функции в формуле многомерной логистической функции — отрицательное число, то результат окажется экспоненциально малым числом, а если положительное, то результат окажется экспоненциально большим числом. Далее, непосредственно перед многомерной логистической функцией на выходе сети предполагается вектор<sup>1</sup>. Наконец, функция логарифма ( $\log$ ) является обратной к экспоненциальной<sup>2</sup>, так что  $\log(\exp(x))$  равно  $x$ . Исходя из этих четырех нюансов, можно выполнить математические упрощения, предполагающие, что лежащая в основе многомерной логистической функции экспонента и используемая при вычислении перекрестной энтропии логарифмическая функция численно устойчивы и избегают действительно больших или малых чисел. Вследствие этих упрощений выходные значения сети (без применения многомерной логистической функции) можно использовать совместно с функцией `CrossEntropyLoss()` фреймворка PyTorch для оптимизации распределения вероятности. Далее, после обучения сети, с помощью многомерной логистической функции можно создать распределение вероятности, как показано в примере 3.8.

**Пример 3.8.** Функция потерь на основе дискретной перекрестной энтропии

Input[0]

```
import torch
import torch.nn as nn

ce_loss = nn.CrossEntropyLoss()
outputs = torch.randn(3, 5, requires_grad=True)
targets = torch.tensor([1, 0, 3], dtype=torch.int64)
loss = ce_loss(outputs, targets)
print(loss)
```

Output[0]

tensor(2.7256)

В этом примере кода с помощью вектора случайных значений сначала моделируется выход сети. Далее создается вектор контрольных данных `targets`, состоящий

<sup>1</sup> В PyTorch на самом деле есть две многомерные логистические функции: `Softmax()` и `LogSoftmax()`. Последняя генерирует логарифмические вероятности с сохранением относительного соотношения любых двух чисел.

<sup>2</sup> Это справедливо только в том случае, когда основание логарифма равно числу  $e$  — как раз основание по умолчанию для функции `log` PyTorch.

из целых чисел, поскольку реализация PyTorch функции `CrossEntropyLoss()` предполагает, что каждому входному параметру соответствует один конкретный класс, а каждому классу — уникальный индекс. Именно поэтому вектор `targets` состоит из трех элементов индекса, соответствующих правильным классам для каждого входного значения. Исходя из этих допущений операция индексации выходных данных модели с вычислительной точки зрения выполняется более эффективно<sup>1</sup>.

## Функция потерь на основе бинарной перекрестной энтропии

Функция потерь на основе дискретной перекрестной энтропии, которую мы рассмотрели в предыдущем разделе, очень удобна для решения задач классификации в случае нескольких классов. Иногда задача (называемая в таком случае *бинарной классификацией*) заключается в разделении только двух классов. В подобных ситуациях целесообразно воспользоваться функцией потерь на основе бинарной перекрестной энтропии (binary cross-entropy, BCE). Мы рассмотрим ее в действии в примере из раздела «Пример: классификация тональностей обзоров ресторанов» на с. 76.

В примере 3.9 мы создаем выходной вектор бинарных вероятностей, `probabilities`, применяя сигма-функцию активации к случайному вектору, представляющему выходные значения сети. Далее создается вектор контрольных значений, состоящий из 0 и 1<sup>2</sup>. Наконец, мы вычисляем BCE на основе вектора бинарной вероятности и вектора контрольных значений.

**Пример 3.9.** Функция потерь на основе бинарной перекрестной энтропии

Input[0]

```
bce_loss = nn.BCELoss()
sigmoid = nn.Sigmoid()
probabilities = sigmoid(torch.randn(4, 1, requires_grad=True))
targets = torch.tensor([1, 0, 1, 0], dtype=torch.float32).view(4, 1)
loss = bce_loss(probabilities, targets)
print(probabilities)
print(loss)
```

<sup>1</sup> Использование унитарных векторов в формуле перекрестной энтропии означает, что результаты всех операций умножения, кроме одной, будут нулевыми. Это колоссальная напрасная трата вычислительных ресурсов.

<sup>2</sup> Обратите внимание, что в примере кода вектор контрольных значений состоит из элементов с плавающей точкой. Хотя бинарная перекрестная энтропия почти аналогична дискретной перекрестной энтропии (просто классов только два), при ее вычислении значения 0 и 1 используются в формуле бинарной перекрестной энтропии, а не служат лишь в качестве индексов, как для дискретной перекрестной энтропии.

Output[0]

```
tensor([[ 0.1625],
        [ 0.5546],
        [ 0.6596],
        [ 0.4284]])
tensor(0.9003)
```

## Углубляемся в обучение с учителем

Задача машинного обучения с учителем состоит в поиске соответствий *наблюдений* (observations) указанным *целевым переменным* (targets) по заданным маркированным выборкам. В этом подразделе мы подробнее рассмотрим данную задачу. Точнее говоря, мы явным образом опишем, как на основе *предсказаний* модели и *функции потерь* произвести градиентную оптимизацию *параметров* модели. Это очень важный материал, лежащий в основе остальной книги, так что внимательно изучите его, даже если вы уже более или менее знакомы с концепцией машинного обучения с учителем.

Как вы помните из главы 1, для машинного обучения с учителем необходимо следующее: модель, функция потерь, обучающие данные и алгоритм оптимизации. Обучающие данные для машинного обучения с учителем представляют собой пары наблюдений и целевых переменных; модель вычисляет предсказания по наблюдениям, а функция потерь оценивает ошибку предсказаний относительно целевых переменных. Цель обучения — подобрать такие параметры модели с помощью градиентного алгоритма оптимизации, чтобы потери были минимальны.

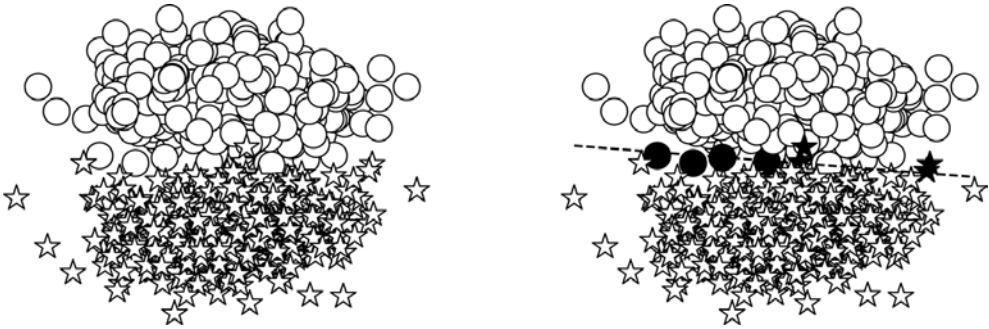
Далее в разделе мы обсудим классическую модельную задачу: классификацию двумерных точек по двум классам. На интуитивном уровне это значит нахождение путем обучения одной линии, называемой *границей принятия решения* (decision boundary) или *гиперплоскостью* (hyperplane), отделяющей точки одного класса от точек другого. Мы пошагово рассмотрим и опишем процесс формирования данных, выбора модели, выбора функции потерь, задания параметров алгоритма оптимизации и, наконец, объединения всего этого.

## Формирование модельных данных

В машинном обучении, чтобы разобраться в алгоритме, очень часто создают искусственные данные с хорошо известными свойствами. В этом разделе мы будем использовать искусственные данные для задачи классификации двумерных точек по двум классам. Для формирования данных произведем выборку<sup>1</sup> точек из двух различных частей *xy*-плоскости, в результате чего получим ситуацию, при которой

<sup>1</sup> Выборку мы производим из двух нормальных распределений с единичной дисперсией. Если вы не понимаете, что это значит, просто считайте, что «форма» данных выглядит так, как показано на рисунке.

обучение для модели будет несложной задачей. Выборки показаны на рис. 3.2. Задача модели — классифицировать звезды (\*) как принадлежащие одному классу и окружности (○) как принадлежащие другому. Это показано на рис. 3.2, *справа*, где все точки данных, расположенные выше прямой, классифицированы не так, как те, что расположены ниже. Код генерации данных (функцию `get_toy_data()`) можно найти в прилагаемом к книге блокноте Python.



**Рис. 3.2.** Создание линейно разделяемого модельного набора данных. Он представляет собой выборку из двух нормальных распределений, по одному для каждого класса. Задача классификации сводится к выяснению, относится точка данных к одному распределению или к другому

## Выбор модели

Воспользуемся здесь моделью, с которой познакомили вас в начале главы, — перцептроном. Гибкость перцептрона состоит в возможности задействовать входные данные любого размера. При типичном моделировании размер входных значений определяется задачей и данными. В нашем модельном примере этот размер равен 2, поскольку мы явным образом расположили данные на двумерной плоскости. Для этой задачи классификации по двум классам мы присвоим классам индексы 0 и 1. Соответствие строковых меток \* и ○ индексам классов может быть любым, главное, чтобы оно было одинаковым на протяжении всей предварительной обработки данных, обучения, оценки и контроля. Важным дополнительным свойством модели является сущность ее выходных данных. Поскольку функция активации перцептрона представляет собой сигма-функцию, то на выходе перцептрона получается вероятность того, что точка данных ( $x$ ) относится к классу 1; то есть  $P(y = 1 | x)$ .

## Преобразование вероятностей в дискретные классы

Для преобразования выходной вероятности в задаче бинарной классификации зафиксируем границу решения,  $\delta$ . Если предсказанная вероятность  $P(y = 1 | x) > \delta$ , то предсказан класс 1, в противном случае — 0. Обычно граница решения задается

равной 0.5, но на практике может понадобиться подобрать значение этого гиперпараметра (на основе оценочного набора данных) так, чтобы добиться желаемой точности классификации.

## Выбор функции потерь

После подготовки данных и выбора архитектуры модели остается выбрать два жизненно важных компонента машинного обучения с учителем: функцию потерь и оптимизатор. В ситуациях, когда выходные данные модели представляют собой вероятность, самое подходящее семейство функций потерь — функции потерь на основе перекрестной энтропии. Поскольку у нас целевые переменные — бинарные, для нашего модельного примера воспользуемся функцией потерь ВСЕ.

## Выбор оптимизатора

Последнее, что осталось выбрать в этом упрощенном примере машинного обучения с учителем, — оптимизатор. Модель генерирует предсказания, функция потерь оценивает расхождение между предсказаниями и целевыми переменными, а оптимизатор обновляет значения весов модели на основе сигнала рассогласования. При простейшем варианте поведение оптимизатора регулируется одним гиперпараметром. Этот гиперпараметр — *скорость обучения* (learning rate) — определяет, насколько сильно сигнал рассогласования влияет на обновление весов. Скорость обучения — критически важный гиперпараметр, рекомендуется попробовать несколько различных его значений и сравнить результаты. Высокая скорость обучения приводит к большим изменениям параметров и может влиять на сходимость. Слишком низкая скорость обучения может приводить к очень медленному продвижению обучения.

В библиотеке PyTorch имеется несколько вариантов оптимизаторов. Традиционно используется алгоритм стохастического градиентного спуска (stochastic gradient descent, SGD), но в сложных задачах оптимизации в нем часто возникают проблемы со сходимостью, приводящие к худшим моделям. В настоящее время предпочитают использовать адаптивные оптимизаторы, в частности Adagrad (adaptive gradient — «адаптивный градиент». — *Примеч. пер.*) и Adam, учитывающие историю обновлений<sup>1</sup>. В следующем примере мы воспользуемся оптимизатором Adam, но никогда не помешает рассмотреть несколько вариантов. В оптимизаторе Adam скорость обучения по умолчанию равна 0.001. Для таких гиперпараметров, как скорость обучения, всегда рекомендуется использовать сначала значения по умолчанию, разве что вы основываетесь на статье, в которой рекомендуется какое-либо другое конкретное значение.

---

<sup>1</sup> В сообществах, посвященных машинному обучению и оптимизации, непрерывно ведутся споры относительно достоинств и недостатков SGD. Нам кажется, что, хотя подобные споры и познавательны, они только мешают.

**Пример 3.10.** Оптимизатор Adam

Input[0]

```
import torch.nn as nn
import torch.optim as optim

input_dim = 2
lr = 0.001

perceptron = Perceptron(input_dim=input_dim)
bce_loss = nn.BCELoss()
optimizer = optim.Adam(params=perceptron.parameters(), lr=lr)
```

## Собираем все вместе: градиентное машинное обучение с учителем

Обучение начинается с вычисления функции потерь, то есть отклонения предсказаний модели от целевых переменных. Градиент функции потерь в свою очередь сигнализирует, «насколько» необходимо поменять параметры. Градиент каждого параметра отражает мгновенную скорость изменений значения потерь при заданном значении параметра. Фактически это значит, что нам известна величина вклада каждого из параметров в функцию потерь. Образно можно представить, что каждый параметр стоит на своем холме и хочет сделать шаг вверх или вниз. В минимальном варианте градиентное обучение модели состоит в итеративном обновлении каждого из параметров с учетом градиента функции потерь относительно этого параметра.

Посмотрим, как этот пошаговый градиентный метод работает. Во-первых, вся вспомогательная информация, хранящаяся внутри объекта (**perceptron**) модели, очищается с помощью функции **zero\_grad()**. Далее модель вычисляет выходные значения (**y\_pred**) по указанным входным данным (**x\_data**). Затем вычисляется функция потерь путем сравнения выходных значений (**y\_pred**) с намеченными целевыми переменными (**y\_target**). Это непосредственно «учительская часть» сигнала обучения с учителем. В объекте потерь (**criterion**) PyTorch есть функция **backward()**, итеративно транслирующая потери обратно по графу вычислений и сообщаящая всем параметрам их градиенты. Наконец, оптимизатор (**opt**) обновляет значения параметров по известным градиентам с помощью функции **step()**.

Весь обучающий набор данных разбивается на *пакеты* (**batches**<sup>1</sup>). Все шаги градиентного этапа выполняются над пакетами данных. Размер пакетов задается гиперпараметром **batch\_size**. А поскольку общий размер обучающего набора данных фиксирован, то увеличение размера пакетов ведет к уменьшению их количества.

<sup>1</sup> В литературе также часто можно встретить прямую кальку с английского «батч». Например, мини-пакетный стохастический градиентный спуск, упоминавшийся в главе 1, называют стохастическим градиентным спуском по мини-батчам. — *Примеч. пер.*



В литературе, в том числе в этой книге, попеременно используются термины «пакет» и «мини-пакет», подчеркивая то, что пакеты по отдельности существенно меньше общего размера обучающей последовательности; например, размер обучающей последовательности может исчисляться миллионами, а мини-пакета — лишь несколькими сотнями.

Обработка определенного числа пакетов (обычно числа пакетов в наборе данных конечной величины) в цикле обучения называется *эпохой* (epoch). Эпоха — одна полная итерация обучения. Если количество пакетов в одной эпохе равно количеству пакетов в наборе данных, то эпоха представляет собой полную итерацию обработки набора данных. Обучение модели состоит из определенного числа эпох. Выбор этого числа — задача нетривиальная, но есть способы, которые мы вскоре обсудим, позволяющие выбрать момент прекращения обучения. Как видно из примера 3.11, цикл обучения с учителем является, таким образом, вложенным: внутренний цикл обрабатывает набор данных или заданное число пакетов, а внешний цикл повторяет внутренний заданное число эпох или использует другой критерий завершения.

**Пример 3.11.** Цикл обучения с учителем для перцептрона и бинарной классификации

```
# Каждая из эпох представляет собой полный проход
# по обучающей последовательности
for epoch_i in range(n_epochs):
    # Проходим во внутреннем цикле по пакетам набора данных
    for batch_i in range(n_batches):
        # Шаг 0: получаем данные
        x_data, y_target = get_toy_data(batch_size)

        # Шаг 1: очищаем значения градиентов
        perceptron.zero_grad()

        # Шаг 2: вычисляем прямой проход модели
        y_pred = perceptron(x_data, apply_sigmoid=True)

        # Шаг 3: вычисляем оптимизируемую величину потерь
        loss = bce_loss(y_pred, y_target)

        # Шаг 4: транслируем сигнал потерь обратно по графу вычислений
        loss.backward()

        # Шаг 5: запускаем выполнение обновления оптимизатором
        optimizer.step()
```

## Вспомогательные понятия машинного обучения

Основная идея градиентного машинного обучения с учителем проста: описывается модель, вычисляются выходные значения, вычисляются градиенты на основе выбранной функции потерь, после чего применяется алгоритм оптимизации для



обновления параметров моделей с учетом градиента. Однако для процесса машинного обучения важны несколько вспомогательных понятий, часть из которых мы рассмотрим в этом разделе.

## Точная оценка качества модели: метрики оценки

Помимо основного цикла обучения с учителем, важнейшим компонентом является объективная оценка качества модели с помощью данных, на которых эта модель не обучалась. Оценка моделей производится с помощью одной или нескольких *метрик оценки* (evaluation metrics). При обработке написанных на естественных языках текстов используется несколько подобных метрик. Самая распространенная, которой мы и воспользуемся в этой главе, — *точность* (ассигасу). Точность представляет собой просто долю правильных предсказаний для не участвовавшего в обучении набора данных.

## Точная оценка качества модели: разбиение набора данных

Важно не забывать, что конечная цель — *обобщить* модель на истинное распределение данных. Что это значит? Если бы нашему алгоритму для просмотра было доступно бесконечное количество данных, он бы обнаружил глобальное распределение данных (*«истинное/ненаблюдаемое распределение»*). Конечно, это невозможно. Нам приходится довольствоваться конечной выборкой — обучающей последовательностью. В этой конечной выборке мы наблюдаем распределение данных, приближенно (неполно) соответствующее истинному. Одна модель считается *лучше обобщающейся*, чем другая, если она снижает погрешность не только в присутствовавших в обучающей последовательности выборках, но и в выборках из распределения, которое она не наблюдала. Поскольку модель стремится снизить потери на обучающей последовательности, то может «переобучиться» и адаптироваться к особенностям, не имеющим отношения к истинному распределению данных.

Для хорошего обобщения принято либо разбивать набор данных на три выбранные случайным образом части (*обучающий* (training), *проверочный* (validation) и *контрольный* (test) наборы данных) или выполнять *k-блочную перекрестную проверку* (k-fold cross-validation). Разбиение на три части — самый простой из этих методов, поскольку требует одного прохода. Следует, впрочем, позаботиться о том, чтобы распределение классов в этих наборах данных было одинаковым. Другими словами, рекомендуется агрегировать набор данных по меткам классов, после чего разбивать каждый из наборов со своей меткой класса на обучающий, проверочный и контрольный. Обычно при разбиении 70 % данных отводится на обучающий набор, 15 % на проверочный и 15 % на контрольный. Но это не обязательно.

В некоторых случаях разбиение на обучающий, проверочный и контрольный наборы может быть уже выполнено; такое часто встречается в наборах данных для задач оценки производительности. В подобных случаях важно использовать обучающие данные только для обновления параметров модели, проверочные данные — только для оценки качества модели в конце каждой эпохи, а тестовые данные применять только один раз, после анализа всех вариантов модели, при выдаче окончательных результатов. Последнее чрезвычайно важно, поскольку, чем больше специалист по машинному обучению «подглядывает» за тем, насколько модель эффективна на контрольном наборе данных, тем больше оказывается систематическая ошибка в сторону значений параметров, хорошо проявляющих себя на контрольном наборе. В подобном случае невозможно определить, насколько хорошо модель будет работать на не виденных ею данных, без сбора дополнительных сведений.

Оценка эффективности модели при  $k$ -блочной перекрестной проверке очень похожа на вариант с предварительной разбивкой набора данных, но содержит еще один предварительный этап разбивки всего набора данных на  $k$  блоков равного размера. Один из этих блоков предназначается для оценки, а оставшиеся  $k - 1$  блоков — для обучения. Этот процесс итеративно повторяется  $k$  раз, всякий раз со сменой оценочного блока. Поскольку блоков всего  $k$ , то каждый из них сможет побывать оценочным, в результате чего получится  $k$  значений точности. Итоговая точность представляет собой просто среднее значение со стандартным отклонением.  $k$ -блочная оценка требует значительных затрат вычислительных ресурсов, но исключительно важна для маленьких наборов данных, когда ошибочное разбиение может привести или к излишнему оптимизму (поскольку контрольный результат оказывается слишком хорошим), или к излишнему пессимизму (поскольку контрольный результат оказывается слишком плохим).

## Когда прекращать обучение

В приведенном ранее примере модель обучалась на протяжении фиксированного количества эпох. Хотя это простейший подход, подобный выбор произволен и не нужен. Одна из важнейших задач должной оценки эффективности модели — определение на основе этой оценки момента для останова обучения. Чаще всего применяют эвристический метод, называемый *ранней остановкой* (early stopping). Ранняя остановка заключается в отслеживании эффективности модели на проверочном наборе данных в каждой эпохе и выявлении момента, когда эффективность перестает улучшаться. Если далее эффективность по-прежнему не увеличивается на протяжении заданного количества эпох, называемого *терпением* (patience), то обучение завершается. Вообще говоря, момент, когда показатели модели перестают улучшаться на каком-либо наборе данных, называется моментом, когда модель *сошлась*. На практике редко дожидаются, чтобы модель полностью сошла, поскольку это требует времени и может привести к переобучению.

## Поиск правильных значений гиперпараметров

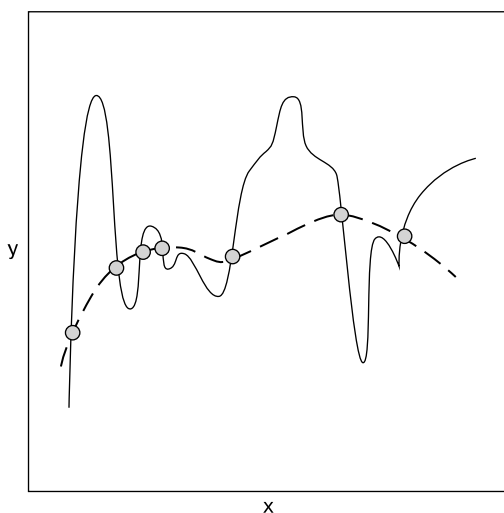
Мы уже выяснили, что параметры (веса) принимают вещественные значения, подстраиваемые оптимизатором с учетом фиксированного поднабора обучающей последовательности, называемого мини-пакетом. *Гиперпараметром* (hyperparameter) называется любая настройка, влияющая на несколько параметров и принимаемые ими значения. Существует множество различных вариантов выбора, определяющих то, как будет обучаться модель. В их числе выбор функции потерь, оптимизатора, скорости (-ей) обучения для оптимизатора, размеров слоев (обсудим это в главе 4), терпения для ранней остановки и различных решений, связанных с регуляризацией (также описываются в главе 4). Важно отдавать себе отчет в том, что эти решения могут существенно влиять на сходимость модели и ее эффективность, так что желательно систематически исследовать все возможные альтернативы.

## Регуляризация

Одно из важнейших понятий глубокого обучения (и машинного обучения вообще) — *регуляризация* (regularization). Истоки понятия регуляризации лежат в теории численной оптимизации. Как вы помните, большинство алгоритмов машинного обучения оптимизируют функцию потерь для поиска наиболее вероятных значений параметров (или «модели»), объясняющих наблюдаемые значения, то есть минимизирующих потери. Для большинства наборов данных и задач существует несколько решений (возможных моделей) задачи оптимизации. Так какое из них нам (или оптимизатору) выбрать? Рассмотрим рис. 3.3, иллюстрирующий задачу подбора кривой для множества точек.

Обе кривые подходят для указанных точек, но какая из них менее вероятна? Обращаясь к бритве Оккама, мы интуитивно понимаем, что простое объяснение лучше сложного. Подобное ограничение гладкости в машинном обучении называется *L2-регуляризацией*. Управлять ею в PyTorch можно с помощью параметра `weight_decay` оптимизатора. Чем больше значение параметра `weight_decay`, тем вероятнее выбор оптимизатором более гладкого (smoother) обоснования (то есть тем сильнее L2-регуляризация).

Кроме L2-регуляризации, часто используется и *L1-регуляризация*, обычно для того, чтобы подтолкнуть алгоритм к более разреженным решениям, то есть таким, в которых большинство параметров модели близки к нулю. В главе 4 мы рассмотрим еще один метод структурной регуляризации — *дропаут* (dropout). Тема регуляризации моделей сейчас активно исследуется, а PyTorch представляет собой гибкий фреймворк для реализации пользовательских методов регуляризации.



**Рис. 3.3.** Обе кривые подходят для точек, но одна из них выглядит более рациональной — с помощью регуляризации нам удастся выбрать более правдоподобное объяснение (взято из «Википедии»: <http://bit.ly/2qAU18y>)

## Пример: классификация тональностей обзоров ресторанов

В предыдущем разделе мы углубились в машинное обучение с учителем и на модельном примере проиллюстрировали многие важнейшие понятия. Здесь мы повторим это, но уже на примере реальной задачи и набора данных: определим классификацию тональностей (позитивная или негативная) обзоров ресторанов на сайте Yelp с помощью перцептрона и машинного обучения с учителем. Поскольку это первый полный пример NLP в нашей книге, мы в мельчайших подробностях опишем вспомогательные структуры данных и процесс обучения. Структура примеров в последующих главах будет очень близкой к структуре примера, приведенного здесь, так что мы рекомендуем внимательно следить за происходящим в этом разделе и обращаться к нему для освежения в памяти нужных моментов<sup>1</sup>.

В начале всех примеров книги мы описываем используемые наборы данных. В этом примере мы задействуем набор данных Yelp, в котором обзорам соответствуют метки тональностей (позитивная или негативная). Кроме того, мы опишем несколько операций над набором данных, связанных с его очисткой и разбиением на обучающий, проверочный и контрольный наборы.

<sup>1</sup> Код для классификации тональностей обзоров на Yelp можно найти в репозитории GitHub этой книги (<https://nlpproc.info/pytorch/repo>).

Если вы разберетесь с набором данных, то увидите повторяющийся во всей книге паттерн описания трех вспомогательных классов, используемых для преобразования текстовых данных в векторную форму: `Vocabulary`, `Vectorizer` и `DataLoader` из фреймворка PyTorch. Класс `Vocabulary` отвечает за согласование отображения целых чисел в токены, обсуждавшееся в разделе «Кодирование наблюдаемых величин и целевых переменных» на с. 23. Класс `Vocabulary` применяется как для отображения текстовых токенов в целочисленные значения, так и для отображения меток классов в целочисленные значения. Далее, класс `Vectorizer` инкапсулирует словари и отвечает за ввод и обработку строковых данных, например текстов обзоров, и преобразование их в числовые векторы для дальнейшего использования в процессе обучения. Последний из вспомогательных классов, `DataLoader` из фреймворка PyTorch, будет у нас отвечать за группировку и упорядочение отдельных векторизованных точек данных в мини-пакеты.

В следующем подразделе мы опишем классификатор на основе перцептрона и его процесс обучения. Процесс обучения практически одинаков для всех примеров книги, но наиболее подробно он будет описан в этом разделе, так что мы опять же рекомендуем обращаться к нему как справочнику. В завершение этого примера мы обсудим результаты и заглянем «под капот», чтобы узнать, чему обучилась модель.

## Набор данных обзоров Yelp

В 2015 году на сайте Yelp проводился конкурс, участники которого должны были предсказать рейтинг ресторана по его обзору. Чжан, Чжао и Лежун в своей статье (Zhang, 2015) упростили этот набор данных, преобразовав рейтинги с одной и двумя звездами в класс «негативных» тональностей, а рейтинги с тремя и четырьмя звездами в класс «позитивных» тональностей, и разбили его на 560 000 обучающих и 38 000 контрольных выборок. В этом примере мы воспользуемся упрощенным набором данных Yelp с двумя небольшими отличиями. В оставшейся части раздела мы опишем процесс небольшой очистки данных и получения нашего итогового набора данных. Далее мы наметим в общих чертах реализацию алгоритма на основе класса `Dataset` PyTorch.

Первое из вышеупомянутых отличий: мы будем использовать «облегченную» версию набора данных, состоящую из 10 % выборок полного набора<sup>1</sup>. Это приводит к тому, что, во-первых, длительность цикла обучения-контроля при маленьком наборе данных невелика, так что мы сможем проводить эксперименты достаточно быстро. Во-вторых, точность модели при этом будет ниже, чем могла бы быть при использовании всех данных. Низкая точность не проблема, ведь всегда можно заново провести обучение на всем наборе данных с учетом знаний, полученных при

---

<sup>1</sup> Код для очистки «облегченной» и «полной» версий набора данных обзоров Yelp можно найти в GitHub.

обучении маленького. Такой трюк очень удобен при работе с моделями глубокого обучения, где данные часто бывают колоссального объема.

Мы разобьем наш маленький набор данных на три части: одну для обучения, вторую для проверки и третью — для контроля. Хотя исходный набор данных состоял только из двух частей, проверочный набор данных важен. При машинном обучении вы часто будете обучать модель на обучающей части набора, выделив предварительно часть данных для оценки эффективности модели. Если решения модели будут основаны на этой выделенной порции данных, неизбежна систематическая ошибка — модель будет демонстрировать лучшие результаты именно на этой выделенной порции данных. А поскольку постепенное улучшение оценок играет очень важную роль, для решения проблемы понадобится третья часть данных, которую стараются как можно меньше использовать при оценке эффективности модели.

Подведем итоги: обучающая часть набора данных должна использоваться для получения параметров модели, проверочная — для выбора гиперпараметров (принятия модельных решений), а контрольная — для окончательной оценки и отчета<sup>1</sup>. В примере 3.12 показано, как мы разбили набор данных. Обратите внимание, что случайное начальное значение (*seed*) — фиксированное и сначала выполняется агрегирование по метке класса, чтобы гарантировать неизменность распределения по классам.

**Пример 3.12.** Создание обучающего, проверочного и контрольного фрагментов (*split*) поднабора данных

```
# Разбиваем поднабор по рейтингу для создания нового обучающего (train),
# проверочного (val) и контрольного (test) фрагментов
by_rating = collections.defaultdict(list)
for _, row in review_subset.iterrows():
    by_rating[row.rating].append(row.to_dict())

# Создаем фрагменты данных
final_list = []
np.random.seed(args.seed)

for _, item_list in sorted(by_rating.items()):
    np.random.shuffle(item_list)

    n_total = len(item_list)
    n_train = int(args.train_proportion * n_total)
```

<sup>1</sup> Разбиение данных на обучающий, проверочный и контрольный наборы хорошо срабатывает в случае больших наборов данных. Иногда, если объем обучающих данных невелик, лучше воспользоваться *k*-блочной перекрестной проверкой. Что означает в данном случае «большие»? Зависит от обучаемой сети, сложности моделируемой задачи, размера входных примеров и т. д., но для многих задач NLP о больших наборах данных можно говорить при сотнях тысяч или миллионах обучающих выборок.

```

n_val = int(args.val_proportion * n_total)
n_test = int(args.test_proportion * n_total)

# Присваиваем точкам данных атрибуты фрагментов
for item in item_list[:n_train]:
    item['split'] = 'train'

for item in item_list[n_train:n_train+n_val]:
    item['split'] = 'val'
for item in item_list[n_train+n_val:n_train+n_val+n_test]:
    item['split'] = 'test'

# Добавляем в итоговый список
final_list.extend(item_list)

final_reviews = pd.DataFrame(final_list)

```

Кроме того, для создания поднабора, состоящего из трех фрагментов — для обучения, проверки и контроля, мы проводим минимальную очистку данных: добавляем пробелы около знаков пунктуации и удаляем посторонние символы, не являющиеся буквенно-цифровыми или знаками пунктуации для всех фрагментов (пример 3.13)<sup>1</sup>.

### Пример 3.13. Минимальная очистка данных

```

def preprocess_text(text):
    text = text.lower()
    text = re.sub(r"([.,!?])", r" \1 ", text)
    text = re.sub(r"^[^a-zA-Z.,!?]+", r" ", text)
    return text

final_reviews.review = final_reviews.review.apply(preprocess_text)

```

## Представление набора данных в PyTorch

Представленный в примере 3.14 класс `ReviewDataset` предполагает, что набор данных уже хотя бы минимально очищен и разбит на три части. В частности, набор данных предполагает возможность разбить обзоры по пробелам, для получения

<sup>1</sup> Очистка данных (предварительная обработка) — важный вопрос, который часто игнорируют во многих книгах по машинному обучению (и даже научных статьях!). Мы намеренно рассматриваем упрощенный вариант, чтобы уделить больше внимания моделированию, но очень рекомендуем вам изучить все доступные утилиты для предварительной обработки текстов, включая NLTK и `sraCy`. Предварительная обработка может как повысить, так и понизить точность, в зависимости от данных и задачи. Старайтесь применять методы, которые хорошо себя зарекомендовали ранее, и чаще экспериментируйте с маленькими наборами данных. Если при реализации описанных в научной статье алгоритмов вы обнаружите, что о предварительной обработке ничего не сказано или сказано слишком расплывчато — обязательно уточните у авторов!

списка токенов в обзоре<sup>1</sup>. Далее предполагается наличие у данных аннотаций, указывающих, к какому фрагменту они относятся. Важно отметить, что для указания метода-точки входа мы используем декоратор `classmethod` языка Python. Мы будем делать так же на протяжении всей книги.

В PyTorch существует абстракция для набора данных — класс `Dataset`. Он представляет собой абстрактный итератор. При использовании PyTorch с новым набором данных необходимо сначала унаследовать класс `Dataset` и реализовать методы `__getitem__()` и `__len__()`. В этом примере мы создадим класс `ReviewDataset`, наследующий класс `Dataset` и реализующий два метода: `__getitem__` и `__len__`. При этом формируется концептуальное соглашение, благодаря которому различные утилиты PyTorch смогут работать с нашим набором данных. Мы рассмотрим одну из этих утилит, а именно `DataLoader`, в следующем разделе. Приведенная ниже реализация в значительной степени основывается на классе `ReviewVectorizer`. Мы опишем класс `ReviewVectorizer` в следующем разделе, но пока можете считать его классом, который занимается преобразованием текстов обзоров в числовой вектор, представляющий конкретные обзоры. Общий паттерн проектирования состоит в реализации класса для набора данных, отвечающего за логику векторизации для одной точки данных. Далее класс `DataLoader` PyTorch (также описанный в следующем разделе) создает мини-пакеты путем выборки их из набора данных и упорядочения.

**Пример 3.14.** Наследование класса `Dataset` библиотеки PyTorch для набора данных обзоров Yelp

```
from torch.utils.data import Dataset

class ReviewDataset(Dataset):
    def __init__(self, review_df, vectorizer):
        """
        Аргументы:
            review_df (pandas.DataFrame): набор данных
            vectorizer (ReviewVectorizer): экземпляр векторизатора,
            полученный на основе набора данных

        self.review_df = review_df
        self._vectorizer = vectorizer

        self.train_df = self.review_df[self.review_df.split=='train']
        self.train_size = len(self.train_df)

        self.val_df = self.review_df[self.review_df.split=='val']
        self.validation_size = len(self.val_df)
```

<sup>1</sup> Как вы помните из главы 2, для некоторых языков разбиение по пробелам — не лучший вариант, но в этом примере мы работаем с очищенными обзорами на английском языке. На всякий случай просмотрите раздел «Корпусы текстов, токены и типы» на с. 48.



```

self.test_df = self.review_df[self.review_df.split=='test']
self.test_size = len(self.test_df)

self._lookup_dict = {'train': (self.train_df, self.train_size),
                      'val': (self.val_df, self.validation_size),
                      'test': (self.test_df, self.test_size)}

self.set_split('train')

@classmethod
def load_dataset_and_make_vectorizer(cls, review_csv):
    """Загружает набор данных и создает новый векторизатор с нуля

    Аргументы:
        review_csv (str): местоположение набора данных
    Возвращает:
        экземпляр ReviewDataset
    """
    review_df = pd.read_csv(review_csv)
    return cls(review_df, ReviewVectorizer.from_dataframe(review_df))

def get_vectorizer(self):
    """ возвращает векторизатор """
    return self._vectorizer

def set_split(self, split="train"):
    """ выбор фрагментов набора данных по столбцу из объекта dataframe

    Аргументы:
        split (str): "train" (обучающий), "val" (проверочный)
        или "test" (контрольный)
    """
    self._target_split = split
    self._target_df, self._target_size = self._lookup_dict[split]

def __len__(self):
    return self._target_size

def __getitem__(self, index):
    """ основной метод-точка входа для наборов данных PyTorch

    Аргументы:
        index (int): индекс точки данных
    Возвращает:
        словарь признаков (x_data) и метки (y_target) точки данных
    """
    row = self._target_df.iloc[index]

    review_vector = \
        self._vectorizer.vectorize(row.review)

    rating_index = \
        self._vectorizer.rating_vocab.lookup_token(row.rating)

```

```
return {'x_data': review_vector,
        'y_target': rating_index}

def get_num_batches(self, batch_size):
    """ Возвращает по заданному размеру пакета число пакетов в наборе

    Аргументы:
        batch_size (int)
    Возвращает:
        число пакетов в наборе данных
    """
    return len(self) // batch_size
```

## Классы Vocabulary, Vectorizer и DataLoader

**Vocabulary**, **Vectorizer** и **DataLoader** — три класса, которые мы будем использовать практически в каждом примере этой книги для важнейшего конвейера: преобразования текстовых входных данных в векторизованные мини-пакеты. Этот конвейер начинается с предварительно обработанного текста; каждая точка данных представляет собой коллекцию токенов. В данном примере токены представляют собой слова, но, как вы увидите в главах 4 и 6, токены могут быть и отдельными символами. Описанные в следующих разделах три класса отвечают за отображение токенов в целочисленные значения, создание на основе этого отображения векторизованной формы каждой из точек данных и последующую группировку векторизованных точек данных в мини-пакет для модели.

### Класс Vocabulary

Первая фаза преобразования текста в векторизованный мини-пакет — отображение токенов в числовую форму. Стандартный его вариант — взаимно однозначное, то есть обратимое отображение — между токенами и числами. На языке Python они будут представлять собой два словаря. Мы инкапсулируем это взаимно однозначное соответствие в классе **Vocabulary**, показанном в примере 3.15. Класс **Vocabulary** не только отвечает за это взаимно однозначное соответствие, благодаря которому пользователь может добавлять новые токены и автоматически наращивать значение индекса, но и поддерживает специальный токен **UNK**<sup>1</sup>, название которого расшифровывается как «неизвестный» (unknown). Благодаря **UNK** можно при контроле обрабатывать токены, которые алгоритм не видел при обучении (например, при контроле могут встретиться слова, которых не было в обучающем наборе данных). Как вы увидите в следующем пункте, мы даже будем явным образом исключать

---

<sup>1</sup> Мы встретимся и с другими специальными токенами при обсуждении моделей последовательностей в главе 6.

нечасто встречающиеся токены из `Vocabulary`, так что в процедуре обучения будут встречаться токены `UNK`. Они играют важную роль в уменьшении объема используемой классом `Vocabulary` памяти<sup>1</sup>. Ожидается, что для добавления новых токенов в `Vocabulary` будет вызываться метод `add_token()`, для извлечения индекса токена — метод `lookup_token()` и для извлечения соответствующего конкретному индексу токена — `lookup_index()`.

**Пример 3.15.** Класс `Vocabulary`, предназначенный для хранения соответствия токенов целым числам, необходимого остальной части конвейера машинного обучения

```
class Vocabulary(object):
    """ Класс, предназначенный для обработки текста и извлечения Vocabulary
        для отображения
    """

    def __init__(self, token_to_idx=None, add_unk=True, unk_token="<UNK>"):
        """
        Аргументы:
            token_to_idx (dict): готовый ассоциативный массив соответствий
            токенов индексам add_unk (bool): флаг, указывающий,
            нужно ли добавлять токен UNK
            unk_token (str): добавляемый в словарь токен UNK
        """

        if token_to_idx is None:
            token_to_idx = {}
        self._token_to_idx = token_to_idx

        self._idx_to_token = {idx: token
                              for token, idx in self._token_to_idx.items()}

        self._add_unk = add_unk
        self._unk_token = unk_token

        self.unk_index = -1
        if add_unk:
            self.unk_index = self.add_token(unk_token)

    def to_serializable(self):
        """ Возвращает словарь с возможностью сериализации """
        return {'token_to_idx': self._token_to_idx,
                'add_unk': self._add_unk,
                'unk_token': self._unk_token}

    @classmethod
```

<sup>1</sup> Слова во всех языках распределяются в соответствии со степенным законом (power law). Количество уникальных слов в корпусе может достигать миллиона, причем большинство из них встречается в обучающем наборе данных лишь несколько раз. Хотя их можно учесть в словаре модели, это повысит потребность в памяти на порядок или еще выше.

```
def from_serializable(cls, contents):
    """ Создает экземпляр Vocabulary на основе сериализованного словаря """
    return cls(**contents)

def add_token(self, token):
    """ Обновляет словари отображения, добавляя в них токен.
    Аргументы:
        token (str): добавляемый в Vocabulary элемент
    Возвращает:
        index (int): соответствующее токenu целочисленное значение
    """
    if token in self._token_to_idx:
        index = self._token_to_idx[token]
    else:
        index = len(self._token_to_idx)
        self._token_to_idx[token] = index
        self._idx_to_token[index] = token
    return index

def lookup_token(self, token):
    """ Извлекает соответствующий токenu индекс
        или индекс UNK, если токен не найден.

    Аргументы:
        token (str): токен для поиска
    Возвращает:
        index (int): соответствующий токenu индекс
    Примечания:
        'unk_index' должен быть >=0 (добавлено в Vocabulary)
        для должного функционирования UNK
    """
    if self.add_unk:
        return self._token_to_idx.get(token, self.unk_index)
    else:
        return self._token_to_idx[token]

def lookup_index(self, index):
    """ Возвращает соответствующий индексу токен

    Аргументы:
        index (int): индекс для поиска
    Возвращает:
        token (str): соответствующий индексу токен
    Генерирует:
        KeyError: если индекс не найден в Vocabulary
    """
    if index not in self._idx_to_token:
        raise KeyError("the index (%d) is not in the Vocabulary" % index)
    return self._idx_to_token[index]

def __str__(self):
    return "<Vocabulary(size=%d)>" % len(self)

def __len__(self):
    return len(self._token_to_idx)
```

## Класс Vectorizer

Вторая фаза преобразования текста в векторизованный мини-пакет — проход в цикле по токенам входных данных и преобразование каждого из них в цифровую форму. В результате этого должен получиться вектор. А поскольку этот вектор затем объединяется с векторами для других точек данных, возникает ограничение, гласящее, что длина создаваемых классом `Vectorizer` векторов должна быть одинаковой.

Для этого класс `Vectorizer` инкапсулирует `Vocabulary` обзора, задающий соответствие слов в обзоре целым числам. В примере 3.16 декоратор `@classmethod` языка Python используется для метода `from_dataframe()` класса `Vectorizer` — указывает точку входа для создания экземпляра `Vectorizer`. Метод `from_dataframe()` проходит в цикле по строкам объекта `DataFrame` библиотеки `Pandas` с двумя целями. Первая цель — подсчет частоты каждого из токенов набора данных. Вторая — создание объекта `Vocabulary`, в котором присутствуют только токены, встречающиеся не реже, чем указано в аргументе `cutoff` данного метода. По существу, этот метод ищет все слова, встречающиеся как минимум `cutoff` раз, и добавляет их в объект `Vocabulary`. Поскольку токен `UNK` также добавлен в `Vocabulary`, то при вызове метода `lookup_token()` класса `Vocabulary` для любого из не добавленных в него слов, будет возвращено значение `unk_index`.

Основная функциональность класса `Vectorizer` инкапсулирована в методе `vectorize()`. Он принимает в качестве аргумента строковое значение, содержащее обзор, и возвращает векторизованное представление этого обзора. В данном примере мы используем свернутое унитарное представление, с которым познакомили вас в главе 1. Оно содержит бинарный вектор — вектор из 0 и 1. Его длина равна длине словаря. На соответствующих словам обзора позициях в векторе содержится 1. Отметим, что у этого представления есть некоторые ограничения. Во-первых, оно является разреженным — количество уникальных слов в обзоре всегда намного меньше количества уникальных слов в словаре. Во-вторых, в нем не учитывается порядок, в котором слова встречаются в обзоре (такой подход называется «мультимножество слов»). В последующих главах мы рассмотрим и другие методы, у которых нет таких ограничений.

**Пример 3.16.** Класс `Vectorizer`, преобразующий текст в числовые векторы

```
class ReviewVectorizer(object):
    """ Векторизатор, приводящий словари в соответствие друг другу
        и использующий их """
    def __init__(self, review_vocab, rating_vocab):
        """
        Аргументы:
            review_vocab (Vocabulary): отображает слова
            в целочисленные значения
```

```
        rating_vocab (Vocabulary): отображает метки классов
        в целочисленные значения
    """
    self.review_vocab = review_vocab
    self.rating_vocab = rating_vocab

def vectorize(self, review):
    """ Создает свернутый унитарный вектор для обзора

    Аргументы:
        review (str): обзор
    Возвращает:
        one_hot (np.ndarray): свернутое унитарное представление
    """
    one_hot = np.zeros(len(self.review_vocab), dtype=np.float32)

    for token in review.split(" "):
        if token not in string.punctuation:
            one_hot[self.review_vocab.lookup_token(token)] = 1
    return one_hot

@classmethod
def from_dataframe(cls, review_df, cutoff=25):
    """ Создает экземпляр векторизатора на основе
    объекта DataFrame набора данных

    Аргументы:
        review_df (pandas.DataFrame): набор данных обзоров
        cutoff (int): параметр для фильтрации по частоте вхождения
    Возвращает:
        экземпляр класса ReviewVectorizer
    """
    review_vocab = Vocabulary(add_unk=True)
    rating_vocab = Vocabulary(add_unk=False)

    # Добавить рейтинги
    for rating in sorted(set(review_df.rating)):
        rating_vocab.add_token(rating)

    # Добавить часто встречающиеся слова, если число вхождений
    # больше указанного
    word_counts = Counter()
    for review in review_df.review:
        for word in review.split(" "):
            if word not in string.punctuation:
                word_counts[word] += 1

    for word, count in word_counts.items():
        if count > cutoff:
            review_vocab.add_token(word)
    return cls(review_vocab, rating_vocab)

@classmethod
def from_serializable(cls, contents):
```

```

""" Создает экземпляр ReviewVectorizer на основе
    сериализуемого словаря

Аргументы:
    contents (dict): сериализуемый словарь
Возвращает:
    экземпляр класса ReviewVectorizer
"""
review_vocab = Vocabulary.from_serializable(contents['review_vocab'])
rating_vocab = Vocabulary.from_serializable(contents['rating_vocab'])
return cls(review_vocab=review_vocab, rating_vocab=rating_vocab)

def to_serializable(self):
    """ Создает сериализуемый словарь для кэширования

Возвращает:
    contents (dict): сериализуемый словарь
"""
return {'review_vocab': self.review_vocab.to_serializable(),
        'rating_vocab': self.rating_vocab.to_serializable()}

```

## Класс DataLoader

Последняя фаза конвейера преобразования текста в векторизованный мини-пакет — собственно группировка векторизованных точек данных. Поскольку группировка в мини-пакеты играет столь важную роль в обучении нейронных сетей, фреймворк PyTorch предоставляет для координации этого процесса встроенный класс `DataLoader`. Для создания экземпляра класса `DataLoader` необходимо передать какой-либо объект `Dataset` PyTorch (например, описанный нами для этого примера `ReviewDataset`), `batch_size` и несколько других поименованных аргументов. В результате получается объект, представляющий собой Python-итератор, группирующий и свертывающий содержащиеся в объекте `Dataset` точки данных<sup>1</sup>. В примере 3.17 мы создадим для `DataLoader` адаптер в виде функции `generate_batches()` — генератора для удобного выбора (switch) данных между CPU и GPU.

### Пример 3.17. Генерация мини-пакетов на основе набора данных

```

def generate_batches(dataset, batch_size, shuffle=True,
                    drop_last=True, device="cpu"):
    """
    Функция-генератор — адаптер для объекта DataLoader фреймворка PyTorch.
    Гарантирует размещение всех тензоров на нужном устройстве.
    """
    dataloader = DataLoader(dataset=dataset, batch_size=batch_size,
                           shuffle=shuffle, drop_last=drop_last)

```

<sup>1</sup> Напомним, что для наследования класса `Dataset` PyTorch разработчик должен реализовать методы `__getitem__()` и `__len__()`, благодаря чему класс `DataLoader` сможет пройти в цикле по набору данных путем итерации по индексам из этого набора.

```

for data_dict in dataloader:
    out_data_dict = {}
    for name, tensor in data_dict.items():
        out_data_dict[name] = data_dict[name].to(device)
    yield out_data_dict

```

## Классификатор-перцептрон

Используемая в этом примере модель представляет собой новую реализацию классификатора *Perceptron*, с которой вы уже встречались в начале главы. Класс *ReviewClassifier* наследует класс *Module* фреймворка PyTorch и создает слой преобразования типа *Linear*, возвращающего один результат. Поскольку речь идет о бинарной классификации (обзор может быть позитивным или негативным), этого вполне достаточно. В качестве завершающего нелинейного преобразования используется сигма-функция (пример 3.18).

Благодаря параметризации метода *forward()* применение сигма-функции необязательно. Чтобы разобраться, для чего это нужно, укажем сначала, что наиболее подходящей функцией потерь для задачи бинарной классификации является бинарная функция потерь на основе перекрестной энтропии (*torch.nn.BCELoss()*). Она специально сформулирована математически в расчете на бинарные вероятности. Однако в случае применения сигма-функции, а затем этой функции потерь возникают проблемы с численной устойчивостью. PyTorch предоставляет пользователям более устойчивый численно вариант — *BCEWithLogitsLoss()*. При использовании функции потерь не следует применять сигма-функцию (поэтому по умолчанию она у нас не применяется). Но если пользователю классификатора хотелось бы получить вероятностное значение, понадобится сигма-функция, так что такую возможность необходимо оставить в качестве необязательной. Мы рассмотрим пример подобного ее использования в примере 3.20.

**Пример 3.18.** Классификатор на основе перцептрона для классификации обзоров Yelp

```

import torch.nn as nn
import torch.nn.functional as F

class ReviewClassifier(nn.Module):
    """ Простой классификатор на основе перцептрона """
    def __init__(self, num_features):
        """
        Аргументы:
            num_features (int): размер входного вектора признаков
        """
        super(ReviewClassifier, self).__init__()
        self.fc1 = nn.Linear(in_features=num_features,
                              out_features=1)

    def forward(self, x_in, apply_sigmoid=False):

```



```
""" Прямой проход классификатора

Аргументы:
    x_in (torch.Tensor): входной тензор данных
    x_in.shape должен быть (batch, num_features)
    apply_sigmoid (bool): флаг для сигма-функции активации
    при использовании функции потерь на основе перекрестной
    энтропии должен равняться false

Возвращает:
    итоговый тензор. tensor.shape должен быть (batch,).
"""
y_out = self.fc1(x_in).squeeze()
if apply_sigmoid:
    y_out = F.sigmoid(y_out)
return y_out
```

## Процедура обучения

В этом подразделе мы в общих чертах опишем компоненты процедуры обучения и их взаимодействие с набором данных и моделью с целью подбора параметров модели и повышения ее эффективности. По сути, процедура обучения отвечает за воплощение модели, проход в цикле по набору данных, вычисление (на основе переданных входных данных) выходных значений модели, определение потерь (того, насколько модель ошиблась в предсказаниях) и обновление модели в соответствии с потерями. Хотя на первый взгляд здесь есть масса нюансов, на самом деле точек модификации процедуры обучения не так уж много, поэтому вы скоро привыкнете к ней в процессе разработки алгоритмов глубокого обучения. Для упрощения управления высокоуровневыми решениями мы сосредоточим в объекте `args` (который показан в примере 3.19) все точки принятия решений<sup>1</sup>.

**Пример 3.19.** Гиперпараметры и настройки программы для классификатора обзоров Yelp на основе перцептрона

```
from argparse import Namespace

args = Namespace(
    # Информация о данных и путях
    frequency_cutoff=25,
    model_state_file='model.pth',
    review_csv='data/yelp/reviews_with_splits_lite.csv',
    save_dir='model_storage/ch3/yelp/',
    vectorizer_file='vectorizer.json',
```

<sup>1</sup> Мы воспользовались классом `Namespace` из встроенного пакета `argparse`, поскольку он прекрасно инкапсулирует словарь свойств и хорошо работает со статическими анализаторами кода. Кроме того, если вам понадобится создавать процедуры обучения моделей на основе командной строки, можете переключиться на класс `ArgumentParser` из пакета `argparse` без необходимости менять остальной код.

```

# Гиперпараметры модели отсутствуют
# Гиперпараметры обучения
batch_size=128,
early_stopping_criteria=5,
learning_rate=0.001,
num_epochs=100,
seed=1337,
# Настройки времени выполнения не приводятся для экономии места
)

```

В оставшейся части подраздела мы опишем *состояние обучения* (training state). Оно будет представлять собой небольшой словарь, с помощью которого мы станем отслеживать информацию о процессе обучения. Этот словарь будет расти по мере увеличения количества отслеживаемых вами нюансов процедуры обучения, и при необходимости вы можете его систематизировать. Представленный в нашем следующем примере словарь содержит лишь основную информацию, отслеживаемую при обучении модели. После описания состояния обучения мы вкратце опишем набор объектов, которые необходимо создать для обучения модели. В их числе будут сама модель, набор данных, оптимизатор и функция потерь. В другие примеры и прилагаемый к книге код включены дополнительные компоненты, но перечислять их в тексте мы не станем. Наконец, завершим раздел кодом самого цикла обучения и продемонстрируем стандартный паттерн оптимизации PyTorch.

## Подготовка условий для обучения

Пример 3.20 демонстрирует компоненты обучения, создаваемые для этого примера. Первый из них — начальное состояние обучения. Функция принимает в качестве аргумента объект `args`, чтобы обеспечить возможность обработки сложной информации, но в книге мы не станем показывать подобных сложностей. В прилагаемых материалах можно найти информацию о том, что еще можно использовать в состоянии обучения. Показанный здесь минимальный набор включает индекс эпохи и списки для значений потерь на этапах обучения и проверки, а также значения точности на этапах обучения и проверки. Он также включает два поля для потерь и точности на этапе контроля.

Далее создаются компоненты для набора данных и модели. В этом примере, а также в остальных примерах книги наборы данных отвечают за создание экземпляров векторизаторов. В прилагаемых к книге материалах создание экземпляра набора данных вложено в оператор `if`, что позволяет или загружать ранее созданные векторизаторы, или создавать новый, с сохранением векторизатора на диск. Важно также, что модель размещается на нужном устройстве в соответствии с пожеланиями пользователя (из параметра `args.cuda`), причем, конечно, есть условный оператор для проверки доступности GPU. Целевое устройство указывается при

вызове функции `generate_batches()` в основном цикле обучения, так что данные и модель будут располагаться на одном устройстве.

Осталось создать еще два компонента — функцию потерь и оптимизатор. В этом примере используется функция потерь `BCEWithLogitsLoss()` (как упоминалось в подразделе «Классификатор-перцептрон» на с. 88, наиболее подходящей функцией потерь для бинарной классификации является бинарная функция потерь на основе перекрестной энтропии, и с точки зрения численной устойчивости лучше использовать функцию `BCEWithLogitsLoss()` и модель, в которой к выходным значениям не применяется сигма-функция, чем функцию `BCELoss()` и модель, в которой к выходным значениям сигма-функция применяется). Мы также используем оптимизатор `Adam`. В целом оптимизатор `Adam`, по сравнению с другими, является вполне конкурентоспособным, и на момент написания книги отсутствуют убедительные аргументы, по которым предпочтительнее какой-то другой оптимизатор. Мы призываем вас проверить этот факт самостоятельно, попробовав другие оптимизаторы и сравнив их эффективность.

**Пример 3.20.** Создание набора данных, модели, функции потерь, оптимизатора и состояния обучения

```
import torch.optim as optim

def make_train_state(args):
    return {'epoch_index': 0,
            'train_loss': [],
            'train_acc': [],
            'val_loss': [],
            'val_acc': [],
            'test_loss': -1,
            'test_acc': -1}

train_state = make_train_state(args)

if not torch.cuda.is_available():
    args.cuda = False
args.device = torch.device("cuda" if args.cuda else "cpu")

# Набор данных и векторизатор
dataset = ReviewDataset.load_dataset_and_make_vectorizer(args.review_csv)
vectorizer = dataset.get_vectorizer()

# Модель
classifier = ReviewClassifier(num_features=len(vectorizer.review_vocab))
classifier = classifier.to(args.device)

# Функция потерь и оптимизатор
loss_func = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(classifier.parameters(), lr=args.learning_rate)
```

## Цикл обучения

Цикл обучения использует созданные в начале объекты для обновления параметров модели так, чтобы ее эффективность со временем росла. Точнее говоря, цикл обучения состоит из двух циклов: внутреннего цикла по мини-пакетам из набора данных и внешнего цикла, повторяющего внутренний нужное число раз. Во внутреннем цикле для каждого мини-пакета вычисляется функция потерь, а параметры модели обновляются с помощью оптимизатора. Соответствующий код приведен в примере 3.21; более подробное описание происходящего следует далее.

### Пример 3.21. Простейший цикл обучения

```
for epoch_index in range(args.num_epochs):
    train_state['epoch_index'] = epoch_index

    # Проход в цикле по обучающему набору данных

    # Настройки: создаем генератор пакетов, устанавливаем значения
    # переменных loss и acc равными 0, включаем режим обучения
    dataset.set_split('train')
    batch_generator = generate_batches(dataset,
                                      batch_size=args.batch_size,
                                      device=args.device)

    running_loss = 0.0
    running_acc = 0.0
    classifier.train()

    for batch_index, batch_dict in enumerate(batch_generator):
        # Процедура обучения состоит из пяти шагов:

        # Шаг 1. Обнуляем градиенты
        optimizer.zero_grad()

        # Шаг 2. Вычисляем выходные значения
        y_pred = classifier(x_in=batch_dict['x_data']).float()

        # Шаг 3. Вычисляем функцию потерь
        loss = loss_func(y_pred, batch_dict['y_target'].float())
        loss_batch = loss.item()
        running_loss += (loss_batch - running_loss) / (batch_index + 1)

        # Шаг 4. Получаем градиенты на основе функции потерь
        loss.backward()

        # Шаг 5. Оптимизатор обновляет значения параметров по градиентам
        optimizer.step()

    # -----
    # Вычисляем точность
    acc_batch = compute_accuracy(y_pred, batch_dict['y_target'])
```

```
running_acc += (acc_batch - running_acc) / (batch_index + 1)

train_state['train_loss'].append(running_loss)
train_state['train_acc'].append(running_acc)

# Проход в цикле по проверочному набору данных

# Настройки: создаем генератор пакетов, устанавливаем значения
# переменных loss и acc равными 0, включаем режим проверки
dataset.set_split('val')
batch_generator = generate_batches(dataset,
                                   batch_size=args.batch_size,
                                   device=args.device)

running_loss = 0.
running_acc = 0.
classifier.eval()

for batch_index, batch_dict in enumerate(batch_generator):

    # Шаг 1. Вычисляем выходные значения
    y_pred = classifier(x_in=batch_dict['x_data'].float())

    # Шаг 2. Вычисляем функцию потерь
    loss = loss_func(y_pred, batch_dict['y_target'].float())
    loss_batch = loss.item()
    running_loss += (loss_batch - running_loss) / (batch_index + 1)

    # Шаг 3. Вычисляем точность
    acc_batch = compute_accuracy(y_pred, batch_dict['y_target'])
    running_acc += (acc_batch - running_acc) / (batch_index + 1)

train_state['val_loss'].append(running_loss)
train_state['val_acc'].append(running_acc)
```

В первой строке используется цикл `for`, в котором мы проходим по эпохам. Количество эпох задается в гиперпараметре. Оно определяет количество проходов по набору данных, выполняемых процедурой обучения. На практике лучше воспользоваться чем-то вроде критерия раннего останова, чтобы завершить этот цикл, а не дожидаться его окончания. В прилагаемых к книге материалах показано, как это сделать.

Вверху цикла `for` видим несколько объявлений и задания начальных значений. Во-первых, устанавливается значение индекса для эпохи состояния обучения. Далее указывается нужный фрагмент набора данных (сначала `'train'`, затем, когда мы хотим оценить эффективность модели в конце эпохи, — `'val'` и, наконец, `'test'` для окончательной оценки эффективности модели). Учитывая архитектуру нашего набора данных, необходимо всегда задавать фрагмент до вызова `generate_batches()`. После создания `batch_generator` задаются начальные значения двух переменных с плавающей точкой для отслеживания изменения потерь

и точности от пакета к пакету. Более подробную информацию о примененной здесь «формуле скользящего среднего» вы можете найти в статье «Википедии» (<http://bit.ly/2Ezb9DP> и [https://ru.wikipedia.org/wiki/Скользящая\\_средняя](https://ru.wikipedia.org/wiki/Скользящая_средняя)). Наконец, мы вызываем метод `.train()` классификатора, указывающий, что модель находится в «режиме обучения» и параметры модели изменяемые. Это также активизирует механизмы регуляризации, например дропаут (см. подраздел «Регуляризация многослойных перцептронов: регуляризация весов и структурная регуляризация» на с. №№).

В следующей части цикла обучения мы проходим по пакетам обучения в `batch_generator` и выполняем нужные для обновления параметров модели операции. В процессе итерации по каждому пакету сначала сбрасываются значения градиентов с помощью метода `optimizer.zero_grad()`. Далее вычисляются выходные значения модели. После этого с помощью функции потерь вычисляются потери — разница между выходными значениями модели и целевыми значениями (фактическими метками классов). Вслед за этим вызывается метод `loss.backward()` объекта потерь (не объекта функции потерь), в результате чего градиенты транслируются всем параметрам. Наконец, на основе этих транслированных значений оптимизатор обновляет параметры с помощью метода `optimizer.step()`. Перечисленные пять шагов — необходимые ступеньки градиентного спуска. Помимо них, требуется еще несколько дополнительных операций для учета и отслеживания. Если точнее, то вычисляются (и сохраняются в обычных переменных Python) значения потерь и точности, а затем они используются для обновления переменных для текущих потерь и текущей точности.

После завершения внутреннего цикла по пакетам обучающего фрагмента выполняется несколько дополнительных операций учета и создания/инициализации. Сначала состояние обучения обновляется на основе окончательных значений потерь и точности. Далее создаются новый генератор пакетов и переменные для текущих потерь и текущей точности. Цикл по проверочным данным почти ничем не отличается от цикла по обучающим, так что мы повторно воспользуемся теми же переменными. Впрочем, есть одно существенное отличие: вызывается метод `.eval()` классификатора, выполняющий обратную по отношению к методу `.train()` классификатора операцию. Метод `.eval()` делает параметры модели неизменяемыми и отключает дропаут. В режиме проверки также отключается вычисление потерь и трансляция градиентов обратно в параметры. Это важно, поскольку подстройка параметров модели под проверочные данные нежелательна. Взамен эти данные должны служить мерой эффективности работы модели. Если эффективность ее работы на обучающих и проверочных данных сильно расходится, то модель, вероятно, переобучилась на обучающих данных и следует поменять модель или процедуру обучения (например, использовать ранний останов, как мы делаем в прилагаемых к книге материалах).

Внешний цикл `for` завершается проходом по проверочным данным и сохранением полученных значений потерь и точности. Все реализованные в этой книге про-

цедуры обучения следуют практически одному паттерну проектирования. На самом деле вообще все алгоритмы градиентного спуска следуют схожим паттернам проектирования. Когда написание этого цикла с нуля станет для вас привычным, вы поймете, что такое градиентный спуск.

## Оценка, вывод и просмотр

Следующие шаги после обучения модели — проверка ее эффективности на заранее выделенной порции данных, использование ее для выполнения вывода на основе новых данных или осмотр весов модели и выяснение, чему же она обучилась. В данном подразделе мы разберем все эти три шага.

## Оценка на контрольных данных

Код оценки эффективности модели на выделенном контрольном наборе почти ничем не отличается от цикла проверки в процедуре обучения из предыдущего примера с одним лишь небольшим нюансом: задан фрагмент `'test'`, а не `'val'`. Различие между этими двумя фрагментами набора данных состоит в том, что контрольный набор следует использовать как можно меньше. При каждом запуске обученной модели на контрольном наборе, принятии на основании этого новых решений относительно модели (например, изменения размера слоев) и повторной оценки заново обученной модели на контрольном наборе модельные решения смещаются в сторону контрольных данных. Другими словами, при частом повторении этого процесса контрольные данные не смогут служить точной мерой, как действительно выделенные данные. В этом можно убедиться на примере 3.22.

### Пример 3.22. Оценка на контрольном наборе

Input[0]

```
dataset.set_split('test')
batch_generator = generate_batches(dataset,
                                   batch_size=args.batch_size,
                                   device=args.device)

running_loss = 0.
running_acc = 0.
classifier.eval()

for batch_index, batch_dict in enumerate(batch_generator):
    # вычисление выходных значений
    y_pred = classifier(x_in=batch_dict['x_data'].float())

    # вычисление потерь
    loss = loss_func(y_pred, batch_dict['y_target'].float())
    loss_batch = loss.item()
    running_loss += (loss_batch - running_loss) / (batch_index + 1)
```

```
# вычисление точности
acc_batch = compute_accuracy(y_pred, batch_dict['y_target'])
running_acc += (acc_batch - running_acc) / (batch_index + 1)

train_state['test_loss'] = running_loss
train_state['test_acc'] = running_acc
```

Input[1]

```
print("Test loss: {:.3f}".format(train_state['test_loss']))
print("Test Accuracy: {:.2f}".format(train_state['test_acc']))
```

Output[1]

```
Test loss: 0.297
Test Accuracy: 90.55
```

## Вывод и классификация новых точек данных

Еще один метод оценки эффективности модели — вывод на основе новых данных и анализ того, работает ли она. Рассмотрим пример 3.23.

**Пример 3.23.** Вывод в консоль предсказаний для образца обзора

Input[0]

```
def predict_rating(review, classifier, vectorizer,
                  decision_threshold=0.5):
    """ Предсказание рейтинга обзора

    Аргументы:
        review (str): текст обзора
        classifier (ReviewClassifier): обученная модель
        vectorizer (ReviewVectorizer): соответствующий векторизатор
        decision_threshold (float): численная граница,
        разделяющая различные классы рейтинга
    """

    review = preprocess_text(review)
    vectorized_review = torch.tensor(vectorizer.vectorize(review))
    result = classifier(vectorized_review.view(1, -1))

    probability_value = F.sigmoid(result).item()

    index = 1
    if probability_value < decision_threshold:
        index = 0

    return vectorizer.rating_vocab.lookup_index(index)

test_review = "this is a pretty awesome book"
prediction = predict_rating(test_review, classifier, vectorizer)
print("{} -> {}".format(test_review, prediction))
```

Output[0]

```
this is a pretty awesome book -> positive
```



## Просмотр весов модели

Наконец, последний способ выяснить, хорошо ли работает модель после окончания обучения, — просмотреть веса модели и принять решение, правильно ли они выглядят. Как демонстрирует пример 3.24, в случае перцептрона и свернутого унитарного кодирования эта задача достаточно проста, поскольку каждому весу модели соответствует ровно одно слово из словаря.

**Пример 3.24.** Просматриваем полученные в результате обучения модели веса

Input[0]

```
# Сортировка весов
fc1_weights = classifier.fc1.weight.detach()[0]
_, indices = torch.sort(fc1_weights, dim=0, descending=True)
indices = indices.numpy().tolist()

# Топ-20 позитивных слов
print("Influential words in Positive Reviews:")
print("-----")
for i in range(20):
    print(vectorizer.review_vocab.lookup_index(indices[i]))
```

Output[0]

```
Influential words in Positive Reviews:
-----
great
awesome
amazing
love
friendly
delicious
best
excellent
definitely
perfect
fantastic
wonderful
vegas
favorite
loved
yummy
fresh
reasonable
always
recommend
```

Input[1]

```
# Топ-20 негативных слов
print("Influential words in Negative Reviews:")
print("-----")
indices.reverse()
for i in range(20):
    print(vectorizer.review_vocab.lookup_index(indices[i]))
```

Output[1]

Influential words in Negative Reviews:

```
-----  
worst  
horrible  
mediocre  
terrible  
not  
rude  
bland  
disgusting  
dirty  
awful  
poor  
disappointing  
ok  
no  
overpriced  
sorry  
nothing  
meh  
manager  
gross
```

## Резюме

Из этой главы вы узнали о некоторых фундаментальных понятиях обучения нейронных сетей с учителем. Мы рассмотрели следующие вопросы.

- ❑ Простейшая нейронная сеть — перцептрон.
- ❑ Такие фундаментальные понятия, как функции активации, функции потерь и различные их виды.
- ❑ Цикл обучения, размеры пакетов и эпохи в контексте модельного примера.
- ❑ Суть обобщения, рекомендуемые практики оценки эффективности обобщения на обучающем/контрольном/проверочном фрагментах.
- ❑ Ранняя остановка и другие критерии для выбора момента завершения или схождения алгоритма обучения.
- ❑ Что такое гиперпараметры, несколько их примеров: размер пакета, скорость обучения и т. д.
- ❑ Классификация обзоров ресторанов на английском языке с сайта Yelp с помощью реализованной на PyTorch модели перцептрона и определение эффективности модели по ее весам.

В главе 4 мы познакомим вас с *упреждающими нейронными сетями* (feed-forward networks). Сначала мы получим модель *многослойного перцептрона* (multilayer

perceptron model) путем вертикального и горизонтального расположения рядом простых перцептронов. Затем изучим новый вид упреждающих сетей, основанных на использовании операций свертки для захвата (capture) внутренней структуры языка.

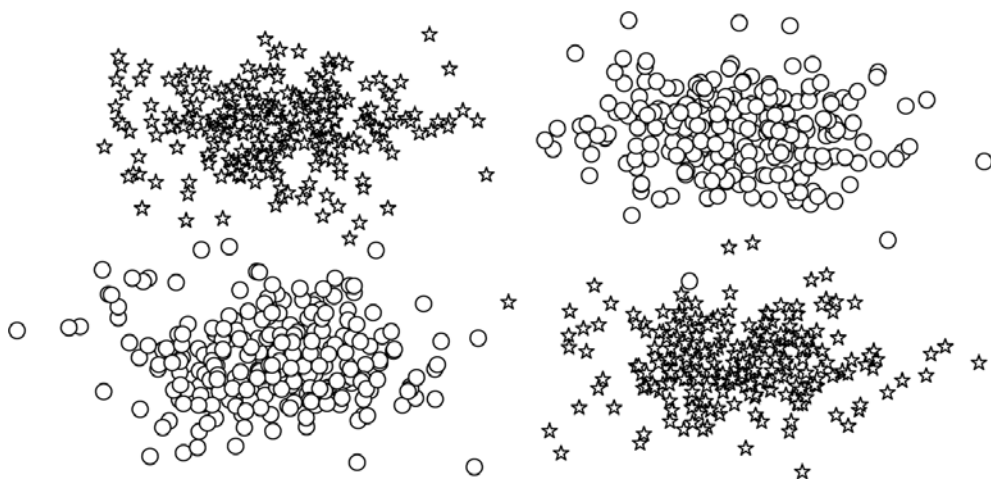
## Библиография

*Zhang X. et al.* Character-Level Convolutional Networks for Text Classification. Proceedings of NIPS, 2015.

# 4

## Использование упреждающих сетей при NLP

В главе 3 мы рассмотрели основные понятия нейронных сетей на примере перцептрона — простейшей из возможных нейронных сетей. Один из недостатков перцептрона — невозможность усвоения им довольно несложных паттернов данных. Например, рассмотрим приведенные на рис. 4.1 точки данных. Этот рисунок эквивалентен ситуации «или — или» (XOR), при которой не существует границы решения в виде одной прямой (то есть невозможно *линейное разделение*). В этом случае перцептрон не подходит.



**Рис. 4.1.** Два класса из набора данных XOR нарисованы в виде кругов и звезд. Обратите внимание, что эти классы не может разделить никакая прямая линия

В этой главе мы изучим семейство моделей нейронных сетей, которые обычно называют *упреждающими сетями* (feed-forward networks). Мы сосредоточим наше внимание на двух видах упреждающих сетей: многослойном перцептроне (multilayer perceptron, MLP) и сверточной нейронной сети (convolutional neural network, CNN)<sup>1</sup>. Многослойный перцептрон представляет собой расширение изученного в главе 3 простого перцептрона — группировку нескольких перцептронов в один слой и размещение нескольких слоев один поверх другого. Мы обсудим многослойные перцептроны буквально через несколько абзацев и разберем пример многоклассовой классификации с их помощью в разделе «Пример: классификация фамилий с помощью MLP» на с. 109.

Создание второго из описываемых в данной главе видов упреждающих нейронных сетей — сверточных нейронных сетей — было вдохновлено использованием оконных фильтров при обработке цифровых сигналов. Благодаря своей оконной природе CNN способны улавливать во входных данных локальные паттерны, что делает их не только основополагающим компонентом машинного зрения, но и идеальным кандидатом для обнаружения подструктур в последовательных данных, например слов и предложений. Мы изучим CNN в разделе «Сверточные нейронные сети» на с. 120 и продемонстрируем их использование в разделе «Пример: классификация фамилий с помощью CNN» на с. 130.

MLP и CNN объединены в этой главе потому, что и те и другие являются упреждающими нейронными сетями и отличаются от другого семейства нейронных сетей — рекуррентных нейронных сетей (recurrent neural networks, RNN), включающих такую обратную связь (циклы), при которой каждое из вычислений получает информацию от предыдущего. В главах 6 и 7 мы разберем различные RNN и расскажем, какую пользу могут принести циклы в сетевой структуре.

При обсуждении этих различных моделей, чтобы лучше разобраться в работе алгоритмов, старайтесь обращать внимание на размер и форму участвующих в вычислениях тензоров данных. Каждый из типов слоев нейронных сетей по-своему влияет на размер и форму тензоров данных, и понимание нюансов этого исключительно важно для полноценного восприятия моделей.

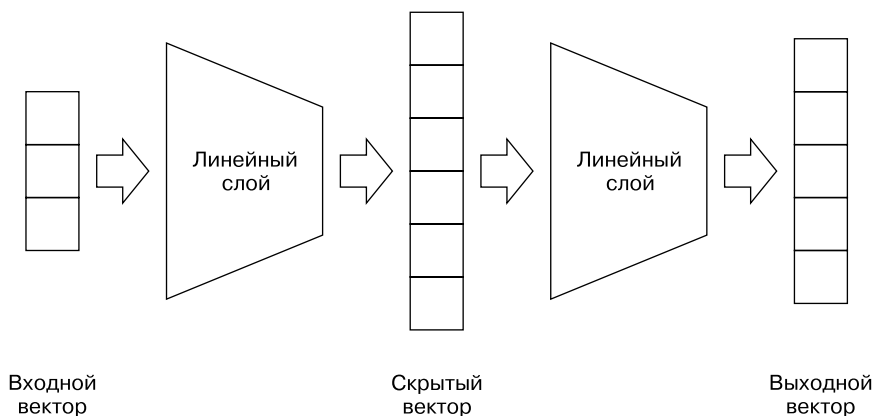
## Многослойный перцептрон

Многослойные перцептроны считаются одними из основных стандартных блоков нейронных сетей. Простейший многослойный перцептрон представляет собой расширение понятия перцептрона из главы 3. Перцептрон принимает на входе вектор

<sup>1</sup> Упреждающей называется любая нейронная сеть, в которой данные движутся в одном направлении (то есть от входа к выходу). Согласно этому определению, перцептрон также является упреждающей моделью, но обычно этот термин применяют для более сложных моделей из нескольких блоков.

данных<sup>1</sup> и вычисляет одно выходное значение. В MLP несколько перцептронов группируются таким образом, что выходной результат отдельного слоя представляет собой новый вектор, а не просто одно значение. Во фреймворке PyTorch, как вы увидите далее, для этого достаточно указать число выходных признаков в слое `Linear`. Дополнительная особенность MLP заключается в том, что между каждыми двумя слоями вставляется нелинейность.

Простейший MLP, показанный на рис. 4.2, состоит из трех фаз представления и двух слоев `Linear`. Первая фаза — *входной вектор* (input vector) — вектор, передаваемый модели. В разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76 входным вектором служило свернутое унитарное представление обзора с сайта Yelp. Первый слой `Linear` вычисляет на основе полученного входного вектора *скрытый вектор* (hidden vector) — вторую фазу представления. Скрытый вектор получил такое название потому, что предоставляет собой выходное значение слоя, расположенного между входом и выходом. Что мы имеем в виду под выходным значением слоя? Содержащиеся в скрытом векторе значения являются выходными для различных перцептронов, из которых слой состоит. На основе этого скрытого вектора второй линейный слой вычисляет *выходной вектор* (output vector). При бинарной задаче вроде классификации обзоров Yelp размер выходного вектора может быть равен 1. При мультиклассовой же классификации, как вам предстоит увидеть в разделе «Пример: классификация фамилий с помощью MLP» на с. 109, размер выходного вектора равен числу классов. Хотя на рис. 4.2 показан только один скрытый вектор, промежуточных фаз может быть



**Рис. 4.2.** Визуальное представление MLP с двумя линейными слоями и тремя фазами представления: входным, скрытым и выходным векторами

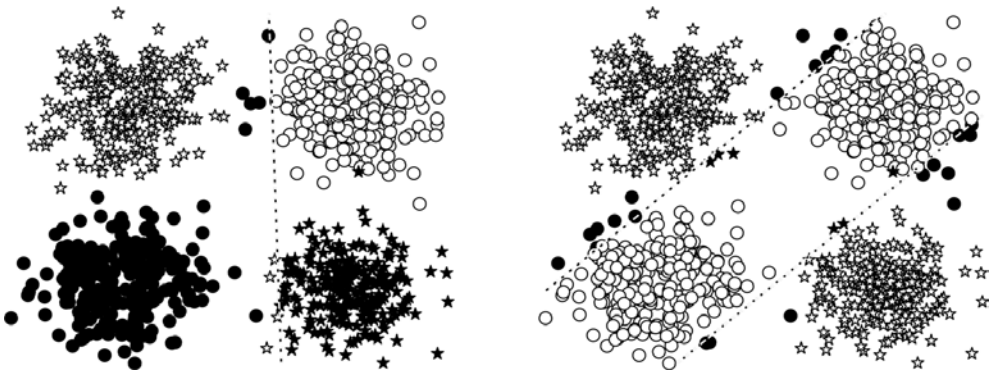
<sup>1</sup> В терминологии PyTorch — тензор. Помните, что вектор представляет собой частный случай тензора. В этой главе и остальной части книги мы будем попеременно использовать слова «вектор» и «тензор», когда это имеет смысл.

несколько, на выходе каждой из которых генерируется скрытый вектор. При этом последний скрытый вектор отображается на выходной вектор путем сочетания линейного слоя и нелинейности.

Функции MLP столь широки именно благодаря добавлению второго линейного слоя и возможности обучения модели *линейно разделяемому* промежуточному представлению, то есть такому, в котором можно различать точки данных по тому, с какой стороны одной прямой (в общем случае гиперплоскости) они находятся. *Обучение промежуточных представлений со специальными свойствами, например линейно разделяемых в задаче классификации, — один из важнейших результатов применения нейронных сетей, сама суть их возможностей моделирования.* В следующем подразделе мы гораздо подробнее обсудим, что это значит.

## Простой пример: XOR

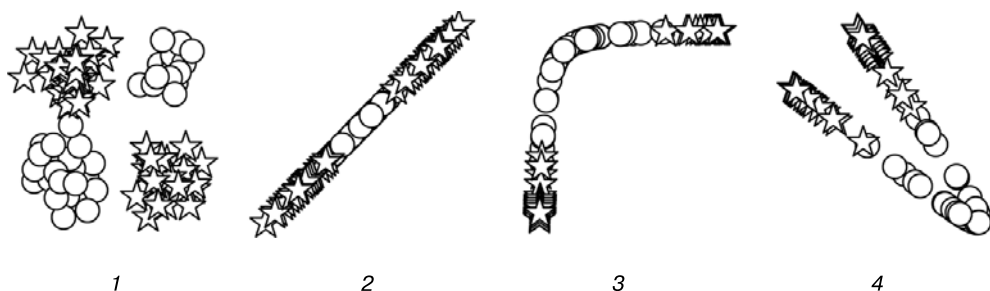
Взглянем на описанный выше пример XOR и сравним использование в нем обычного перцептрона с многослойным. В этом примере производится обучение как обычного перцептрона, так и MLP в задаче бинарной классификации: идентификации звезд и кругов. Все точки данных представляют собой двумерные координаты. Окончательная модель предсказаний (без подробностей реализации) показана на рис. 4.3. На этом графике неправильно классифицированные точки данных закрашены черным, а правильно классифицированные — не закрашены. Слева видно, что у обычного перцептрона не получается найти границу решений для разделения кругов и звезд, как свидетельствуют заполненные фигуры.



**Рис. 4.3.** Решения, полученные в результате обучения обычного перцептрона (слева) и MLP (справа) для задачи XOR. Фактический класс каждой из точек данных определяет ее форму: звезду или круг. Неправильно классифицированные точки данных закрашены черным, а правильно классифицированные — нет. Пунктирные линии соответствуют границам принятия решений для каждой из моделей. Слева в результате обучения перцептрона получается граница решений, не отделяющая правильно звезды от кругов. На самом деле одной прямой для этого недостаточно. Справа MLP успешно обучается разделять звезды и круги

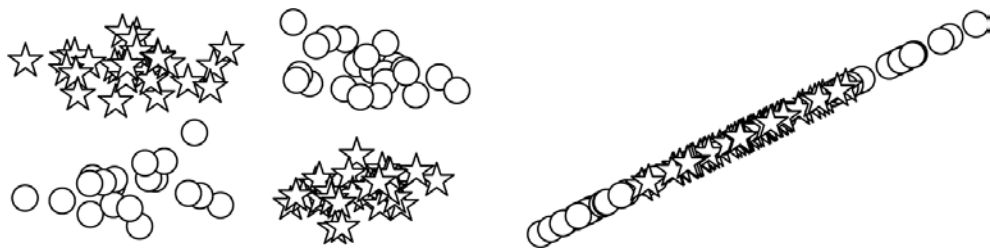
MLP же (справа) гораздо точнее находит границу решений для классификации кругов и звезд.

Хотя из графика кажется, что границ решений у MLP две и в этом состоит его преимущество, на самом деле граница решений всего одна! Граница решений кажется двойной, поскольку промежуточное представление так трансформирует пространство, что одна гиперплоскость оказывается в двух этих местах. На рис. 4.4 можно увидеть вычисленные MLP промежуточные значения. Форма точек указывает на их класс (звезда или круг). Как видим, нейронная сеть (в данном случае MLP) обучается так деформировать пространство данных, что к моменту прохода через последний слой оказывается возможно разделить набор данных одной плоскостью.



**Рис. 4.4.** Входные данные и промежуточные представления MLP: 1 — входные данные сети; 2 — выходные данные первого линейного модуля; 3 — выходные данные первой нелинейности; 4 — выходные данные второго линейного модуля. Как вы можете видеть, в выходных данных первого линейного модуля круги и звезды сгруппированы, а в выходных данных второго линейного модуля точки данных реорганизованы таким образом, что стали линейно разделяемы

Напротив, как демонстрирует рис. 4.5, у обычного перцептрона нет дополнительного слоя, который позволил бы преобразовать формы данных так, чтобы их можно было разделить линейно.



**Рис. 4.5.** Входное и выходное представления перцептрона. Поскольку в нем отсутствует промежуточное представление для группировки и реорганизации данных, как у MLP, разделить звезды и круги он не может



## Реализация многослойных перцептронов в PyTorch

В предыдущем подразделе мы вкратце описали основные идеи MLP. Здесь шаг за шагом рассмотрим его реализацию на PyTorch. Как мы описали выше, MLP, в отличие от обычного перцептрона из главы 3, включает еще один слой вычислений. В реализации, представленной в примере 4.1, воплощена идея с двумя модулями `Linear` PyTorch. Объекты `Linear` называются здесь `fc1` и `fc2` по общепринятому соглашению, именуемому модуль `Linear` полносвязным слоем (fully connected layer, или для краткости `fc layer`)<sup>1</sup>. Помимо этих двух линейных слоев, в нем присутствует нелинейность — выпрямленный линейный блок (ReLU, с которым мы познакомились в разделе «Функции активации» на с. 60) — он применяется к выходным результатам первого линейного слоя до отправки их на вход второму линейному слою. По причине последовательной сущности слоев необходимо убедиться в том, что количество выходных значений слоя равно количеству входных значений следующего слоя. Нелинейный блок между двумя линейными слоями нужен, поскольку без него два линейных слоя математически эквивалентны одному<sup>2</sup>, а значит, не смогут моделировать сложные паттерны. В нашей реализации MLP выполняется только прямой проход метода обратного распространения ошибки. Дело в том, что фреймворк PyTorch автоматически выясняет, как выполнить обратный проход и обновить градиенты, на основе определения модели и реализации прямого прохода.

**Пример 4.1.** Реализация многослойного перцептрона с помощью фреймворка PyTorch

```
import torch.nn as nn
import torch.nn.functional as F

class MultilayerPerceptron(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        Аргументы:
            input_dim (int): размер входных векторов
            hidden_dim (int): размер выходных результатов первого линейного слоя
            output_dim (int): размер выходных результатов второго линейного слоя
        """
        super(MultilayerPerceptron, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """ Прямой проход MLP

        Аргументы:
```

<sup>1</sup> Это распространенная практика в литературе по глубокому обучению. Если полносвязных слоев более одного, они нумеруются слева направо: `fc-1`, `fc-2` и т. д.

<sup>2</sup> Это можно легко доказать, записав уравнения линейных слоев. Предлагаем вам сделать это в качестве упражнения.

```

x_in (torch.Tensor): тензор входных данных
    Значение x_in.shape должно быть (batch, input_dim)
apply_softmax (bool): флаг для многомерной логистической функции
активации. При использовании функции потерь на основе
перекрестной энтропии должен равняться false
Возвращает:
    итоговый тензор. Значение tensor.shape должно
    быть (batch, output_dim)
"""
intermediate = F.relu(self.fc1(x_in))
output = self.fc2(intermediate)

if apply_softmax:
    output = F.softmax(output, dim=1)
return output

```

В примере 4.2 мы воплощаем MLP. Благодаря обобщенному характеру реализации MLP входные данные могут быть любого размера. Для демонстрации возьмем входные данные размерностью 3, выходные — размерностью 4, а скрытое представление — размерностью 100. Обратите внимание, как в выводимых оператором `print` результатах количества блоков в слоях выстраиваются таким образом, чтобы для входных данных размерностью 3 сгенерировать выходные данные размерностью 4.

#### Пример 4.2. Пример воплощения MLP

Input[0]

```

batch_size = 2 # число вводимых за один раз выборков
input_dim = 3
hidden_dim = 100
output_dim = 4

# Инициализация модели
mlp = MultilayerPerceptron(input_dim, hidden_dim, output_dim)
print(mlp)

```

Output[0]

```

MultilayerPerceptron(
  (fc1): Linear(in_features=3, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=4, bias=True)
  (relu): ReLU()
)

```

Для быстрой проверки работы модели мы можем передать в нее случайные входные данные, как показано в примере 4.3. Поскольку модель пока не обучена, выходные данные также носят случайный характер. Подобная предварительная проверка работоспособности полезна перед тем, как тратить время на обучение модели. Обратите внимание, как благодаря диалоговому режиму PyTorch все это можно сделать в режиме реального времени при разработке примерно так же, как при использовании NumPy и Pandas.

**Пример 4.3.** Тестирование MLP на случайных входных данных

Input[0]

```
def describe(x):
    print("Type: {}".format(x.type()))
    print("Shape/size: {}".format(x.shape))
    print("Values: \n{}".format(x))

x_input = torch.rand(batch_size, input_dim)
describe(x_input)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.8329,  0.4277,  0.4363],
        [ 0.9686,  0.6316,  0.8494]])
```

Input[1]

```
y_output = mlp(x_input, apply_softmax=False)
describe(y_output)
```

Output[1]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 4])
Values:
tensor([[ -0.2456,  0.0723,  0.1589, -0.3294],
        [-0.3497,  0.0828,  0.3391, -0.4271]])
```

Важно разобраться, как читать входные и выходные данные моделей PyTorch. В предыдущем примере выходные данные модели MLP представляли собой тензор из двух строк и четырех столбцов. Строки тензора соответствуют измерению пакета — числу точек данных в мини-пакете. Столбцы — итоговые векторы признаков для каждой из точек данных<sup>1</sup>. В некоторых случаях, например при классификации, вектор признаков является *вектором предсказаний* (prediction vector). Название «вектор предсказаний» означает, что он соответствует распределению вероятности. Судьба вектора предсказаний зависит от того, выполняем мы обучение или вывод. Во время обучения выходные данные используются «как есть», вместе с функцией потерь и представлением целевых меток классов<sup>2</sup>. Мы обсудим это подробнее в разделе «Пример: классификация фамилий с помощью MLP» на с. 109.

<sup>1</sup> Иногда называемые «векторами представления».

<sup>2</sup> Выходные данные модели и функции потерь во фреймворке PyTorch должны быть согласованы между собой. Подробнее это описано в документации (<http://bit.ly/2RFOIjM>); например, там указано, какие функции потерь требуют предшествующего многомерной логистической функции вектора предсказаний, а какие — нет. Причины этого основываются на математических упрощениях и соображениях численной устойчивости.

Впрочем, чтобы на основе вектора предсказаний получить вероятности, необходим дополнительный шаг. А именно функция активации на основе многомерной логистической функции для преобразования вектора значений в вероятности. Исторически многомерная логистическая функция применялась во многих сферах. В физике она носит название распределения Больцмана или Гиббса; в статистике это полиномиальная логистическая регрессия; а в сообществе специалистов по NLP она известна в качестве классификатора методом максимальной энтропии (MaxEnt)<sup>1</sup>. Какое бы название ни использовалось, все равно понятно, что при больших положительных значениях вероятности будут больше, а при меньших отрицательных — меньше. В примере 4.3 этот дополнительный шаг определяется аргументом `apply_softmax`. В примере 4.4 выводится то же самое, но флаг `apply_softmax` теперь равен `true`.

**Пример 4.4.** Вывод вероятностей с помощью классификатора на основе многослойного перцептрона (обратите внимание на параметр `apply_softmax=True`)

Input[0]

```
y_output = mlp(x_input, apply_softmax=True)
describe(y_output)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 4])
Values:
tensor([[ 0.2087,  0.2868,  0.3127,  0.1919],
        [ 0.1832,  0.2824,  0.3649,  0.1696]])
```

В заключение упомянем, что многослойные перцептроны представляют собой расположенные ярусами линейные слои, отображающие тензоры в другие тензоры. Между каждой парой линейных слоев располагается нелинейность, прерывающая линейную связь, благодаря которой модель может деформировать окружающее векторное пространство. При классификации подобная деформация должна обеспечивать линейную разделяемость классов. Кроме того, можно воспользоваться многомерной логистической функцией для интерпретации результатов MLP как вероятностей, но не следует применять многомерную логистическую функцию с определенными функциями потерь<sup>2</sup>, поскольку в основе их реализации могут лежать различные математические/вычислительные упрощения.

<sup>1</sup> Это очень существенный вопрос. Более глубокое его изучение выходит за рамки данной книги, но мы рекомендуем вам прочитать посвященное этой теме руководство Френка Ферраро (Frank Ferraro) и Джейсона Айзнера (Jason Eisner) (<http://bit.ly/2rPeW8G>).

<sup>2</sup> Хотя мы и упоминаем эту особенность, но не станем углубляться в подробности взаимодействий между выходными нелинейностями и функциями потерь. Этот вопрос подробно и понятно изложен в документации фреймворка PyTorch (<http://bit.ly/2RFOIjM>) — именно к ней вам следует обращаться при необходимости.

## Пример: классификация фамилий с помощью MLP

В этом разделе мы воспользуемся MLP для классификации фамилий по стране их происхождения. Вывод демографической информации (такой как национальность) на основе общедоступных данных применяется во множестве сфер, начиная от рекомендаций товаров до обеспечения равных возможностей пользователей из разных стран. Тем не менее демографические характеристики и другие не обезличенные атрибуты относятся к «защищаемым законом атрибутам». Использовать их при моделировании и в программных продуктах следует осторожно<sup>1</sup>. Мы начнем с разбиения фамилий на символы и их обработки методом, аналогичным обработке слов, описанной в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76. Помимо различий в данных, модели на уровне символов практически схожи по структуре и реализации с основанными на словах моделями<sup>2</sup>.

Важный урок, который следует вынести из этого примера: реализация и обучение MLP представляет собой очевидное развитие методики реализации и обучения перцептрона из главы 3. На самом деле мы будем на протяжении всей книги вспоминать пример из главы 3 для более подробного описания этих компонентов. Далее, мы не станем включать код, приведенный в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76. Чтобы увидеть код примеров целиком, обратитесь к прилагаемым к книге материалам<sup>3</sup>.

Данный раздел начинается с описания набора данных фамилий и его предварительной обработки. Далее мы пошагово опишем конвейер преобразования строки с фамилией в векторизованный мини-пакет с помощью классов `Vocabulary`, `Vectorizer` и `DataLoader`. Прочитавшие главу 3 встретят эти классы как старых приятелей, только с небольшими модификациями.

Далее в этом разделе вас ждет описание модели `surnameclassifier` и обоснование ее архитектуры. MLP здесь схож с примером перцептрона из главы 3, но, помимо изменений модели, в этом примере появятся многоклассовые выходные данные и соответствующие функции потерь. После описания модели мы рассмотрим

<sup>1</sup> Обсуждение этических вопросов NLP можно найти на сайте [ethicsinml.org](https://ethicsinml.org).

<sup>2</sup> Интересно, что недавние исследования показали: включение моделей на уровне символов в модели на уровне слов могут повысить эффективность последних. См. книгу Петерса и др. (Peters et al., 2018).

<sup>3</sup> См. блокнот `/chapters/chapter_4/4_2_mlp_surnames/4_2_Classifying_Surnames_with_an_MLP.ipynb` из репозитория GitHub этой книги (<https://nlproc.info/PyTorchNLPBook/hero/>).

процедуру обучения. Она очень напоминает уже виденную вами в подразделе «Процедура обучения» на с. 114, так что для краткости мы не станем рассматривать ее так подробно, как там. Мы очень рекомендуем вам вернуться к этому подразделу за дополнительными пояснениями.

Завершается этот пример оценкой модели на контрольной порции набора данных и описанием процедуры вывода для новой фамилии. В многоклассовых предсказаниях хорошо то, что можно получить не только наиболее вероятное предсказание, и мы дополнительно обсудим, как вывести  $k$  наиболее вероятных предсказаний для новой фамилии.

## Набор данных фамилий

В этом примере мы воспользуемся *набором данных фамилий*, состоящим из 10 000 фамилий людей 18 разных национальностей, собранных авторами из различных источников в Интернете. Этот набор данных будет использоваться в нескольких примерах в книге. Он обладает несколькими интересными свойствами. Первое из них — относительная несбалансированность. Три наиболее многочисленных класса охватывают более 60 % данных: 27 % английских, 21 % русских и 14 % арабских. Частота оставшихся 15 национальностей убывает — свойство, присущее и языкам. Второе свойство — достоверная и интуитивно понятная связь между национальностью и написанием фамилии. Многие особенности написания фамилий сильно связаны со страной происхождения (например, O'Neill (О'Нил), Antonopoulos (Антонопулос), Nagasawa (Нагасава) или Zhu (Чжу)).

Начнем создание итогового набора данных с несколько менее обработанной версии, чем та, что предложена в прилагаемых к книге материалах, и произведем несколько операций модификации набора данных. Первая нацелена на исправление дисбаланса — в исходном наборе данных было более 70 % русских фамилий, возможно, из-за систематической ошибки выборки или большого числа неповторяющихся русских фамилий. Для этого выполним прореживание этого слишком широко представленного класса, выбрав случайным образом подмножество фамилий из числа помеченных как русские. Далее группируем набор данных по национальностям и разбиваем его на три фрагмента: 70 % отводим на обучающий набор данных, 15 % на проверочный и 15 % на контрольный, чтобы распределения меток классов между фрагментами были сравнимыми.

Реализация `SurnameDataset` практически идентична классу `ReviewDataset`, показанному в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76, с единственными небольшими отличиями в реализации метода `__getitem__()`<sup>1</sup>. Напомним, что представленные в книге классы наборов данных

<sup>1</sup> Изменены также некоторые имена переменных, чтобы отразить их роль/содержимое.

наследуют класс `Dataset` фреймворка PyTorch, поэтому нам нужно реализовать два метода: `__getitem__()`, возвращающий точку данных по индексу, и `__len__()`, возвращающий длину набора данных. Различие этого примера и примера из главы 3 — в методе `__getitem__()` (пример 4.5). Он возвращает не векторизованный обзор, как в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76, а векторизованную фамилию и индекс соответствующей национальности.

**Пример 4.5.** Реализация метода `SurnameDataset.__getitem__()`

```
class SurnameDataset(Dataset):
    # Реализация практически идентична примеру 3.14

    def __getitem__(self, index):
        row = self._target_df.iloc[index]
        surname_vector = \
            self._vectorizer.vectorize(row.surname)
        nationality_index = \
            self._vectorizer.nationality_vocab.lookup_token(row.nationality)

        return {'x_surname': surname_vector,
                'y_nationality': nationality_index}
```

## Классы `Vocabulary`, `Vectorizer` и `DataLoader`

Для классификации фамилий по символам мы воспользуемся классами `Vocabulary`, `Vectorizer` и `DataLoader`, чтобы преобразовать фамилии в виде строковых значений в векторизованные мини-пакеты. Мы используем те же структуры данных, что и в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76, демонстрируя тем самым полиморфизм, позволяющий обрабатывать символьные токены фамилий аналогично токенам-словам из обзоров Yelp. Вместо векторизации путем отображения токенов-слов в целочисленные значения данные векторизуются с помощью отображения символов в целочисленные значения.

### Класс `Vocabulary`

Используемый в этом примере класс `Vocabulary` в точности аналогичен тому, который использовался в примере 3.16 для отображения слов обзоров Yelp в соответствующие целочисленные значения. Отметим, что `Vocabulary` представляет собой сочетание двух словарей Python, формирующих взаимно однозначное соответствие между токенами (в данном случае символьными) и целочисленными значениями; а именно, первый словарь задает соответствие символов целочисленным индексам, а второй задает соответствие целочисленных индексов символам. Метод `add_token()` служит для добавления новых токенов в `Vocabulary`, метод `lookup_token()` — для извлечения индекса, а `lookup_index()` — для извлечения токена по заданному индексу (что удобно в фазе вывода). В отличие от `Vocabulary`

для обзоров Yelp, мы используем унитарное представление<sup>1</sup>, не подсчитываем частоту вхождения символов и ограничиваемся только часто встречающимися элементами. В основном потому, что набор данных невелик и большинство символов встречаются достаточно часто.

## Класс SurnameVectorizer

Если класс `Vocabulary` преобразует отдельные токены (символьные) в целые числа, то класс `SurnameVectorizer` отвечает за применение `Vocabulary` и преобразование фамилии в вектор. Создание экземпляра и его использование аналогичны работе класса `ReviewVectorizer` в пункте «Класс `Vectorizer`» на с. 85, но с одним ключевым отличием: строковое значение не разбивается по пробелам. Фамилии представляют собой последовательности символов, каждый из которых — отдельный токен в нашем словаре. Однако вплоть до раздела «Сверточные нейронные сети» на с. 120 мы будем игнорировать информацию о последовательности и создавать свернутое унитарное представление входных данных, проходя в цикле по всем символам входной строки. Для ранее не встреченных символов мы выделим специальный токен `UNK`. Он используется в словаре символов, поскольку мы создаем экземпляр `Vocabulary` на основе только обучающих данных, и в проверочных/контрольных данных вполне могут оказаться уникальные символы<sup>2</sup>.

Следует заметить, что, хотя в этом примере используется свернутое унитарное представление, в последующих главах вы узнаете и о других методах векторизации, альтернативных, а иногда и превосходящих унитарное кодирование. А именно, в разделе «Пример: классификация фамилий с помощью CNN» на с. 130 вы встретите матрицу из уникальных векторов, в которой каждый символ занимает определенную позицию и обладает своим собственным унитарным вектором. Далее, в главе 5, вы узнаете о слое вложений — векторизации, которая возвращает вектор целочисленных значений, и создании на их основе матрицы плотных векторов. А пока рассмотрим код класса `SurnameVectorizer` (пример 4.6).

### Пример 4.6. Реализация класса `SurnameVectorizer`

```
class SurnameVectorizer(object):
    """ Векторизатор, приводящий словари в соответствие друг другу
        и использующий их"""
    def __init__(self, surname_vocab, nationality_vocab):
        self.surname_vocab = surname_vocab
        self.nationality_vocab = nationality_vocab

    def vectorize(self, surname):
```

<sup>1</sup> См. описание унитарных представлений в подразделе «Унитарное представление» на с. 24.

<sup>2</sup> А при нашем разбиении данных в проверочном наборе действительно есть уникальные символы, которые могут привести к сбою обучения, если не использовать `UNK`.



```

""" Векторизация передаваемой фамилии

Аргументы:
    surname (str): фамилия
Возвращает:
    one_hot (np.ndarray): свернутое унитарное представление
"""
vocab = self.surname_vocab
one_hot = np.zeros(len(vocab), dtype=np.float32)
for token in surname:
    one_hot[vocab.lookup_token(token)] = 1
return one_hot

@classmethod
def from_dataframe(cls, surname_df):
    """ Создает экземпляр векторизатора на основе объекта DataFrame
        набора данных

    Аргументы:
        surname_df (pandas.DataFrame): набор данных фамилий
    Возвращает:
        экземпляр SurnameVectorizer
    """
    surname_vocab = Vocabulary(unk_token="@")
    nationality_vocab = Vocabulary(add_unk=False)

    for index, row in surname_df.iterrows():
        for letter in row.surname:
            surname_vocab.add_token(letter)
            nationality_vocab.add_token(row.nationality)

    return cls(surname_vocab, nationality_vocab)

```

## Модель SurnameClassifier

Класс `SurnameClassifier` (пример 4.7) — реализация MLP, представленного ранее в главе. Первый линейный слой отображает входные векторы в промежуточный вектор, к которому применяется нелинейность. Второй линейный слой отображает промежуточный вектор в вектор предсказаний.

На последнем шаге может применяться многомерная логистическая функция для приведения суммы выходных значений к 1, то есть их интерпретации как «вероятностей»<sup>1</sup>. Причина необязательности ее применения кроется в математической формулировке используемой функции потерь — функции потерь на основе

<sup>1</sup> Мы намеренно написали слово «вероятностей» в кавычках, чтобы подчеркнуть, что это вовсе не настоящие вероятности в байесовском смысле, но поскольку сумма выходных значений равна 1, то это допустимое распределение, которое можно интерпретировать как вероятности. Это одно из самых занудных примечаний в данной книге, так что можете смело его игнорировать, просто закройте на это глаза и называйте их вероятностями.

перекрестной энтропии, с которой мы познакомились в разделе «Функции потерь» на с. 64. Напомним, что функция потерь на основе перекрестной энтропии лучше всего подходит для многоклассовой классификации, но вычисление многомерной логистической функции во время обучения — напрасная трата ресурсов. Кроме того, во многих случаях она может приводить к численной неустойчивости.

**Пример 4.7.** Реализация класса SurnameClassifier с помощью MLP

```
import torch.nn as nn
import torch.nn.functional as F

class SurnameClassifier(nn.Module):
    """ Многослойный перцептрон с двумя слоями для классификации фамилий """
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        Аргументы:
            input_dim (int): размер входных векторов
            hidden_dim (int): размер выходных векторов первого линейного слоя
            output_dim (int): размер выходных векторов второго линейного слоя
        """
        super(SurnameClassifier, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """ Прямой проход классификатора

        Аргументы:
            x_in (torch.Tensor): входной тензор данных
                Значение x_in.shape должно быть (batch, input_dim)
            apply_softmax (bool): флаг для многомерной логистической функции
                активации. При использовании функции потерь на основе
                перекрестной энтропии должен равняться false

        Возвращает:
            итоговый тензор. Значение tensor.shape должно
                быть (batch, output_dim).
        """
        intermediate_vector = F.relu(self.fc1(x_in))
        prediction_vector = self.fc2(intermediate_vector)

        if apply_softmax:
            prediction_vector = F.softmax(prediction_vector, dim=1)

        return prediction_vector
```

## Процедура обучения

Хотя в этом примере мы используем другую модель, набор данных и функцию потерь, процедура обучения не отличается от описанной в предыдущей главе. Поэтому в примере 4.8 мы приведем только `args` и основные различия процедуры обучения между этим примером и примером из раздела «Пример: классификация тональностей обзоров ресторанов» на с. 76.

**Пример 4.8.** Гиперпараметры и настройки программы для классификатора фамилий на основе MLP

```
args = Namespace(
    # Информация о данных и путях
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch4/surname_mlp",
    # Гиперпараметры модели
    hidden_dim=300
    # Гиперпараметры обучения
    seed=1337,
    num_epochs=100,
    early_stopping_criteria=5,
    learning_rate=0.001,
    batch_size=64,
    # Настройки времени выполнения не приводятся для экономии места
)
```

Наиболее заметное различие в обучении относится к выходным данным модели и используемой функции потерь. В этом примере выходные данные представляют собой вектор многоклассовых предсказаний, который можно преобразовать в вероятности. Для таких выходных данных подходят только функции потерь `CrossEntropyLoss()` и `NLLLoss()`. Воспользуемся первой.

В примере 4.9 приведено создание экземпляров набора данных, модели, функции потерь и оптимизатора. Они практически идентичны тем, что описывались в примере из главы 3. Собственно, это справедливо практически для всех примеров в последующих главах.

**Пример 4.9.** Создание экземпляров набора данных, модели, функции потерь и оптимизатора

```
dataset = SurnameDataset.load_dataset_and_make_vectorizer(args.surname_csv)
vectorizer = dataset.get_vectorizer()

classifier = SurnameClassifier(input_dim=len(vectorizer.surname_vocab),
                              hidden_dim=args.hidden_dim,
                              output_dim=len(vectorizer.nationality_vocab))

classifier = classifier.to(args.device)

loss_func = nn.CrossEntropyLoss(dataset.class_weights)
optimizer = optim.Adam(classifier.parameters(), lr=args.learning_rate)
```

## Цикл обучения

Цикл обучения для этого примера практически идентичен описанному в пункте «Цикл обучения» на с. 92, за исключением названий переменных. А именно, из примера 4.10 видно, что для получения данных из `batch_dict` используются другие ключи. Помимо этого незначительного различия, функциональность цикла

обучения осталась такой же. На основе обучающих данных вычисляются выходные данные модели, потеря и градиентов. Затем модель обновляется на основе градиентов.

**Пример 4.10.** Фрагмент цикла обучения

# Процедура обучения состоит из следующих пяти шагов:

```
# -----  
# Шаг 1. Обнуляем градиенты  
optimizer.zero_grad()  
  
# Шаг 2. Вычисляем выходные значения  
y_pred = classifier(batch_dict['x_surname'])  
  
# Шаг 3. Вычисляем функцию потерь  
loss = loss_func(y_pred, batch_dict['y_nationality'])  
loss_batch = loss.to("cpu").item()  
running_loss += (loss_batch - running_loss) / (batch_index + 1)  
  
# Шаг 4. Получаем градиенты на основе функции потерь  
loss.backward()  
  
# Шаг 5. Оптимизатор обновляет значения параметров по градиентам  
optimizer.step()
```

## Оценка модели и получение предсказаний

Чтобы выяснить эффективность модели, необходимо проанализировать модель с помощью количественных и качественных методов. Количественные методы включают оценку погрешности на специально выделенных контрольных данных для определения того, способен ли классификатор выполнить обобщение на еще не виденные выборки. Качественные методы включают ваше личное представление о том, чему обучилась модель, на основе первых  $k$  предсказаний классификатора для новой выборки.

### Оценка на контрольном наборе данных

Для оценки `SurnameClassifier` на контрольных данных мы поступим так же, как и в примере с классификацией текстов обзоров ресторанов в подразделе «Оценка, вывод и просмотр» на с. 95: выберем фрагмент данных `'test'`, вызовем метод `classifier.eval()` и пройдем в цикле по контрольным данным аналогично прочим фрагментам. В этом примере фреймворк PyTorch из-за вызова метода `classifier.eval()` не обновляет параметры модели при использовании контрольных/проверочных данных.

Точность модели на контрольных данных составляет около 50 %. Если запустить процедуру обучения из прилагаемого к книге блокнота, вы увидите, что эффективность на обучающих данных выше. Дело в том, что модель всегда лучше приспособлена к данным, на которых обучается, так что эффективность на обучающих данных не показательна для обучения на новых данных. Если вы проверяете работу кода примеров, рекомендуем попробовать различные размеры скрытого измерения. При этом вы должны наблюдать рост эффективности<sup>1</sup>. Впрочем, этот рост не очень велик (особенно по сравнению с моделью из раздела «Пример: классификация фамилий с помощью CNN» на с. 130). Основная причина: неудачность свернутой унитарной векторизации как метода представления. Хотя каждая из фамилий компактно представляется в виде одного вектора, при этом теряется информация об упорядоченности символов, играющая большое значение при идентификации происхождения фамилии.

## Классификация новой фамилии

Пример 4.11 демонстрирует код для классификации новой фамилии. Функция сначала запускает процесс векторизации по отношению к полученной в виде строкового значения фамилии, а затем получает предсказания модели. Обратите внимание на включенный флаг `apply_softmax` — это означает, что переменная `result` содержит вероятности. Предсказание модели в полиномиальном случае представляет собой список вероятностей различных классов. Для получения оптимального класса, соответствующего наибольшей предсказанной вероятности, воспользуемся функцией `max()` тензора PyTorch.

**Пример 4.11.** Вывод с помощью существующей модели (классификатора). Предсказание национальности по фамилии

```
def predict_nationality(name, classifier, vectorizer):
    vectorized_name = vectorizer.vectorize(name)
    vectorized_name = torch.tensor(vectorized_name).view(1, -1)
    result = classifier(vectorized_name, apply_softmax=True)

    probability_values, indices = result.max(dim=1)
    index = indices.item()

    predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
    probability_value = probability_values.item()

    return {'nationality': predicted_nationality,
            'probability': probability_value}
```

<sup>1</sup> Напомним уже сказанное в главе 3: при экспериментах с гиперпараметрами, такими как размер скрытого измерения и количество слоев, оценка должна производиться на проверочном наборе данных, а не на контрольном. Если же текущие гиперпараметры вас устраивают, можете выполнить оценку на контрольных данных.

## Извлечение $k$ наилучших предсказаний для новой фамилии

Часто бывает полезно взглянуть не только на одно лучшее предсказание. Например, в NLP стандартной практикой считается извлечение  $k$  наилучших предсказаний и повторное ранжирование их с помощью другой модели. Для получения этих предсказаний во фреймворке PyTorch есть удобная функция `torch.topk()` (пример 4.12).

**Пример 4.12.** Предсказание  $k$  лучше всего подходящих национальностей

```
def predict_topk_nationality(name, classifier, vectorizer, k=5):
    vectorized_name = vectorizer.vectorize(name)
    vectorized_name = torch.tensor(vectorized_name).view(1, -1)
    prediction_vector = classifier(vectorized_name, apply_softmax=True)
    probability_values, indices = torch.topk(prediction_vector, k=k)

    # Возвращаемый размер 1,k
    probability_values = probability_values.detach().numpy()[0]
    indices = indices.detach().numpy()[0]

    results = []
    for prob_value, index in zip(probability_values, indices):
        nationality = vectorizer.nationality_vocab.lookup_index(index)
        results.append({'nationality': nationality,
                       'probability': prob_value})

    return results
```

## Регуляризация многослойных перцептронов: регуляризация весов и структурная регуляризация

В главе 3 мы объяснили, почему регуляризация является решением проблемы переобучения, и изучили два важных типа регуляризации весов — L1 и L2. Эти методы регуляризации весов применимы как к MLP, так и к сверточным нейронным сетям, которые мы рассмотрим в следующем разделе. Помимо регуляризации весов, для глубоких моделей (то есть моделей из нескольких слоев), таких как обсуждавшиеся в этой главе упреждающие сети, важнейшее значение имеет подход структурной регуляризации под названием *дропаут* (dropout).

Попросту говоря, во время обучения дропаут отбрасывает на вероятностной основе связи между элементами из двух смежных слоев. В чем польза такого подхода? Начнем с интуитивно понятного (юмористического) объяснения Стивена Мерити (Stephen Merity)<sup>1</sup>: «Дропаут, попросту говоря, означает, что если у вас хорошо

---

<sup>1</sup> Это определение взято из весьма забавной первоапрельской «статьи» Стивена Мерити (<http://bit.ly/2Cq1FJR>).

*получается выполнять какую-либо задачу, будучи мертвецки пьяным, то в трезвом состоянии вы тем более сможете ее выполнить. Этому наблюдению обязаны собой множество достижений современной науки, так что зарождается целое движение против использования дропаута в нейронных сетях».*

Нейронные сети — особенно глубокие сети с большим количеством слоев — способны создавать интересные коадаптации элементов. «Коадаптация» — термин из нейронаук, означающий просто ситуацию, при которой связь между двумя элементами становится чрезвычайно сильной за счет ослабления связей между другими элементами. Обычно это приводит к переобучению модели на текущих данных. Вероятностное отбрасывание связей между элементами позволяет гарантировать, что никакой конкретный элемент не окажется в постоянной зависимости от другого конкретного элемента, благодаря чему модели становятся ошибкоустойчивыми. Дропаут не требует добавления в модель дополнительных параметров, а лишь одного гиперпараметра — «вероятности отброса»<sup>1</sup>. Это, как легко догадаться, вероятность, при которой отбрасываются связи между элементами. Обычно ее задают равной 0.5. Пример 4.13 демонстрирует реализацию MLP с дропаутом.

**Пример 4.13.** MLP с дропаутом

```
import torch.nn as nn
import torch.nn.functional as F

class MultilayerPerceptron(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        Аргументы:
            input_dim (int): размер входных векторов
            hidden_dim (int): размер на выходе первого линейного слоя
            output_dim (int): размер на выходе второго линейного слоя
        """
        super(MultilayerPerceptron, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """ Прямой проход MLP

        Аргументы:
            x_in (torch.Tensor): входной тензор данных
                Значение x_in.shape должно быть (batch, input_dim)
            apply_softmax (bool): флаг для многомерной логистической функции
                активации. При использовании функции потерь на основе
                перекрестной энтропии должен равняться false
        Возвращает:
```

<sup>1</sup> Некоторые библиотеки глубокого обучения по непонятной причине называют (и интерпретируют) эту вероятность «вероятностью сохранения», то есть придают ей в точности противоположный смысл.

```

        Итоговый тензор. Значение tensor.shape должно
        быть (batch, output_dim).
    """
    intermediate = F.relu(self.fc1(x_in))
    output = self.fc2(F.dropout(intermediate, p=0.5))

    if apply_softmax:
        output = F.softmax(output, dim=1)
    return output

```

Важно отметить, что *дропаут применяется только во время обучения, а не оценки*. В качестве упражнения рекомендуем вам поэкспериментировать с моделью `SurnameClassifier` с дропаутом и посмотреть, как изменятся результаты.

## Сверточные нейронные сети

В первой части главы мы подробно изучили MLP — нейронные сети, построенные из ряда линейных слоев и нелинейных функций. MLP — не лучший инструмент для того, чтобы использовать возможности последовательных паттернов<sup>1</sup>. Например, в наборе данных фамилий определенные участки могут в значительной степени раскрывать национальность их носителей (как O' в O'Neil, opulos в Antonopoulos, sawa в Nagasawa или Zh в Zhu). Длины этих участков могут быть различными, сложность состоит в том, чтобы уловить их без явного кодирования.

Рассмотрим сверточные нейронные сети — тип нейронных сетей, хорошо подходящий для обнаружения пространственной субструктуры (а значит, и создания осмысленной пространственной субструктуры). CNN добиваются этого за счет небольшого количества весов, используемых для просмотра входящих тензоров данных. В результате этого просмотра генерируются выходные тензоры, отражающие обнаружение (или не обнаружение) субструктур.

Мы начнем с описания способов функционирования CNN и вопросов их проектирования. Подробно обсудим гиперпараметры CNN, чтобы вы научились интуитивно чувствовать их поведение и влияние на выходные результаты. И наконец, мы шаг за шагом пройдем по нескольким простым примерам, иллюстрирующим механизм работы CNN. В разделе «Пример: классификация фамилий с помощью CNN» на с. 130 мы рассмотрим более масштабный пример.

<sup>1</sup> Можно спроектировать такой MLP, который бы принимал на входе символьные биграммы, чтобы уловить некоторые из подобных зависимостей. Количество биграмм для английского языка (алфавит которого состоит из 26 букв) составляет 325. Так что при скрытом слое из 100 узлов число параметров для входного скрытого слоя будет  $325 \times 100$ . А если рассматривать все возможные символьные триграммы, то получим еще  $2600 \times 100$  параметров. Как мы увидим, сверточные нейронные сети позволяют уловить ту же информацию при намного меньшем числе параметров благодаря совместному использованию параметров.

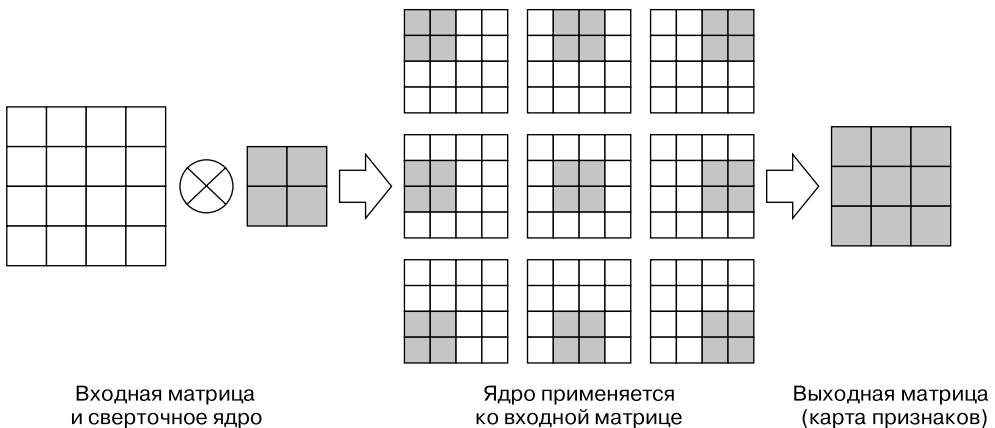


### ИСТОРИЧЕСКИЙ КОНТЕКСТ

Название и основная функциональность CNN ведут свое начало от классической математической операции *свертки* (convolution). Свертка уже многие десятилетия применяется в различных инженерных дисциплинах, включая цифровую обработку сигналов и компьютерную графику. Обычно в свертке использовались задаваемые программистом параметры, соответствующие какому-либо функциональному замыслу, например подсвечиванию краев или глушению высокочастотных звуков. На самом деле многие фильтры Photoshop представляют собой применение к изображениям фиксированных операций свертки. Однако в глубоком обучении и этой главе параметры сверточных фильтров определяются путем обучения на данных, поскольку для решения имеющейся задачи это оптимальный вариант.

## Гиперпараметры CNN

Рассмотрим рис. 4.6, чтобы лучше разобраться во влиянии различных проектных решений на CNN. Здесь к входной матрице применяется одно «ядро». Точное математическое выражение для операции свертки (линейный оператор) не важно для понимания этого раздела; важно уяснить себе из рисунка, что ядро — маленькая квадратная матрица, согласованным образом применяемая к различным позициям входной матрицы.



**Рис. 4.6.** Двумерная операция свертки. Входная матрица свертывается с одним сверточным ядром, в результате чего получается выходная матрица (называемая также картой признаков). Свертка представляет собой применение ядра к каждой из позиций входной матрицы. При каждом таком применении значения входной матрицы умножаются на значения ядра, после чего полученные результаты суммируются. В этом примере значения гиперпараметров ядра следующие:  $\text{kernel\_size}=2$ ,  $\text{stride}=1$ ,  $\text{padding}=0$  и  $\text{dilation}=1$ . Смысл этих гиперпараметров объясняется далее

Классические варианты свертки<sup>1</sup> ориентированы на указание значений ядра<sup>2</sup>, CNN ориентированы на задание значений гиперпараметров, определяющих поведение CNN, с последующим поиском оптимальных параметров для заданного набора данных методом градиентного спуска. Два основных гиперпараметра определяют форму свертки (`kernel_size`) и позиции (шаг свертки. — *Примеч. пер.*) во входном тензоре данных, которые свертка будет перемножать (`stride`). Есть и дополнительные гиперпараметры для управления степенью дополнения входного тензора данных нулями (`padding`) и дистанция между умножаемыми ячейками при применении к входному тензору данных (`dilation`). В следующих подразделах мы расскажем, как лучше чувствовать эти гиперпараметры.

## Размерность операции свертки

Первое понятие, с которым нам нужно разобраться, — *размерность* (dimensionality) операции свертки. На рис. 4.6 и остальных рисунках этого подраздела используется двумерная свертка, но бывают свертки и других размерностей, более подходящие к данным определенных видов. Во фреймворке PyTorch свертки могут быть одномерными, двумерными и трехмерными (реализуются модулями `Conv1d`, `Conv2d` и `Conv3d` соответственно). Одномерная свертка удобна для работы с временными рядами, в которых каждому шагу по времени соответствует вектор признаков. В подобной ситуации можно обучаться паттернам в пространственном измерении последовательности. Большинство операций свертки в NLP одномерные. Двумерная свертка, с другой стороны, предназначена для улавливания пространственно-временных паттернов по двум направлениям в данных, например в изображениях — по осям высоты и ширины, вследствие чего двумерные свертки так часто применяются для обработки изображений. Аналогично в трехмерных свертках улавливаются пространственно-временные паттерны по трем направлениям в данных. Например, в видеоданных информация является трехмерной (два измерения отражают кадр изображения, а измерение времени отражает последовательность кадров). В этой книге мы в основном будем использовать `Conv1d`.

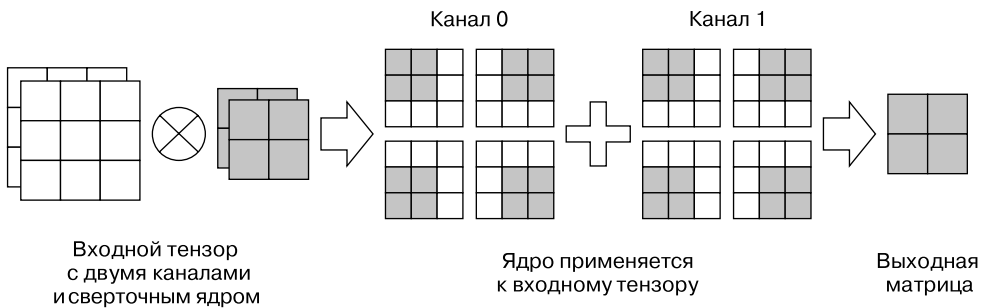
## Каналы

Неофициально *канал* (channel) — одно из измерений пространства признаков по точкам входных данных. Например, в изображениях каждому пикселу соответствует три канала (по одному для каждого из компонентов RGB). Это понятие в контексте свертки можно перенести и на текстовые данные. Фактически, если

<sup>1</sup> См. дополнительную информацию о классических вариантах сверток во врезке «Исторический контекст» на с. 121.

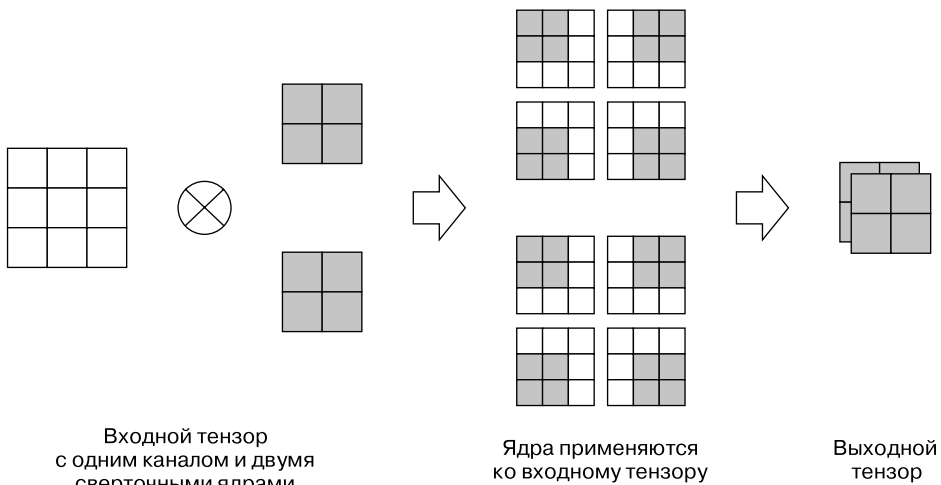
<sup>2</sup> Подобным образом работает множество фильтров в программах редактирования изображений наподобие Photoshop. Например, для подсветки границ можно воспользоваться специально разработанным для этой цели сверточным фильтром.

рассматривать слова в текстовом документе как «пиксели», то число каналов соответствует размеру словаря. Если уменьшить уровень разбиения и рассмотреть свертку по символам, то число каналов будет равно размеру набора символов (в нашем случае словаря). В реализации сверток фреймворка PyTorch число каналов на входе задается аргументом `in_channels`. На выходе операции свертки может быть несколько каналов (аргумент `out_channels`). Ее можно рассматривать как оператор свертки, «отображающий» входное пространство признаков на выходное пространство признаков. Эта идея проиллюстрирована на рис. 4.7 и 4.8.



**Рис. 4.7.** Операция свертки с двумя входными матрицами (два входных канала).

Соответствующее ядро также состоит из двух слоев; оно отдельно перемножает каждый из слоев, после чего суммирует результаты. Настройки: `input_channels=2`, `output_channels=1`, `kernel_size=2`, `stride=1`, `padding=0` и `dilation=1`

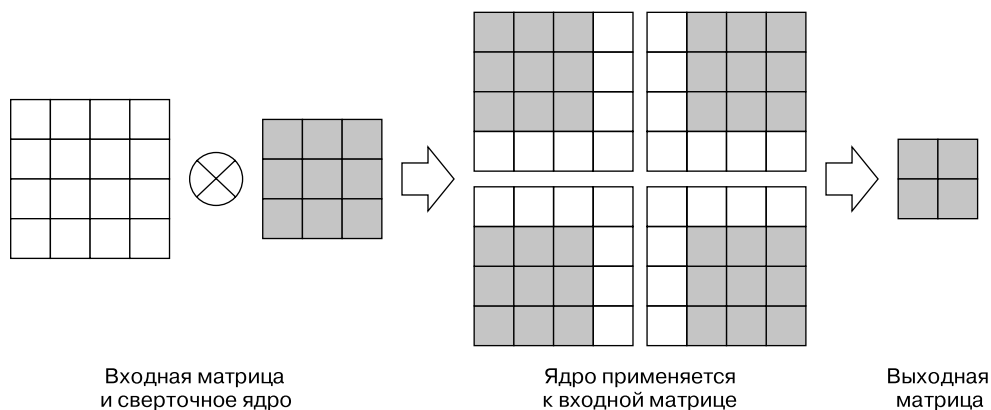


**Рис. 4.8.** Операция свертки с одной входной матрицей (один входной канал) и двумя сверточными ядрами (два выходных канала). Ядра применяются по отдельности к входной матрице и располагаются ярусами в выходном тензоре. Настройки: `input_channels=1`, `output_channels=2`, `kernel_size=2`, `stride=1`, `padding=0` и `dilation=1`

Сложно сразу понять, сколько каналов нужно для решения имеющейся задачи. Для упрощения предположим, что их количество ограничено одним снизу и 1024 сверху. Определившись с этими ограничениями, можно приступить к выяснению количества входных каналов. Часто используется паттерн проектирования, согласно которому от одного до другого сверточного слоя число каналов не может меняться более чем в два раза. Это правило нельзя считать непреложным, но с его помощью можно составить представление о том, каким должно быть значение `out_channels`.

## Размер ядра

Ширина матрицы ядра называется *размером ядра* (параметр `kernel_size` в PyTorch). На рис. 4.6 размер ядра был равен 2, а на рис. 4.9 мы покажем ядро размером 3. Вы должны уяснить себе, что свертка приводит к пространственному (или временному) объединению локальной информации из входных данных, причем количество локальной информации в каждой из операций свертки определяется размером ядра. Однако при увеличении размера ядра размер выходных данных уменьшается (см. книгу Дюмулена и Визина [Dumoulin, Visin, 2016]). Именно поэтому размер выходной матрицы на рис. 4.9 составляет  $2 \times 2$ , где размер ядра равен 3, а на рис. 4.6 —  $3 \times 3$ , где размер ядра равен 2.



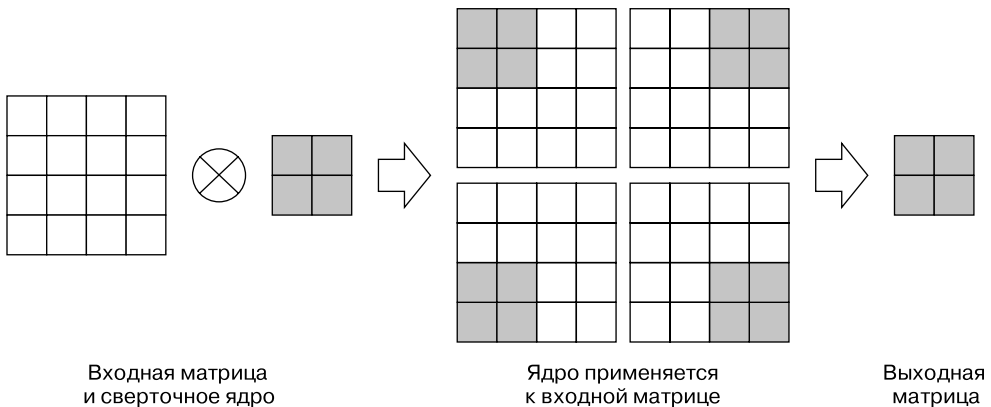
**Рис. 4.9.** К входной матрице применяется операция свертки с `kernel_size=3`. В результате достигается компромисс: при каждом применении ядра к матрице используется больше локальной информации, но размер выходной матрицы меньше

Кроме того, поведение размера ядра в NLP-приложениях в каком-то смысле аналогично поведению  $n$ -грамм, захватывающих паттерны языка путем анализа групп слов. Чем меньше размеры ядер, тем меньшие и более часто встречающиеся паттерны захватываются, в то время как большие размеры ядер приводят к захва-

ту более крупных паттернов — возможно, более осмысленных, но и более редких. Меньшие размеры ядер приводят к более «мелкозернистым» признакам на выходе, а бóльшие — к более «грубозернистым» признакам.

## Шаг свертки

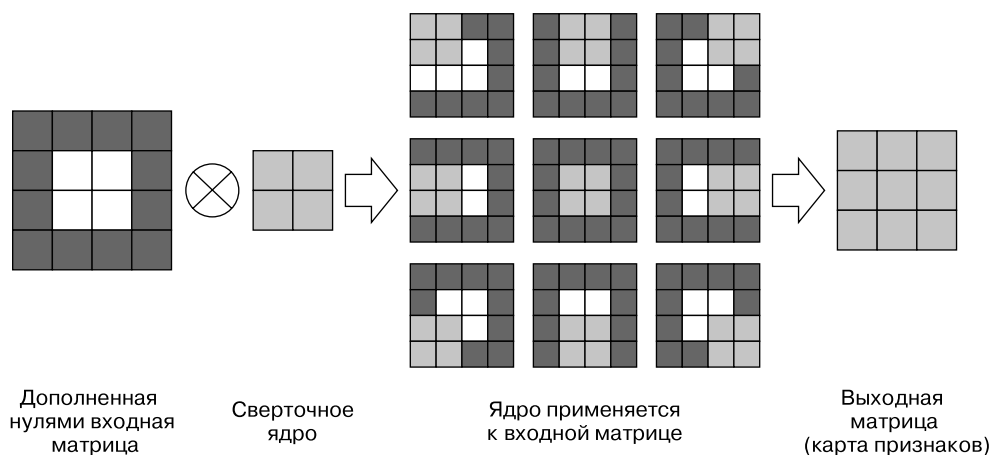
*Шаг свертки* (stride) определяет размер шага между свертками. Если размер шага свертки совпадает с размером ядра, то вычисления ядра не перекрываются. Если же он равен 1, то перекрытие максимально. Путем увеличения шага свертки можно нарочно сжать выходной тензор для обобщения информации, как показано на рис. 4.10.



**Рис. 4.10.** К входной матрице применяется операция свертки с `kernel_size=2` при гиперпараметре `stride=2`. В результате шаги ядра увеличиваются, а размер выходной матрицы уменьшается. Это удобно для прореживания входной матрицы

## Дополнение нулями

Хотя параметры `kernel_size` и `stride` позволяют управлять областью, занимаемой каждым из вычисленных значений признаков, у них есть неприятный и обычно непреднамеренный побочный эффект, заключающийся в сокращении общего размера карты признаков (выходного результата свертки). Для нейтрализации этого эффекта искусственно увеличивают длину входного тензора данных (если он одно-, дву- или трехмерный), высоту (если он дву- или трехмерный) и глубину (если он трехмерный), вставляя в начало каждого из соответствующих измерений нули. Из этого следует *увеличение* числа выполняемых CNN сверток, но и появляется возможность контролировать форму выходных данных без компромиссов относительно размера ядра, шага свертки или расширения. Дополнение нулями в действии показано на рис. 4.11.



**Рис. 4.11.** К входной матрице высотой и шириной 2 применяется операция свертки с `kernel_size=2`. Однако высоту и ширину входной матрицы можно увеличить за счет дополнения нулями (темно-серые ячейки). Чаще всего это выполняется для ядер размером 3, чтобы размеры выходной и входной матриц были одинаковы

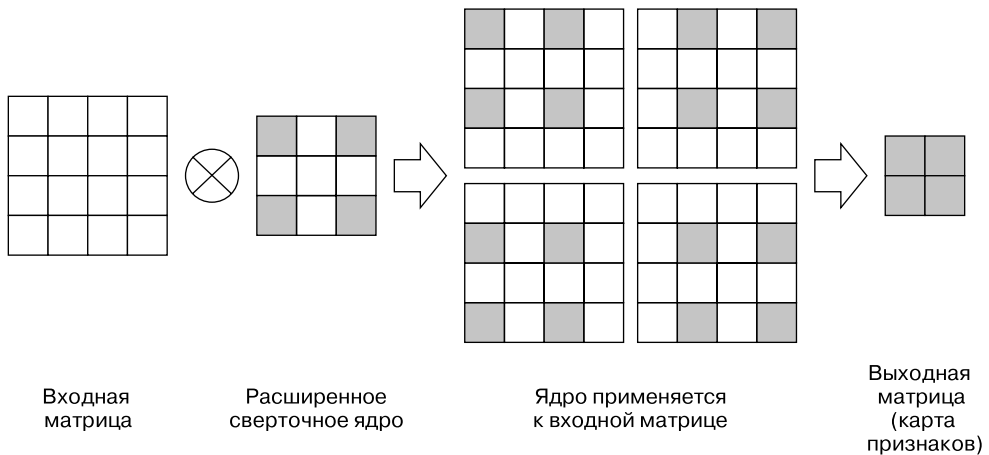
## Расширение

*Расширение* (dilation) определяет способ применения сверточного ядра к входной матрице. На рис. 4.12 показано, что увеличение степени расширения с 1 (значение по умолчанию) до 2 означает, что элементы ядра при применении к входной матрице находятся через один друг от друга. Можно также рассматривать это как аналог шага свертки в самом ядре — определенный размер шага между элементами ядра либо применение ядра с «отверстиями». Это может пригодиться для агрегирования больших областей входного пространства без увеличения количества параметров. Расширяемые свертки оказались очень удобны при многоярусном расположении сверточных слоев. Последовательное применение расширяемых сверток экспоненциально повышает размер «принимающего поля», то есть размер входного пространства, видимого сети до предсказания.

## Реализация сверточных нейронных сетей в PyTorch

В этом разделе мы рассмотрим комплексный пример, в котором используются введенные в предыдущем разделе понятия. Обычно целью проектирования нейронной сети является нахождение подходящего набора гиперпараметров. Мы вновь обратимся к уже знакомой нам из раздела «Пример: классификация фамилий с помощью MLP» на с. 109 задаче классификации фамилий, но теперь воспользуемся

сверточными нейронными сетями, а не MLP. Нам по-прежнему придется применить завершающий линейный слой, который сможет создать вектор предсказаний на основе созданного последовательностью сверточных слоев вектора признаков. Это означает, что наша цель — определить конфигурацию сверточных слоев, которая приводит к желаемому вектору признаков. Все приложения CNN строятся подобным образом: начальный набор сверточных слоев извлекает карту признаков, которая затем служит входными данными для какой-либо последующей обработки. При классификации эта последующая обработка почти всегда представляет собой применение линейного (fc) слоя.



**Рис. 4.12.** К входной матрице применяется операция свертки с  $\text{kernel\_size}=2$  при гиперпараметре  $\text{dilation}=2$ . Увеличение гиперпараметра  $\text{dilation}$  по сравнению со значением по умолчанию означает больший разброс элементов матрицы ядра при перемножении с входной матрицей. Дальнейшее увеличение  $\text{dilation}$  еще более усиливает этот разброс

Наше обсуждение реализации в этом разделе концентрируется на проектных решениях, необходимых для создания вектора признаков<sup>1</sup>. Мы начнем с формирования искусственного тензора данных, по форме имитирующего настоящие данные. Этот тензор будет трехмерным — таков размер мини-пакета векторизованных текстовых данных. Поскольку для каждого из символов последовательности

<sup>1</sup> Многие книги по глубокому обучению игнорируют этот нюанс и демонстрируют читателю итоговую нейронную сеть, как будто она была ниспослана свыше. Мы рассчитываем в этом разделе провести незнакомого с данным вопросом читателя по всему процессу получения конкретной сверточной нейронной сети, начиная с входных данных. В большинстве ситуаций, особенно при реализации научной статьи, это не требуется. Можно просто воспользоваться приведенными значениями.

используется унитарный вектор, то последовательность унитарных векторов формирует матрицу, а мини-пакет унитарных матриц — трехмерный тензор. На языке сверток размер каждого из унитарных векторов (обычно равный размеру словаря) представляет собой число «входных каналов», а длина последовательности символов — «ширину».

Как показано в примере 4.14, первый шаг конструирования вектора признаков — применение экземпляра класса `Conv1d` фреймворка PyTorch к трехмерному тензору данных. По информации о размере выходных данных можно составить представление о том, насколько сильно был уменьшен тензор. Наглядная иллюстрация причин сжатия выходного тензора приведена на рис. 4.9.

**Пример 4.14.** Искусственные данные и применение класса `Conv1d`<sup>1</sup>

Input[0]

```
batch_size = 2
one_hot_size = 10
sequence_width = 7
data = torch.randn(batch_size, one_hot_size, sequence_width)
conv1 = Conv1d(in_channels=one_hot_size, out_channels=16,
               kernel_size=3)
intermediate1 = conv1(data)
print(data.size())
print(intermediate1.size())
```

Output[0]

```
torch.Size([2, 10, 7])
torch.Size([2, 16, 5])
```

Существует три основных метода для дальнейшего сжатия выходного тензора. Первый состоит в создании дополнительных операций свертки и последовательного их применения. В конце концов у соответствующего `sequence_width (dim=2)` измерения получится `size=1`. Мы покажем результаты применения двух дополнительных сверток в примере 4.15. Вообще, процесс применения сверток для уменьшения размера выходного тензора носит итеративный характер и требует определенных допущений. Наш пример устроен таким образом, что в последнем измерении полученного после трех сверток результата получится `size=1`<sup>2</sup>.

<sup>1</sup> Для работы этого примера следует указать пакет для `Conv1d`. Например, вот так:

```
import torch.nn as nn
```

...

```
conv1 = nn.Conv1d(... — Примеч. пер.
```

<sup>2</sup> Для тензоров большего размера требуется больше сверток. Кроме того, необходимо также модифицировать сами свертки для более быстрого сжатия тензора. Для этого можно попробовать увеличить значение таких гиперпараметров, как `stride`, `dilation` и `kernel_size`.



**Пример 4.15.** Многократное применение сверток к данным

Input[0]

```
conv2 = nn.Conv1d(in_channels=16, out_channels=32, kernel_size=3)
conv3 = nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3)

intermediate2 = conv2(intermediate1)
intermediate3 = conv3(intermediate2)

print(intermediate2.size())
print(intermediate3.size())
```

Output[0]

```
torch.Size([2, 32, 3])
torch.Size([2, 64, 1])
```

Input[1]

```
y_output = intermediate3.squeeze()
print(y_output.size())
```

Output[1]

```
torch.Size([2, 64])
```

С каждой сверткой размер измерения канала растет, поскольку измерение канала служит вектором признаков для каждой точки данных. Чтобы тензор действительно стал вектором признаков, осталось удалить мешающее нам измерение с `size=1`. Для этого можно воспользоваться методом `squeeze()`. При этом отбрасываются все измерения с `size=1` и возвращается результат. Полученные векторы признаков можно далее использовать совместно с другими компонентами нейронных сетей, например линейными слоями, для вычисления векторов предсказаний.

Существует еще два метода сжатия тензора до одного вектора признаков на точку данных: свертывание оставшихся значений в вектор признаков и усреднение<sup>1</sup> по лишним измерениям. Эти два метода показаны в примере 4.16. При использовании первого все векторы просто свертываются в один с помощью метода `view()` фреймворка PyTorch<sup>2</sup>. Второй метод основан на использовании какой-либо математической операции для агрегирования содержащейся в векторах информации. Чаще всего с этой целью применяется арифметическое среднее, но суммирование и вычисление максимального значения по измерениям карты признаков также нередко встречаются. У каждого из этих подходов есть свои достоинства и недостатки. При свертывании сохраняется вся информация, но векторы признаков могут оказаться больше, чем хотелось бы (или чем допустимо с вычислительной

<sup>1</sup> На самом деле можно также вычислить их сумму или воспользоваться механизмом на основе внимания, как описано в главе 8. Для простоты мы обсудим только эти два метода.

<sup>2</sup> Подробности относительно метода `view()` вы можете найти в главе 1.

точки зрения). Усреднение не зависит от размера дополнительных измерений, но может приводить к потере информации<sup>1</sup>.

**Пример 4.16.** Два дополнительных метода свертки к векторам признаков

Input[0]

```
# Метод 2 свертки к векторам признаков
print(intermediate1.view(batch_size, -1).size())

# Метод 3 свертки к векторам признаков
print(torch.mean(intermediate1, dim=2).size())
# print(torch.max(intermediate1, dim=2).size())
# print(torch.sum(intermediate1, dim=2).size())
```

Output[0]

```
torch.Size([2, 80])
torch.Size([2, 16])
```

Этот метод проектирования рядов сверток носит эмпирический характер: выбирается желаемый размер данных и после экспериментов с последовательностью сверток в конце концов получается подходящий вектор признаков. Хотя на практике это отлично работает, существует и другой метод вычисления выходного размера тензора по гиперпараметрам свертки и входному тензору: с помощью математической формулы, выведенной на основе самой операции свертки.

## Пример: классификация фамилий с помощью CNN

С целью демонстрации эффективности сверточных нейронных сетей воспользуемся простой моделью CNN для задачи классификации фамилий<sup>2</sup>. Многие детали остались такими же, как и в предыдущем примере решения этой задачи с помощью MLP, но поменялись конструкция модели и процесс векторизации. Входные данные модели теперь представляют собой не свернутый унитарный

<sup>1</sup> Кроме того, нужно аккуратно хранить всю информацию, необходимую для последовательностей переменной длины, о которых мы поговорим в главе 6. Если коротко, для учета последовательностей переменной длины некоторые позиции должны содержать только нули. В подобной ситуации при усреднении будет выполняться суммирование по дополнительным измерениям и деление результата на число ненулевых элементов.

<sup>2</sup> Блокнот для этого примера находится в репозитории GitHub (<https://nlpproc.info/PyTorchNLPBook/repo/>) в папке /chapters/chapter\_4/4\_4\_cnn\_surnames/4\_4\_Classifying\_Surnames\_with\_a\_CNN.ipynb.

вектор, как в предыдущем примере, а матрицу унитарных векторов. Благодаря подобной архитектуре CNN сможет лучше «разглядеть» расположение символов и закодировать информацию последовательности, утерянную в свернутом унитарном представлении из раздела «Пример: классификация фамилий с помощью MLP» на с. 109.

## Класс SurnameDataset

Набор данных фамилий уже описывался в подразделе «Набор данных фамилий» на с. 110. Мы воспользуемся здесь тем же набором данных, но с одним отличием в реализации: набор данных состоит из матрицы унитарных векторов, а не свернутого унитарного вектора. Для этого реализуем класс для набора данных, который будет отслеживать самую длинную фамилию и передавать ее векторизатору в качестве количества включаемых в матрицу строк. Количество столбцов равно размеру унитарных векторов (размеру словаря). Пример 4.17 демонстрирует изменения в методе `SurnameDataset.__getitem__()`, а в следующем подразделе мы покажем изменения в методе `SurnameVectorizer.vectorize()`.

**Пример 4.17.** Класс `SurnameDataset`, модифицированный для передачи максимальной длины фамилии

```
class SurnameDataset(Dataset):
    # ... существующая реализация из раздела
    # "Пример: классификация фамилий с помощью MLP" на с. 109

    def __getitem__(self, index):
        row = self._target_df.iloc[index]

        surname_matrix = \
            self._vectorizer.vectorize(row.surname, self._max_seq_length)

        nationality_index = \
            self._vectorizer.nationality_vocab.lookup_token(row.nationality)

        return {'x_surname': surname_matrix,
                'y_nationality': nationality_index}
```

Есть две причины, по которым для управления размером унитарной матрицы используется самая длинная фамилия из набора данных. Во-первых, каждый из мини-пакетов матриц фамилий объединяется в трехмерный тензор и все они должны быть одинакового размера. Во-вторых, при использовании самой длинной фамилии из набора данных все мини-пакеты обрабатываются одинаково<sup>1</sup>.

<sup>1</sup> Можно было вместо этого использовать максимальную длину фамилии в каждом мини-пакете, но единое значение для всего набора данных упрощает задачу.

## Классы Vocabulary, Vectorizer и DataLoader

В этом примере, хотя классы `Vocabulary` и `DataLoader` реализованы совершенно аналогично примеру из подраздела «Классы `Vocabulary`, `Vectorizer` и `DataLoader`» на с. 82, мы изменили метод `vectorize()` класса `Vectorizer` соответственно потребностям CNN модели. А именно, как мы покажем в коде примера 4.18, эта функция задает соответствие каждого из символов в строке целочисленному значению, после чего использует это целое число для формирования матрицы унитарных векторов. Важно отметить, что все столбцы в матрице представляют собой различные унитарные векторы. Основная причина этого: используемые нами слои `Conv1d` требуют, чтобы в тензорах данных в нулевом измерении содержался пакет, в первом — каналы, а во втором — признаки.

Помимо изменений, внесенных для использования унитарной матрицы, мы модифицируем класс `Vectorizer` так, чтобы он вычислял и сохранял максимальную длину фамилии в виде переменной `max_surname_length`.

### Пример 4.18. Реализация класса `SurnameVectorizer` для CNN

```
class SurnameVectorizer(object):
    """ Векторизатор, приводящий словари в соответствие друг другу
        и использующий их """
    def vectorize(self, surname):
        """
        Аргументы:
            surname (str): фамилия
        Возвращает:
            one_hot_matrix (np.ndarray): матрица унитарных векторов
        """

        one_hot_matrix_size = (len(self.character_vocab), self.max_surname_length)
        one_hot_matrix = np.zeros(one_hot_matrix_size, dtype=np.float32)

        for position_index, character in enumerate(surname):
            character_index = self.character_vocab.lookup_token(character)
            one_hot_matrix[character_index][position_index] = 1

        return one_hot_matrix

    @classmethod
    def from_dataframe(cls, surname_df):
        """ Создает экземпляр векторизатора на основе объекта DataFrame
            набора данных

        Аргументы:
            surname_df (pandas.DataFrame): набор данных фамилий
        Возвращает:
            экземпляр SurnameVectorizer
        """
        character_vocab = Vocabulary(unk_token="@")
```

```

nationality_vocab = Vocabulary(add_unk=False)
max_surname_length = 0

for index, row in surname_df.iterrows():
    max_surname_length = max(max_surname_length, len(row.surname))
    for letter in row.surname:
        character_vocab.add_token(letter)
    nationality_vocab.add_token(row.nationality)

return cls(character_vocab, nationality_vocab, max_surname_length)

```

## Заново реализуем класс SurnameClassifier с помощью сверточных нейронных сетей

Используемая в этом примере модель построена с помощью методов, рассмотренных в разделе «Сверточные нейронные сети» на с. 120. По сути, созданные нами для тестирования сверточных слоев «искусственные» данные в точности соответствуют размеру тензоров данных в наборе данных фамилий при использовании класса `Vectorizer` из этого примера. В примере 4.19 можно видеть как сходство с показанной в том разделе последовательностью `Conv1d`, так и новые дополнения, требующие пояснений. А именно, модель напоминает предыдущую тем, что использует ряды одномерных сверток для постепенного вычисления большего числа признаков с получением в результате векторов из одного признака.

Новое в этом примере — использование модулей PyTorch `Sequential` и `ELU`. Модуль `Sequential` — удобная обертка, инкапсулирующая линейную последовательность операций. В данном случае она инкапсулирует применение последовательности `Conv1d`. `ELU` представляет собой нелинейность, аналогичную блоку `ReLU`, с которым мы познакомились в главе 3, но вместо обрезания отрицательных значений выполняется их потенцирование. `ELU` — нелинейность, использование которой между сверточными слоями выглядит весьма многообещающим (см. статью Клеверта и др. [Clevert et al., 2015]).

В этом примере количество каналов сверток определяется гиперпараметром `num_channels`. Можно выбрать другое количество каналов для каждой из операций свертки отдельно. Но это повлекло бы за собой необходимость оптимизации большего числа гиперпараметров. Мы обнаружили, что 256 — достаточно большое количество для модели, чтобы достичь удовлетворительной эффективности.

### Пример 4.19. Класс SurnameClassifier на основе CNN

```

import torch.nn as nn
import torch.nn.functional as F

class SurnameClassifier(nn.Module):
    def __init__(self, initial_num_channels, num_classes, num_channels):
        """

```

```
Аргументы:
    initial_num_channels (int): размер входного вектора признаков
    num_classes (int): размер выходного вектора предсказаний
    num_channels (int): единый размер канала для всей сети
"""

super(SurnameClassifier, self).__init__()

self.convnet = nn.Sequential(
    nn.Conv1d(in_channels=initial_num_channels,
              out_channels=num_channels, kernel_size=3),
    nn.ELU(),
    nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
              kernel_size=3, stride=2),
    nn.ELU(),
    nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
              kernel_size=3, stride=2),
    nn.ELU(),
    nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
              kernel_size=3),
    nn.ELU()
)
self.fc = nn.Linear(num_channels, num_classes)

def forward(self, x_surname, apply_softmax=False):
    """ Прямой проход классификатора

    Аргументы:
        x_surname (torch.Tensor): входной тензор данных
        Значение x_surname.shape должно
        быть (batch, initial_num_channels, max_surname_length)
        apply_softmax (bool): флаг для многомерной логистической функции
        активации. При использовании функции потерь на основе
        перекрестной энтропии должен равняться false
    Возвращает:
        Итоговый тензор. Значение tensor.shape должно
        быть (batch, num_classes).
    """
    features = self.convnet(x_surname).squeeze(dim=2)
    prediction_vector = self.fc(features)

    if apply_softmax:
        prediction_vector = F.softmax(prediction_vector, dim=1)

    return prediction_vector
```

## Процедура обучения

Процедура обучения состоит из следующих уже привычных для нас операций: создания экземпляров набора данных, модели, функции потерь, оптимизатора, прохода в цикле по обучающему фрагменту набора данных и обновления параметров модели, прохода в цикле по проверочному фрагменту набора данных и оценки

эффективности модели, а потом — повтора итераций определенное количество раз. Это уже третья реализация процедуры обучения в книге, так что вы уже должны были хорошо усвоить эту последовательность операций. Мы не станем подробно описывать процедуру обучения для этого примера, поскольку она в точности соответствует таковой из раздела «Пример: классификация фамилий с помощью MLP» на с. 109. Входные аргументы, впрочем, отличаются, как вы можете видеть в примере 4.20.

**Пример 4.20.** Входные аргументы классификатора фамилий на основе CNN

```
args = Namespace(
    # Информация о данных и путях
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch4/cnn",
    # Гиперпараметры модели
    hidden_dim=100,
    num_channels=256,
    # Гиперпараметры обучения
    seed=1337,
    learning_rate=0.001,
    batch_size=128,
    num_epochs=100,
    early_stopping_criteria=5,
    dropout_p=0.1,
    # Настройки времени выполнения не приводятся для экономии места
)
```

## Оценка эффективности модели и предсказание

Для оценки эффективности модели требуются качественные и количественные метрики эффективности. Ниже описаны основные компоненты этих метрик. Мы рекомендуем вам подробно изучить их, чтобы лучше разобраться с моделью и результатами ее обучения.

### Оценка на контрольном наборе данных

Код оценки качества модели не поменялся с предыдущего примера, как и процедура обучения. Вкратце резюмируем происходящее: мы вызываем метод `eval()` классификатора, чтобы предотвратить обратное распространение ошибки, и проходим в цикле по контрольному набору данных. Точность этой модели на контрольном наборе данных составляет около 56 %, в то время как точность MLP была равна примерно 50 %. Хотя эти конкретные показатели отнюдь не предел возможностей данных архитектур, улучшение показателей в результате применения относительно простой модели CNN выглядит достаточно убедительно, чтобы вам стоило попробовать сверточные нейронные сети на текстовых данных.

## Классификация/извлечение наилучших предсказаний для новой фамилии

В этом примере изменена только одна часть функции `predict_nationality()` (пример 4.21): вместо того чтобы менять форму только что созданного тензора данных, добавив измерение пакетов с помощью метода `view()`, воспользуемся функцией `unsqueeze()` для добавления измерения с `size=1` для пакета. То же изменение выполнено и в функции `predict_topk_nationality()`.

**Пример 4.21.** Использование обученной модели для предсказания

```
def predict_nationality(surname, classifier, vectorizer):
    """ Предсказание национальности для новой фамилии

    Аргументы:
        surname (str): фамилия для классификатора
        classifier (SurnameClassifier): экземпляр классификатора
        vectorizer (SurnameVectorizer): соответствующий векторизатор
    Возвращает:
        Словарь, содержащий наиболее вероятную национальность
        и соответствующую вероятность
    """
    vectorized_surname = vectorizer.vectorize(surname)
    vectorized_surname = torch.tensor(vectorized_surname).unsqueeze(0)
    result = classifier(vectorized_surname, apply_softmax=True)

    probability_values, indices = result.max(dim=1)
    index = indices.item()

    predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
    probability_value = probability_values.item()

    return {'nationality': predicted_nationality,
            'probability': probability_value}
```

## Прочие вопросы CNN

В завершение дискуссии в этом разделе мы вкратце поговорим о нескольких дополнительных темах, не самых важных для сверточных нейронных сетей, но играющих основную роль в их повседневном применении. В частности, опишем операции субдискретизации, пакетной нормализации, связей «сеть в сети» и остаточных связей.

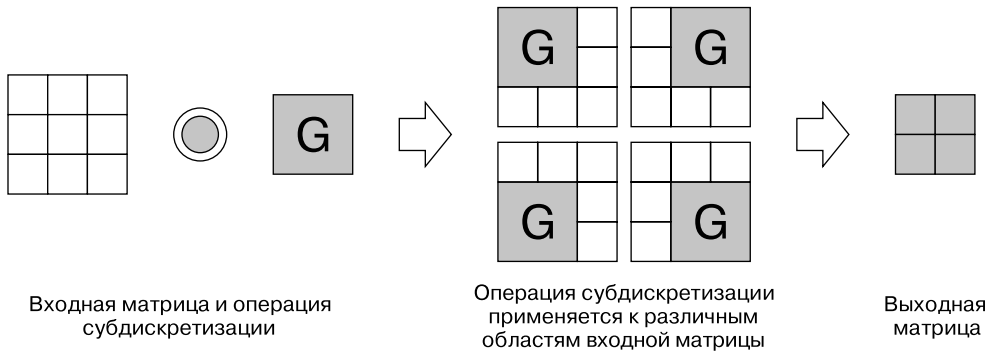
### Субдискретизация

Субдискретизация — операция, состоящая в агрегировании карты признаков более высокой размерности в карту признаков более низкой размерности. Результатом свертки становится карта признаков, значения в которой агрегируют какую-либо



область входных данных. Вследствие естественных при свертках перекрытий многие вычисленные признаки оказываются избыточными.

Субдискретизация — способ сведения карты признаков более высокой размерности (вероятно, избыточной) в карту признаков более низкой размерности. Формально субдискретизация представляет собой арифметический оператор вроде суммирования для вычисления среднего значения или максимума по локальной области карты признаков, а получающиеся в результате операции называются *субдискретизацией посредством суммирования* (sum pooling), *субдискретизацией посредством усреднения* (average pooling) и *субдискретизацией посредством вычисления максимума* (max pooling). Субдискретизацию можно также рассматривать как способ усиления статистической мощности путем преобразования более крупной, но и более слабой статистически карты признаков в более сильную. Субдискретизация показана на рис. 4.13.



**Рис. 4.13.** Операция субдискретизации функционально идентична свертке: она применяется к различным областям входной матрицы. Однако вместо умножения и суммирования значений входной матрицы операция субдискретизации применяет к входной матрице некую функцию  $G$ , которая выполняет дискретизацию значений.  $G$  может представлять собой любую операцию, но чаще всего встречаются суммирование, поиск максимума и вычисление среднего значения

## Пакетная нормализация (BatchNorm)

Пакетная нормализация (BatchNorm) — инструмент, часто применяемый при проектировании CNN. BatchNorm масштабирует функции активации так, чтобы у них были нулевое математическое ожидание и единичная дисперсия, применяя к выходным значениям CNN соответствующее преобразование. Используемые ею для Z-преобразования<sup>1</sup> значения математического ожидания и дисперсии обновляются

<sup>1</sup> Обоснование вы можете найти в статье «Википедии» по адресам <http://bit.ly/2V0x64z> и <https://ru.wikipedia.org/wiki/Z-оценка>.

попакетно так, что их отклонения в отдельных пакетах ни на что сильно не влияют. Благодаря BatchNorm модели оказываются менее чувствительными к начальным параметрам, а подстройка скорости обучения упрощается (см. статью Иоффе и Шегеди [Ioffe, Szegedy, 2015]). Во фреймворке PyTorch BatchNorm описан в модуле nn. Пример 4.22 демонстрирует создание экземпляра и использование BatchNorm со сверткой и линейными слоями.

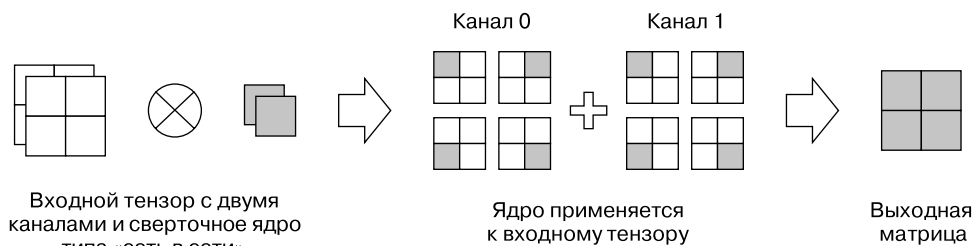
**Пример 4.22.** Использование слоя Conv1D с пакетной нормализацией

```
# ...
self.conv1 = nn.Conv1d(in_channels=1, out_channels=10,
                        kernel_size=5,
                        stride=1)
self.conv1_bn = nn.BatchNorm1d(num_features=10)
# ...

def forward(self, x):
    # ...
    x = F.relu(self.conv1(x))
    x = self.conv1_bn(x)
    # ...
```

## Связи типа «сеть в сети» (свертки $1 \times 1$ )

Связи типа «сеть в сети» (network-in-network, NiN) — это сверточные ядра с `kernel_size=1` и несколькими интересными свойствами. В частности, свертка  $1 \times 1$  ведет себя по отношению к каналам так же, как и полносвязный линейный слой<sup>1</sup>. Это удобно при отображении карт признаков с большим количеством каналов в менее «глубокие» карты признаков. На рис. 4.14 мы покажем применение отдельной связи типа NiN к входной матрице. Как вы можете видеть, два канала



**Рис. 4.14.** Пример операции свертки  $1 \times 1$  в действии. Обратите внимание, как операция свертки  $1 \times 1$  уменьшает число каналов с двух до одного

<sup>1</sup> Если вы вспомните предыдущие схемы, для каждого входящего канала есть по параметру, так что сверточное ядро с `kernel_size=1` представляет собой вектор, размер которого соответствует количеству входящих каналов.

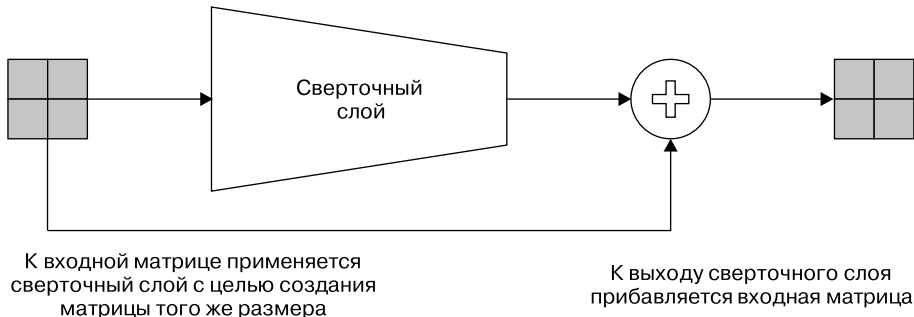
здесь свертываются в один. Таким образом, NiN (свертка  $1 \times 1$ ) представляет собой малозатратный способ подключения дополнительных нелинейностей с небольшим количеством параметров (см. статью Лина и др. [Lin et al., 2013]).

## Остаточные связи/остаточный блок

Одна из важнейших тенденций в сверточных нейронных сетях, делающая возможными по-настоящему глубокие сети (более 100 слоев), — остаточная связь, называемая также *связью с пропуском слоя* (skip-layer connection). Если обозначить функцию свертки как *conv*, то на выходе остаточного слоя получим<sup>1</sup>:

$$\text{выходные данные} = \text{conv}(\text{входные данные}) + \text{входные данные}.$$

Эта операция, впрочем, отличается неприметной уловкой, которую мы продемонстрируем на рис. 4.15. Для добавления входных данных к выходным данным свертки их формы должны совпадать. Для этого перед сверткой обычно выполняют дополнение нулями. На рис. 4.15 производится дополнение нулями матрицы размером 1 для свертки размером 3. Больше об остаточных связях вы можете узнать из статьи-первоисточника (Хи и др. [He et al., 2016]). Пример использования остаточных сетей в NLP можно найти в статье Хуанга и Ванга (Huang, Wang, 2017).



**Рис. 4.15.** Остаточная связь — метод прибавления исходной матрицы к выходному результату свертки. Выше это проиллюстрировано наглядно, на применении к входной матрице сверточного слоя и прибавлении к ней же результата. Для получения выходных данных того же размера, что и входные, часто задают следующие значения гиперпараметров: *kernel\_size*=3 и *padding*=1. Вообще говоря, любое нечетное значение гиперпараметра *kernel\_size* при *padding*=(*floor(kernel\_size)/2* – 1) приведет к выходным данным того же размера, что и входные. Наглядную иллюстрацию дополнения нулями при свертке можно найти на рис. 4.11. Получаемая от сверточного слоя матрица прибавляется к входным данным, а итоговый результат представляет собой результат вычислений остаточной связи (этот рисунок основан на рисунке 2 из статьи Хи и др. (He et al., 2016))

<sup>1</sup> В данном случае под входными данными имеются в виду входные данные остаточного блока, а не обязательно входные данные всей нейронной сети.

## Резюме

В главе мы познакомили вас с двумя основными упреждающими архитектурами: многослойным перцептроном (MLP; также называемым полносвязной сетью) и сверточной нейронной сетью (CNN). Мы наблюдали широкие возможности MLP по аппроксимации произвольных нелинейных функций и показали применение сверточных нейронных сетей в NLP на примере классификации фамилий по национальности. Мы исследовали один из основных недостатков/ограничений MLP — отсутствие разделения параметров — и продемонстрировали архитектуру сверточных нейронных сетей в качестве возможного решения этой проблемы. CNN, изначально предназначенные для машинного зрения, стали главным направлением NLP в основном благодаря высокоэффективным реализациям и низким требованиям к памяти. Мы изучили несколько вариантов сверток — с дополнением нулями, расширением и шагом свертки — и преобразование ими входного пространства. Немалая часть дискуссии в этой главе посвящена также практическому вопросу выбора входных и выходных размеров фильтров сверток. Мы показали, как операция свертки помогает захватывать информацию о субструктурах языка, для чего воспользовались в примере классификации фамилий сверточными нейронными сетями. Наконец, мы обсудили некоторые побочные, но важные вопросы, связанные с архитектурой сверточных нейронных сетей: 1) субдискретизацию; 2) пакетную нормализацию; 3) свертку  $1 \times 1$ ; 4) остаточные связи. В современной архитектуре CNN часто используется одновременно несколько подобных трюков. Например, в архитектуре Inception (см. статью Шегеди и др. [Szegedy et al., 2015]), где благодаря внимательному применению подобных уловок стало возможно создавать сверточные нейронные сети глубиной в сотни слоев — не только точные, но и быстро обучаемые. В главе 5 мы исследуем вопрос обучения и использования представлений для таких дискретных блоков, как слова, предложения, документы и другие типы признаков, с применением *вложений*.

## Библиография

1. Lin M., Chen Q., Yan S. Network in network. arXiv preprint arXiv: 1312.4400. 2013.
2. Szegedy C., Liu W., Jia Y., Sermanet P., Reed S., Anguelov D., Erhan D., Vanhoucke V., Rabinovich A. Going deeper with convolutions // CVPR. 2015.
3. Clevert D.-A., Unterthiner T., Hochreiter S. Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv: 1511.07289. 2015.
4. Ioffe S., Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv: 1502.03167. 2015.

5. *Dumoulin V., Visin F.* A guide to convolution arithmetic for deep learning. arXiv preprint arXiv: 1603.07285. 2016.
6. *He K., Zhang X., Ren S., Sun J.* Identity mappings in deep residual networks // ECCV. 2016.
7. *Huang Y. Y., Wang W. Y.* Deep Residual Learning for Weakly-Supervised Relation Extraction. arXiv preprint arXiv: 1707.08866. 2017.

# 5

## Вложение слов и прочих типов

При решении задач обработки текстов на естественных языках приходится иметь дело с различными видами дискретных типов данных. Самый очевидный пример — слова. Множество слов (словарь) конечно. В числе других примеров символы, метки частей речи, поименованные сущности, поименованные типы сущностей, признаки, связанные с синтаксическим разбором, позиции в каталоге товаров и т. д. Фактически к *дискретному типу* относится любой входной признак, взятый из конечного (или бесконечного, но счетного) множества.

В основе успешного применения глубокого обучения в NLP лежит представление дискретных типов данных (например, слов) в виде плотных (dense) векторов. Термины «обучение представлением» (representation learning) и «вложение» (embedding) означают обучение отображению/представлению из дискретного типа данных в точку векторного пространства. Если дискретные типы представляют собой слова, то плотное векторное представление называется *вложением слов* (word embedding). Мы уже встречали примеры методов вложения *на основе количества вхождений*, например TF-IDF («частотность термина — обратная частотность документа») в главе 2. В этой главе мы сосредоточим внимание на методах вложения *на основе обучения* и методах вложения *на основе предсказания* (см. статью Барони и др. [Baroni et al., 2014]), в которых обучение представлением производится путем максимизации целевой функции для конкретной задачи обучения; например, предсказания слова по контексту. Методы вложения на основе обучения в настоящий момент являются стандартом благодаря широкой применимости и высокой эффективности. На самом деле вложения слов в задачах NLP так распространены, что их называют «сирача NLP» (sriracha of NLP), поскольку можно ожидать, что их использование в любой задаче обеспечит повышение эффективности решения<sup>1</sup>. Но

---

<sup>1</sup> Для тех, кто не слышал об этом соусе, сирача (или шрирача) — популярная в США приправа на основе перца чили.

это прозвище немного вводит в заблуждение, ведь, в отличие от сирачи, вложения обычно не добавляются в модель постфактум, а представляют собой ее базовый компонент.

В этой главе мы обсудим векторные представления в связи с вложениями слов: методы вложения слов, методы оптимизации вложений слов для задач обучения как с учителем, так и без учителя, методы визуализации вложений слов, а также методы сочетания вложений слов для предложений и документов. Впрочем, не забывайте, что описываемые здесь методы применимы к любому дискретному типу.

## Зачем нужно обучение вложениям

В предыдущих главах мы продемонстрировали вам обычные методы создания векторных представлений слов. А именно, вы узнали, как использовать унитарные представления — векторы длиной, совпадающей с размером словаря, с нулями на всех позициях, кроме одной, содержащей значение 1, соответствующее конкретному слову. Кроме того, вы встречались с представлениями количеств вхождений — векторами длиной, равной числу уникальных слов в модели, содержащими количества вхождений слов в предложение на соответствующих позициях. Такие представления называются также *дистрибутивными* (distributional representations), поскольку их значимое содержание/смысл отражается несколькими измерениями вектора. История дистрибутивных представлений насчитывает уже много десятилетий (см. статью Ферта [Firth, 1935]), они отлично подходят для множества моделей машинного обучения и нейронных сетей. Эти представления конструируются эвристически<sup>1</sup>, а не обучаются на данных.

Распределенные представления (distributed representation) получили свое название оттого, что слова в них представлены плотным вектором намного меньшей размерности (допустим,  $d=100$  вместо размера всего словаря, который может быть порядка  $10^5$  или  $10^6$ ), а смысл и другие свойства слова распределены по нескольким измерениям этого плотного вектора.

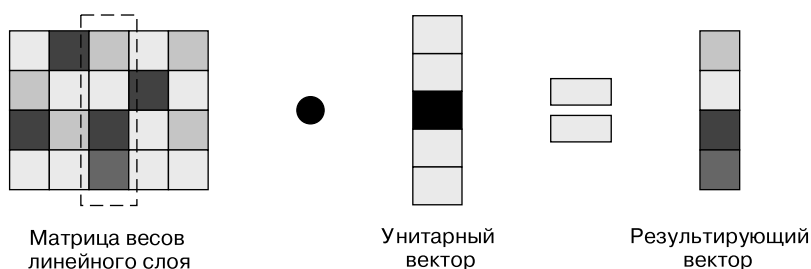
У низкоразмерных плотных представлений, полученных в результате обучения, есть несколько преимуществ по сравнению с унитарными и содержащими количества вхождений векторами, с которыми мы сталкивались в предыдущих главах. Во-первых, понижение размерности эффективно с вычислительной точки зрения. Во-вторых, представления на основе количества вхождений приводят к векторам высокой размерности с излишним кодированием одной и той же информации

<sup>1</sup> Прекрасный обзор традиционных (не основанных на нейронных сетях) методов вложения вы можете найти в статье «Распределительные подходы к определению смысла слов» (<https://stanford.io/2LuklLp>) Криса Поттса (Chris Potts) из Стэнфордского университета.

в различных измерениях, и их статистическая мощность не слишком велика. В-третьих, слишком большая размерность входных данных может привести к проблемам при машинном обучении и оптимизации — феномен, часто называемый *проклятием размерности* (<http://bit.ly/2CrhQXm>). Для решения этой проблемы с размерностью применяются различные способы понижения размерности, например сингулярное разложение (singular-value decomposition, SVD) и метод главных компонент (principal component analysis, PCA), но, по иронии судьбы, эти подходы плохо масштабируются на размерности порядка миллионов (типичный случай в NLP). В-четвертых, представления, усвоенные из (или подогнанные на основе) относящихся к конкретной задаче данных, оптимально подходят именно для этой задачи. В случае эвристических алгоритмов вроде TF-IDF и методов понижения размерности наподобие SVD непонятно, подходит ли для конкретной задачи целевая функция оптимизации при таком способе вложения.

## Эффективность вложений

Чтобы понять, как работают вложения, рассмотрим пример унитарного вектора, на который умножается матрица весов в линейном слое, как показано на рис. 5.1. В главах 3 и 4 размер унитарных векторов совпадал с размером словаря. Вектор называется унитарным потому, что содержит 1 на позиции, соответствующей конкретному слову, указывая таким образом на его наличие.



**Рис. 5.1.** Пример умножения матрицы для случая унитарного вектора и матрицы весов линейного слоя. Поскольку унитарный вектор содержит все нули и только одну единицу, позиция этой единицы играет роль оператора выбора при умножении матрицы. На рисунке это показано в виде затемнения ячеек матрицы весов и итогового вектора. Подобный метод поиска хотя и работает, но требует большого расхода вычислительных ресурсов и неэффективен, поскольку унитарный вектор умножается на каждое из чисел матрицы весов и вычисляется сумма по строкам

По определению число строк<sup>1</sup> матрицы весов линейного слоя, принимающей на входе унитарный вектор, должно быть равно размеру этого унитарного вектора. При умножении матрицы, как показано на рис. 5.1, итоговый вектор фактически

<sup>1</sup> Вероятно, здесь и далее в этом абзаце автор имеет в виду столбцы, а не строки. — *Примеч. пер.*



представляет собой строку, соответствующую ненулевому элементу унитарного вектора. С учетом этого наблюдения можно пропустить шаг умножения и воспользоваться целочисленным значением в качестве индекса для извлечения нужной строки.

Еще одно, последнее примечание относительно эффективности вложений: несмотря на пример с рис. 5.1, где размерность матрицы весов совпадает с размерностью входного унитарного вектора, так бывает далеко не всегда. На самом деле вложения часто применяются для представления слов из пространства меньшей размерности, чем понадобилось бы при использовании унитарного вектора или представления количества вхождений. Типичный размер вложений в научных статьях — от 25 до 500 измерений, причем выбор конкретного значения сводится к объему доступной памяти GPU.

## Подходы к обучению вложениям слов

Задача этой главы — не научить вас конкретным методикам вложений слов, а помочь разобраться, что такое вложения, как и где их можно применять, как лучше использовать их в моделях, а также каковы их ограничения. Дело в том, что на практике редко приходится писать новые алгоритмы обучения вложениям слов. Однако в данном подразделе мы сделаем краткий обзор современных подходов к такому обучению. Обучение во всех методах вложения слов производится с помощью одних только слов (то есть немаркированных данных), однако *с учителем*. Это становится возможно благодаря созданию вспомогательных задач обучения с учителем, в которых данные маркируются неявно, из тех соображений, что оптимизированное для решения вспомогательной задачи представление должно захватывать множество статистических и лингвистических свойств корпуса текста, чтобы приносить хоть какую-то пользу. Вот несколько примеров подобных вспомогательных задач.

- ❑ Предсказать следующее слово по заданной последовательности слов. Носит также название *задачи моделирования языка* (language modelling).
- ❑ Предсказать пропущенное слово по словам, расположенным до и после него.
- ❑ Предсказать слова в рамках определенного окна, независимо от позиции, по заданному слову.

Конечно, этот список не полон и выбор вспомогательной задачи зависит от интуиции разработчика алгоритма и вычислительных затрат. В числе примеров *GloVe*, *непрерывное мультимножество слов* (Continuous Bag-of-Words, CBOW), *Skipgrams* и т. д. Подробности можно найти в главе 10 книги Голдберга (Goldberg, 2017), но модель CBOW мы вкратце обсудим здесь. Впрочем, в большинстве случаев вполне достаточно воспользоваться предобученными вложениями слов и подогнать их к имеющейся задаче.

## Практическое применение предобученных вложений слов

Основная часть этой главы, а также остальная часть книги касается использования *предобученных вложений слов*. Предобученные с помощью одного из множества описанных выше методов на большом корпусе — например, на полном корпусе Google News, «Википедии» или Common Crawl<sup>1</sup> — вложения слов можно свободно скачивать и использовать. Далее в главе мы покажем, как грамотно находить и загружать эти вложения, изучим некоторые свойства вложений слов и приведем примеры применения предобученных вложений слов в задачах NLP.

### Загрузка вложений

Вложения слов стали настолько популярными и распространенными, что для скачивания доступно множество различных их вариантов, начиная с первоначального Word2Vec<sup>2</sup> до стэнфордского GloVe (<https://stanford.io/2PSivPZ>), в том числе FastText<sup>3</sup> компании Facebook (<https://fasttext.cc/>) и многие другие. Обычно вложения поставляются в следующем формате: каждая строка начинается со слова/типа, за которым идет последовательность чисел (то есть векторное представление). Длина этой последовательности равна размерности представления (*размерности вложения*). Размерность вложений обычно порядка сотен. Количество типов токенов чаще всего равно размеру словаря и составляет порядка миллиона. Например, вот первые семь измерений векторов *dog* и *cat* из GloVe.

dog	-1.242	-0.360	0.573	0.367	0.600	-0.189	1.273	...
cat	-0.964	-0.610	0.674	0.351	0.413	-0.212	1.380	...

Для эффективной загрузки и обработки вложений мы опишем вспомогательный класс `PreTrainedEmbeddings` (пример 5.1). В нем создается хранимый в оперативной памяти индекс всех векторов слов для упрощения быстрого поиска и запросов ближайших соседей с помощью пакета приближенного вычисления ближайших соседей, `annoy`.

<sup>1</sup> Common Crawl — лицензированный по лицензии Creative Commons корпус сосканированных сайтов, доступный по адресу [commoncrawl.org](http://commoncrawl.org).

<sup>2</sup> Word2Vec — набор методов вложения. В этой главе мы рассмотрим вложение типа «непрерывное множество слов» из статьи Word2Vec. Чтобы скачать Word2Vec, перейдите по адресу <https://goo.gl/ZER2d5>.

<sup>3</sup> На момент написания данной книги FastText — единственный из известных нам пакетов, предлагающий вложения на различных языках. И не только вложения.

**Пример 5.1.** Использование предобученных вложений слов<sup>1</sup>

Input[0]

```

import numpy as np
from annoy import AnnoyIndex

class PreTrainedEmbeddings(object):
    def __init__(self, word_to_index, word_vectors):
        """
        Аргументы:
            word_to_index (dict): отображение слов
            в целочисленные значения
            word_vectors (список массивов numpy)
        """
        self.word_to_index = word_to_index
        self.word_vectors = word_vectors
        self.index_to_word = \
            {v: k for k, v in self.word_to_index.items()}
        self.index = AnnoyIndex(len(word_vectors[0]),
                                metric='euclidean')
        for _, i in self.word_to_index.items():
            self.index.add_item(i, self.word_vectors[i])
        self.index.build(50)

    @classmethod
    def from_embeddings_file(cls, embedding_file):
        """ Создаем экземпляр на основе файла векторов,
            заранее полученных в результате обучения

        Формат файла векторов должен быть следующим:
            word0 x0_0 x0_1 x0_2 x0_3 ... x0_N
            word1 x1_0 x1_1 x1_2 x1_3 ... x1_N

        Аргументы:
            embedding_file (str): местоположение файла
        Возвращает:
            экземпляр PretrainedEmbeddings
        """
        word_to_index = {}
        word_vectors = []
        with open(embedding_file) as fp:

```

<sup>1</sup> При выполнении примера 5.1 может возникнуть ошибка кодировки из-за содержащихся в наборе Glove символов не из таблицы ASCII. Простейший способ решения этой проблемы — воспользоваться в методе `from_embeddings_file` функцией `open` из пакета `io`, указав кодировку UTF, вот так:

```
import io
```

```
...
```

```
    with io.open(embedding_file, encoding='utf-8') as fp:
```

... — *Примеч. пер.*

```

for line in fp.readlines():
    line = line.split(" ")
    word = line[0]
    vec = np.array([float(x) for x in line[1:]])

    word_to_index[word] = len(word_to_index)
    word_vectors.append(vec)
return cls(word_to_index, word_vectors)

```

```
Input[1]
```

```

embeddings = \
    PreTrainedEmbeddings.from_embeddings_file('glove.6B.100d.txt')

```

В этих примерах мы используем вложения слов GloVe. Их необходимо скачать и создать экземпляр класса `PreTrainedEmbeddings`, как показано в `Input[1]` из примера 5.1.

## Связи между вложениями слов

Ключевое свойство вложений слов — кодирование синтаксических и семантических связей, проявляющихся в виде закономерностей использования слов. Например, о котах и собаках обычно говорят очень схоже (обсуждают своих питомцев, особенности кормления и т. п.). В результате вложения для слов *cats* и *dogs* гораздо ближе друг к другу, чем к вложениям для названий других животных, скажем уток и слонов.

Изучать семантические связи, закодированные во вложениях слов, можно по-разному. Один из самых популярных методов — использовать задачу на аналогию (один из частых видов задач на логическое мышление на таких экзаменах, как SAT):

Слово1 : Слово2 :: Слово3 : \_\_\_\_\_

В этой задаче необходимо по заданным трем словам определить четвертое, отвечающее связи между первыми двумя<sup>1</sup>. С помощью вложений слов эту задачу можно кодировать пространственно. Во-первых, вычитаем *Слово2* из *Слово1*. Вектор разности между ними кодирует связь между *Слово1* и *Слово2*. Эту разность затем можно прибавить к *Слово3* и получить в результате вектор, ближайший к четвертому, пропущенному слову. Для решения задачи на аналогию достаточно выполнить запрос ближайших соседей по индексу с помощью этого полученного вектора. Соответствующая функция, показанная в примере 5.2, делает в точности описанное выше: она с помощью векторной арифметики и приближенного индекса ближайших соседей находит недостающий элемент в аналогии.

<sup>1</sup> Имеется в виду, что четвертое слово должно соотноситься с третьим так же, как второе с первым. — *Примеч. пер.*

**Пример 5.2.** Решение задачи на аналогию с помощью вложений слов

Input[0]

```

import numpy as np
from annoy import AnnoyIndex

class PreTrainedEmbeddings(object):
    """ Продолжение реализации из предыдущего примера """
    def get_embedding(self, word):
        """
        Аргументы:
            word (str)
        Возвращает
            вложение (numpy.ndarray)
        """
        return self.word_vectors[self.word_to_index[word]]
    def get_closest_to_vector(self, vector, n=1):
        """ Возвращает n ближайших соседей заданного вектора
        Аргументы:
            vector (np.ndarray): размер его должен соответствовать
            размеру векторов в индексе Annoy
            n (int): требуемое число соседей
        Возвращает:
            [str, str, ...]: ближайшие к заданному вектору слова
            По расстоянию эти слова не упорядочиваются
        """

        nn_indices = self.index.get_nns_by_vector(vector, n)
        return [self.index_to_word[neighbor]
                for neighbor in nn_indices]

    def compute_and_print_analogy(self, word1, word2, word3):
        """ Выводит в консоль решения задачи на аналогию,
            с помощью вложений слов
            word1 по отношению к word2 аналогично word3 по отношению
            к __
            Данный метод выводит в консоль: word1 : word2 :: word3 :
            word4

        Аргументы:
            word1 (str)
            word2 (str)
            word3 (str)
        """
        vec1 = self.get_embedding(word1)
        vec2 = self.get_embedding(word2)
        vec3 = self.get_embedding(word3)

        # Простая гипотеза: аналогия представляет собой
        # пространственную связь
        spatial_relationship = vec2 - vec1
        vec4 = vec3 + spatial_relationship

        closest_words = self.get_closest_to_vector(vec4, n=4)

```

```

existing_words = set([word1, word2, word3])
closest_words = [word for word in closest_words
                  if word not in existing_words]

if len(closest_words) == 0:
    print("Could not find nearest neighbors for the
          vector!")
    return

for word4 in closest_words:
    print("{} : {} :: {} : {}".format(word1, word2,
                                      word3, word4))

```

Что интересно, с помощью простой словесной аналогии можно продемонстрировать, как вложения слов способны улавливать разнообразные семантические и синтаксические связи (пример 5.3).

**Пример 5.3.** Кодирование с помощью вложений слов множества лингвистических связей на примере задач на аналогии SAT

Input[0]

```

# Связь 1: связь между гендерно-дифференцированными
# существительными и местоимениями
embeddings.compute_and_print_analogy('man', 'he', 'woman')

```

Output[0]

man : he :: woman : she

Input[1]

```

# Связь 2: связи "глагол – существительное"
embeddings.compute_and_print_analogy('fly', 'plane', 'sail')

```

Output[1]

fly : plane :: sail : ship

Input[2]

```

# Связь 3: связи "существительное – существительное"
embeddings.compute_and_print_analogy('cat', 'kitten', 'dog')

```

Output[2]

cat : kitten :: dog : puppy

Input[3]

```

# Связь 4: гиперонимия (обобщение)
embeddings.compute_and_print_analogy('blue', 'color', 'dog')

```

Output[3]

blue : color :: dog : animal

Input[4]

```

# Связь 5: меронимия (отношение "часть – целое")
embeddings.compute_and_print_analogy('toe', 'foot', 'finger')

```

Output[4]

toe : foot :: finger : hand

```
Input[5] # Связь 6: тропонимия (отличия в способе осуществления действия)
embeddings.compute_and_print_analogy('talk', 'communicate', 'read')
```

```
Output[5] talk : communicate :: read : interpret
```

```
Input[6] # Связь 7: метонимия (соглашения/фигуры речи)
embeddings.compute_and_print_analogy('blue', 'democrat', 'red')
```

```
Output[6] blue : democrat :: red : republican
```

```
Input[7] # Связь 8: степени прилагательных
embeddings.compute_and_print_analogy('fast', 'fastest', 'young')
```

```
Output[7] fast : fastest :: young : youngest
```

Хотя может показаться, что связи четко отражают функционирование языка, не все так просто. Как демонстрирует пример 5.4, связи могут неверно определяться, поскольку векторы слов определяются на основе их совместной встречаемости.

**Пример 5.4.** Пример, иллюстрирующий опасность кодирования смысла слов на основе совместной встречаемости, — иногда это не работает!

```
Input[0] embeddings.compute_and_print_analogy('fast', 'fastest', 'small')
```

```
Output[0] fast : fastest :: small : largest
```

Пример 5.5 иллюстрирует одно из самых распространенных сочетаний при кодировании гендерных ролей.

**Пример 5.5.** Осторожнее с защищаемыми атрибутами, например с гендером, кодируемыми вложениями слов. Они могут приводить к нежелательной предвзятости в дальнейших моделях

```
Input[0] embeddings.compute_and_print_analogy('man', 'king', 'woman')
```

```
Output[0] man : king :: woman : queen
```

Оказывается, довольно сложно различить закономерности языка и закоренелые культурные предубеждения. Например, доктора отнюдь не всегда мужчины, а медсестры не всегда женщины, но подобные предубеждения настолько устоялись, что отразились в языке, а в результате и в векторах слов, как показано в примере 5.6.

**Пример 5.6.** «Зашитые» в векторы слов культурные предубеждения

```
Input[0] embeddings.compute_and_print_analogy('man', 'doctor', 'woman')
```

```
Output[0] man : doctor :: woman : nurse
```

Не следует забывать о возможных систематических ошибках во вложениях с учетом роста их популярности и распространенности в NLP-приложениях. Искоренение систематических ошибок во вложениях слов — новая и очень интересная сфера научных исследований (см. статью Болукбаши и др. [Bolukbasi et al., 2016]). Рекомендуем заглянуть на сайт [ethicsinnlp.org](https://ethicsinnlp.org), где можно найти актуальную информацию по вопросам интерсекциональности этики и NLP.

## Пример: обучение вложениям модели непрерывного мультимножества слов

В этом примере мы рассмотрим одну из самых известных моделей, предназначенных для конструирования универсальных вложений слов и обучения им — модель Word2Vec непрерывного мультимножества слов<sup>1</sup>. В этом разделе, говоря о задаче CBOW или задаче классификации CBOW, мы подразумеваем, что конструируем задачу классификации с целью обучения вложениям CBOW. Модель CBOW представляет собой задачу многоклассовой классификации, включающую просмотр текстов, создание контекстных окон слов, исключение из этих окон центральных слов и классификацию контекстных окон по этим отсутствующим словам. Ее можно рассматривать как задачу заполнения пропусков: дано предложение с пропущенным словом и задача модели — выяснить, какое слово там должно стоять.

Цель данного примера — познакомить вас со слоем вложений `nn.Embedding` — модулем фреймворка PyTorch, инкапсулирующим матрицу вложения. С помощью слоя вложений можно отобразить целочисленный идентификатор токена в вектор, пригодный для применения в вычислениях нейронной сети. При обновлении оптимизатором весов модели с целью минимизации потерь обновляются также значения этого вектора. Во время этого процесса модель обучается создавать вложения слов оптимальным для конкретной задачи образом.

В оставшейся части данного примера мы будем следовать нашему стандартному формату примеров. В первом разделе познакомим вас с набором данных — оцифрованной версией романа Мэри Шелли «Франкенштейн». Далее мы обсудим

<sup>1</sup> См. блокнот `/chapters/chapter_5/5_2_CBOW/5_2_Continuous_Bag_of_Words_CBOW.ipynb` в репозитории GitHub этой книги ([https://nlpproc.info/PyTorchNLPBook/](https://nlpproc.info/PyTorchNLPBook/repo/)repo/).



конвейер векторизации: от токенов до векторизованных мини-пакетов. После этого вкратце опишем модель классификации CBOW и применение слоя вложений. Далее мы рассмотрим процедуру обучения (хотя, если вы читали главы по порядку, она уже хорошо вам знакома). Наконец, попробуем оценить эффективность модели, обсудим ее вывод и способы ее проверки.

## Набор данных «Франкенштейн»

Для этого примера мы сформируем текстовый набор данных на основе оцифрованной версии романа Мэри Шелли «Франкенштейн», доступной на сайте проекта «Гутенберг» (<http://bit.ly/2T5iU8J>). В подразделе мы обсудим предварительную обработку, создание класса `Dataset` фреймворка PyTorch для этого набора данных и, наконец, разбиение набора данных на обучающий, проверочный и контрольный наборы.

Начнем с неформатированного текстового файла и выполним минимальную предварительную обработку: воспользуемся лексическим анализатором Punkt из набора инструментов NLTK (<http://bit.ly/2GvR09j>) для разбиения текста на отдельные предложения, после чего преобразуем все предложения в нижний регистр и удалим знаки пунктуации. Такая предварительная обработка позволит нам в дальнейшем разбить строковые значения по пробелам и получить список токенов (или лексем). Для нее мы снова воспользуемся функцией из раздела «Пример: классификация тональностей обзоров ресторанов» на с. 76.

Следующий этап: пронумеровать набор данных как последовательность окон для оптимизации модели CBOW. Для этого мы пройдем в цикле по списку токенов каждого из предложений и сгруппируем их в окна заданного размера<sup>1</sup>, как наглядно показано на рис. 5.2.

Последний этап формирования набора данных: разбиение данных на обучающий, проверочный и контрольный наборы. Напомним, что обучающий и проверочный наборы используются при обучении модели: обучающий — для обновления параметров, а проверочный — для оценки эффективности модели<sup>2</sup>. Контрольный

<sup>1</sup> Точный размер окна представляет собой критически важный для CBOW гиперпараметр. При слишком большом размере окон модель перестанет улавливать закономерности, при слишком маленьком — может упустить из виду интересные зависимости.

<sup>2</sup> Повторим, что не следует ни при каких обстоятельствах интерпретировать оценки эффективности, полученные на данных, на которых модель обучалась, как окончательные оценки эффективности модели. На проверочном наборе модель не обучается, так что эффективность ее работы на нем гораздо лучше отражает окончательную эффективность. Впрочем, ваши решения, как решения экспериментатора, могут оказаться необъективными при использовании данных об эффективности на проверочном наборе, а показатели эффективности модели — более высокими, чем были бы на новых данных.

набор задействуется не более одного раза, для повышения объективности оценки. В этом примере (как и в большинстве примеров книги) набор данных разбивается в соотношении 70 % на обучающий, 15 % на проверочный и 15% на контрольный набор данных.

Предварительно обработанное предложение	i pitied frankenstein my pity amounted to horror i abhorred myself
Окно 1	i pitied frankenstein my pity amounted to horror i abhorred myself
Окно 2	i pitied frankenstein my pity amounted to horror i abhorred myself
Окно 3	i pitied frankenstein my pity amounted to horror i abhorred myself
Окно 4	i pitied frankenstein my pity amounted to horror i abhorred myself

**Рис. 5.2.** Задача CBOW: предсказание слова по окружающему его контексту (с обеих сторон). Длина контекстных окон равна двум словам как слева, так и справа. В результате скольжения окна по тексту получается множество выборок с учителем, каждая со своим целевым словом (по центру). Окна, длина которых отличается от 2, дополняются пробелами до нужного размера. Например, для окна 3 по контекстам i pitied и my pity классификатор CBOW выдает предсказание frankenstein

Полученный набор данных с окнами и целями загружается с помощью объекта `DataFrame` библиотеки `Pandas` и индексируется в классе `CBOWDataset`. В примере 5.7 показан фрагмент кода метода `__getitem__()`, в котором используется векторизатор для преобразования контекста — правого и левого окон — в вектор. Целевое слово по центру окна преобразуется в целое число с помощью словаря.

**Пример 5.7.** Формирование класса набора данных для задачи CBOW

```
class CBOWDataset(Dataset):
    # ... существующая реализация из примера 3.15
    @classmethod
    def load_dataset_and_make_vectorizer(cls, cbow_csv):
        """ Загружает набор данных и создает новый векторизатор с нуля

        Аргументы:
            cbow_csv (str): местоположение набора данных
        Возвращает:
            экземпляр CBOWDataset
        """
        cbow_df = pd.read_csv(cbow_csv)
        train_cbow_df = cbow_df[cbow_df.split=='train']
        return cls(cbow_df, CBOWVectorizer.from_dataframe(train_cbow_df))

    def __getitem__(self, index):
        """ Основной метод — точка входа для наборов данных PyTorch

        Аргументы:
```

```

        index (int): индекс точки данных
    Возвращает:
        словарь с признаками (x_data) и меткой (y_target) точки данных
    """
    row = self._target_df.iloc[index]

    context_vector = \
        self._vectorizer.vectorize(row.context, self._max_seq_length)
    target_index = self._vectorizer.cbow_vocab.lookup_token(row.target)

    return {'x_data': context_vector,
            'y_target': target_index}

```

## Классы Vocabulary, Vectorizer и DataLoader

В задаче классификации CBOW конвейер преобразования текста в векторизованный мини-пакет не отличается ничем особенным: классы `Vocabulary` и `DataLoader` функционируют точно так же, как и в примере с классификацией тональностей обзоров ресторанов. Однако, в отличие от векторизаторов из глав 3 и 4, векторизатор в данном случае не формирует унитарные векторы. Вместо этого он формирует и возвращает вектор целых чисел, представляющих индексы контекста. Код функции `vectorize()` приведен в примере 5.8.

### Пример 5.8. Векторизатор для данных CBOW

```

class CBOWVectorizer(object):
    """ Векторизатор, приводящий словари в соответствие друг другу
        и использующий их """
    def vectorize(self, context, vector_length=-1):
        """
        Аргументы:
            context (str): строка разделенных пробелами слов
            vector_length (int): аргумент, жестко задающий длину вектора индексов
        """

        indices = \
            [self.cbow_vocab.lookup_token(token) for token in context.split(' ')]
        if vector_length < 0:
            vector_length = len(indices)

        out_vector = np.zeros(vector_length, dtype=np.int64)
        out_vector[:len(indices)] = indices
        out_vector[len(indices):] = self.cbow_vocab.mask_index

        return out_vector

```

Обратите внимание, что, если число токенов в контексте меньше максимальной длины, оставшиеся записи заполняются нулями. Это можно назвать *дополнением* нулями.

## Модель CBOWClassifier

Показанный в примере 5.9 класс `CBOWClassifier` выполняет три важных действия. Во-первых, для всех слов из контекста создаются векторы с помощью слоя вложений и соответствующих этим словам индексов. Во-вторых, задача состоит в сочетании векторов таким образом, чтобы уловить общий контекст. В данном примере мы суммируем значения из векторов. Впрочем, можно также находить максимальное или среднее значение либо даже воспользоваться многослойным перцептроном. В-третьих, с помощью линейного слоя и вектора контекста вычисляется вектор предсказаний, представляющий собой распределение вероятности для всего словаря. Максимальное (наиболее вероятное) значение в векторе предсказаний указывает на наиболее подходящее предсказание для целевого слова — пропущенного в контексте центрального слова.

Используемый здесь слой вложений параметризуется в основном двумя числами: количеством вложений (размером словаря) и размером вложений. Во фрагменте кода из примера 5.9 есть и третий аргумент: `padding_idx`. Он используется как значение-индикатор в подобных нашему случаях, когда длины точек данных могут различаться<sup>1</sup>. Слой вложений делает так, чтобы соответствующий этому индексу вектор и все его градиенты были равны нулю.

### Пример 5.9. Модель CBOWClassifier

```
class CBOWClassifier(nn.Module):
    def __init__(self, vocabulary_size, embedding_size, padding_idx=0):
        """
        Аргументы:
            vocabulary_size (int): количество записей в словаре,
                определяет число вложений и размер вектора предсказаний
            embedding_size (int): размер вложений
            padding_idx (int): по умолчанию 0; слой Embedding этот
                индекс не использует
        """
        super(CBOWClassifier, self).__init__()

        self.embedding = nn.Embedding(num_embeddings=vocabulary_size,
                                      embedding_dim=embedding_size,
                                      padding_idx=padding_idx)
        self.fc1 = nn.Linear(in_features=embedding_size,
                             out_features=vocabulary_size)

    def forward(self, x_in, apply_softmax=False):
        """ Прямой проход классификатора
```

<sup>1</sup> Этот паттерн указания индекса для дополнения нулями (чтобы можно было использовать точки данных переменной длины) будет повторяться во многих примерах, поскольку так часто встречается в лингвистических данных.

Аргументы:

`x_in (torch.Tensor)`: тензор входных данных  
 Значение `x_in.shape` должно быть `(batch, input_dim)`  
`apply_softmax (bool)`: флаг для многомерной логистической функции активации. При использовании функции потерь на основе перекрестной энтропии должен равняться `false`

Возвращает:

итоговый тензор. Значение `tensor.shape` должно быть `(batch, output_dim)`.

```
"""
x_embedded_sum = self.embedding(x_in).sum(dim=1)
y_out = self.fc1(x_embedded_sum)
```

```
if apply_softmax:
    y_out = F.softmax(y_out, dim=1)
```

```
return y_out
```

## Процедура обучения

В этом примере процедура обучения следует образцу, применявшемуся на протяжении всей книги. Сначала инициализируем набор данных, векторизатор, модель, функцию потерь и оптимизатор. Затем проходим в цикле по обучающему и проверочному фрагментам набора данных на протяжении определенного количества эпох, минимизируем потери на обучающем фрагменте и оцениваем эффективность модели на проверочном. Больше подробностей относительно процедуры обучения вы можете найти в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76, где мы рассмотрели ее подробнее. В примере 5.10 приведены используемые при обучении аргументы.

### Пример 5.10. Аргументы для сценария обучения CBOW

Input[0]

```
args = Namespace(
    # Информация о данных и путях
    cbow_csv="data/books/frankenstein_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch5/cbow",
    # Гиперпараметры модели
    embedding_size=300,
    # Гиперпараметры обучения
    seed=1337,
    num_epochs=100,
    learning_rate=0.001,
    batch_size=128,
    early_stopping_criteria=5,
    # Настройки времени выполнения не приводятся для экономии места
)
```

## Оценка модели и получение предсказаний

Оценка в этом примере основывается на предсказании целевого слова по заданному контексту для каждого из целевых слов и пар контекстов в контрольном наборе<sup>1</sup>. Правильно предсказанное слово означает, что модель обучилась предсказанию слов по контексту. В данном примере точность классификации целевых слов на контрольном наборе данных достигает 15 %. Этот результат не слишком высок по нескольким причинам. Во-первых, архитектура CBOW в примере должна была иллюстрировать конструирование универсальных вложений. Многие свойства исходной реализации были опущены, поскольку только вносили ненужные для изучения метода сложности (но при этом были необходимы для максимальной эффективности). Во-вторых, используемый набор данных очень мал — всего одна книга на примерно 70 000 слов, что совершенно недостаточно для выявления многих закономерностей при обучении с нуля. А созданные по последнему слову науки и техники вложения обычно обучаются на наборах данных, состоящих из терабайт текста<sup>2</sup>.

В этом примере мы показали, как с помощью слоя `nn.Embedding` фреймворка PyTorch с нуля обучать вложения путем подготовки искусственной задачи обучения с учителем — классификации CBOW. В следующем примере мы выясним, как приспособить уже обученное на одном корпусе вложение к другой задаче. В МО использование модели, обученной на одной задаче, для решения другой называется *переносом обучения* (transfer learning).

## Пример: перенос обучения для классификации документов с применением предобученных вложений

В предыдущем примере слой вложений использовался для простой классификации. Текущий пример расширяет его тремя способами: сначала загружает предобученные вложения слов, затем подгоняет их путем классификации целых статей из новостей и, наконец, с помощью сверточной нейронной сети улавливает пространственные связи между словами.

В этом примере мы воспользуемся набором данных AG News. Для моделирования последовательностей слов из AG News создадим вариант `SequenceVocabulary` класса `Vocabulary`, включающий нескольких специальных токенов, жизненно важных для моделирования предложений. Класс `Vectorizer` демонстрирует использование этого словаря.

<sup>1</sup> Состав набора данных описан в подразделе «Набор данных “Франкенштейн”» ранее.

<sup>2</sup> Набор данных Common Crawl содержит более 100 Тбайт данных.

После описания набора данных и способа формирования векторизованных мини-пакетов мы рассмотрим загрузку предобученных векторов слов в слой вложений и покажем, как подогнать их под нашу задачу. Далее, модель сочетает предобученный слой вложений с CNN, использовавшимся в разделе «Пример: классификация фамилий с помощью CNN» на с. 130. В попытке повышения сложности модели до более реалистического уровня мы также отметим места, где в качестве метода регуляризации применяется дропаут. Затем мы обсудим процедуру обучения. Наверное, вы не удивитесь, узнав, что она практически не поменялась с предыдущих примеров в главе 4 и этой главе. Наконец, мы завершим пример оценкой эффективности модели на контрольном наборе и обсудим результаты.

## Набор данных AG News

Набор данных AG News (<http://bit.ly/2SbWzpl>) представляет собой коллекцию из более чем миллиона новостных статей, собранных учеными в 2005 году для экспериментов с методами интеллектуального анализа данных и извлечения информации. Цель этого примера — продемонстрировать эффективность применения предобученных вложений слов при классификации текстов. В примере мы воспользуемся сокращенной версией набора данных, состоящей из 120 000 новостных статей, распределенных равномерно по четырем категориям: спорт, наука/технология, мир и бизнес. Помимо сокращения объема набора данных, мы будем рассматривать в качестве наблюдений заголовки статей и создадим задачу многоклассовой классификации для предсказания категории по заголовку.

Как и ранее, выполним предварительную обработку текста, удалим знаки препинания, добавив вокруг них (запятых, апострофов и точек) пробелы, и преобразуем текст в нижний регистр. Кроме того, мы разобьем набор данных на обучающий, проверочный и контрольный наборы, сначала агрегировав точки данных по метке класса, а затем распределив их по трем фрагментам. Таким образом, мы гарантируем одинаковое распределение классов во всех трех фрагментах.

Метод `NewsDataset.__getitem__()`, показанный в примере 5.11, следует уже знакомой вам простой формуле: из конкретной строки набора данных извлекается строковое значение, представляющее входные данные модели, векторизуется и объединяется с целочисленным значением, соответствующим категории новости (метке класса).

### Пример 5.11. Метод `NewsDataset.__getitem__()`

```
class NewsDataset(Dataset):
    @classmethod
    def load_dataset_and_make_vectorizer(cls, news_csv):
        """ Загружает набор данных и создает новый векторизатор с нуля

        Аргументы:
            news_csv (str): местоположение набора данных
```

```

Возвращает:
    экземпляр NewsDataset
"""
news_df = pd.read_csv(news_csv)
train_news_df = news_df[news_df.split=='train']
return cls(news_df, NewsVectorizer.from_dataframe(train_news_df))

def __getitem__(self, index):
    """ Основной метод — точка входа для наборов данных PyTorch

    Аргументы:
        index (int): индекс точки данных
    Возвращает:
        словарь с признаками (x_data) и меткой (y_target) точки данных
    """
    row = self._target_df.iloc[index]

    title_vector = \
        self._vectorizer.vectorize(row.title, self._max_seq_length)

    category_index = \
        self._vectorizer.category_vocab.lookup_token(row.category)

    return {'x_data': title_vector,
            'y_target': category_index}

```

## Классы Vocabulary, Vectorizer и DataLoader

В этом примере появился класс `SequenceVocabulary` — подкласс стандартного класса `Vocabulary`, включающий четыре специальных токена, используемых для данных последовательности: токены `UNK`, `MASK`, `BEGIN-OF-SEQUENCE` и `END-OF-SEQUENCE`. Мы опишем их подробнее в главе 6, но, если говорить вкратце, они служат для трех различных целей. Благодаря токenu `UNK` (сокращение от `unknown` — «неизвестный»), с которым мы сталкивались в главе 4, модель получает возможность обучаться представлениям для редких слов, а значит, обрабатывать во время контроля не встречавшиеся ей во время обучения слова. Токен `MASK` служит индикатором для слоев вложений и вычислений функций потерь в случае последовательностей переменной длины. Наконец токены `BEGIN-OF-SEQUENCE` и `END-OF-SEQUENCE` указывают нейронной сети на границы последовательности. Рисунок 5.3 демонстрирует результаты использования этих специальных токенов в конвейере векторизации.

Вторую часть конвейера для преобразования текста в векторизованный мини-пакет составляет векторизатор `NewsVectorizer`, в котором создается экземпляр и инкапсулируется применение `SequenceVocabulary`. В данном примере векторизатор следует паттерну, продемонстрированному в пункте «Класс Vectorizer» на с. 85: ограничению общего набора возможных слов путем подсчета вхождений и поро-  
го-



вой фильтрации при определенной частоте. Основная цель этого действия — улучшение качества сигнала для модели и снижение использования моделью памяти за счет отброса зашумленных, редко встречающихся слов.

SequenceVocabulary { '<MASK>': 0, '<UNK>': 1, '<BEGIN-OF-SEQUENCE>': 2, '<END-OF-SEQUENCE>': 3, 'is': 4, 'happy': 5 } 	Jerry is happy	Шаг 0: исходное предложение
	1 4 5	Шаг 1: используем Vocabulary для отображения слов в целые числа
	2 1 4 5 3	Шаг 2: охватываем предложение маркерами границ
	2 1 4 5 3 0 0	Шаг 3: дополняем векторы нулями для выравнивания их длины

**Рис. 5.3.** Простой пример конвейера векторизации с сокращенным до минимума классом SequenceVocabulary, содержащим четыре описанных выше специальных токена. Во-первых, он используется для отображения слов в последовательность целых чисел. Поскольку слова Jerry нет в SequenceVocabulary, оно отображается в целочисленное значение UNK. Далее к целым числам в начале и конце присоединяются специальные токены, отмечающие границы предложения. Наконец, целочисленные значения дополняются справа нулями до заданной длины, чтобы все векторы в наборе данных были одинаковой длины

После создания экземпляра класса `Vectorizer` его метод `vectorize()` принимает на входе название новостной статьи и возвращает вектор, длина которого соответствует самому длинному заголовку статьи из набора данных (пример 5.12). Он отвечает за два основных действия. Во-первых, локально сохраняет максимальную длину последовательности. Обычно за ее отслеживание отвечает набор данных, и во время вывода в качестве вектора применяется длина контрольной последовательности. Впрочем, при использовании CNN-модели важно поддерживать статический размер даже во время вывода. Во-вторых, как показано во фрагменте кода из примера 5.11, он возвращает дополненный нулями вектор целых чисел, отражающий слова из последовательности. Кроме того, в начало этого вектора целых чисел добавлено число для токена `BEGIN-OF-SEQUENCE`, а в конец — для `END-OF-SEQUENCE`. С точки зрения классификатора эти специальные токены отмечают границы последовательности, благодаря чему он может обрабатывать слова возле границ не так, как слова ближе к центру<sup>1</sup>.

<sup>1</sup> Важно отметить, что такое поведение допустимо, но не обязательно. В наборе данных должны быть свидетельства практической пользы такого поведения и его благоприятного влияния на итоговые потери.

**Пример 5.12.** Реализация векторизатора для набора данных AG News

```
class NewsVectorizer(object):
    def vectorize(self, title, vector_length=-1):
        """
        Аргументы:
            title (str): строка разделенных пробелами слов
            vector_length (int): аргумент, жестко задающий длину вектора индексов
        Возвращает:
            векторизованный заголовок статьи (numpy.array)
        """
        indices = [self.title_vocab.begin_seq_index]
        indices.extend(self.title_vocab.lookup_token(token)
                       for token in title.split(" "))
        indices.append(self.title_vocab.end_seq_index)

        if vector_length < 0:
            vector_length = len(indices)

        out_vector = np.zeros(vector_length, dtype=np.int64)
        out_vector[:len(indices)] = indices
        out_vector[len(indices):] = self.title_vocab.mask_index

        return out_vector

    @classmethod
    def from_dataframe(cls, news_df, cutoff=25):
        """ Создает экземпляр векторизатора на основе
            объекта DataFrame набора данных

        Аргументы:
            news_df (pandas.DataFrame): целевой набор данных
            cutoff (int): пороговое значение частоты вхождений
                для включения в словарь
        Возвращает:
            экземпляр NewsVectorizer
        """
        category_vocab = Vocabulary()
        for category in sorted(set(news_df.category)):
            category_vocab.add_token(category)

        word_counts = Counter()
        for title in news_df.title:
            for token in title.split(" "):
                if token not in string.punctuation:
                    word_counts[token] += 1

        title_vocab = SequenceVocabulary()
        for word, word_count in word_counts.items():
            if word_count >= cutoff:
                title_vocab.add_token(word)

        return cls(title_vocab, category_vocab)
```

## Модель NewsClassifier

Ранее в этой главе мы показали вам, как загрузить с диска предобученные вложения и эффективно использовать их с помощью алгоритма приближенного вычисления ближайших соседей из библиотеки `annoy` сервиса Spotify. В этом примере в результате сравнения векторов друг с другом мы обнаружим интересные лингвистические закономерности. Впрочем, предобученные векторы слов можно использовать с гораздо большей отдачей: для инициализации матрицы вложений слоя `Embedding`.

Для использования вложений слов в качестве начальной матрицы вложений необходимо сначала загрузить вложения с диска, затем выбрать нужное подмножество вложений для имеющихся в наших данных слов и, наконец, задать загруженное подмножество в качестве матрицы весов слоя вложений. Первый и второй из этих этапов показаны в примере 5.13. Одна из часто возникающих при этом проблем состоит в наличии в наборе данных слов, отсутствующих в предобученных вложениях GloVe. Распространенный метод решения — использовать один из методов инициализации из библиотеки PyTorch, например метод равномерной инициализации Ксавье<sup>1</sup>, как показано в примере 5.13 (см. статью Глоро и Бенжио<sup>2</sup> [Glorot, Bengio, 2010]).

**Пример 5.13.** Выбор подмножества вложений слов в соответствии со словарем

```
def load_glove_from_file(glove_filepath):
    """ Загрузка вложений GloVe

    Аргументы:
        glove_filepath (str): путь к файлу вложений GloVe
    Возвращает:
        word_to_index (dict), embeddings (numpy.ndarray)
    """
    word_to_index = {}
    embeddings = []
    with open(glove_filepath, "r") as fp:
        for index, line in enumerate(fp):
            line = line.split(" ") # each line: word num1 num2 ...
            word_to_index[line[0]] = index # word = line[0]
            embedding_i = np.array([float(val) for val in line[1:]])
            embeddings.append(embedding_i)
    return word_to_index, np.stack(embeddings)

def make_embedding_matrix(glove_filepath, words):
    """ Создает матрицу вложений для конкретного мультимножества слов

    Аргументы:
        glove_filepath (str): путь к файлу вложений GloVe
        words (list): список слов в наборе данных
```

<sup>1</sup> Получил название в честь Ксавье Глоро (Xavier Glorot), также известен как инициализатор Глоро. — *Примеч. пер.*

<sup>2</sup> См. [https://ru.wikipedia.org/wiki/Бенжио,\\_Йошуа](https://ru.wikipedia.org/wiki/Бенжио,_Йошуа).

```

Возвращает:
    final_embeddings (numpy.ndarray): матрица вложений
"""
word_to_idx, glove_embeddings = load_glove_from_file(glove_filepath)
embedding_size = glove_embeddings.shape[1]
final_embeddings = np.zeros((len(words), embedding_size))

for i, word in enumerate(words):
    if word in word_to_idx:
        final_embeddings[i, :] = glove_embeddings[word_to_idx[word]]
    else:
        embedding_i = torch.ones(1, embedding_size)
        torch.nn.init.xavier_uniform_(embedding_i)
        final_embeddings[i, :] = embedding_i

return final_embeddings

```

`NewsClassifier` в этом примере дополняет классификатор `ConvNet` из главы 4, где мы классифицировали фамилии с помощью CNN, — теперь будем использовать унитарные вложения символов. А именно, мы применяем слой вложений, отображающий индексы входных токенов в векторное представление. Мы заменяем матрицу весов слоя вложений и используем предобученное подмножество вложений, как показано в примере 5.14<sup>1</sup>. Вложения затем используются в методе `forward()` для отображения индексов в векторы. Помимо слоя вложений, все осталось таким же, как в примере из раздела «Пример: классификация фамилий с помощью CNN» на с. 130.

#### Пример 5.14. Реализация класса `NewsClassifier`

```

class NewsClassifier(nn.Module):
    def __init__(self, embedding_size, num_embeddings, num_channels,
                  hidden_dim, num_classes, dropout_p,
                  pretrained_embeddings=None, padding_idx=0):
        """
        Аргументы:
            embedding_size (int): размер векторов вложений
            num_embeddings (int): количество векторов вложений
            filter_width (int): ширина сверточных ядер
            num_channels (int): количество сверточных ядер в каждом слое
            hidden_dim (int): размер скрытого измерения
            num_classes (int): число классов в классификации
            dropout_p (float): параметр дропаута
            pretrained_embeddings (numpy.array): предобученные вложения слов.
            По умолчанию None. При наличии padding_idx (int):

```

<sup>1</sup> В более новых версиях фреймворка PyTorch замена матрицы весов слоя вложений абстрагирована — достаточно просто передать подмножество вложений в конструктор класса `Embedding`. Актуальную информацию об этом можно найти в документации PyTorch (<http://bit.ly/2T0cdVp>).

```

        индекс, соответствующий начальной позиции
    """
    super(NewsClassifier, self).__init__()

    if pretrained_embeddings is None:
        self.emb = nn.Embedding(embedding_dim=embedding_size,
                                num_embeddings=num_embeddings,
                                padding_idx=padding_idx)

    else:
        pretrained_embeddings = torch.from_numpy(pretrained_embeddings).float()
        self.emb = nn.Embedding(embedding_dim=embedding_size,
                                num_embeddings=num_embeddings,
                                padding_idx=padding_idx,
                                _weight=pretrained_embeddings)

    self.convnet = nn.Sequential(
        nn.Conv1d(in_channels=embedding_size,
                  out_channels=num_channels, kernel_size=3),
        nn.ELU(),
        nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                  kernel_size=3, stride=2),
        nn.ELU(),
        nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                  kernel_size=3, stride=2),
        nn.ELU(),
        nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                  kernel_size=3),
        nn.ELU()
    )

    self._dropout_p = dropout_p
    self.fc1 = nn.Linear(num_channels, hidden_dim)
    self.fc2 = nn.Linear(hidden_dim, num_classes)

def forward(self, x_in, apply_softmax=False):
    """ Прямой проход классификатора

    Аргументы:
        x_in (torch.Tensor): тензор входных данных
            Значение x_in.shape должно
            быть (batch, dataset._max_seq_length)
        apply_softmax (bool): флаг для многомерной логистической функции
            активации. При использовании функции потерь на основе
            перекрестной энтропии должен равняться false

    Возвращает:
        итоговый тензор. Значение tensor.shape должно
        быть (batch, num_classes).
    """

    # вложение и перестановка, чтобы сделать из признаков каналы
    x_embedded = self.emb(x_in).permute(0, 2, 1)
    features = self.convnet(x_embedded)

```

```

# усреднение и удаление лишнего измерения
remaining_size = features.size(dim=2)
features = F.avg_pool1d(features, remaining_size).squeeze(dim=2)
features = F.dropout(features, p=self._dropout_p)

# завершающий линейный слой, генерирующий результаты классификации
intermediate_vector = F.relu(F.dropout(self.fc1(features),
                                       p=self._dropout_p))
prediction_vector = self.fc2(intermediate_vector)

if apply_softmax:
    prediction_vector = F.softmax(prediction_vector, dim=1)

return prediction_vector

```

## Процедура обучения

Процедура обучения состоит из такой последовательности операций, как создание набора данных, модели, векторизатора, функции потерь, оптимизатора, проход в цикле по обучающему фрагменту набора данных и обновление параметров модели, проход в цикле по проверочному фрагменту набора данных и оценка эффективности, а также повтор итераций над набором данных определенное число раз. Эта последовательность действий уже должна быть вам хорошо знакома. Соответствующие гиперпараметры и другие необходимые для обучения аргументы приведены в примере 5.15.

**Пример 5.15.** Аргументы классификатора NewsClassifier на основе CNN с использованием предобученных вложений

```

args = Namespace(
    # Информация о данных и путях
    news_csv="data/ag_news/news_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch5/document_classification",
    # Гиперпараметры модели
    glove_filepath='data/glove/glove.6B.100d.txt',
    use_glove=False,
    embedding_size=100,
    hidden_dim=100,
    num_channels=100,
    # Гиперпараметры обучения
    seed=1337,
    learning_rate=0.001,
    dropout_p=0.1,
    batch_size=128,
    num_epochs=100,
    early_stopping_criteria=5,
    # Настройки времени выполнения не приводятся для экономии места
)

```

## Оценка модели и получение предсказаний

Задача в этом примере состояла в классификации заголовков новостей по соответствующим им категориям. Как вы видели в предыдущих примерах, существует два вида методов, позволяющих понять, насколько хорошо модель выполняет свою работу: количественная оценка с помощью контрольного набора данных и качественная оценка, когда сам эксперт изучает результаты классификации.

### Оценка на контрольном наборе данных

Хотя с задачей классификации заголовков новостей мы встречаемся впервые, количественная оценка не отличается от предыдущих процедур: задать для модели режим оценки, чтобы отключить дропаут и обратное распространение ошибки (с помощью метода `classifier.eval()`), после чего пройти в цикле по контрольному набору данных аналогично обучающему и проверочному наборам. При обычных настройках следует попробовать различные варианты обучения и, добившись удовлетворительных результатов, оценить эффективность модели. Мы оставим это вам в качестве упражнения. Какой итоговой точности можно достичь на этом контрольном наборе данных? Помните, что за весь процесс исследования контрольный набор данных можно использовать лишь один раз.

### Предсказание категорий заголовков свежих новостей

Конечная цель обучения классификатора — развертывание его в промышленной эксплуатации для вывода или предсказания на основе еще не виденных им данных. Предсказание категории еще не обработанного заголовка новостей требует нескольких шагов. Во-первых, нужно выполнить предварительную обработку текста аналогично предварительной обработке данных при обучении. Для вывода применим к данным ту же функцию предварительной обработки, что и для обучения. Полученная строка векторизуется с помощью векторизатора и преобразуется в тензор PyTorch. Далее к ней применяется классификатор. Вычисляется максимум вектора предсказаний, и находится вероятное название категории. Соответствующий код приведен в примере 5.16.

**Пример 5.16.** Выполнение предсказаний с применением обученной модели

```
def predict_category(title, classifier, vectorizer, max_length):
    """ Предсказание категории новостей для нового названия

    Аргументы:
        title (str): необработанное строковое значение с названием
        classifier (NewsClassifier): экземпляр обученного классификатора
        vectorizer (NewsVectorizer): соответствующий векторизатор
        max_length (int): максимальная длина последовательности
```

Примечание: сверточные нейронные сети чувствительны к размеру тензора входных данных.

Это гарантирует одинаковый его размер с обучающими данными.

```
"""
title = preprocess_text(title)
vectorized_title = \
    torch.tensor(vectorizer.vectorize(title, vector_length=max_length))
result = classifier(vectorized_title.unsqueeze(0), apply_softmax=True)
probability_values, indices = result.max(dim=1)
predicted_category = vectorizer.category_vocab.lookup_index(indices.item())

return {'category': predicted_category,
        'probability': probability_values.item()}
```

## Резюме

В этой главе мы изучили вложения слов — способ представления дискретных элементов вроде слов в виде векторов фиксированной размерности в пространстве, где расстояние между векторами кодирует множество разнообразных лингвистических свойств. Важно не забывать, что описанные здесь методики применимы к любым дискретным элементам — предложениям, абзацам, документам, записям базы данных и т. д. Это делает методики вложения незаменимыми для глубокого обучения, особенно при NLP. Мы показали, как применять предобученные вложения в стиле черного ящика. Мы вкратце обсудили несколько способов обучения этим вложениям непосредственно из данных, включая метод непрерывного мультимножества слов (CBOW). Далее мы показали, как обучить модель CBOW в контексте языкового моделирования. Наконец, мы прошли по примеру использования предобученных вложений и проанализировали тонкую подстройку вложений для таких задач, как классификация документов.

К сожалению, из-за нехватки места мы обошли множество важных тем, например исключение предвзятости вложений слов, моделирование контекста и многозначность. Языковые данные отражают реальный мир. Предрассудки общества часто отражаются в моделях в виде предвзятых обучающих корпусов текста. В одном исследовании ближайшими к местоимению «она» были слова «домохозяйка», «медсестра», «секретарь», «библиотекарь», «парикмахер» и т. д., а ближайшими к «он» — «хирург», «протезе», «философ», «архитектор», «финансист» и т. п. Обученные на подобных предвзятых вложениях слов модели и сами продолжают принимать решения, приводящие к несправедливым исходам. Исключение предвзятости вложений слов — еще очень юная сфера исследований; мы рекомендуем вам прочитать статью Болукбаши и др. (Bolukbasi et al., 2016) и цитирующие эту работу современные статьи. Отметим также, что в использованных нами вложениях слов не учитывался контекст. Например, в зависимости от контекста слово «партия» может иметь различный смысл, но все обсуждавшиеся здесь методы вложения схлопывают эти смыслы. Недавние работы, такие как статья Петерса и др. (Peters et al., 2018), предлагают методы создания адаптированных к контексту вложений.



## Библиография

1. *Firth J.* The Technique of Semantics // Transactions of the Philological Society. 1935.
2. *Baroni M., Dinu G., Kruszewski G.* Don't Count, Predict! A Systematic Comparison of Context-Counting vs. Context-Predicting Semantic Vectors // Proceedings of the 52nd Annual Meeting of the ACL. 2014.
3. *Bolukbasi T. et al.* Man Is to Computer Programmer as Woman Is to Homemaker? Debiasing Word Embeddings // NIPS. 2016.
4. *Goldberg Y.* Neural Network Methods for Natural Language Processing. Morgan and Claypool, 2017.
5. *Peters M. et al.* Deep Contextualized Word Representations. arXiv preprint arXiv:1802.05365. 2018.
6. *Glorot X., Bengio Y.* Understanding the Difficulty of Training Deep Feedforward Neural Networks // Proceedings of the 13th International Conference on Artificial Intelligence and Statistics. 2010.

# 6

## Моделирование последовательностей для обработки текстов на естественных языках

Последовательность — это упорядоченный набор элементов. В традиционном машинном обучении предполагается, что точки данных распределены независимо и одинаково (*independently and identically distributed, IID*), но во многих случаях, например при работе с языковыми, речевыми и временными данными, каждый элемент данных зависит от предшествующих и следующих за ним элементов. Такие данные называются *последовательностью* (*sequence*). Последовательная информация в естественных языках встречается повсеместно. Например, речь можно рассматривать как последовательность базовых элементов — *фонем*. В таких языках, как английский, каждое слово расположено в предложении отнюдь не произвольным образом. Его место может определяться расположенными до или после него словами. В частности, за предлогом *of* с большой вероятностью следует артикль *the*, например *The lion is the king of the jungle*. Отметим также, что во многих языках число (единственное или множественное) сказуемого в предложении должно соответствовать числу подлежащего. Вот пример:

The book is on the table.  
The books are on the table.

Иногда длина подобных зависимостей (ограничений) может оказаться произвольно большой. Например:

The book that I got yesterday is on the table.  
The books read by the second-grade children are shelved in the lower rack.

Короче говоря, для понимания естественного языка необходимо понимать, как устроены предложения. В предыдущих главах мы познакомились с упреждающими нейронными сетями, такими как многослойный перцептрон и сверточные нейронные сети, а также с возможностями векторных представлений. Хотя с помощью этих методик можно решить множество различных задач обработки текстов на естественных языках, нельзя утверждать, что они правильно моделируют последовательности<sup>1</sup>.

Вполне применимы и традиционные подходы к моделированию последовательностей в NLP, включая скрытые марковские модели, условные случайные поля и другие графические модели, хотя в этой книге мы их обсуждать не станем<sup>2</sup>.

В глубоком обучении моделирование последовательностей требует хранения скрытой информации о состоянии (*скрытого состояния*). По мере обхода элементов последовательности, например просмотра моделью слов в предложении, это скрытое состояние обновляется. Таким образом, скрытое состояние (обычно это вектор) инкапсулирует все просмотренные последовательностью на текущий момент данные<sup>3</sup>. Этим вектором скрытого состояния, называемым также *представлением последовательности* (sequence representation), можно затем воспользоваться во множестве задач моделирования последовательностей самыми разнообразными способами, в зависимости от решаемой задачи, начиная от классификации последовательностей до их предсказания. В этой главе мы изучим классификацию данных, а в главе 7 рассмотрим применение моделей последовательностей для генерации последовательностей.

Начнем с простейшей модели последовательности на основе нейронной сети — *рекуррентной нейронной сети* (recurrent neural network, RNN). После этого мы приведем комплексный пример применения RNN к классификации. А именно, вам предстоит увидеть пример использования символьной RNN для классификации фамилий по национальной принадлежности. Пример с фамилиями демонстрирует, что модели последовательностей могут улавливать орфографические (на уровне частей слов) языковые паттерны. Он реализован таким образом, чтобы читатель мог применить эту модель к другим задачам, включая моделирование последовательностей текста, в котором элементы данных представляют собой слова, а не символы.

---

<sup>1</sup> За исключением сверточных нейронных сетей. Как мы расскажем в главе 9, с помощью CNN можно эффективно собирать информацию о последовательности.

<sup>2</sup> Подробности вы можете найти в книге Коллера и Фридмана (Koller, Friedman, 2009).

<sup>3</sup> В главе 7 мы рассмотрим различные варианты моделей последовательностей, умеющие «забывать» старую, ненужную информацию.

## Введение в рекуррентные нейронные сети

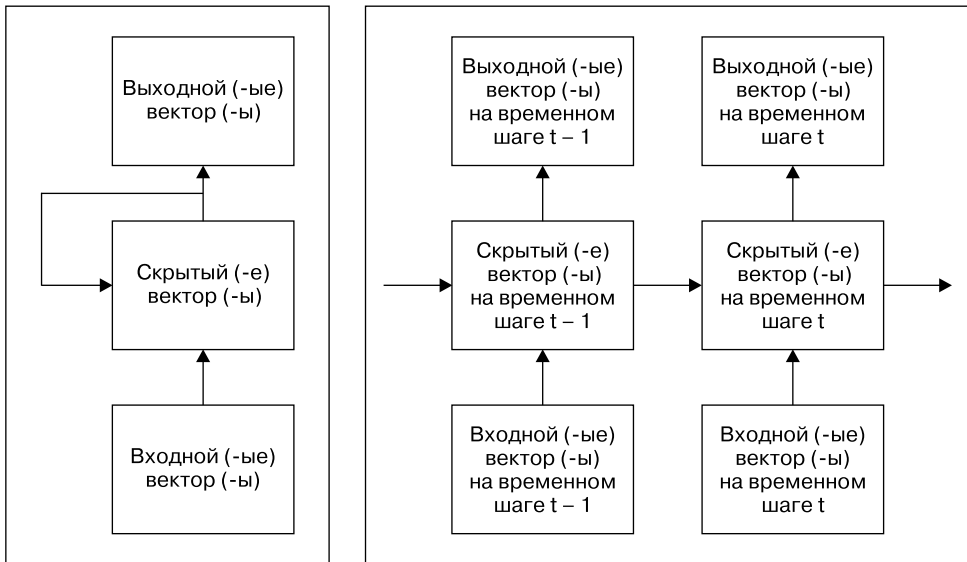
Цель рекуррентных нейронных сетей состоит в моделировании последовательностей тензоров<sup>1</sup>. Рекуррентные нейронные сети, подобно упреждающим, представляют собой целое семейство моделей. В семействе RNN содержится несколько различных моделей, из которых мы будем использовать только простейший вариант, иногда называемый *RNN Элмана* (Elman RNN)<sup>2</sup>. Задача рекуррентных сетей — как простейшей сети Элмана, так и более сложных форм, описанных в главе 7, — обучение представлению последовательности. Для этого предусмотрен вектор скрытого состояния, захватывающий текущее состояние последовательности. Он вычисляется на основе текущего входного вектора и предыдущего вектора скрытого состояния. Эти связи показаны на рис. 6.1, где демонстрируются как функциональная (*слева*), так и «развернутая» схемы вычислительных зависимостей. На обеих иллюстрациях выходной вектор совпадает со скрытым. Это не всегда так, но в случае RNN Элмана предсказания совпадают со скрытым вектором.

Рассмотрим более подробное описание, чтобы понять, что происходит внутри RNN Элмана. Как показано в «развернутом» представлении на рис. 6.1, известном также под названием «*обратное распространение ошибок во времени*» (backpropagation through time, BPTT), входной вектор с текущего шага времени и скрытый вектор с предыдущего шага отображаются в скрытый вектор состояния текущего шага. Показанный подробнее на рис. 6.2 новый скрытый вектор вычисляется с помощью матрицы весов «скрытый в скрытый» — для отображения предыдущего скрытого вектора состояния и матрицы весов «входной в скрытый» — для отображения входного вектора.

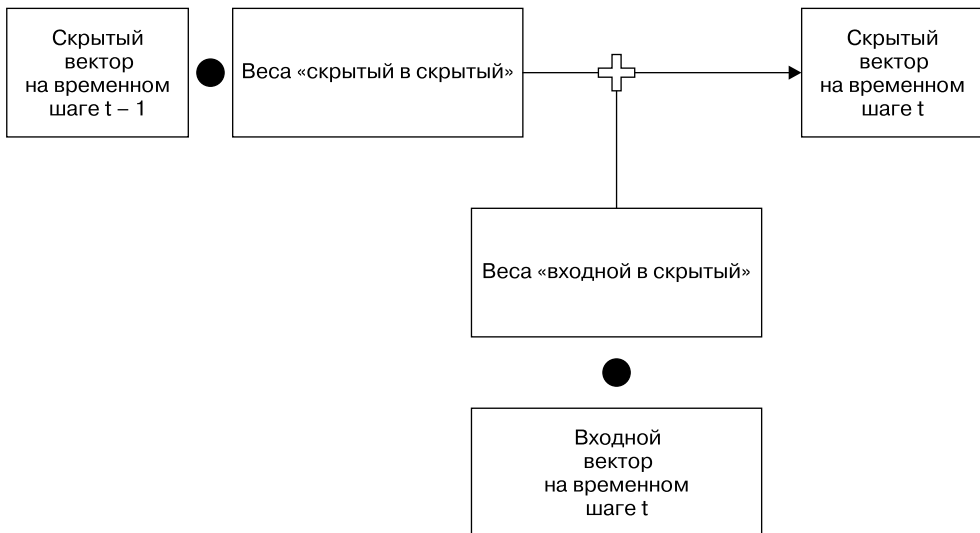
Критически важно, чтобы веса «скрытый в скрытый» и «входной в скрытый» могли совместно использоваться на различных временных шагах. Из этого факта следует сделать вывод, что при обучении эти веса необходимо подогнать таким образом, чтобы RNN обучалась учитывать поступающую информацию и поддерживать представление состояния, которое бы подытоживало все обработанные до сих пор входные данные. RNN ниоткуда не может узнать, на каком временном шаге она находится. Вместо этого она просто обучается переходу с одного шага на другой с хранением представления состояния, которое бы обеспечивало минимальную функцию потерь.

<sup>1</sup> Как вы помните из главы 1, в виде тензора можно выразить все что угодно. В данном случае RNN моделирует последовательность элементов в заданные моменты времени. Каждый из этих элементов можно выразить в виде тензора. В данной главе мы иногда будем использовать вместо слова «тензор» слово «вектор». Размерность будет понятна из контекста.

<sup>2</sup> В этой главе под RNN мы будем понимать RNN Элмана. На самом деле все, что мы будем в главе моделировать, можно смоделировать с помощью других рекуррентных нейронных сетей (чему посвящена глава 8), но для простоты ограничимся RNN Элмана. Имейте это в виду при чтении главы.



**Рис. 6.1.** Функциональное представление RNN Элмана (*слева*) демонстрирует рекуррентные связи в виде петли обратной связи для скрытых векторов. «Развернутое» представление (*справа*) более ясно показывает вычислительные связи; скрытый вектор на каждом временном шаге зависит как от входных данных этого шага, так и от скрытого вектора с предыдущего временного шага



**Рис. 6.2.** Явные вычисления внутри RNN Элмана показаны в виде сложения двух величин: скалярного произведения скрытого вектора с предыдущего шага на матрицу весов «скрытый в скрытый» и скалярного произведения входного вектора на матрицу весов «входной в скрытый»

Применение одних и тех же весов на всех временных шагах для преобразования входных данных в выходные — еще один пример совместного использования параметров. В главе 4 мы видели совместное использование параметров сверточными нейронными сетями в смысле пространства. CNN задействуют параметры, называемые *ядрами*, с целью вычисления выходных данных для различных подобластей входных данных. Сверточные ядра смещаются по входным данным, вычисляя выходные данные для всех возможных мест, чтобы усвоить трансляционную инвариантность. Напротив, RNN используют одни и те же параметры для вычисления выходных данных на всех временных шагах, захватывая состояние последовательности с помощью скрытого вектора состояния. Таким образом, задача RNN состоит в усвоении инвариантности благодаря возможности вычисления любых нужных выходных данных по заданному скрытому вектору состояния и входному вектору. Можно считать, что RNN совместно используют параметры во времени, а CNN — параметры в пространстве.

Поскольку длины слов и предложений могут различаться, RNN или любая другая модель последовательности должна уметь обрабатывать *последовательности переменной длины* (variable-length sequences). Один из методов достижения этого — искусственное ограничение длины последовательностей неким фиксированным значением. В этой книге мы применим для обработки последовательностей переменной длины другой метод — *маскирование* (masking), воспользовавшись знаниями длин последовательностей. Если вкратце, при маскировании данные могут сигнализировать, что определенные входные векторы не должны учитываться в градиенте или итоговых выходных данных. Фреймворк PyTorch предоставляет примитивы для работы с последовательностями переменной длины в классе `PackedSequence` — он служит для создания плотных тензоров из более разреженных. Пример этого показан в разделе «Пример: классификация национальной принадлежности фамилий с помощью символьного RNN» на с. 177<sup>1</sup>.

## Реализация RNN Элмана

Чтобы разобраться в нюансах RNN, пройдемся по простой реализации RNN Элмана. Фреймворк PyTorch предоставляет множество удобных классов и вспомогательных функций для построения RNN. Класс `RNN` фреймворка PyTorch реализует RNN Элмана. Вместо того чтобы применять непосредственно этот класс, мы воспользуемся `RNNCell` — абстракцией для отдельного временного шага RNN и сформируем RNN на ее основе. Благодаря этому мы сможем явным образом показать вам все выполняемые RNN вычисления. Приведенный в примере 6.1 класс

<sup>1</sup> В книгах и статьях, посвященных глубокому обучению, часто лишь вскользь затрагивают «подробности реализации» маскирования и упакованных последовательностей. Хотя для общего понимания RNN это не столь важно, углубленное знакомство с данными понятиями, которое вы получите в этой главе, для специалиста-практика жизненно важно. Обратите на это особое внимание!

ElmanRNN с помощью RNNCell создает вышеописанные матрицы весов «скрытый в скрытый» и «входной в скрытый». Каждый вызов RNNCell() принимает в качестве аргументов матрицу входных и матрицу скрытых векторов и возвращает матрицу скрытых векторов, получившуюся в результате данного шага.

**Пример 6.1.** Реализация RNN Элмана с помощью класса RNNCell фреймворка PyTorch

```
class ElmanRNN(nn.Module):
    """ RNN Элмана, созданная с помощью RNNCell """
    def __init__(self, input_size, hidden_size, batch_first=False):
        """
        Аргументы:
            input_size (int): размер входных векторов
            hidden_size (int): размер векторов скрытого состояния
            batch_first (bool): будут ли в нулевом измерении располагаться
                               данные пакета
        """
        super(ElmanRNN, self).__init__()

        self.rnn_cell = nn.RNNCell(input_size, hidden_size)

        self.batch_first = batch_first
        self.hidden_size = hidden_size

    def _initialize_hidden(self, batch_size):
        return torch.zeros((batch_size, self.hidden_size))

    def forward(self, x_in, initial_hidden=None):
        """ Прямой проход ElmanRNN
        Аргументы:
            x_in (torch.Tensor): тензор входных данных
            Если self.batch_first = true: x_in.shape =
                (batch_size, seq_size, feat_size)
            В противном случае: x_in.shape =
                (seq_size, batch_size, feat_size)
            initial_hidden (torch.Tensor): начальное скрытое состояние для RNN
        Возвращает:
            hiddens (torch.Tensor): выходные векторы RNN на каждом из шагов
            Если self.batch_first = true:
                hiddens.shape = (batch_size, seq_size, hidden_size)
            В противном случае: hiddens.shape =
                (seq_size, batch_size, hidden_size)
        """
        if self.batch_first:
            batch_size, seq_size, feat_size = x_in.size()
            x_in = x_in.permute(1, 0, 2)
        else:
            seq_size, batch_size, feat_size = x_in.size()

        hiddens = []

        if initial_hidden is None:
            initial_hidden = self._initialize_hidden(batch_size)
```

```
initial_hidden = initial_hidden.to(x_in.device)

hidden_t = initial_hidden

for t in range(seq_size):
    hidden_t = self.rnn_cell(x_in[t], hidden_t)
    hiddens.append(hidden_t)

hiddens = torch.stack(hiddens)

if self.batch_first:
    hiddens = hiddens.permute(1, 0, 2)

return hiddens
```

В RNN, помимо гиперпараметров, управляющих размером входных и скрытых векторов, есть булев аргумент, позволяющий указать, должно ли данное измерение находиться на месте нулевого. Этот флаг есть во всех реализациях RNN фреймворка PyTorch. Если он равен `True`, RNN меняет местами нулевое и первое измерения входного тензора.

Метод `forward()` в классе `ElmanRNN` проходит в цикле по входному тензору, вычисляя вектор скрытого состояния для каждого из временных шагов. Обратите внимание, что в нем есть аргумент для задания начального скрытого состояния, но если его не задать, то будет использоваться вектор скрытого состояния, состоящий из одних `0`. При проходе в цикле по входному вектору класс `ElmanRNN` вычисляет новое скрытое состояние. Эти скрытые состояния агрегируются и в конце концов располагаются ярусами<sup>1</sup>.

Перед их возвратом снова проверяется значение флага `batch_first`. Если оно равно `True`, выходные скрытые векторы переставляются так, что данные пакета снова будут находиться в нулевом измерении.

Выходные данные класса представляют собой трехмерный тензор — по вектору скрытого состояния для каждой точки данных в пакетном измерении и каждого шага по времени. Эти скрытые векторы можно использовать различными способами, в зависимости от имеющейся задачи. Один из этих способов: классификация с их помощью каждого из временных шагов по какому-либо дискретному набору вариантов. Данный метод означает, что веса RNN адаптируются к отслеживанию информации, связанной с предсказаниями на каждом из шагов. Кроме того, итоговый вектор можно использовать для классификации предложения в целом. Это значит, что веса RNN можно приспособить для отслеживания важной для итоговой классификации информации. В данной главе мы рассмотрим только классификацию, но в следующих двух главах изучим подробнее пошаговые предсказания.

---

<sup>1</sup> Обсуждение операции расположения ярусами PyTorch можно найти в подразделе «Операции над тензорами» на с. 35.



## Пример: классификация национальной принадлежности фамилий с помощью символьного RNN

Теперь, когда мы вкратце описали основные свойства RNN и прошли по реализации ElmanRNN, можно воспользоваться этими знаниями для решения какой-нибудь задачи. Рассмотрим задачу классификации фамилий из главы 4, где последовательности символов (фамилии) классифицируются по национальной принадлежности.

### Класс SurnameDataset

В этом примере мы воспользуемся набором данных фамилий, описанным в главе 4. Точки данных представляют собой фамилии и соответствующие национальности. Мы не станем повторяться, рассказывая обо всех нюансах набора данных, рекомендуем вам заглянуть в подраздел «Набор данных фамилий» на с. 110.

В этом примере, как и в разделе «Пример: классификация фамилий с помощью CNN» на с. 130, мы рассматриваем фамилии как последовательности символов. Как и всегда, мы реализуем класс набора данных, показанный в примере 6.2, который возвращает векторизованную фамилию вместе с соответствующим ее национальной принадлежности целочисленным значением. Кроме того, возвращается длина последовательности, используемая в вычислениях далее по конвейеру для определения местоположения завершающего вектора в последовательности. Это одна из частей уже привычной нам последовательности шагов — реализовать Dataset, Vectorizer и Vocabulary перед собственно обучением.

#### Пример 6.2. Реализация класса SurnameDataset

```
class SurnameDataset(Dataset):
    @classmethod
    def load_dataset_and_make_vectorizer(cls, surname_csv):
        """ Загружает набор данных и создает новый векторизатор с нуля

        Аргументы:
            surname_csv (str): местоположение набора данных
        Возвращает:
            экземпляр SurnameDataset
        """
        surname_df = pd.read_csv(surname_csv)
        train_surname_df = surname_df[surname_df.split=='train']
        return cls(surname_df, SurnameVectorizer.from_dataframe(train_surname_df))

    def __getitem__(self, index):
        """ Основной метод — точка входа для наборов данных PyTorch

        Аргументы:
            index (int): индекс точки данных
```

```

Возвращает:
    словарь, содержащий признаки (x_data), метку (y_target)
    и длину признака (x_length) точки данных
"""
row = self._target_df.iloc[index]

surname_vector, vec_length = \
    self._vectorizer.vectorize(row.surname, self._max_seq_length)

nationality_index = \
    self._vectorizer.nationality_vocab.lookup_token(row.nationality)

return {'x_data': surname_vector,
        'y_target': nationality_index,
        'x_length': vec_length}

```

## Структуры данных для векторизации

Первый этап конвейера векторизации — отображение токенов всех символов фамилии в уникальные целочисленные значения. Для этого воспользуемся структурой данных `SequenceVocabulary`, описанной в разделе «Пример: перенос обучения для классификации документов с использованием предобученных вложений» на с. 158. Напомним, что эта структура данных не только задает соответствия символов в фамилиях целочисленным значениям, но и включает четыре специальных токена: `UNK`, `MASK`, `BEGIN-OF-SEQUENCE` и `END-OF-SEQUENCE`. Первые два из них жизненно важны для языковых данных: `UNK` используется для ранее не виденных и отсутствующих в словаре токенов во входных данных, а `MASK` позволяет работать с последовательностями переменной длины. Последние два токена служат для модели признаками границ предложения и присоединяются к предложению спереди и сзади соответственно. Более подробное описание `SequenceVocabulary` вы можете найти в разделе «Классы `Vocabulary`, `Vectorizer` и `DataLoader`» на с. 160.

Общую процедуру векторизации выполняет класс `SurnameVectorizer`, использующий `SequenceVocabulary` для отображения символов в фамилиях в целочисленные значения. В примере 6.3 приведена его реализация, которая должна выглядеть для вас очень знакомой; в предыдущей главе мы обсуждали классификацию заголовков новостных статей по категориям и конвейер векторизации был практически точно таким же.

### Пример 6.3. Векторизатор для фамилий

```

class SurnameVectorizer(object):
    """Векторизатор, приводящий словари в соответствие друг другу
    и использующий их"""
    def vectorize(self, surname, vector_length=-1):
        """
        Аргументы:
            title (str): строка символов
            vector_length (int): аргумент, жестко задающий длину

```

```

        вектора индексов
        """
        indices = [self.char_vocab.begin_seq_index]
        indices.extend(self.char_vocab.lookup_token(token)
                       for token in surname)
        indices.append(self.char_vocab.end_seq_index)

        if vector_length < 0:
            vector_length = len(indices)

        out_vector = np.zeros(vector_length, dtype=np.int64)
        out_vector[:len(indices)] = indices
        out_vector[len(indices):] = self.char_vocab.mask_index

        return out_vector, len(indices)

    @classmethod
    def from_dataframe(cls, surname_df):
        """ Создает экземпляр векторизатора на основе
            объекта DataFrame набора данных

        Аргументы:
            surname_df (pandas.DataFrame): набор данных фамилий
        Возвращает:
            экземпляр SurnameVectorizer
        """
        char_vocab = SequenceVocabulary()
        nationality_vocab = Vocabulary()

        for index, row in surname_df.iterrows():
            for char in row.surname:
                char_vocab.add_token(char)
            nationality_vocab.add_token(row.nationality)

        return cls(char_vocab, nationality_vocab)

```

## Модель SurnameClassifier

Модель `SurnameClassifier` состоит из слоя вложений, RNN Элмана и линейного слоя. Мы предполагаем, что ее входными данными служат токены, представленные в виде набора целочисленных значений, после их отображения в целые числа с помощью `SequenceVocabulary`. Модель сначала выполняет вложение этих целых чисел с помощью соответствующего слоя вложений. Затем с применением RNN вычисляются векторы представления последовательности. Они соответствуют скрытым состояниям для каждого из символов фамилии. Поскольку задача — классифицировать фамилии, извлекается вектор, соответствующий позиции завершающего символа в каждой из фамилий. Его можно рассматривать как результат прохода по всей входной последовательности, а значит, сводный вектор для фамилии. Путем передачи этих сводных векторов через линейный слой вычисляется вектор предсказаний. Далее либо этот вектор предсказаний используется при вычислении

функции потерь, либо применяется многомерная логистическая функция для создания распределения вероятности по фамилиям<sup>1</sup>.

Аргументы модели: размер вложений, число вложений (то есть размер словаря), число классов и размер скрытого состояния RNN. Два из этих аргументов — число вложений и число классов — определяются данными. Остальные гиперпараметры: размер вложений и размер скрытого состояния. Хотя они могут принимать произвольные значения, обычно имеет смысл начать с маленьких значений, чтобы выполнить обучение быстро и проверить, что модель работает.

**Пример 6.4.** Реализация модели SurnameClassifier с помощью RNN Элмана

```
class SurnameClassifier(nn.Module):
    """ RNN для извлечения признаков & и MLP для классификации """
    def __init__(self, embedding_size, num_embeddings, num_classes,
                  rnn_hidden_size, batch_first=True, padding_idx=0):
        """
        Аргументы:
            embedding_size (int): размер вложений символов
            num_embeddings (int): количество символов для создания вложений
            num_classes (int): размер вектора предсказаний
                Примечание: количество национальностей
            rnn_hidden_size (int): размер скрытого состояния RNN
            batch_first (bool): указывает, будут ли в нулевом измерении
                входных тензоров находиться данные пакета или последовательности
            padding_idx (int): индекс для дополнения нулями тензора;
                см. torch.nn.Embedding
        """
        super(SurnameClassifier, self).__init__()
        self.emb = nn.Embedding(num_embeddings=num_embeddings,
                                embedding_dim=embedding_size,
                                padding_idx=padding_idx)
        self.rnn = ElmanRNN(input_size=embedding_size,
                             hidden_size=rnn_hidden_size,
                             batch_first=batch_first)
        self.fc1 = nn.Linear(in_features=rnn_hidden_size,
                              out_features=rnn_hidden_size)
        self.fc2 = nn.Linear(in_features=rnn_hidden_size,
                              out_features=num_classes)

    def forward(self, x_in, x_lengths=None, apply_softmax=False):
        """ Прямой проход классификатора

        Аргументы:
            x_in (torch.Tensor): тензор входных данных
                Значение x_in.shape должно быть (batch, input_dim)
            x_lengths (torch.Tensor): длины всех последовательностей пакета,
```

<sup>1</sup> В этом примере число классов невелико. Во многих же ситуациях в NLP число классов на выходе может достигать тысяч или даже сотен тысяч. В подобных случаях будет оправданным использование иерархической многомерной логистической функции вместо «наивной» многомерной.

```

используемые для поиска завершающих векторов последовательностей
apply_softmax (bool): флаг для многомерной логистической функции
активации. При использовании функции потерь на основе
перекрестной энтропии должен равняться false
Возвращает:
    out (torch.Tensor); `out.shape = (batch, num_classes)`
"""
x_embedded = self.emb(x_in)
y_out = self.rnn(x_embedded)

if x_lengths is not None:
    y_out = column_gather(y_out, x_lengths)
else:
    y_out = y_out[:, -1, :]

y_out = F.dropout(y_out, 0.5)
y_out = F.relu(self.fc1(y_out))
y_out = F.dropout(y_out, 0.5)
y_out = self.fc2(y_out)

if apply_softmax:
    y_out = F.softmax(y_out, dim=1)

return y_out

```

Как вы увидите, для функции `forward()` требуются длины последовательностей. Они используются для извлечения завершающих векторов каждой из последовательностей в тензоре, возвращаемых из RNN функцией `column_gather()` (пример 6.5). Она проходит в цикле по индексам строк пакета и извлекает вектор, находящийся на позиции, которая соответствует длине последовательности.

**Пример 6.5.** Извлечение завершающего выходного вектора каждой из последовательностей с помощью метода `column_gather()`

```

def column_gather(y_out, x_lengths):
    """ Извлекает определенный вектор из каждой точки данных пакета в `y_out`.

    Аргументы:
        y_out (torch.FloatTensor, torch.cuda.FloatTensor)
            shape: (batch, sequence, feature)
        x_lengths (torch.LongTensor, torch.cuda.LongTensor)
            shape: (batch,)

    Возвращает:
        y_out (torch.FloatTensor, torch.cuda.FloatTensor)
            shape: (batch, feature)
    """
    x_lengths = x_lengths.long().detach().cpu().numpy() - 1

    out = []
    for batch_index, column_index in enumerate(x_lengths):
        out.append(y_out[batch_index, column_index])

    return torch.stack(out)

```

## Процедура обучения и результаты

Процедура обучения следует стандартной схеме. Для каждого пакета данных применяется модель и вычисляются векторы предсказаний. С помощью функции `CrossEntropyLoss()` и эталонных данных вычисляется функция потерь. На базе значений потерь и оптимизатора вычисляются градиенты, а на их основе обновляются веса модели. Этот процесс повторяется для каждого из пакетов в обучающих данных. Аналогично поступаем с проверочными данными, но модель устанавливается в режим оценки, чтобы предотвратить обратное распространение ошибки. Проверочные данные используются только для получения менее предвзятого представления об эффективности работы модели. Данная процедура повторяется определенное количество эпох. Код вы можете найти в прилагаемых к книге материалах.

Рекомендуем вам поэкспериментировать с гиперпараметрами, чтобы лучше почувствовать, как они влияют на эффективность и насколько, и составить таблицу полученных результатов. Мы также оставляем вам в качестве упражнения написание подходящей базовой модели для этой задачи<sup>1</sup>. Реализованная в подразделе «Модель SurnameClassifier» на с. 179 модель носит общий характер, ее применимость не ограничивается символами. Слой вложений в той модели способен отобразить любой дискретный элемент в последовательность дискретных элементов; например, предложение — это последовательность слов. Попробуйте код из примера 6.6 и для других задач классификации, например классификации предложений.

### Пример 6.6. Аргументы для SurnameClassifier на основе RNN

```
args = Namespace(
    # Информация о данных и путях
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch6/surname_classification",
    # Гиперпараметры модели
    char_embedding_size=100,
    rnn_hidden_size=64,
    # Гиперпараметры обучения
    num_epochs=100,
```

<sup>1</sup> Для разминки представьте себе MLP, получающий на входе набор символьных униграмм. Затем представьте на его входе набор символьных биграмм. Вполне возможно, что базовые модели окажутся для этой задачи более подходящими, чем простая модель на основе RNN. Это весьма поучительно: для проектирования признаков достаточно сообщить базовой модели, что в символьных биграммах есть сигнал. Подсчитайте количество параметров в вариантах с униграммами и биграммами в качестве входных данных и сравните их с RNN-моделью из этой главы. Больше или меньше параметров оказалось и почему? Наконец, можете ли вы придумать более простую, чем MLP, эффективную базовую модель для данной задачи?

```
learning_rate=1e-3,  
batch_size=64,  
seed=1337,  
early_stopping_criteria=5,  
# Настройки времени выполнения не приводятся для экономии места  
)
```

## Резюме

В этой главе мы познакомили вас с рекуррентными нейронными сетями и рассказали, как их применять для моделирования данных последовательностей. Мы рассмотрели один из простейших вариантов рекуррентных сетей — RNN Элмана. Установили, что целью моделирования последовательностей является обучение представлению (то есть вектору) последовательности. Такое усвоенное представление можно использовать различными способами, в зависимости от задачи. Мы рассмотрели пример задачи, включающей классификацию подобного представления скрытого состояния по одному из нескольких классов. На примере задачи классификации фамилий мы продемонстрировали использование RNN для захвата информации на уровне частей слов.

## Библиография

*Koller D., Friedman N. Probabilistic Graphical Models: Principles and Techniques. MIT Press, 2009.*

# 7 Продолжаем моделирование последовательностей для обработки текстов на естественных языках

Наша цель в этой главе — научиться *предсказанию последовательностей*. Задача предсказания последовательности требует маркирования всех ее элементов. Подобные задачи часто встречаются при обработке текстов на естественных языках. В числе их примеров *моделирование языка* (рис. 7.1), при котором на каждом шаге по заданной последовательности слов предсказывается следующее слово; *частеречная разметка* (part-of-speech tagging), когда предсказывается грамматическая часть речи для каждого слова; *распознавание поименованных сущностей* (named entity recognition), при котором предсказывается, является ли каждое из слов частью поименованной сущности, и т. д. Иногда в литературе, посвященной NLP, задачи предсказания последовательностей также называются *маркированием последовательностей* (sequence labeling).

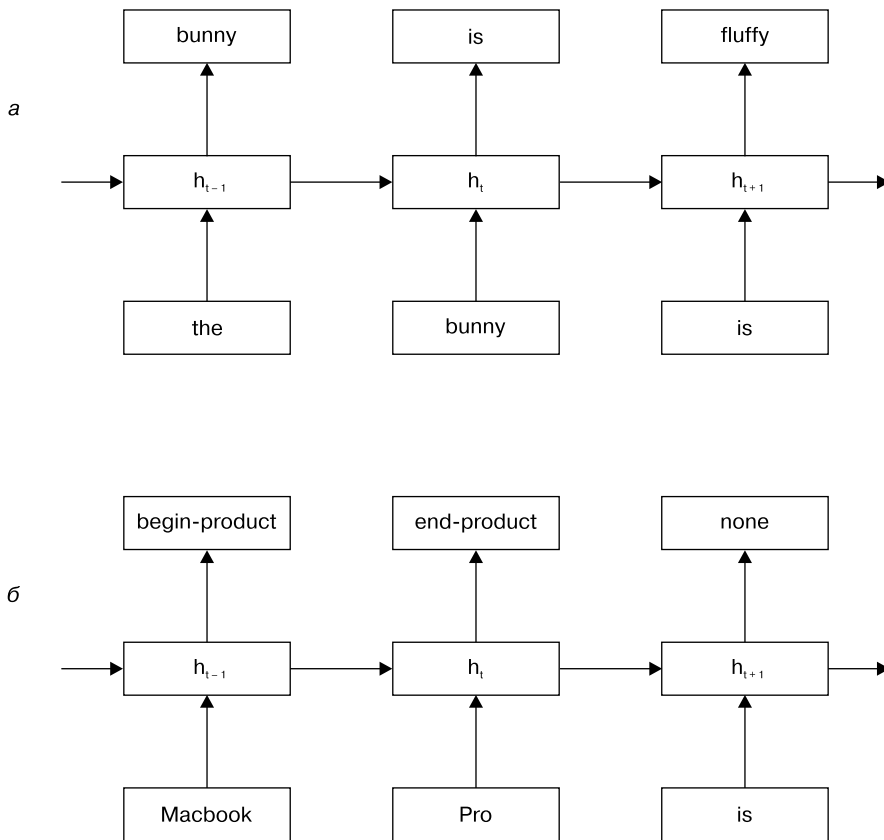
Хотя теоретически для задач предсказания последовательностей можно воспользоваться рекуррентными нейронными сетями Элмана, с которыми мы познакомились в главе 6, но эти RNN не способны уловить «далекие» зависимости и на практике работают плохо. В этой главе мы немного поговорим о том, почему так происходит, и расскажем о новом типе архитектуры RNN — *шлюзовых сетях*<sup>1</sup> (gated networks).

Мы также расскажем вам, как применять предсказание последовательностей для решения задачи *генерации естественного языка* (natural language generation), и изучим контекстно обусловленную генерацию, при которой выходная последовательность каким-либо образом ограничена.

---

<sup>1</sup> В русскоязычной литературе отсутствует устоявшийся термин для подобных нейронных сетей, их называют также «управляемыми» и «вентильными», но вариант «шлюзовые» в наибольшей степени отражает их сущность. — *Примеч. пер.*





**Рис. 7.1.** Два примера задач предсказания последовательностей: *а* — моделирование языка, задача которого — предсказание очередного слова в последовательности; *б* — распознавание поименованных сущностей, цель которого — предсказание границ строк-сущностей из текста вместе с их типами

## Проблемы «наивных» RNN (RNN Элмана)

Хотя «наивные» RNN и RNN Элмана, обсуждавшиеся в главе 6, хорошо подходят для моделирования последовательностей, существуют две проблемы, из-за которых эти сети не годятся для многих задач: невозможность хранения информации для долгосрочных предсказаний и нестабильность градиента. Чтобы понять, в чем именно эти проблемы, вспомним, что RNN, по сути, вычисляют вектор скрытого состояния на каждом временном шаге на основе вектора скрытого состояния с предыдущего шага и входного вектора текущего шага. Именно за счет этого вычисления RNN обладают столь большими возможностями, но оно же приводит к фундаментальным численным проблемам.

Первая проблема с RNN Элмана — трудности хранения долгосрочной информации. Например, в случае RNN из главы 6 на каждом временном шаге мы просто обновляли вектор скрытого состояния независимо от того, имело ли это смысл. В результате RNN никак не контролирует, какие значения сохраняются в скрытом состоянии, а какие отбрасываются — это полностью определяется входными данными. На интуитивном уровне это совершенно бессмысленно. Хотелось бы, чтобы RNN умела определять, требуется ли обновление, и если да, то насколько именно нужно обновить и какие именно части вектора состояния.

Вторая проблема с RNN Элмана — их склонность к выходу градиентов из-под контроля и стремлению в сторону нуля или бесконечности. Такие нестабильные градиенты называются соответственно *исчезающими* или *растущими взрывным образом*, в зависимости от направления уменьшения/роста их абсолютных значений. Действительно большое абсолютное значение градиента или действительно маленькое (меньше 1) может привести к нестабильности процедуры оптимизации (см. статьи Хохрайтера и др. [Hochreiter et al., 2001]; Пашкану и др. [Pascanu et al., 2013]).

Существуют методы решения этих проблем с градиентами в «наивных» RNN, например с помощью выпрямленных линейных блоков (rectified linear units, ReLU), отсекающие градиентов (gradient clipping) или просто продуманной инициализации. Но ни один из этих методов не работает столь надежно, как методика под названием «шлюзование» (gating).

**Шлюзование как решение проблем «наивных» RNN.** Чтобы понять на интуитивном уровне, что такое шлюзование, представьте, что вам нужно сложить две величины,  $a$  и  $b$ , но при этом хотелось бы контролировать, какая часть величины  $b$  будет включена в сумму. Математически для этого сумму  $a + b$  можно переписать в таком виде:

$$a + \lambda b,$$

где  $\lambda$  представляет собой значение от 0 до 1. Если  $\lambda = 0$ , то  $b$  не вносит никакого вклада в сумму, а если  $\lambda = 1$ , то  $b$  включается в сумму полностью. Таким образом,  $\lambda$  можно рассматривать как своеобразный переключатель или шлюз, определяющий включаемую в сумму долю  $b$ . Именно так работает механизм шлюзования. Вернемся теперь к RNN Элмана и посмотрим, как встроить шлюзование в «наивную» RNN, чтобы иметь возможность выполнять условные обновления. Если предыдущее скрытое состояние обозначить  $h_{t-1}$ , а текущие входные данные —  $x_t$ , то очередное обновление в RNN Элмана будет выглядеть следующим образом:

$$h_t = h_{t-1} + F(h_{t-1}, x_t),$$

где  $F$  — очередное вычисление RNN. Конечно, это безусловное суммирование со всеми присущими ему недостатками, описанными выше. Теперь представьте себе,

что вместо константы  $\lambda$  в предыдущем примере у нас есть функция от предыдущего вектора скрытого состояния  $h_{t-1}$  и текущих входных данных  $x_t$ , тоже производящая шлюзование, то есть ее значение находится в промежутке от 0 до 1. При такой функции шлюзования уравнение обновления RNN будет выглядеть следующим образом:

$$h_t = h_{t-1} + \lambda(h_{t-1}, x_t)F(h_{t-1}, x_t).$$

Теперь становится ясно, что функция  $\lambda$  управляет тем, какая доля текущего входного вектора участвует в обновлении состояния  $h_{t-1}$ . Более того,  $\lambda$  зависит от контекста. Так работают все шлюзовые нейронные сети. Функция  $\lambda$  обычно представляет собой сигма-функцию, которая, как мы знаем из главы 3, возвращает значение от 0 до 1.

В случае сетей с долгой краткосрочной памятью (long short-term memory network, LSTM (Hochreiter, Schmidhuber, 1997)) этот механизм аккуратно расширяется, включая не только условные обновления, но и умышленное «забывание» значений из предыдущего скрытого состояния  $h_{t-1}$ . Подобное «забывание» выполняется путем умножения предыдущего скрытого состояния  $h_{t-1}$  на еще одну функцию,  $\mu$ , также возвращающую значения от 0 до 1 и зависящую от текущих входных данных:

$$h_t = \mu(h_{t-1}, x_t)h_{t-1} + \lambda(h_{t-1}, x_t)F(h_{t-1}, x_t).$$

Как вы могли догадаться,  $\mu$  — тоже функция шлюзования. Настоящее описание LSTM более запутанное из-за параметризации функций шлюзования, что приводит к довольно сложной (для непосвященных) последовательности операций. Но вооружившись полученным в этом разделе представлением о шлюзовании, вы уже готовы подробнее рассмотреть механизмы обновления LSTM. Рекомендуем вам классическую статью Кристофера Олаха (Christopher Olah) (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>). Мы воздержимся от освещения этих вопросов в книге, поскольку такие подробности не важны для применения LSTM в приложениях NLP.

LSTM — лишь один из множества шлюзовых вариантов RNN. Еще один вариант, все более популярный в последние годы, — шлюзовой рекуррентный блок (gated recurrent unit, GRU (Chung et al., 2015)). К счастью, в PyTorch для перехода на LSTM можно просто заменить `nn.RNN` или `nn.RNNCell` на `nn.LSTM` или `nn.LSTMCell` соответственно без каких-либо других модификаций кода (и аналогично для GRU)!

Механизм шлюзования — эффективное решение проблем, перечисленных выше в этом разделе. Он позволяет не только управлять обновлениями, но и контролировать градиенты, да и обучение при этом упрощается. Без дальнейших проволочек мы теперь покажем на двух примерах шлюзовые архитектуры в действии.

## Пример: символьная RNN для генерации фамилий

В этом примере<sup>1</sup> мы рассмотрим простую задачу предсказания последовательностей: генерацию фамилий с помощью RNN. На практике это означает, что для каждого временного шага RNN вычисляет распределение вероятности по множеству возможных символов в фамилии. Эти распределения вероятности можно использовать или для оптимизации сети, чтобы улучшить выдаваемые ею предсказания (при условии, что мы знаем, какие символы должны быть предсказаны), или для генерации совершенно новых фамилий.

Хотя набор данных для этой задачи уже использовался в предыдущих примерах и должен быть вам хорошо знаком, способ формирования выборок данных для предсказания последовательности несколько отличается. После описания набора данных и задачи вкратце обрисует вспомогательные структуры данных, обеспечивающие возможность предсказания последовательностей.

Далее мы опишем две модели генерации фамилий: контекстно не обусловленную (unconditioned) `SurnameGenerationModel` и контекстно обусловленную (conditioned) `SurnameGenerationModel`. Контекстно не обусловленная модель предсказывает последовательности символов фамилий без какой-либо информации о национальной принадлежности. И напротив, контекстно обусловленная модель использует конкретные вложения для национальностей в качестве начального скрытого состояния RNN для смещения (bias) предсказаний последовательностей.

## Класс `SurnameDataset`

Впервые приведенный в разделе «Пример: классификация фамилий с помощью MLP» на с. 109, набор данных фамилий представляет собой список фамилий и стран их происхождения. Пока мы использовали его для задач классификации — правильного определения по новой фамилии страны ее происхождения. В этом же примере мы покажем, как на основе этого набора данных обучить модель распределять вероятности по последовательностям символов и генерировать новые последовательности.

Класс `SurnameDataset` практически не отличается от приведенного в предыдущих главах: мы используем объект `DataFrame` библиотеки `Pandas` для загрузки набора данных и создаем векторизатор для инкапсуляции отображения токенов в целочисленные значения, необходимые для модели и имеющейся задачи. Чтобы отразить различие задач, метод `SurnameDataset.__getitem__()` модифицируется:

<sup>1</sup> Код для него вы можете найти в каталоге `/chapters/chapter_7/7_3_surname_generation` в репозитории GitHub этой книги (<https://nlproc.info/PyTorchNLPBook/repo/>).

он должен возвращать последовательность целочисленных значений для целей предсказания, как показано в примере 7.1. Этот метод обращается к векторизатору для вычисления последовательности целых чисел, играющих роль входных данных (`from_vector`), и последовательности целых чисел, играющих роль выходных данных (`to_vector`). Реализация метода `vectorize()` описана в следующем подразделе.

**Пример 7.1.** Метод `SurnameDataset.__getitem__()` для задачи предсказания последовательностей

```
class SurnameDataset(Dataset):
    @classmethod
    def load_dataset_and_make_vectorizer(cls, surname_csv):
        """ Загружает набор данных и создает с нуля новый векторизатор

        Аргументы:
            surname_csv (str): местоположение набора данных
        Возвращает:
            экземпляр SurnameDataset
        """

        surname_df = pd.read_csv(surname_csv)
        return cls(surname_df, SurnameVectorizer.from_dataframe(surname_df))

    def __getitem__(self, index):
        """ Основной метод — точка входа для наборов данных PyTorch

        Аргументы:
            index (int): индекс точки данных
        Возвращает:
            словарь, содержащий точку данных: (x_data, y_target, class_index)
        """
        row = self._target_df.iloc[index]

        from_vector, to_vector = \
            self._vectorizer.vectorize(row.surname, self._max_seq_length)

        nationality_index = \
            self._vectorizer.nationality_vocab.lookup_token(row.nationality)

        return {'x_data': from_vector,
                'y_target': to_vector,
                'class_index': nationality_index}
```

## Структуры данных для векторизации

Как и в предыдущих примерах, для преобразования последовательностей символов фамилий в векторизованную форму используются три основные структуры данных: `SequenceVocabulary` — для отображения токенов в целые числа, `SurnameVectorizer` — для общего согласования этих отображений и `DataLoader` — для группировки

результатов `SurnameVectorizer` в мини-пакеты. Поскольку в этом примере реализация объекта `DataLoader` и его использование не отличаются от предыдущих, мы опустим подробности<sup>1</sup>.

## Класс `SurnameVectorizer` и токен `END-OF-SEQUENCE`

Процедура обучения для задачи предсказания последовательностей ожидает на входе две последовательности целочисленных значений, представляющие токены-наблюдения и целевые токены на каждом из временных шагов. Обычно необходимо предсказать ту же самую последовательность, на которой производится обучение, в нашем случае — фамилии. Это значит, что работать придется только с одной последовательностью токенов и с одним предложением для формирования наблюдений и целей.

Чтобы сделать из этого задачу предсказания последовательностей, всем токенам с помощью `SequenceVocabulary` ставятся в соответствие индексы. Далее в начало последовательности добавляется индекс токена `BEGIN-OF-SEQUENCE`, `begin_seq_index`, а в ее конец — индекс токена `END-OF-SEQUENCE`, `end_seq_index`. На этом этапе каждая точка данных представляет собой последовательность индексов, причем первые и последние индексы одинаковы. Для создания необходимых для процедуры обучения входной и выходной последовательностей мы просто воспользуемся двумя срезами последовательности индексов: первый будет включать все индексы токенов, за исключением последнего, а второй — все индексы токенов, за исключением первого. После выравнивания и попарного сочетания эти последовательности станут подходящими индексами для входа-выхода.

Для большей ясности приведем код для метода `SurnameVectorizer.vectorize()` в примере 7.2. Первый этап — отображение строки символов `surname` в список `indices` соответствующих им целочисленных значений. Далее к `indices` добавляются индексы начала и конца последовательности: а именно, в начало `indices` добавляется `begin_seq_index`, а в конец — `end_seq_index`. Потом проверяем значение `vector_length`, которое обычно передается во время выполнения (хотя код написан так, что допускает любую длину вектора). Во время обучения наличие `vector_length` важно, поскольку мини-пакеты формируются на основе многоярусных векторных представлений. А если длины векторов различны, то уложить их в одну матрицу не получится. После проверки `vector_length` создаются два вектора: `from_vector` и `to_vector`. Срез индексов, не содержащий последнего индекса, помещается в `from_vector`, а срез индексов, не содержащий первого, — в `to_vector`. Оставшиеся позиции векторов заполняются `mask_index`. Важно, чтобы после-

---

<sup>1</sup> См. более подробную информацию о `SequenceVocabulary` в подразделе «Классы `Vocabulary`, `Vectorizer` и `DataLoader`» на с. 132 и вводное описание структур данных `Vocabulary` и `Vectorizer` в подразделе «Классы `Vocabulary`, `Vectorizer` и `DataLoader`» на с. 82.

довательности заполнялись (дополнялись) справа, поскольку пустые позиции меняют выходной вектор и хотелось бы, чтобы это произошло после просмотра последовательности.

**Пример 7.2.** Код метода `SurnameVectorizer.vectorize()` для задачи предсказания последовательности

```
class SurnameVectorizer(object):
    """ Векторизатор, приводящий словари в соответствие друг другу
        и использующий их"""
    def vectorize(self, surname, vector_length=-1):
        """ Векторизует фамилию в вектор наблюдений и целей

        Аргументы:
            surname (str): векторизуемая фамилия
            vector_length (int): аргумент, жестко задающий длину вектора индексов
        Возвращает:
            кортеж: (from_vector, to_vector)
                from_vector (numpy.ndarray): вектор наблюдений
                to_vector (numpy.ndarray): вектор целевых предсказаний
        """
        indices = [self.char_vocab.begin_seq_index]
        indices.extend(self.char_vocab.lookup_token(token)
                       for token in surname)
        indices.append(self.char_vocab.end_seq_index)

        if vector_length < 0:
            vector_length = len(indices) - 1

        from_vector = np.zeros(vector_length, dtype=np.int64)
        from_indices = indices[:-1]
        from_vector[:len(from_indices)] = from_indices
        from_vector[len(from_indices):] = self.char_vocab.mask_index

        to_vector = np.empty(vector_length, dtype=np.int64)
        to_indices = indices[1:]
        to_vector[:len(to_indices)] = to_indices
        to_vector[len(to_indices):] = self.char_vocab.mask_index

        return from_vector, to_vector

    @classmethod
    def from_dataframe(cls, surname_df):
        """ Создает экземпляр векторизатора на основе
            объекта DataFrame набора данных

        Аргументы:
            surname_df (pandas.DataFrame): набор данных фамилий
        Возвращает:
            экземпляр SurnameVectorizer
        """
        char_vocab = SequenceVocabulary()
```

```
nationality_vocab = Vocabulary()

for index, row in surname_df.iterrows():
    for char in row.surname:
        char_vocab.add_token(char)
        nationality_vocab.add_token(row.nationality)

return cls(char_vocab, nationality_vocab)
```

## От RNN Элмана к GRU

На практике переход от «наивной» RNN к шлюзовому варианту очень прост. В следующих моделях можно с легкостью использовать и LSTM, хотя мы применяем GRU вместо «наивной» RNN. Для использования GRU мы создаем экземпляр класса `torch.nn.GRU` с теми же параметрами, что и для `ElmanRNN` из главы 6.

## Модель 1. Контекстно не обусловленная модель SurnameGenerationModel

Первая из двух наших моделей будет контекстно не обусловленной: перед генерацией фамилии она не наблюдает никаких национальностей. На практике необусловленность контекстом означает, что выполняемые GRU вычисления не смещены в сторону какой-либо национальности. В примере 7.4 это смещение создается искусственно с помощью исходного скрытого вектора. В этом же примере исходный вектор скрытого состояния состоит из всех нулей и не вносит никакого вклада в вычисления<sup>1</sup>.

В целом `SurnameGenerationModel` (пример 7.3) включает вложение индексов символов, вычисление их последовательного состояния с помощью GRU и вычисление вероятности предсказаний токенов с помощью линейного слоя. Если подробнее, контекстно не обусловленная модель `SurnameGenerationModel` начинается с инициализации слоя вложений, GRU и линейного слоя. Подобно моделям последовательностей из главы 6, входными данными для модели служит матрица целых чисел. Для преобразования этих целых чисел в трехмерный тензор (последовательность векторов для каждого элемента пакета) используется экземпляр `Embedding` фреймворка PyTorch, `char_emb`. Этот тензор передается в GRU, вычисляющий векторы состояния для всех позиций во всех последовательностях.

**Пример 7.3.** Контекстно не обусловленная модель для генерации фамилий

```
class SurnameGenerationModel(nn.Module):
    def __init__(self, char_embedding_size, char_vocab_size,
```

---

<sup>1</sup> Матричное умножение на состоящий из нулей вектор скрытого состояния даст в результате матрицу из всех нулей.



```

        rnn_hidden_size, batch_first=True, padding_idx=0,
        dropout_p=0.5):
    """
    Аргументы:
        char_embedding_size (int): размер вложений символов
        char_vocab_size (int): количество символов для создания вложений
        rnn_hidden_size (int): размер скрытого состояния RNN
        batch_first (bool): указывает, будут ли в нулевом измерении
        входных тензоров располагаться данные пакета или последовательности
        padding_idx (int): индекс для дополнения тензора;
            см. torch.nn.Embedding
        dropout_p (float): вероятность обнуления при использовании
        метода дропаута
    """
    super(SurnameGenerationModel, self).__init__()

    self.char_emb = nn.Embedding(num_embeddings=char_vocab_size,
                                embedding_dim=char_embedding_size,
                                padding_idx=padding_idx)
    self.rnn = nn.GRU(input_size=char_embedding_size,
                      hidden_size=rnn_hidden_size,
                      batch_first=batch_first)
    self.fc = nn.Linear(in_features=rnn_hidden_size,
                        out_features=char_vocab_size)
    self._dropout_p = dropout_p

def forward(self, x_in, apply_softmax=False):
    """ Прямой проход модели

    Аргументы:
        x_in (torch.Tensor): тензор входных данных
            Значение x_in.shape должно быть (batch, input_dim)
        apply_softmax (bool): флаг для многомерной логистической функции
        активации во время обучения должен равняться 0
    Возвращает:
        итоговый тензор. Значение tensor.shape должно быть (batch, output_dim)
    """
    x_embedded = self.char_emb(x_in)
    y_out, _ = self.rnn(x_embedded)
    batch_size, seq_size, feat_size = y_out.shape
    y_out = y_out.contiguous().view(batch_size * seq_size, feat_size)
    y_out = self.fc(F.dropout(y_out, p=self._dropout_p))

    if apply_softmax:
        y_out = F.softmax(y_out, dim=1)

    new_feat_size = y_out.shape[-1]
    y_out = y_out.view(batch_size, seq_size, new_feat_size)

    return y_out

```

Основное различие между задачами классификации последовательностей из главы 6 и задачами предсказания последовательностей из этой главы — обработка

вычисляемых RNN векторов состояния. В главе 6 мы извлекали по вектору для каждого индекса пакета и выполняли предсказания на их основе. В данном примере мы сменим форму нашего тензора с трехмерного на двумерный (то есть матрицу), измерение строк в котором будет соответствовать выборкам (пакетам и индексам последовательности). С помощью этой матрицы и линейного слоя вычислим векторы предсказаний для каждой из выборок. В завершение вычислений мы снова преобразуем матрицу обратно в трехмерный тензор. Поскольку при операциях смены формы упорядоченность сохраняется, то пакеты и индексы последовательностей останутся на своих местах. Изменение формы необходимо потому, что слою `Linear` на входе требуется матрица.

## Модель 2. Контекстно обусловленная модель SurnameGenerationModel

Наша вторая модель учитывает национальную принадлежность генерируемой фамилии. На деле это значит существование механизма для смещения поведения модели относительно конкретной фамилии. В этом примере мы параметризуем начальное скрытое состояние RNN, вкладывая каждую из национальностей в виде вектора размера скрытого состояния. Это значит, что при подстройке значений параметров модель также подстраивает значения в матрице вложений так, чтобы сделать предсказания более чувствительными к какой-либо конкретной национальности и закономерностям соответствующих фамилий. Например, вектор для ирландской национальности смещен в сторону последовательностей, начинающихся с `Mc` и `O`.

Пример 7.4 демонстрирует отличия контекстно обусловленной модели. А именно, в ней появляется дополнительный слой вложений для отображения индексов национальностей в векторы, размер которых совпадает со скрытым слоем RNN. Далее в функции `forward` выполняется вложение индексов национальностей, и они просто передаются в качестве начального скрытого слоя RNN. Хотя изменения, по сравнению с первой моделью, очень просты, они сильно влияют на возможность изменения поведения модели в зависимости от национальности, соответствующей генерируемой фамилии.

**Пример 7.4.** Контекстно обусловленная модель для генерации фамилий

```
class SurnameGenerationModel(nn.Module):
    def __init__(self, char_embedding_size, char_vocab_size,
                  num_nationalities, rnn_hidden_size, batch_first=True,
                  padding_idx=0, dropout_p=0.5):
        # ...
        self.nation_embedding = nn.Embedding(embedding_dim=rnn_hidden_size,
                                              num_embeddings=num_nationalities)

    def forward(self, x_in, nationality_index, apply_softmax=False):
        # ...
```

```

x_embedded = self.char_embedding(x_in)
# hidden_size: (num_layers * num_directions, batch_size, rnn_hidden_size)
nationality_embedded = self.nation_emb(nationality_index).unsqueeze(0)
y_out, _ = self.rnn(x_embedded, nationality_embedded)
# ...

```

## Процедура обучения и результаты

В этом примере представлена задача предсказания последовательностей символов для генерации фамилий. Хотя во многих отношениях детали реализации и процедуры обучения напоминают примеры классификации последовательностей из главы 6, есть и несколько отличий. В данном разделе мы сосредоточим наше внимание на этих отличиях, используемых гиперпараметрах и результатах.

Для вычисления функции потерь необходимо внести два изменения, по сравнению с предыдущими примерами, поскольку предсказания выполняются на каждом временном шаге последовательности. Во-первых, нужно преобразовать трехмерные тензоры<sup>1</sup> в двумерные (матрицы), чтобы удовлетворить вычислительные ограничения. Во-вторых, нужно согласовать `mask_index`, благодаря которому возможна работа с последовательностями переменной длины, с функцией потерь, чтобы она не учитывала маскированные места в расчетах.

Для решения обеих этих проблем — с трехмерными тензорами и последовательностями переменной длины — воспользуемся кодом из примера 7.5. Во-первых, выполним нормализацию предсказаний и целей до ожидаемых функций потерь размеров (двумерные для предсказаний и одномерные для целей). Теперь каждая строка соответствует отдельной выборке: одному временному шагу в одной последовательности. Далее вычисляется функция потерь на основе перекрестной энтропии (при значении параметра `ignore_index`, равном `mask_index`). В результате этого функция потерь будет игнорировать все позиции в целевых векторах, соответствующие `ignore_index`.

**Пример 7.5.** Преобразование трехмерных тензоров и вычисление потерь в масштабах последовательности

```

def normalize_sizes(y_pred, y_true):
    """ Нормализация размеров тензоров

    Аргументы:
        y_pred (torch.Tensor): выходные данные модели
            Если они представляют собой трехмерный тензор, преобразуем
            его в матрицу
        y_true (torch.Tensor): целевые предсказания
            Если они представляют собой матрицу, преобразуем ее в вектор
    """

```

<sup>1</sup> В трехмерных тензорах информация о пакете располагается в первом измерении, последовательность находится во втором, а векторы предсказаний — в третьем.

```

if len(y_pred.size()) == 3:
    y_pred = y_pred.contiguous().view(-1, y_pred.size(2))
if len(y_true.size()) == 2:
    y_true = y_true.contiguous().view(-1)
return y_pred, y_true

def sequence_loss(y_pred, y_true, mask_index):
    y_pred, y_true = normalize_sizes(y_pred, y_true)
    return F.cross_entropy(y_pred, y_true, ignore_index=mask_index)

```

На основе этой модифицированной функции вычисления потерь создадим процедуру обучения, аналогичную той, что приводилась во всех прочих примерах книги. Она начинается с прохода в цикле по обучающему набору данных, по одному мини-пакету за раз. Выходные данные модели вычисляются на основе входных для каждого из мини-пакетов. А поскольку на каждом временном шаге выполняются предсказания, выходные данные модели представляют собой трехмерный тензор. С помощью вышеописанного метода `sequence_loss` и оптимизатора вычисляется сигнал рассогласования для предсказаний модели, на основе которого обновляются ее параметры.

Большая часть гиперпараметров модели определяется размером символического словаря. Он равен количеству дискретных токенов, потенциально наблюдаемых на входе модели, и количеству классов в выходной классификации на каждом временном шаге. Кроме этого, к гиперпараметрам модели относятся размер вложений символов и размер внутреннего скрытого состояния RNN. Эти гиперпараметры и параметры обучения приведены в примере 7.6.

#### Пример 7.6. Гиперпараметры для генерации фамилий

```

args = Namespace(
    # Информация о данных и путях
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch7/model1_unconditioned_surname_generation",
    # или: save_dir="model_storage/ch7/model2_conditioned_surname_generation",
    # Гиперпараметры модели
    char_embedding_size=32,
    rnn_hidden_size=32,
    # Гиперпараметры обучения
    seed=1337,
    learning_rate=0.001,
    batch_size=128,
    num_epochs=100,
    early_stopping_criteria=5,
    # Настройки времени выполнения не приводятся для экономии места
)

```

И хотя мера эффективности модели — точность предсказаний в пересчете на символ, в данном примере лучше будет выполнить качественную оценку, внимательно просмотрев генерируемые моделью фамилии. Напишем код для прохода в новом

цикле по модифицированной версии шагов из метода `forward()` и вычисления предсказаний на каждом из временных шагов, которые затем послужат входными данными для следующего временного шага. Соответствующий код показан в примере 7.7. На выходе модели на каждом временном шаге получается вектор предсказаний, который преобразуется в распределение вероятности с помощью многомерной логистической функции. С этим распределением вероятности мы можем воспользоваться функцией выборки `torch.multinomial()`, выбирающей индексы с пропорциональной их вероятности частотой. Выборка представляет собой стохастическую процедуру, результаты которой каждый раз новые.

**Пример 7.7.** Выборка из контекстно не обусловленной модели генерации

```
def sample_from_model(model, vectorizer, num_samples=1, sample_size=20,
                    temperature=1.0):
    """ Выборка последовательности индексов из модели

    Аргументы:
        model (SurnameGenerationModel): обученная модель
        vectorizer (SurnameVectorizer): соответствующий векторизатор
        num_samples (int): число выборок
        sample_size (int): максимальная длина выборки
        temperature (float): подчеркивает или сворачивает распределение
            При 0.0 < temperature < 1.0 максимумы заостряются
            temperature > 1.0 делает распределение более равномерным
    Возвращает:
        indices (torch.Tensor): матрица индексов
        форма: (num_samples, sample_size)
    """
    begin_seq_index = [vectorizer.char_vocab.begin_seq_index
                       for _ in range(num_samples)]
    begin_seq_index = torch.tensor(begin_seq_index,
                                   dtype=torch.int64).unsqueeze(dim=1)
    indices = [begin_seq_index]
    h_t = None

    for time_step in range(sample_size):
        x_t = indices[time_step]
        x_emb_t = model.char_emb(x_t)
        rnn_out_t, h_t = model.rnn(x_emb_t, h_t)
        prediction_vector = model.fc(rnn_out_t.squeeze(dim=1))
        probability_vector = F.softmax(prediction_vector / temperature, dim=1)
        indices.append(torch.multinomial(probability_vector, num_samples=1))
    indices = torch.stack(indices).squeeze().permute(1, 0)
    return indices
```

Необходимо преобразовать полученные из функции `sample_from_model` выборки индексов в строковое значение, удобное для чтения человеком. Как демонстрирует пример 7.8, мы воспользуемся для этого `SequenceVocabulary`, который служил для векторизации фамилий. При создании строкового значения станем использовать только индексы, предшествующие индексу `END-OF-SEQUENCE`. А значит, модель теперь будет «понимать», где должны заканчиваться фамилии.

**Пример 7.8.** Отображение выборок индексов в строковые значения для фамилий

```
def decode_samples(sampled_indices, vectorizer):
    """ Преобразование индексов в строковое представление фамилий

    Аргументы:
        sampled_indices (torch.Tensor): индексы из функции `sample_from_model`
        vectorizer (SurnameVectorizer): соответствующий векторизатор
    """
    decoded_surnames = []
    vocab = vectorizer.char_vocab

    for sample_index in range(sampled_indices.shape[0]):
        surname = ""
        for time_step in range(sampled_indices.shape[1]):
            sample_item = sampled_indices[sample_index, time_step].item()
            if sample_item == vocab.begin_seq_index:
                continue
            elif sample_item == vocab.end_seq_index:
                break
            else:
                surname += vocab.lookup_index(sample_item)
        decoded_surnames.append(surname)
    return decoded_surnames
```

С помощью этих функций можно посмотреть результаты работы модели, показанные в примере 7.9, и составить представление о том, научилась ли она генерировать адекватные фамилии. Какие выводы можно сделать из просмотра этих выходных данных? Хотя фамилии вроде бы следуют нескольким морфологическим паттернам, не похоже, чтобы каждая из них соответствовала одной национальности. Возможно, наша общая модель фамилий при обучении перепутала распределения символов между различными национальностями. Контекстно обусловленная модель `SurnameGenerationModel` призвана решить эту проблему.

**Пример 7.9.** Выборка из контекстно не обусловленной модели

Input[0]

```
samples = sample_from_model(unconditioned_model, vectorizer,
                             num_samples=10)
decode_samples(samples, vectorizer)
```

Output[0]

```
['Aqtaliby',
 'Yomaghev',
 'Mauasheev',
 'Unander',
 'Virrovo',
 'NInev',
 'Bukhumohe',
 'Burken',
 'Rati',
 'Jzirmar']
```

Для контекстно обусловленной `SurnameGenerationModel` мы модифицируем функцию `sample_from_model()` так, чтобы она принимала на входе список индексов национальностей вместо количества выборок. В примере 7.10 эта модифицированная функция на основе индексов и вложений национальностей формирует начальное скрытое состояние GRU. Дальнейшая процедура выборки выглядит точно так же, как и в контекстно не обусловленной модели.

**Пример 7.10.** Выборка из модели последовательности

```
def sample_from_model(model, vectorizer, nationalities, sample_size=20,
                      temperature=1.0):
    """ Выборка последовательности индексов из модели

    Аргументы:
        model (SurnameGenerationModel): обученная модель
        vectorizer (SurnameVectorizer): соответствующий векторизатор
        nationalities (list): список соответствующих национальностям целых чисел
        sample_size (int): максимальная длина выборки
        temperature (float): подчеркивает или сворачивает распределение
            При 0.0 < temperature < 1.0 максимумы заостряются
            temperature > 1.0 делает распределение более равномерным

    Возвращает:
        indices (torch.Tensor): матрица индексов
        форма: (num_samples, sample_size)
    """
    num_samples = len(nationalities)
    begin_seq_index = [vectorizer.char_vocab.begin_seq_index
                       for _ in range(num_samples)]
    begin_seq_index = torch.tensor(begin_seq_index,
                                   dtype=torch.int64).unsqueeze(dim=1)
    indices = [begin_seq_index]
    nationality_indices = torch.tensor(nationalities,
                                       dtype=torch.int64).unsqueeze(dim=0)
    h_t = model.nation_emb(nationality_indices)

    for time_step in range(sample_size):
        x_t = indices[time_step]
        x_emb_t = model.char_emb(x_t)
        rnn_out_t, h_t = model.rnn(x_emb_t, h_t)
        prediction_vector = model.fc(rnn_out_t.squeeze(dim=1))
        probability_vector = F.softmax(prediction_vector / temperature, dim=1)
        indices.append(torch.multinomial(probability_vector, num_samples=1))
    indices = torch.stack(indices).squeeze().permute(1, 0)
    return indices
```

Выборка с использованием контекстного вектора позволяет влиять на генерируемые выходные данные. В примере 7.11 мы проходим в цикле по индексам национальностей и производим выборку из каждого. Для экономии места мы покажем лишь небольшую часть результатов. Из них видно, что модель действительно улавливает некоторые орфографические особенности фамилий.

**Пример 7.11.** Выборка из контекстно обусловленной модели SurnameGenerationModel (показаны не все результаты)

Input[0]

```
for index in range(len(vectorizer.nationality_vocab)):
    nationality = vectorizer.nationality_vocab.lookup_index(index)

    print("Sampled for {}: ".format(nationality))

    sampled_indices = sample_from_model(model=conditioned_model,
                                       vectorizer=vectorizer,
                                       nationalities=[index] * 3,
                                       temperature=0.7)

    for sampled_surname in decode_samples(sampled_indices,
                                         vectorizer):
        print("- " + sampled_surname)
```

Output[0]

```
Sampled for Arabic:
- Khatso
- Salbwa
- Gadi
Sampled for Chinese:
- Lie
- Puh
- Pian
Sampled for German:
- Lenger
- Schanger
- Schumper
Sampled for Irish:
- Mcochin
- Corran
- O'Baintin
Sampled for Russian:
- Mahghatsunkov
- Juhin
- Karkovin
Sampled for Vietnamese:
- Lo
- Tham
- Tou
```

## Полезные советы по обучению моделей последовательностей

Обучение моделей последовательностей — непростая задача, в процессе которой возникает множество проблем. В этом разделе мы приведем пару советов и опишем несколько приемов, которые пригодились нам в работе и встречались в литературе.



- ❑ *По возможности применяйте шлюзовые варианты архитектуры.* Шлюзовые архитектуры упрощают обучение, решая многие проблемы с численной устойчивостью, присущие нешлюзовым вариантам.
- ❑ *По возможности используйте GRU вместо LSTM.* Эффективность работы GRU близка к LSTM при намного меньшем числе параметров и требуемых ресурсов. К счастью, с точки зрения фреймворка PyTorch, для перехода с LSTM на GRU требуется просто другой класс `Module`.
- ❑ *Используйте Adam в качестве оптимизатора.* В главах 6–8 в качестве оптимизатора мы применяем только Adam, и не без причин: он надежен и обычно сходится быстрее, чем альтернативные оптимизаторы. В наибольшей степени это справедливо для моделей последовательностей. Если по какой-либо причине ваша модель не сходится с помощью Adam, попробуйте воспользоваться стохастическим градиентным спуском.
- ❑ *Обрезка градиентов.* Если вы заметили многочисленные числовые ошибки при реализации описанных в этой главе приемов, добавьте в свой код инструменты для вывода графиков значений градиентов в процессе обучения. Выяснив диапазон этих значений, обрежьте все аномальные значения. Это сделает обучение более плавным. Во фреймворке PyTorch предусмотрена удобная утилита для этой цели — `clip_grad_norm()` (пример 7.12). Мы рекомендуем вам выработать у себя привычку выполнять обрезку градиентов.

**Пример 7.12.** Применение обрезки градиентов в PyTorch

```
# описание модели последовательности
model = ..
# описание функции потерь
loss_function = ..

# цикл обучения
for _ in ...:
    ...
    model.zero_grad()
    output, hidden = model(data, hidden)
    loss = loss_function(output, targets)
    loss.backward()
    torch.nn.utils.clip_grad_norm(model.parameters(), 0.25)
    ...
```

- ❑ *Ранняя остановка.* В случае моделей последовательностей нередко происходит переобучение. Мы рекомендуем вам останавливать процедуру обучения пораньше, как только начинает расти погрешность оценки, определенная на проверочном наборе данных.

В главе 8 мы продолжим обсуждение моделей последовательностей и обсудим предсказание и генерацию последовательностей с длинами, отличающимися от длин входных последовательностей, с применением моделей «последовательность в последовательность» и других вариантов.

## Библиография

1. *Hochreiter S., Schmidhuber J.* Long Short-Term Memory // *Neural Computation*, 15. 1997.
2. *Hochreiter S. et al.* Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies // *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.
3. *Pascanu R., Mikolov T., Bengio Y.* On the Difficulty of Training Recurrent Neural Networks // *Proceedings of the 30th International Conference on Machine Learning*. 2013.
4. *Chung J. et al.* Gated Feedback Recurrent Neural Networks // *Proceedings of the 32nd International Conference on Machine Learning*. 2015.

# 8

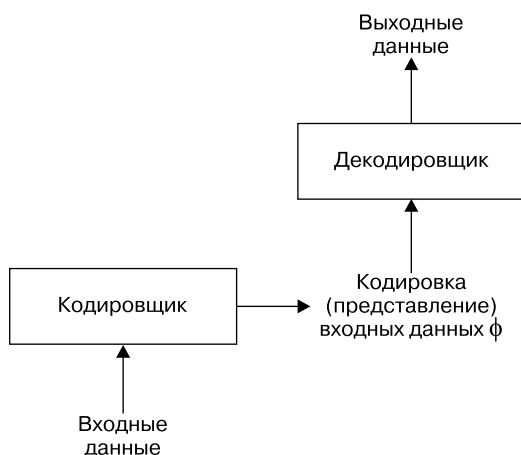
## Продвинутое моделирование последовательностей для обработки текстов на естественных языках

В этой главе мы продолжим обсуждать концепции моделирования последовательностей, о которых говорили в главах 6 и 7, и еще подробнее рассмотрим *моделирование преобразования последовательностей в последовательности* (sequence-to-sequence modeling), когда модель получает на входе последовательность и возвращает другую последовательность, возможно, другой длины. Примеры задач преобразования последовательностей в последовательности встречаются повсеместно. Например, по сообщению электронной почты предсказать ответ, предсказать английский перевод французской фразы или по статье составить ее краткое изложение. Мы также обсудим отличающиеся структурно варианты моделей последовательностей, в частности двунаправленные модели. Чтобы добиться от представления последовательностей максимальной эффективности, познакомим вас с механизмом внимания и подробнее остановимся на этом вопросе. Наконец, эта глава завершается подробным описанием машинного перевода с помощью нейронных сетей (neural machine translation, NMT) как реализации описанных здесь концепций.

### Модели преобразования последовательностей в последовательности, модели типа «кодировщик-декодировщик» и контекстно обусловленная генерация

Модели преобразования последовательностей в последовательности (S2S) — частный случай общего семейства моделей типа «кодировщик-декодировщик». Модель типа «кодировщик-декодировщик» представляет собой композицию двух моделей (рис. 8.1), кодировщика и декодировщика, обычно обучаемых совместно.

Модель кодировщика выдает на выходе кодированное представление ( $\phi$ )<sup>1</sup> входных данных — обычно вектор. Задача кодировщика — захватить важные для текущей задачи свойства входных данных. Задача декодировщика — на основе кодированных входных данных сформировать желаемые выходные. Исходя из этого, мы дадим определение моделей S2S как моделей типа «кодировщик-декодировщик», в которых кодировщик и декодировщик представляют собой модели последовательностей, а входные и выходные данные — последовательности, возможно, различной длины.



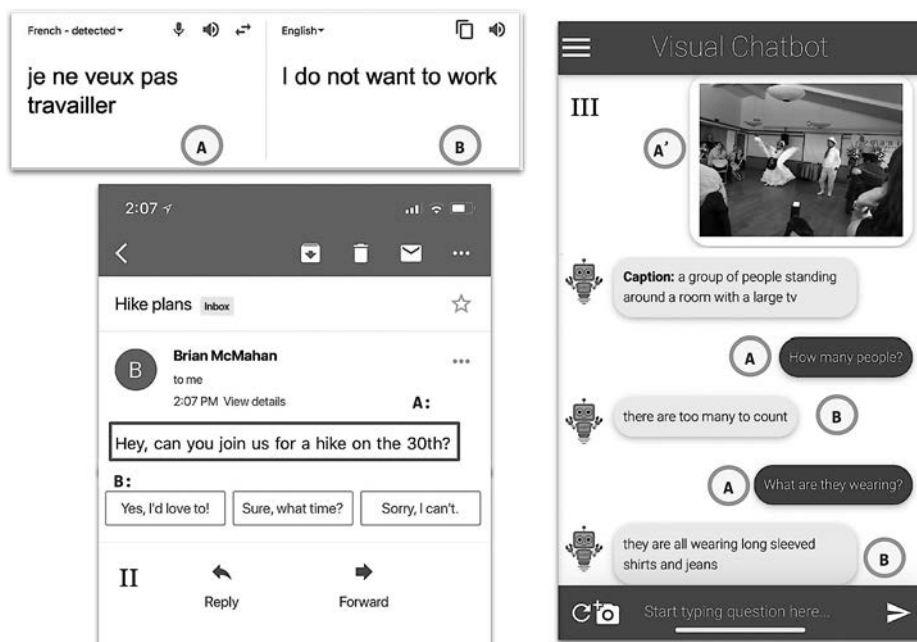
**Рис. 8.1.** Модель типа «кодировщик-декодировщик» — композиция двух совместно обучаемых моделей. Кодировщик генерирует представление (кодировку) входных данных,  $\phi$ , на основе которого декодировщик формирует выходные данные

Модели типа «кодировщик-декодировщик» можно рассматривать в том числе как частный случай так называемых *моделей контекстно обусловленной генерации* (conditioned generation models). При контекстно обусловленной генерации на возвращаемый декодировщиком результат влияет не входное представление  $\phi$ , а общий контекст  $c$ . При поступлении этого общего обуславливающего контекста  $c$  от модели кодировщика контекстно обусловленная генерация происходит так же, как и в моделях типа «кодировщик-декодировщик». Но не все модели контекстно обусловленной генерации представляют собой модели типа «кодировщик-декодировщик», поскольку обуславливающий контекст может извлекаться из какого-либо структурированного источника.

Рассмотрим пример генератора сводок погоды. Показатели температуры, влажности, а также скорости и направления ветра могут «обуславливать» генерируемую декодировщиком текстовую сводку погоды. В подразделе «Модель 2. Контекстно обусловленная модель SurnameGenerationModel» на с. 194 вы видели пример

<sup>1</sup> Символ  $\phi$  в этой главе будет использоваться для кодировок.

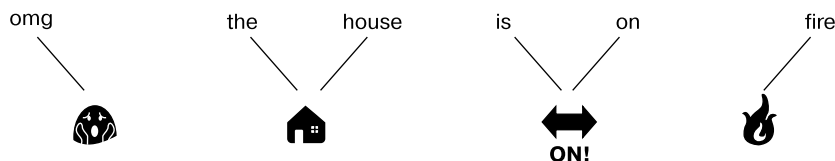
генерации фамилий, контекстно обусловленных национальностями. На рис. 8.2 приведены несколько реальных примеров моделей контекстно обусловленной генерации.



**Рис. 8.2.** Примеры задач, решаемых моделями типа «кодировщик-декодировщик», включают машинный перевод (*слева вверху*: входные данные A — фраза на французском языке, выходные данные B — фраза на английском языке) и предложения вариантов ответов на сообщения электронной почты (*слева внизу*: входные данные A — текст сообщения электронной почты, выходные данные B — один из множества возможных ответов на него). Пример справа несколько сложнее: в нем чат-бот отвечает на вопросы, помеченные буквой A, относительно входного изображения (A'), причем генерируемый ответ (B) обуславливается контекстом кодирований A и A'. Все эти задачи можно рассматривать как задачи контекстно обусловленной генерации

В этой главе мы подробно изучим модели S2S и проиллюстрируем их применение для задач машинного перевода. Представьте себе «интеллектуальную» клавиатуру для iOS/Android, автоматически преобразующую текст в эмодзи во время набора. Один токен на входе может превратиться в несколько токенов на выходе или ни в один. Например, при наборе фразы *omg the house is on fire*<sup>1</sup> клавиатура должна вывести что-то вроде 🏠🔥👉👉👉👉. Обратите внимание, что длина выходных данных (четыре токена) отличается от длины входных (шесть токенов). Отображение входных данных в выходные, показанное на рис. 8.3, называется *выравниванием* (alignment).

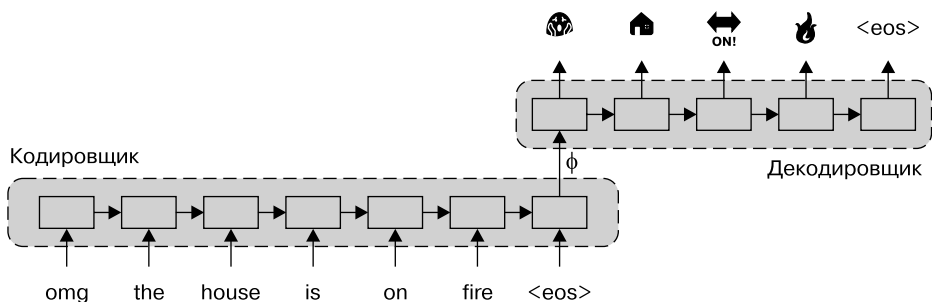
<sup>1</sup> О боже, дом в огне! — *Примеч. пер.*



**Рис. 8.3.** Преобразование в эмодзи как S2S-задача предсказания: выравнивание токенов в двух фразах указывает на эквивалентность перевода

Традиционно для решения S2S-задач применялись статистические подходы со сложной архитектурой и эвристическими алгоритмами. Хотя обзор этих подходов выходит за рамки данной главы, мы рекомендуем вам прочитать статью Кёна (Koehn, 2009) и взглянуть на перечисленные на сайте [statmt.org](http://statmt.org) ресурсы. Из главы 6 мы узнали, что модель последовательности способна кодировать последовательность произвольной длины в вектор. А в главе 7 видели, что одного вектора достаточно, чтобы рекуррентная нейронная сеть генерировала различные фамилии в зависимости от контекста. Естественным расширением этих концепций являются S2S-модели.

На рис. 8.4 показано, как кодировщик «кодирует» все входные данные в представление  $\phi$ , обуславливающее выдачу декодировщиком правильных выходных данных. В качестве кодировщика можно использовать любую RNN, будь то RNN Элмана, LSTM или GRU. В следующих двух разделах мы покажем два важнейших компонента современных S2S-моделей. Во-первых, рассмотрим двунаправленную рекуррентную модель, в которой для создания лучших представлений объединяются прямой и обратный проходы по последовательности. Во-вторых, в разделе «Захватываем больше информации из последовательности: внимание» на с. 209 мы представим и обсудим механизм внимания, с помощью которого удобно фокусироваться на различных частях входных данных, в зависимости от задачи. Обе темы очень важны для создания сложных решений на основе S2S-моделей.



**Рис. 8.4.** S2S-модель для преобразования английского текста в эмодзи

## Захватываем больше информации из последовательности: двунаправленные рекуррентные модели

Рекуррентную модель для простоты можно рассматривать как черный ящик, кодирующий последовательность в вектор. При моделировании последовательности полезно наблюдать не только предыдущие слова, но и те слова, которые появятся в будущем<sup>1</sup>. Рассмотрим следующее предложение<sup>2</sup>:

The man who hunts ducks out on the weekends.

Если модель читает это предложение строго слева направо, то ее представление слова *ducks*<sup>3</sup> будет не таким, как у модели, читающей слова так же и еще справа налево. Люди постоянно подсознательно корректируют смысл подобным ретроспективным образом.

Информации из прошлого и будущего вместе достаточно для полноценного представления смысла слова в предложении. В этом и состоит цель двунаправленных рекуррентных моделей. В двунаправленной форме могут применяться любые модели из рекуррентного семейства — RNN Элмана, LSTM или GRU. Двунаправленные модели, как и их однонаправленные аналоги, рассмотренные в главах 6 и 7, можно использовать как для классификации, так и для маркировки последовательностей, при которой необходимо предсказать по одной метке для каждого слова из входных данных.

Рисунки 8.5 и 8.6 подробно иллюстрируют вышеописанное.

Обратите внимание на рис. 8.5, как модель «читает» предложение в двух направлениях и генерирует представление предложения  $\phi$ , являющееся композицией прямого и обратного представлений. На этом рисунке не показан завершающий слой классификации, состоящий из линейного слоя и многомерной логистической функции.

$\phi_{love}$  на рис. 8.6 — представление (кодировка) скрытого состояния сети на временном шаге, когда входные данные представляют собой слово *love*. Эта информация

<sup>1</sup> В потоковых приложениях это невозможно, но NLP в любом случае часто применяется в пакетном (не потоковом) контексте.

<sup>2</sup> Предложения, подобные приведенному в этом примере, называются предложениями «с подвохом» (garden-path sentence) ([https://en.wikipedia.org/wiki/Garden\\_path\\_sentence](https://en.wikipedia.org/wiki/Garden_path_sentence)) и встречаются чаще, чем можно подумать; например, в заголовках газет подобные конструкции — обычное дело.

<sup>3</sup> У английского слова *duck* есть как минимум два значения: «утка» (существительное) и «уклоняться» (глагол).

о состоянии сыграет важную роль в разделе «Захватываем больше информации из последовательности: внимание» на с. 209, когда мы будем обсуждать механизм внимания.

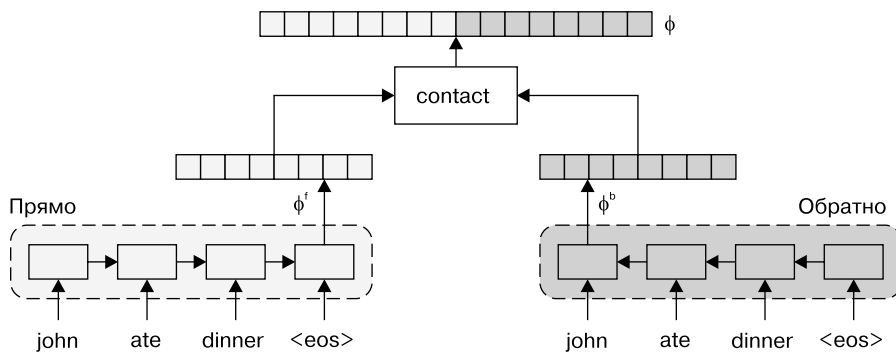


Рис. 8.5. Двухнаправленная модель RNN для классификации последовательностей

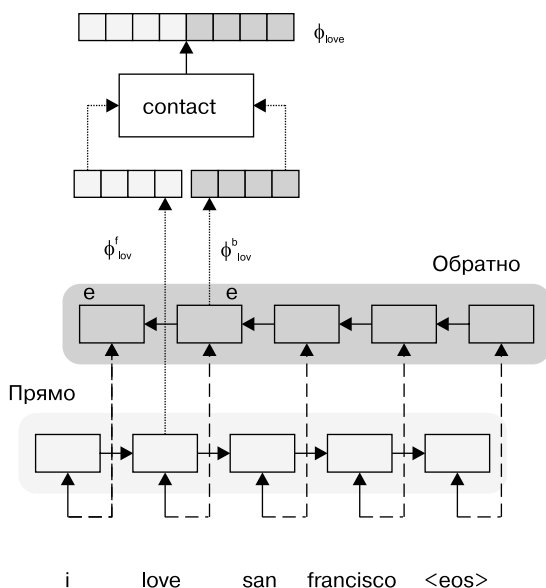


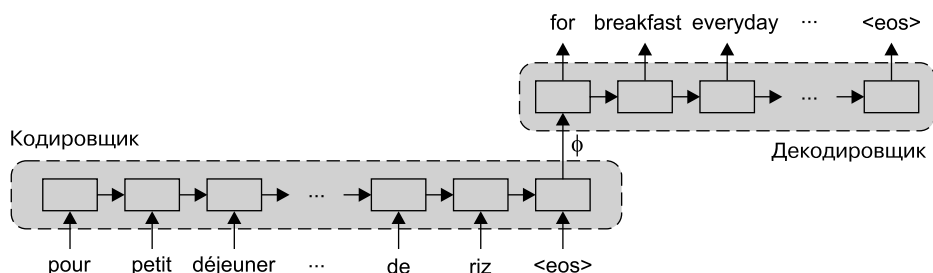
Рис. 8.6. Двухнаправленная рекуррентная модель для маркировки последовательностей

Обратите внимание, что для каждого из слов во входных данных существуют «прямое» и «обратное» представления, в результате объединения которых и получается итоговое представление интересующего нас слова. На рис. 8.6 не показан завершающий слой классификации, на каждом временном шаге состоящий из линейного слоя и многомерной логистической функции.



## Захватываем больше информации из последовательности: внимание

Одна из проблем с формой S2S-модели, описанной в начале главы на с. №№, состоит в том, что она «втискивает» (кодирует) целое входное предложение в один вектор,  $\phi$ , на основе которого и генерирует выходные данные (рис. 8.7). Хотя для очень коротких предложений это может сработать, в случае длинных подобные модели оказываются неспособны уловить информацию из всех входных данных<sup>1</sup>. Такие ограничения накладывает использование в качестве кодировки одного лишь итогового скрытого состояния. Еще одна проблема длинных элементов входных данных — «исчезновение» градиентов по мере обратного распространения ошибки во времени, что затрудняет обучение.



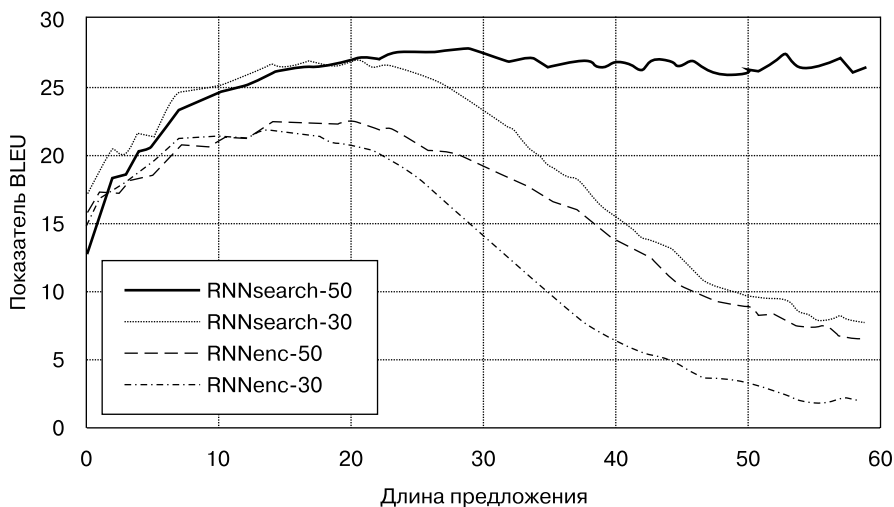
**Рис. 8.7.** Перевод длинного предложения на французском языке на английский с помощью модели типа «кодировщик-декодировщик». Итоговое представление  $\phi$  оказывается неспособно захватить «далекие» зависимости во входных данных, что затрудняет обучение

Такой процесс кодирования и последующего декодирования может показаться странным двуязычным/многоязычным читателям, которым приходилось когда-либо заниматься переводами. Живым людям не свойственно докапываться до смысла предложения, генерируя перевод на основе этого смысла. Например, на рис. 8.7, видя французское слово *pour*, мы понимаем, что в английском переводе будет *for*, аналогично при виде *petit-déjeuner* у нас в уме всплывает слово *breakfast* и т. д. Другими словами, при генерации выходных данных наш мозг концентрируется на соответствующих частях входных. Этот феномен носит название *внимания* (attention). Внимание широко изучалось в нейронауках и смежных дисциплинах, именно оно обеспечивает наши успехи, несмотря на ограниченность нашей памяти. Внимание встречается повсеместно. На самом деле это происходит прямо сейчас с вами, дорогой читатель. Вы. Уделяете. Внимание. Каждому. Слову. Которое. Сейчас. Читаете. Даже если у вас феноменальная память, вряд ли вы читаете эту книгу строго последовательно. Читая слово, вы обращаете внимание на соседние слова, а возможно, и на название раздела и главы и т. д.

<sup>1</sup> См., например, статьи Бенжио и др. (Bengio et al., 1994) или Ли и Зейдема (Le, Zuidema, 2016).

Хотелось бы, чтобы наши модели генерации последовательностей точно так же уделяли внимание различным частям входных данных, а не только общим итогам входных данных в целом. Это называется *механизмом внимания* (attention mechanism). Кстати, первыми моделями, в которых он использовался для обработки написанных на естественных языках текстов, были как раз модели машинного перевода Богданова (Bahdanau et al., 2015). С тех пор было предложено несколько типов механизмов внимания и несколько подходов к повышению внимания.

В этом разделе мы рассмотрим пару простейших механизмов внимания и приведем соответствующую терминологию. Внимание оказалось чрезвычайно полезным для повышения эффективности систем глубокого обучения со сложными входными и выходными данными. Фактически Богданов и его коллеги показали, что эффективность системы машинного перевода, согласно показателю BLEU (который мы обсудим в следующем разделе), без механизма внимания ухудшается по мере удлинения элементов входных данных (рис. 8.8). Добавление механизма внимания решает эту проблему.



**Рис. 8.8.** График демонстрирует изменения показателей BLEU систем машинного перевода с механизмом внимания (RNNsearch-30, RNNsearch-50) и без него (RNNenc-30, RNNenc-50).

Системы RNN\*-30 и RNN\*-50 обучались на предложениях длиной до 30 и до 50 слов соответственно. В системах машинного перевода без механизма внимания эффективность системы ухудшается по мере увеличения длины предложений. При наличии механизма внимания показатели перевода более длинных предложений улучшаются, но стабильность эффективности машинного перевода зависит от длины предложений, на которых они обучались (рисунок взят из статьи Богданова и др. [Bahdanau et al., 2015])

**Внимание в глубоких нейронных сетях.** Внимание — общий механизм, который можно использовать с любыми обсуждавшимися ранее в книге моделями. Но здесь

мы опишем его применительно к моделям типа «кодировщик-декодировщик», поскольку именно в них механизмы внимания проявляют себя особенно ярко. Рассмотрим S2S-модель. Напомним, что в обычной S2S-модели на каждом шаге формируется представление скрытого состояния, обозначаемое  $\varphi_w$ , свое для каждого конкретного временного шага в кодировщике (это видно из рис. 8.6). Для использования механизма внимания мы учтем не только завершающее скрытое состояние кодировщика, но и скрытые состояния для всех промежуточных шагов. Для этих скрытых состояний кодировщика используется, возможно, не слишком информативное название *значения*, или *ценности* (values), а в некоторых случаях — *ключи* (keys). Внимание также зависит от предыдущего скрытого состояния декодировщика, называемого *запросом* (query)<sup>1</sup>.

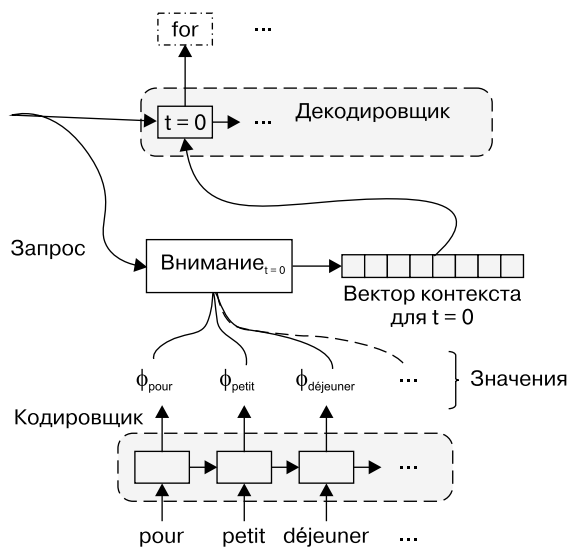
На рис. 8.9 все это проиллюстрировано для временного шага 0. Вектор запроса для временного шага  $t = 0$  — фиксированный гиперпараметр. Внимание представлено вектором, размерность которого соответствует количеству значений, на которые обращается внимание. Он называется *вектором внимания* (attention vector) или *весами внимания* (attention weights), а иногда *выравниванием* (alignment). На основе весов внимания в сочетании с состояниями кодировщика (значениями) генерируется *вектор контекста* (context vector), иногда называемый *glimpse* («беглый взгляд»). Такой вектор контекста служит входными данными для декодировщика, поэтому не требуется полная кодировка предложения. С помощью *функции совместимости* (compatibility function) обновляется вектор внимания для следующего временного шага. Конкретный характер функции совместимости зависит от используемого механизма внимания.

Существует несколько способов реализации внимания. Простейший и чаще всего используемый — *механизм внимания с учетом содержимого* (content-aware). Увидеть его в действии можно в разделе «Пример: нейронный машинный перевод» на с. 215. Еще один распространенный механизм внимания — *внимание с учетом местоположения* (location-aware), зависит только от вектора запроса и ключа. Веса внимания обычно представляют собой значения с плавающей точкой от 0 до 1. Такой механизм внимания называется *мягким* (soft). В случае же бинарного (0/1) вектора внимания оно называется *жестким* (hard).

Приведенный на рис. 8.9 механизм внимания зависит от состояний кодировщика для всех временных шагов входных данных. Такой механизм называется

<sup>1</sup> Начинаящих термины «ключи», «значения» («ценности») и «запрос» могут только запутать, но мы все равно будем ими пользоваться, поскольку они стали общепринятыми. Имеет смысл прочитать данный раздел несколько раз, чтобы полностью уяснить эти понятия. Такая терминология возникла потому, что изначально внимание рассматривалось как задача поиска. Расширенное описание этих понятий и механизма внимания в целом можно найти в статье Лилян Вэн «Внимание? Внимание!» (Weng L. Attention? Attention!) (<https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html>).

глобальным вниманием (global attention). Локальное же внимание представляет собой механизм, зависящий только от некоего окна входных данных около текущего временного шага.



**Рис. 8.9.** Внимание в действии на временном шаге  $t = 0$  декодировщика. Предсказанные выходные данные — *for*, и блок внимания учитывает скрытые состояния кодировщика  $\phi_w$  для всех входных слов

Иногда, особенно при машинном переводе, можно явным образом передать информацию о выравнивании в виде части обучающих данных. В подобных случаях можно создать *механизм внимания с учителем* (supervised attention mechanism), при котором функция внимания усваивается с помощью отдельной, совместно обучаемой нейронной сети. Для входных данных большого размера (например, документов) можно разработать более грубый или более тонкий (*иерархический*) механизм внимания, который не только концентрируется на непосредственных входных данных, но и учитывает структуру документа — абзац, раздел, главу и т. д.

В посвященной нейронным сетям с преобразованиями (transformer networks) работе Васвани и др. (Vaswani et al., 2017) описывается механизм *мультивнимания* (multiheaded attention), при котором несколько векторов внимания используются для отслеживания различных областей входных данных. В ней также описано *аутовнимание* (self-attention), механизм, с помощью которого модель узнает об областях входных данных и их влиянии друг на друга.

Если входные данные комбинированные (например, включают как речь, так и изображение), можно создать механизм *комбинированного* (multimodal) внимания.

Несмотря на новизну темы, сейчас можно найти много литературы, посвященной вниманию, что указывает на важность этой тематики. Подробный обзор всех подходов выходит за рамки книги, так что в качестве отправного пункта мы рекомендуем вам обратиться к статьям Луона, Фама и Мэннинга (Luong, Pham, Manning, 2015) и Васвани и др. (Vaswani et al., 2017).

## Оценка эффективности моделей генерации последовательностей

Такие метрики классификации, как точность, чувствительность<sup>1</sup> и F1, ничего не дают моделям при наличии нескольких возможных ответов, как мы видели в задачах генерации, — у одного французского предложения может быть несколько переводов на английский. Модели последовательностей оцениваются путем сравнения с ожидаемыми выходными данными — *эталонными выходными данными* (reference output). При сравнении различных моделей применяются показатели *согласия* (goodness) — близости выходных данных модели к эталонным. Например, в таких задачах, как машинный перевод, не стоит штрафовать отклонившуюся на одно слово модель так же сильно, как модель, выходной текст которой совершенно невозможно понять. Для одной входной выборки может существовать несколько эталонных выходных данных, например несколько (чуть отличающихся) допустимых переводов на английский конкретного французского предложения. Предусмотрено два вида оценки эффективности для моделей генерации последовательностей: оценка человеком и автоматическая оценка.

При оценке человеком эффективности машинного перевода несколько человек либо оценивают (положительно или отрицательно) результаты работы модели, либо исправляют перевод. В результате получается простой показатель частоты появления ошибок, очень хорошо отражающий нашу конечную цель: оценить близость результатов работы системы к результатам перевода человеком. Оценка человеком важна, но редко используется из-за дороговизны и медленной работы экспертов. Мнения людей также часто различаются, поэтому в качестве золотого стандарта при человеческой оценке применяется показатель частоты взаимного согласия оценщиков (inter-annotator agreement rate).

Измерение частоты взаимного согласия оценщиков также требует немалых затрат ресурсов. Одна из часто применяемых метрик оценки человеком — человеко-ориентированная частота ошибок перевода (human-targeted translation error rate, HTER) — взвешенное редакторское расстояние. Оно определяется путем подсчета количества вставок, удалений и перестановок, которые нужно выполнить человеку, чтобы «исправить» результат работы машинного перевода и получить текст с приемлемым смыслом и достаточно «гладкий» перевод (рис. 8.10).

<sup>1</sup> Иногда называется «полнота». — *Примеч. пер.*

### Judge Sentence

You have already judged 14 of 3064 sentences, taking 86.4 seconds per sentence.

**Source:** les deux pays constituent plutôt un laboratoire nécessaire au fonctionnement interne de l'ue .

**Reference:** rather , the two countries form a laboratory needed for the internal working of the eu .

Translation	Adequacy	Fluency
both countries are rather a necessary laboratory the internal operation of the eu .	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>
both countries are a necessary laboratory at internal functioning of the eu .	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>
the two countries are rather a laboratory necessary for the internal workings of the eu .	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>
the two countries are rather a laboratory for the internal workings of the eu .	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>
the two countries are rather a necessary laboratory internal workings of the eu .	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>1 2 3 4 5</div>
<b>Annotator:</b> Philipp Koehn <b>Task:</b> WMT06 French-English	<div>Annotate</div>	
Instructions	5= All Meaning 4= Most Meaning 3= Much Meaning 2= Little Meaning 1= None	5= Flawless English 4= Good English 3= Non-native English 2= Disfluent English 1= Incomprehensible

**Рис. 8.10.** Процесс оценки эффективности работы модели человеком (предоставлено Филиппом Кёном)

С другой стороны, автоматическая оценка проста и выполняется очень быстро. Существует два вида метрик для автоматической оценки сгенерированных предложений: *метрики на основе пересечения  $n$ -грамм* ( $n$ -gram overlap-based metrics) и *перплексия* (perplexity). Мы вновь воспользуемся в качестве примера машинным переводом, но эти метрики применимы к любой задаче, связанной с генерацией последовательностей. Метрики на основе пересечения  $n$ -грамм оценивают близость выходных данных к эталонным, вычисляя сводный показатель пересечения  $n$ -грамм. Примеры метрик на основе пересечения  $n$ -грамм: BLEU, ROUGE и METEOR. Из этих трех наиболее проверенной временем и предпочитаемой в литературе по машинному переводу является BLEU (BiLingual Evaluation Understudy)<sup>1</sup>. Мы не станем приводить здесь точную формулировку BLEU и рекомендуем вам обратиться к статье Папинени и др. (Papineni et al., 2002). На практике

<sup>1</sup> Настолько, что исходная статья, в которой BLEU была предложена, получила в 2018 году премию Test-of-Time (<https://naacl2018.wordpress.com/2018/03/22/test-of-time-award-papers/>).

для вычисления оценок мы будем использовать пакеты NLTK<sup>1</sup> или SacreBLEU<sup>2</sup>. Определение самой метрики BLEU при наличии эталонных данных происходит достаточно быстро и просто.

Перплексия — еще одна основанная на теории информации метрика автоматической оценки. Применима в любой ситуации, где можно определить вероятность выходной последовательности. Для последовательности  $x$ , вероятность которой составляет  $P(x)$ , перплексия определяется как:

$$\text{перплексия}(x) = 2^{-P(x)\log P(x)}.$$

Благодаря этому можно легко сравнивать различные модели генерации последовательностей, просто определяя перплексию модели для выделенного набора данных. Хотя вычислить перплексию несложно, с ее использованием для оценки эффективности генерации последовательностей связано много проблем. Во-первых, это «раздувающая» метрика. Обратите внимание, что в выражении для перплексии есть возведение в степень. В результате мелкие различия эффективности моделей (правдоподобия) приводят к большим различиям в перплексии, создавая иллюзию значительного прогресса. Во-вторых, изменения перплексии не обязательно отражаются в соответствующие изменения частоты ошибок моделей с точки зрения других метрик. Наконец, как и в случае BLEU и других метрик на основе  $n$ -грамм, улучшение перплексии не всегда приводит к ощутимым улучшениям перевода с точки зрения человека.

В следующем разделе мы приведем пример машинного перевода и соединим все описанные понятия воедино в реализации с помощью фреймворка PyTorch.

## Пример: нейронный машинный перевод

В этом разделе мы рассмотрим реализацию наиболее распространенного применения S2S-моделей — машинного перевода. По мере роста популярности глубокого обучения в начале 2010-х стало очевидно, что вложения слов и RNN — исключительно мощные инструменты для перевода с одного языка на другой — при условии достаточного количества данных. Модели машинного перевода еще больше усовершенствовались с появлением механизма внимания, описанного в разделе «Оценка эффективности моделей генерации последовательностей» на с. 213. В этом разделе мы опишем реализацию, в основе которой лежит упрощенный подход к вниманию в S2S-моделях (Luong, Pham, Manning, 2015).

<sup>1</sup> Пример можно найти по адресу [https://github.com/nltk/nltk/blob/develop/nltk/translate/bleu\\_score.py](https://github.com/nltk/nltk/blob/develop/nltk/translate/bleu_score.py).

<sup>2</sup> Пакет SacreBLEU (<https://github.com/mjpost/sacreBLEU>) — общепринятый ориентир при оценке машинного перевода.

Начнем с общего описания набора данных и вспомогательных структур данных, необходимых для нейронного машинного перевода. Набор данных представляет собой корпус параллельных текстов, состоящий из пар английских предложений и соответствующих французских переводов. Поскольку речь идет о парах предложений, длины которых могут различаться, нужно отслеживать максимальные длины и словари как входных, так и выходных предложений. По большей части пример представляет собой естественное расширение того, что наши внимательные читатели уже видели в предыдущих главах.

Описав набор данных и вспомогательные структуры данных, мы рассмотрим модель и генерацию ею целевой последовательности, обращая внимание на различные места исходной последовательности. Кодировщик в нашей модели вычисляет векторы для каждого из мест исходной последовательности на основе информации от всех частей последовательности, используя двунаправленный шлюзовый рекуррентный блок (bi-GRU). Для этого воспользуемся структурой данных `PackedSequence` фреймворка PyTorch. К выходным данным bi-GRU применяется механизм внимания (см. раздел «Захватываем больше информации из последовательности: внимание» на с. 209), контекстно обуславливающий генерацию целевой последовательности. Мы обсудим результаты работы модели и возможности их улучшения в подразделе «Процедура обучения и результаты» на с. 232.

## Набор данных для машинного перевода

Для этого примера мы воспользуемся набором данных англо-французских пар предложений из проекта Tatoeba<sup>1</sup>. Предварительная обработка данных начинается с преобразования всех предложений в нижний регистр и применения к каждой паре предложений токенизаторов для английского и французского языков из пакета NLTK. Далее мы создадим списки токенов с помощью токенизаторов слов для соответствующих языков. И хотя далее мы выполняем дополнительные действия, этот список токенов представляет собой предварительно обработанный набор данных.

Мы не только выполним описанную выше стандартную предварительную обработку, но и воспользуемся приведенным ниже списком синтаксических паттернов для выбора поднабора данных, чтобы упростить задачу обучения. По существу, мы сужаем область определения данных до ограниченного диапазона синтаксических паттернов. В свою очередь, это значит, что модель при обучении будет наблюдать меньшее разнообразие данных и продемонстрирует более высокую эффективность при более коротком времени обучения.



При создании новых моделей и экспериментировании с новыми архитектурами следует выбирать параметры моделей, которые обеспечивают более короткие циклы итерации.

<sup>1</sup> Этот набор данных взят с сайта <http://www.manythings.org/anki/>.



Выбранный нами поднабор данных состоит из английских фраз, начинающихся с *I am, he is, she is, they are, you are* или *we are*<sup>1</sup>. Это приводит к сокращению набора данных с 135 842 до всего лишь 13 062 пар предложений, то есть в десять раз<sup>2</sup>. В завершение настроек обучения разобьем оставшиеся 13 062 пары предложений на обучающий (70 %), проверочный (15 %) и контрольный (15 %) наборы данных. Чтобы доля предложений, начинающихся с приведенного выше списка, была постоянной, мы сначала сгруппируем по начальным словам предложения, создадим фрагменты на основе этих групп, после чего объединим фрагменты из каждой группы.

## Конвейер векторизации для NMT

Для векторизации исходных английских и целевых французских предложений требуется более сложный конвейер, чем в предыдущих главах. Причин у этого две. Во-первых, исходные и целевые предложения играют разные роли в модели, относятся к различным языкам и векторизуются по-разному. Во-вторых, необходимым условием для использования структуры `PackedSequence` фреймворка PyTorch является сортировка каждого из мини-пакетов по длине исходных предложений<sup>3</sup>. В качестве подготовки к этим двум нюансам объект класса `NMTVectorizer` инициализируется двумя различными объектами типа `SequenceVocabulary` и двумя значениями максимальной длины последовательности (пример 8.1).

### Пример 8.1. Формирование объекта `NMTVectorizer`

```
class NMTVectorizer(object):
    """ Векторизатор, приводящий словари в соответствие друг другу
        и использующий их """
    def __init__(self, source_vocab, target_vocab, max_source_length,
                  max_target_length):
        """
        Аргументы:
            source_vocab (SequenceVocabulary): соответствие исходных
            слов целым числам
            target_vocab (SequenceVocabulary): соответствие целевых слов
            целым числам
            max_source_length (int): максимальная длина последовательности
            в исходном наборе данных
```

<sup>1</sup> Мы также включаем туда случаи сокращенного написания этих пар «подлежащее — сказуемое», например *i'm, we're* и *he's*.

<sup>2</sup> А значит, модель сможет просмотреть весь набор данных в десять раз быстрее. Впрочем, из этого не следует, что процесс обучения сойдется ровно за 1/10 времени, поскольку модели может потребоваться просмотреть набор данных за меньшее число эпох или может помешать еще какой-то фактор.

<sup>3</sup> Для сортировки предложений по порядку используется низкоуровневый примитив CUDA RNN.

```

        max_target_length (int): максимальная длина последовательности
        в целевом наборе данных
    """
    self.source_vocab = source_vocab
    self.target_vocab = target_vocab

    self.max_source_length = max_source_length
    self.max_target_length = max_target_length

    @classmethod
    def from_dataframe(cls, bitext_df):
        """ Создает экземпляр векторизатора на основе
        объекта DataFrame набора данных

        Аргументы:
            bitext_df (pandas.DataFrame): параллельный набор текста
        Возвращает:
            экземпляр NMTVectorizer
        """
        source_vocab = SequenceVocabulary()
        target_vocab = SequenceVocabulary()
        max_source_length, max_target_length = 0, 0

        for _, row in bitext_df.iterrows():
            source_tokens = row["source_language"].split(" ")
            if len(source_tokens) > max_source_length:
                max_source_length = len(source_tokens)
            for token in source_tokens:
                source_vocab.add_token(token)

            target_tokens = row["target_language"].split(" ")
            if len(target_tokens) > max_target_length:
                max_target_length = len(target_tokens)
            for token in target_tokens:
                target_vocab.add_token(token)

        return cls(source_vocab, target_vocab, max_source_length,
                    max_target_length)

```

Прежде всего сложность повышается из-за различий в способах обработки исходной и целевой последовательностей. Исходная последовательность векторизуется со вставкой токенов **BEGIN-OF-SEQUENCE** и **END-OF-SEQUENCE** в начале и конце соответственно. Для создания сводных векторов для каждого из токенов исходной последовательности модель использует **bi-GRU**, причем в этом процессе большую пользу приносит наличие границ последовательностей. И напротив, целевая последовательность векторизуется в виде двух копий, смещенных друг относительно друга на один токен: первой копии требуется токен **BEGIN-OF-SEQUENCE**, а второй копии — **END-OF-SEQUENCE**. Как вы помните из главы 7, в задачах предсказания последовательностей необходимо наблюдать входные и выходные токены на каждом временном шаге. Эту задачу в **S2S**-модели выполняет декодировщик, но при условии доступности контекста кодировщика.

Для решения проблемы мы создадим базовый метод векторизации, `_vectorize()`, для которого не важно, обрабатывать ли входные или целевые индексы. Далее мы напишем два отдельных метода для обработки входных и целевых индексов. Наконец, эти наборы индексов согласовываются с помощью вызываемого объектом набора данных метода `NMTVectorizer.vectorize`. Соответствующий код приведен в примере 8.2.

### Пример 8.2. Функции векторизации из класса NMTVectorizer

```

class NMTVectorizer(object):
    """ Векторизатор, приводящий словари в соответствие друг другу
    и использующий их"""
    def _vectorize(self, indices, vector_length=-1, mask_index=0):
        """ Векторизация передаваемых индексов

        Аргументы:
            indices (list): список представляющих последовательность
            целочисленных значений
            vector_length (int): жестко задает длину вектора индексов
            mask_index (int): используемый mask_index; практически всегда 0
        """
        if vector_length < 0:
            vector_length = len(indices)
        vector = np.zeros(vector_length, dtype=np.int64)
        vector[:len(indices)] = indices
        vector[len(indices):] = mask_index
        return vector

    def _get_source_indices(self, text):
        """ Возвращает векторизованный исходный текст

        Аргументы:
            text (str): исходный текст; токены должны отделяться пробелами
        Возвращает:
            indices (list): список целых чисел, представляющих текст
        """
        indices = [self.source_vocab.begin_seq_index]
        indices.extend(self.source_vocab.lookup_token(token)
                       for token in text.split(" "))
        indices.append(self.source_vocab.end_seq_index)
        return indices

    def _get_target_indices(self, text):
        """ Возвращает векторизованный целевой текст

        Аргументы:
            text (str): целевой текст; токены должны отделяться пробелами
        Возвращает:
            кортеж: (x_indices, y_indices)
            x_indices (list): список целочисленных значений;
            наблюдения в целевом декодировщике
            y_indices (list): список целочисленных значений;
            предсказания в целевом декодировщике
        """

```

```

"""
indices = [self.target_vocab.lookup_token(token)
            for token in text.split(" ")]
x_indices = [self.target_vocab.begin_seq_index] + indices
y_indices = indices + [self.target_vocab.end_seq_index]
return x_indices, y_indices

def vectorize(self, source_text, target_text, use_dataset_max_lengths=True):
    """ Возвращает векторизованный исходный и целевой текст

    Аргументы:
        source_text (str): текст на исходном языке
        target_text (str): текст на целевом языке
        use_dataset_max_lengths (bool): использовать ли
            максимальные длины векторов
    Возвращает:
        векторизованную точку данных в виде словаря с ключами:
            source_vector, target_x_vector, target_y_vector, source_length
    """
    source_vector_length = -1
    target_vector_length = -1

    if use_dataset_max_lengths:
        source_vector_length = self.max_source_length + 2
        target_vector_length = self.max_target_length + 1

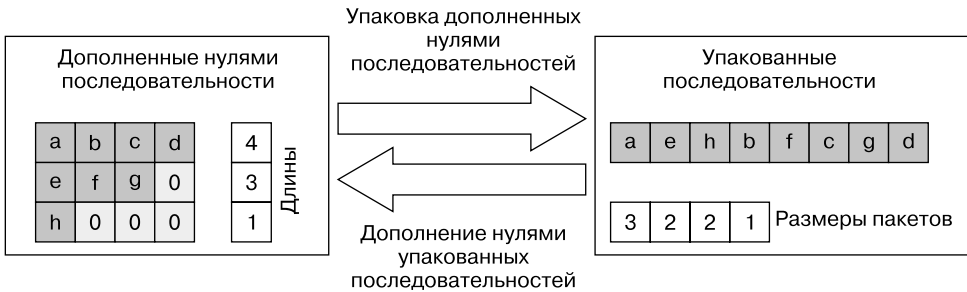
    source_indices = self._get_source_indices(source_text)
    source_vector = self._vectorize(source_indices,
                                    vector_length=source_vector_length,
                                    mask_index=self.source_vocab.mask_index)

    target_x_indices, target_y_indices = self._get_target_indices(
        target_text)
    target_x_vector = self._vectorize(target_x_indices,
                                       vector_length=target_vector_length,
                                       mask_index=self.target_vocab.mask_index)
    target_y_vector = self._vectorize(target_y_indices,
                                       vector_length=target_vector_length,
                                       mask_index=self.target_vocab.mask_index)

    return {"source_vector": source_vector,
            "target_x_vector": target_x_vector,
            "target_y_vector": target_y_vector,
            "source_length": len(source_indices)}

```

Вторая причина повышения сложности также кроется в исходной последовательности. Для кодирования исходной последовательности с помощью bi-GRU мы применяем структуру данных `PackedSequence` фреймворка PyTorch. Обычно мини-пакет из последовательностей переменной длины представляется численно в виде строк матрицы целых чисел, в которой все последовательности выравниваются влево и дополняются нулями для учета различий в длинах. Структура данных `PackedSequence` представляет последовательности переменной длины в виде массива. Для этого выполняется конкатенация данных последовательностей на каждом временном шаге, один за другим, и отслеживание числа последовательностей на каждом временном шаге (рис. 8.11).



**Рис. 8.11.** Слева показана матрица дополненных нулями последовательностей и их длины. Дополненная матрица — стандартный вариант представления последовательностей переменной длины. Они дополняются *справа* нулями и располагаются ярусами, в виде строк. Во фреймворке PyTorch допустимо и более сжатое представление дополненных последовательностей, с помощью структуры данных `PackedSequence`, приведенной справа вместе с размерами пакетов. Благодаря этому представлению GPU может проходить последовательность, отслеживая количество последовательностей на каждом временном шаге (размеров пакетов)

Для создания `PackedSequence` необходимо знать длину всех последовательностей, кроме того, последовательности должны быть отсортированы в порядке убывания длины исходной последовательности. Для отражения этой вновь отсортированной матрицы оставшиеся в мини-пакете тензоры сортируются в том же порядке, чтобы соответствовать кодировке исходной последовательности. В примере 8.3 показаны изменения, внесенные в функцию `generate_batches()`, после которых она превратилась в `generate_nmt_batches()`.

### Пример 8.3. Генерация мини-пакетов для примера NMT

```
def generate_nmt_batches(dataset, batch_size, shuffle=True,
                        drop_last=True, device="cpu"):
    """ Функция-генератор — адаптер для объекта DataLoader
        фреймворка PyTorch; NMT-версия """
    dataloader = DataLoader(dataset=dataset, batch_size=batch_size,
                           shuffle=shuffle, drop_last=drop_last)

    for data_dict in dataloader:
        lengths = data_dict['x_source_length'].numpy()
        sorted_length_indices = lengths.argsort()[::-1].tolist()

        out_data_dict = {}
        for name, tensor in data_dict.items():
            out_data_dict[name] = data_dict[name][sorted_length_indices].to(device)
        yield out_data_dict
```

## Кодирование и декодирование в NMT-модели

В этом примере мы начинаем с исходной последовательности — предложения на английском языке — и генерируем целевую — соответствующий перевод на французский. Стандартный подход — использовать модели типа «кодировщик-декодировщик», как описано в первом разделе этой главы. В представленной в примерах 8.4 и 8.5



```

    decoded_states = self.decoder(encoder_state=encoder_state,
                                   initial_hidden_state=final_hidden_states,
                                   target_sequence=target_sequence)

    return decoded_states

```

**Пример 8.5.** Кодировщик выполняет вложения исходных слов и извлекает признаки с помощью bi-GRU

```

class NMTEncoder(nn.Module):
    def __init__(self, num_embeddings, embedding_size, rnn_hidden_size):
        """
        Аргументы:
            num_embeddings (int): размер исходного словаря
            embedding_size (int): размер векторов вложений
            rnn_hidden_size (int): размер векторов скрытого состояния RNN
        """
        super(NMTEncoder, self).__init__()

        self.source_embedding = nn.Embedding(num_embeddings, embedding_size,
                                              padding_idx=0)
        self.birnn = nn.GRU(embedding_size, rnn_hidden_size, bidirectional=True,
                             batch_first=True)

    def forward(self, x_source, x_lengths):
        """ Прямой проход модели

        Аргументы:
            x_source (torch.Tensor): тензор входных данных
            Значение x_source.shape равно (batch, seq_size)
            x_lengths (torch.Tensor): вектор длин всех элементов пакета
        Возвращает:
            кортеж: x_unpacked (torch.Tensor), x_birnn_h (torch.Tensor)
            x_unpacked.shape = (batch, seq_size, rnn_hidden_size * 2)
            x_birnn_h.shape = (batch, rnn_hidden_size * 2)
        """
        x_embedded = self.source_embedding(x_source)
        # создает PackedSequence; x_packed.data.shape=(number_items,
        #                                           embedding_size)
        x_lengths = x_lengths.detach().cpu().numpy()
        x_packed = pack_padded_sequence(x_embedded, x_lengths,
                                       batch_first=True)

        # x_birnn_h.shape = (num_rnn, batch_size, feature_size)
        x_birnn_out, x_birnn_h = self.birnn(x_packed)
        # перестановка в (batch_size, num_rnn, feature_size)
        x_birnn_h = x_birnn_h.permute(1, 0, 2)
        # свертываем признаки; изменяем форму
        # на (batch_size, num_rnn * feature_size)
        # (напомним: -1 охватывает оставшиеся позиции,
        # свертывая два скрытых вектора RNN в 1)
        x_birnn_h = x_birnn_h.contiguous().view(x_birnn_h.size(0), -1)

        x_unpacked, _ = pad_packed_sequence(x_birnn_out, batch_first=True)
        return x_unpacked, x_birnn_h

```

В целом, кодировщик принимает на входе последовательность целочисленных значений и создает по вектору признаков для каждой позиции. На выходе кодировщик выдает эти векторы и завершающее скрытое состояние bi-GRU, использовавшееся для создания векторов признаков. Это скрытое состояние применяется для инициализации скрытого состояния декодировщика в следующем подразделе.

Если углубиться в описание работы кодировщика, то сначала выполняется вложение входной последовательности с помощью слоя вложений. Обычно, чтобы модель могла работать с последовательностями переменной длины, достаточно установить флаг `padding_idx` для слоя вложений, поскольку любой равной `padding_idx` позиции присваивается нулевой вектор, который не обновляется в процессе оптимизации. Напомним, что это называется *маской*. Впрочем, в такой модели типа «кодировщик-декодировщик» маскированные позиции необходимо обрабатывать по-другому, поскольку для кодирования исходной последовательности мы применяем bi-GRU. Основная причина состоит в том, что маскированные позиции могут влиять на обратную компоненту архитектуры bi-GRU пропорционально числу встреченных им маскированных позиций до начала обработки последовательности<sup>1</sup>.

Для обработки маскированных позиций в последовательностях переменной длины в bi-GRU мы используем структуру данных `PackedSequence` фреймворка PyTorch. Вид `PackedSequence` определяет способ пакетной обработки последовательностей переменной длины в CUDA. Любую дополненную нулями последовательность, например, вложенную исходную последовательность из представленного в примере 8.6 кодировщика можно преобразовать в `PackedSequence` при соблюдении двух условий: указания длин всех последовательностей и сортировки мини-пакета в соответствии с этими длинами. Это наглядно показано на рис. 8.11, а поскольку это непростая задача, мы проиллюстрируем ее в примере 8.6 и результатах его выполнения<sup>2</sup>.

**Пример 8.6.** Простая демонстрация функций `pack_padded_sequence()` и `pad_packed_sequence()`

Input[0]

```
abcd_padded = torch.tensor([1, 2, 3, 4], dtype=torch.float32)
efg_padded = torch.tensor([5, 6, 7, 0], dtype=torch.float32)
h_padded = torch.tensor([8, 0, 0, 0], dtype=torch.float32)

padded_tensor = torch.stack([abcd_padded, efg_padded, h_padded])

describe(padded_tensor)
```

<sup>1</sup> Чтобы убедиться в этом, попробуйте представить в уме или изобразить схематически ход вычислений. В качестве подсказки: рассмотрите очередной шаг — входные данные и последнее скрытое состояние умножаются на веса и складываются с учетом смещения. Какое влияние смещение окажет на выходные данные, если они состояли исключительно из 0?

<sup>2</sup> Для этого мы воспользуемся функцией `describe()`, описанной в подразделе «Создание тензоров» на с. 31.



```
Output[0]
Type: torch.FloatTensor
Shape/size: torch.Size([3, 4])
Values:
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  0.],
        [ 8.,  0.,  0.,  0.]])
```

```
Input[1]
lengths = [4, 3, 1]
packed_tensor = pack_padded_sequence(padded_tensor, lengths,
                                     batch_first=True)
packed_tensor
```

```
Output[1]
PackedSequence(data=tensor([ 1.,  5.,  8.,  2.,  6.,  3.,  7.,  4.]),
               batch_sizes=tensor([ 3,  2,  2,  1]))
```

```
Input[2]
unpacked_tensor, unpacked_lengths = \
    pad_packed_sequence(packed_tensor, batch_first=True)

describe(unpacked_tensor)
describe(unpacked_lengths)
```

```
Output[2]
Type: torch.FloatTensor
Shape/size: torch.Size([3, 4])
Values:
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  0.],
        [ 8.,  0.,  0.,  0.]])
Type: torch.LongTensor
Shape/size: torch.Size([3])
Values:
tensor([ 4,  3,  1])
```

Как описано в предыдущем подразделе, сортировка производится при генерации мини-пакетов. Далее, как показано в примере 8.5, вызывается функция `pack_padded_sequence()` фреймворка PyTorch и ей передаются вложенные последовательности, их длины и булев флаг, указывающий, что первое измерение содержит пакет. Эта функция возвращает объект `PackedSequence`, который служит входными данными bi-GRU с целью создания векторов состояния для расположенного далее по конвейеру декодировщика. Результаты работы bi-GRU распаковываются в полный тензор с помощью еще одного булева флага, указывающего, что пакет находится в первом измерении. В результате операции распаковки (рис. 8.11) на всех маскированных позициях<sup>1</sup> оказываются нулевые векторы, что гарантирует целостность вычислений далее по конвейеру.

<sup>1</sup> Все позиции за пределами известной длины последовательности, начиная слева направо в измерении последовательности, считаются маскированными.

**Пример 8.7.** Класс NMTDecoder служит для формирования целевого предложения из кодированного исходного предложения

```
class NMTDecoder(nn.Module):
    def __init__(self, num_embeddings, embedding_size, rnn_hidden_size,
                  bos_index):
        """
        Аргументы:
            num_embeddings (int): число вложений; также число уникальных
            слов в целевом словаре
            embedding_size (int): размер вектора вложения
            rnn_hidden_size (int): размер скрытого состояния RNN
            bos_index(int): индекс BEGIN-OF-SEQUENCE
        """
        super(NMTDecoder, self).__init__()
        self.rnn_hidden_size = rnn_hidden_size
        self.target_embedding = nn.Embedding(num_embeddings=num_embeddings,
                                              embedding_dim=embedding_size,
                                              padding_idx=0)
        self.gru_cell = nn.GRUCell(embedding_size + rnn_hidden_size,
                                    rnn_hidden_size)
        self.hidden_map = nn.Linear(rnn_hidden_size, rnn_hidden_size)
        self.classifier = nn.Linear(rnn_hidden_size * 2, num_embeddings)
        self.bos_index = bos_index

    def _init_indices(self, batch_size):
        """ возвращает вектор индексов BEGIN-OF-SEQUENCE """
        return torch.ones(batch_size, dtype=torch.int64) * self.bos_index

    def _init_context_vectors(self, batch_size):
        """ возвращает нулевой вектор для инициализации контекста """
        return torch.zeros(batch_size, self.rnn_hidden_size)

    def forward(self, encoder_state, initial_hidden_state, target_sequence):
        """ Прямой проход модели
        Аргументы:
            encoder_state (torch.Tensor): выходные данные NMTEncoder
            initial_hidden_state (torch.Tensor): последнее скрытое состояние
            в NMTEncoder
            target_sequence (torch.Tensor): целевой тензор текстовых данных
            sample_probability (float): параметр плановой выборки
            вероятность использования предсказаний модели
            на каждом шаге декодировщика
        Возвращает:
            output_vectors (torch.Tensor): векторы предсказаний
            на каждом из шагов
        """
        # Здесь мы предполагаем, что пакет находится в первом измерении
        # Входные данные: (Пакет, Последовательность)
        # Мы хотим пройти в цикле по последовательности и переставить
        # измерения местами, чтобы получилось (Последовательность, Пакет)
        target_sequence = target_sequence.permute(1, 0)

        # Используем в качестве начального скрытого состояния
        # переданное скрытое состояние кодировщика
```

```

h_t = self.hidden_map(initial_hidden_state)

batch_size = encoder_state.size(0)
# Инициализируем векторы контекста нулями
context_vectors = self._init_context_vectors(batch_size)
# Инициализируем первое y_t слово как BOS [BEGIN-OF-SEQUENCE]
y_t_index = self._init_indices(batch_size)

h_t = h_t.to(encoder_state.device)
y_t_index = y_t_index.to(encoder_state.device)
context_vectors = context_vectors.to(encoder_state.device)

output_vectors = []
# Перемещаем из GPU все закэшированные тензоры и сохраняем
# для последующего анализа
self._cached_p_attn = []
self._cached_ht = []
self._cached_decoder_state = encoder_state.cpu().detach().numpy()

output_sequence_size = target_sequence.size(0)
for i in range(output_sequence_size):

    # Шаг 1: выполняем вложение слова и его конкатенацию
    # с предыдущим контекстом
    y_input_vector = self.target_embedding(target_sequence[i])
    rnn_input = torch.cat([y_input_vector, context_vectors], dim=1)

    # Шаг 2: выполняем шаг GRU, получая новый скрытый вектор
    h_t = self.gru_cell(rnn_input, h_t)
    self._cached_ht.append(h_t.cpu().data.numpy())

    # Шаг 3: обращаем внимание на состояние кодировщика,
    # используя текущий скрытый вектор
    context_vectors, p_attn, _ = \
        verbose_attention(encoder_state_vectors=encoder_state,
                          query_vector=h_t)

    # вспомогательный: кэшируем вероятности внимания
    # для визуализации
    self._cached_p_attn.append(p_attn.cpu().detach().numpy())

    # Шаг 4: на основе текущего скрытого вектора
    # и вектора контекста генерируем предсказание
    # относительно следующего слова
    prediction_vector = torch.cat((context_vectors, h_t), dim=1)
    score_for_y_t_index = self.classifier(prediction_vector)

    # вспомогательный: получаем показатели эффективности
    # предсказания
    output_vectors.append(score_for_y_t_index)

```

После создания кодировщиком векторов состояния с помощью bi-GRU и упаковки-распаковки декодировщик проходит в цикле по временным шагам и генерирует выходную последовательность. Функционально этот цикл очень похож на

цикл генерации из главы 7, но с несколькими отличиями, очевидно, результатами методологического выбора при используемом Луоном, Фамом и Мэннингом стиле внимания. Во-первых, целевая последовательность представляет собой наблюдения на каждом из временных шагов<sup>1</sup>. С помощью GRUCell вычисляются скрытые состояния. Начальное скрытое состояние определяется путем применения линейного слоя к сцепленным итоговым скрытым состояниям кодировщика bi-GRU<sup>2</sup>. Входные данные декодировщика GRU на каждом из временных шагов представляют собой сцепленный вектор вложенного входного токена и вектора контекста с предыдущего шага. Вектор контекста должен захватывать полезную для текущего временного шага информацию и контекстно обуславливать выходные данные модели. На первом временном шаге вектор контекста состоит из нулей, представляя, таким образом, отсутствие контекста, благодаря чему какой-либо вклад в вычисления GRU математически вносят только входные данные.

При новом скрытом состоянии в качестве вектора запроса создается новый набор векторов контекста с помощью механизма внимания для текущего временного шага. В результате конкатенации этих векторов контекста со скрытым состоянием создается вектор, представляющий информацию декодировки для соответствующего временного шага. Этот вектор состояния информации декодировки используется в классификаторе (в данном случае в простом линейном слое) для создания вектора предсказаний, `score_for_y_t_index`. Из полученных векторов предсказаний с помощью многомерной логистической функции можно сделать распределения вероятностей для выходного словаря или с помощью функции потерь на основе перекрестной энтропии задействовать их для оптимизации по эталонным целевым значениям. Прежде чем говорить о том, как векторы предсказаний используются в процедуре обучения, взглянем на вычисления, связанные с собственно вниманием.

## Подробнее изучаем механизм внимания

Важно понимать, как именно работает механизм внимания в данном примере. Как вы помните из пункта «Внимание в глубоких нейронных сетях» на с. 210, механизм внимания можно описать в терминах *запросов*, *ключей* и *значений*. Функция вычисления показателей принимает на входе вектор *запроса* и векторы *ключей*, на основе которых вычисляет набор весов для выбора векторов *значений*. В данном примере ее роль играет скалярное произведение, но возможны и другие<sup>3</sup>. В этом примере скрытое состояние декодировщика выступает в роли

<sup>1</sup> Векторизатор присоединяет к началу последовательности токен BEGIN-OF-SEQUENCE, так что первое наблюдение всегда представляет собой специальный токен, указывающий границу последовательности.

<sup>2</sup> Обсуждение вопросов соединения кодировщиков и декодировщиков в нейронном машинном переводе можно найти в разделе 7.3 книги Нойбига (Neubig, 2007).

<sup>3</sup> В статье Луона, Фама и Мэннинга (Luong, Pham, Manning, 2015) описываются три различные функции вычисления показателей.

вектора *запроса*, а набор векторов состояния кодировщика — векторов как *ключей*, так и *значений*.

Скалярное произведение скрытого состояния декодировщика с векторами из состояния кодировщика дает в результате по скалярному значению для каждого из элементов закодированной последовательности. После применения многомерной логистической функции эти скалярные значения становятся распределениями вероятностей для векторов состояния кодировщика<sup>1</sup>. Эти вероятности применяются для умножения векторов состояния кодировщика на веса перед получением в результате их сложения. Используется один вектор для каждого элемента пакета. В завершение скрытое состояние декодировщика может выборочно умножать состояние кодировщика на веса на каждом из временных шагов. Это служит своеобразным «проектором», благодаря которому модель может научиться «выделять» информацию, необходимую для генерации выходной последовательности.

В примере 8.8 мы продемонстрируем этот вариант механизма внимания. Первая функция пытается подробно описать операции. Кроме того, в ней используется операция `view()`, добавляющая измерения размером 1 для последующего транслирования тензора с другим тензором<sup>2</sup>. В версии `terse_attention()` операция `view()` заменяется наиболее популярной операцией `unsqueeze()`. Кроме того, вместо поэлементного умножения и суммирования используется более эффективная математически операция `matmul()`.

**Пример 8.8.** Механизм внимания, при котором поэлементное умножение и суммирование выполняются более явным образом

```
def verbose_attention(encoder_state_vectors, query_vector):
    """
    encoder_state_vectors: трехмерный вектор из bi-GRU в кодировщике
    query_vector: скрытое состояние GRU декодировщика
    """
    batch_size, num_vectors, vector_size = encoder_state_vectors.size()
    vector_scores = \
        torch.sum(encoder_state_vectors * query_vector.view(batch_size, 1,
                                                                vector_size),
                  dim=2)
    vector_probabilities = F.softmax(vector_scores, dim=1)
    weighted_vectors = \
        encoder_state_vectors * vector_probabilities.view(batch_size,
                                                            num_vectors, 1)
```

<sup>1</sup> Каждый элемент мини-пакета представляет собой последовательность, а сумма вероятностей для каждой последовательности равна 1.

<sup>2</sup> Транслирование происходит, если в тензоре есть измерение размером 1. Пусть этот тензор называется тензор А. При выполнении поэлементной операции (например, сложения или вычитания) над этим и еще одним тензором (назовем его тензор В) их формы должны совпадать, за исключением измерения размерностью 1. Операция над тензором А и тензором В повторяется для каждой из позиций тензора В. Если форма тензора А — (10, 1, 10), а тензора В — (10, 5, 10), то при операции А + В сложение тензора А будет повторено для каждой из пяти позиций тензора В.

```

context_vectors = torch.sum(weighted_vectors, dim=1)
return context_vectors, vector_probabilities

def terse_attention(encoder_state_vectors, query_vector):
    """
    encoder_state_vectors: трехмерный вектор из bi-GRU в кодировщике
    query_vector: скрытое состояние
    """
    vector_scores = torch.matmul(encoder_state_vectors,
                                  query_vector.unsqueeze(dim=2)).squeeze()
    vector_probabilities = F.softmax(vector_scores, dim=-1)
    context_vectors = torch.matmul(encoder_state_vectors.transpose(-2, -1),
                                    vector_probabilities.unsqueeze(dim=2)).squeeze()
    return context_vectors, vector_probabilities

```

## Обучение поиску и плановая выборка

В текущем виде наша модель предполагает наличие целевой последовательности, которая используется в качестве входных данных на каждом из временных шагов в декодировщике. Во время тестирования это допущение нарушается, поскольку модель не может «жульничать» и знать последовательность, которую пытается сгенерировать. Для решения этой проблемы предусмотрена методика, с помощью которой модель может использовать во время обучения свои собственные предсказания. В литературе данная методика известна под названием «обучение поиску» (learning to search) и «плановая выборка» (scheduled sampling)<sup>1</sup>.

Чтобы уяснить себе сущность этой методики, рассмотрим задачу предсказания как задачу поиска. На каждом из временных шагов модель может выбрать один из множества путей (количество вариантов равно размеру целевого словаря), а данные представляют собой наблюдения правильных путей. Во время тестирования модель наконец-то может «выйти из колеи», поскольку правильный путь для вычисления распределений вероятностей ей более не известен. Таким образом, методика, которая разрешает модели производить выборку собственного пути, предоставляет способ оптимизации модели и получения ею лучших распределений вероятностей при отклонении от целевых последовательностей из набора данных.

Чтобы модель могла выполнять выборку собственных предсказаний во время обучения, необходимо внести в код три основных изменения. Во-первых, сделать начальные индексы более явными в виде индексов токена **BEGIN-OF-SEQUENCE**. Во-вторых, на каждом временном шаге цикла генерации будет выбираться случайная выборка, и если эта случайная выборка меньше вероятности собственных предсказаний модели, во время текущей итерации нужно использовать предсказания модели<sup>2</sup>.

<sup>1</sup> Подробности вы можете найти в статьях Дауме, Лэнгфорда и Марку (Daumé, Langford, Marcu, 2009) и Бенжио и др. (Bengio et al., 2015).

<sup>2</sup> Если вы знакомы с выборкой по методу Монте-Карло для таких способов оптимизации, как методы Монте-Карло по схеме марковских цепей, то наверняка узнаете этот паттерн.

Наконец, при условии `if use_sample` выполняется сама выборка. В примере 8.9 закомментированные строки демонстрируют использование предсказания максимума, а не закомментированные строки демонстрируют возможность выборки индексов пропорционально их вероятностям.

**Пример 8.9.** Декодировщик со встроенной процедурой выборки (выделено жирным шрифтом)

```
class NMTDecoder(nn.Module):
    def __init__(self, num_embeddings, embedding_size, rnn_size, bos_index):
        super(NMTDecoder, self).__init__()
        # ... остальной код инициализации ...

        # задается произвольно; подойдет любая небольшая константа
        self._sampling_temperature = 3

    def forward(self, encoder_state, initial_hidden_state, target_sequence,
                sample_probability=0.0):
        if target_sequence is None:
            sample_probability = 1.0
        else:
            # Здесь мы предполагаем, что пакет находится в первом измерении
            # Входные данные: (Пакет, Последовательность)
            # Мы хотим пройти в цикле по последовательности и переставить
            # измерения местами, чтобы получить (Последовательность, Пакет)

            target_sequence = target_sequence.permute(1, 0)
            output_sequence_size = target_sequence.size(0)

            # ... с предыдущей реализации ничего не поменялось

            output_sequence_size = target_sequence.size(0)
            for i in range(output_sequence_size):
                # новый код: вспомогательное булево значение
                # и "учитель" y_t_index
                use_sample = np.random.random() < sample_probability
                if not use_sample:
                    y_t_index = target_sequence[i]

                # Шаг 1: выполняем вложение слова и его конкатенацию
                # с предыдущим контекстом
                # ... код опущен для экономии места
                # Шаг 2: выполняем шаг GRU, получая новый скрытый вектор
                # ... код опущен для экономии места
                # Шаг 3: обращаем внимание на состояние кодировщика,
                # используя текущий скрытый вектор
                # ... код опущен для экономии места
                # Шаг 4: на основе текущего скрытого вектора
                # и вектора контекста генерируем предсказание
                # относительно следующего слова
                prediction_vector = torch.cat((context_vectors, h_t), dim=1)
                score_for_y_t_index = self.classifier(prediction_vector)
                # новый код: если булево значение истинно, производим выборку
                if use_sample:
                    # температура выборки обеспечивает заострение максимумов
```

```

p_y_t_index = F.softmax(score_for_y_t_index *
                        self._sampling_temperature, dim=1)
# метод 1: выбираем наиболее вероятное слово
# _, y_t_index = torch.max(p_y_t_index, 1)
# метод 2: выборка из распределения
y_t_index = torch.multinomial(p_y_t_index, 1).squeeze()

# вспомогательный код: получаем показатели эффективности
# предсказания
output_vectors.append(score_for_y_t_index)

output_vectors = torch.stack(output_vectors).permute(1, 0, 2)

return output_vectors

```

## Процедура обучения и результаты

Процедура обучения для этого примера практически идентична процедурам обучения, которые вы видели в предыдущих главах<sup>1</sup>. Мы проходим в цикле по набору данных по порциям-мини-пакетам в течение определенного числа эпох. Впрочем, здесь каждый мини-пакет состоит из четырех тензоров: матрицы целых чисел для исходной последовательности, двух матриц целых чисел для целевой последовательности и вектора целых чисел для длин исходной последовательности. Две матрицы целевой последовательности представляют собой смещение целевой последовательности на один токен и дополнены либо токенами **BEGIN-OF-SEQUENCE** в качестве наблюдений целевой последовательности, либо токенами **END-OF-SEQUENCE** в качестве меток предсказаний целевой последовательности. В виде входных данных модель принимает исходную последовательность и наблюдения целевой последовательности и генерирует предсказания целевой последовательности. Метки предсказаний целевой последовательности используются в функции потерь для вычисления потерь на основе перекрестной энтропии, которые затем распространяются обратно по всем параметрам модели, для выяснения градиентов. После этого вызывается оптимизатор и обновляет все параметры модели пропорционально градиентам.

Помимо прохода в цикле по обучающему фрагменту набора данных, выполняется и проход по проверочному фрагменту. Полученные при проверке сведения об эффективности служат более объективным показателем улучшения работы модели. Процедура идентична процедуре обучения, но модель работает в режиме оценки и не обновляется в соответствии с проверочными данными.

После обучения модели возникает важная и непростая задача оценки эффективности ее работы. В разделе «Оценка эффективности моделей генерации после-

<sup>1</sup> В основном благодаря тому, что градиентный спуск и автоматическое дифференцирование — изящные абстракции, располагающиеся между определениями модели и их оптимизацией.



довательностей» на с. 213 описано несколько метрик оценки генерации последовательностей, но такие метрики, как BLEU, оценивающие пересечение  $n$ -грамм между предсказанными и эталонными предложениями, стали стандартом в сфере машинного перевода. Код оценки с агрегированием результатов мы здесь приводить не станем, но вы можете найти его в репозитории GitHub по адресу <https://nlpproc.info/PyTorchNLPBook/repo/>. В этом коде выходные данные модели агрегируются с исходным предложением, эталонным целевым предложением и матрицей вероятностей внимания для данного примера. Наконец, для каждой пары исходного и сгенерированного предложений вычисляется BLEU-4.

Для качественной, а не количественной оценки работы модели мы визуализируем матрицу вероятностей внимания в виде выравниваний между исходным и сгенерированным текстом. Впрочем, важно отметить: недавние исследования показали, что выравнивания на основе внимания отнюдь не то же самое, что в классическом машинном переводе. Показатели выравнивания на основе внимания не говорят о синонимии переводов, как выравнивания между словами и фразами, но могут указывать на полезную для декодировщика информацию, например обращать внимание на подлежащее при генерации выходного сказуемого (см. статью Кёна, Ноулза [Koehn, Knowles, 2017]).

Две версии нашей модели отличаются по типу своего взаимодействия с целевым предложением. Первая использует имеющиеся целевые предложения в качестве входных данных на каждом из временных шагов в декодировщике. Вторая версия с помощью плановой выборки предоставляет модели возможность наблюдать ее собственные предсказания в качестве входных данных декодировщика. Преимущество этой версии — в возможности оптимизировать модель на основе ее собственной погрешности.

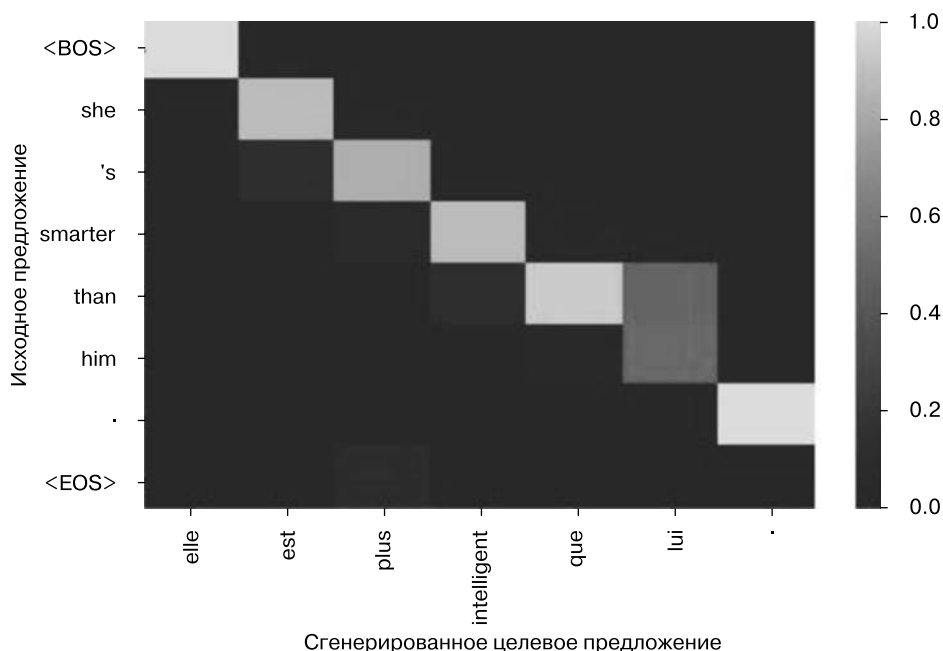
В табл. 8.1 приведены показатели BLEU. Важно не забывать, что для упрощения обучения мы выбрали облегченную версию стандартной задачи NMT, поэтому показатели кажутся выше, чем обычно встречаются в научной литературе. Хотя у второй модели, с плановой выборкой, показатель BLEU выше, их показатели довольно близки. Но что эти показатели значат на самом деле? Для выяснения этого необходимо выполнить качественную оценку модели.

**Таблица 8.1.** Показатели BLEU для двух показанных выше моделей; BLEU вычисляется как простое среднее значение пересечений 1-, 2-, 3- и 4-грамм

Название модели	Показатель BLEU
Модель без плановой выборки	46,8
Модель с плановой выборкой	48,1

Для углубленной оценки построим график показателей внимания, чтобы увидеть, дают ли они какую-либо информацию о выравнивании между исходными и целевыми предложениями. При этом оказывается, что между двумя моделями есть

разительное различие<sup>1</sup>. Рисунок 8.12 демонстрирует распределение вероятности внимания каждого из временных шагов декодировщика для модели с плановой выборкой. В ней веса внимания выстраиваются достаточно хорошо для предложения, выбранного из проверочного фрагмента набора данных.



**Рис. 8.12.** График матрицы весов внимания для модели с плановой выборкой как качественная оценка эффективности работы модели

## Резюме

Эта глава была посвящена генерации выходных предложений на основе обуславливающего контекста с помощью так называемых моделей контекстно обусловленной генерации. Модели, где обуславливающий контекст сам выявляется из другого предложения, называются моделями преобразования последовательностей в последовательности (S2S). Мы также обсудили, что S2S-модели — частный случай моделей типа «кодировщик-декодировщик». Мы рассмотрели структурные варианты описанных в главах 6 и 7 моделей последовательностей, а именно двунаправленные

<sup>1</sup> График для первой модели приводить не станем, поскольку там акцент делается только на завершающем состоянии RNN кодировщика. Как отмечают Кён и Ноулз (Koehn, Knowles, 2017), веса внимания свои для множества различных ситуаций. Мы подозреваем, что веса внимания в первой модели не зависят настолько от внимания потому, что необходимая для модели информация уже закодирована в состояниях GRU кодировщика.

модели, позволяющие выжать из последовательности максимум. Мы также научились использовать механизм внимания для эффективного захвата зависимостей из более «далекого» контекста. Наконец, поговорили об оценке моделей преобразования последовательностей в последовательности и рассмотрели соответствующий комплексный пример машинного перевода. До сих пор каждая глава книги была посвящена конкретной архитектуре нейронной сети. В следующей главе мы соберем изложенное во всех предыдущих главах воедино и рассмотрим примеры создания реальных систем путем сочетания различных архитектур моделей.

## Библиография

1. *Bengio Y., Simard P., Frasconi P.* Learning Long-Term Dependencies with Gradient Descent is Difficult // *IEEE Transactions on Neural Networks* 5. 1994.
2. *Bahdanau D., Cho K., Bengio Y.* Neural Machine Translation by Jointly Learning to Align and Translate // *Proceedings of the International Conference on Learning Representations*. 2015.
3. *Papineni K. et al.* BLEU: A Method for Automatic Evaluation of Machine Translation // *Proceedings of the 40th Annual Meeting of the ACL*. 2002.
4. *Daumé III H., Langford J., Marcu D.* Search-Based Structured Prediction // *Machine Learning Journal*. 2009.
5. *Bengio S. et al.* Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks // *Proceedings of NIPS*. 2015.
6. *Luong M.-T., Pham H., Manning C. D.* Effective Approaches to Attention-Based Neural Machine Translation // *Proceedings of EMNLP*. 2015.
7. *Le P., Zuidema W.* Quantifying the Vanishing Gradient and Long Distance Dependency Problem in Recursive Neural Networks and Recursive LSTMs // *Proceedings of the 1st Workshop on Representation Learning for NLP*. 2016.
8. *Koehn P., Knowles R.* Six Challenges for Neural Machine Translation // *Proceedings of the 1st Workshop on Neural Machine Translation*. 2017.
9. *Neubig G.* Neural Machine Translation and Sequence-to-Sequence Models: A Tutorial. 2017. arXiv: 1703.01619.
10. *Vaswani A. et al.* Attention is all you need // *Proceedings of NIPS*. 2017.

# 9

## Классические методы и перспективные направления

В этой главе мы взглянем на предыдущие главы в целом и покажем, как связаны различные, вроде бы не зависящие друг от друга темы и как исследователи могут сочетать и комбинировать описанные идеи для решения имеющихся задач. Мы также вкратце резюмируем кое-какие классические вопросы NLP, для подробного обсуждения которых не хватило места в этой книге. Наконец мы укажем на перспективные направления в данной сфере, по состоянию на 2018 год.

### Какие темы мы уже изучили

Мы начали с парадигмы обучения с учителем и использования абстракции графа вычислений для кодирования сложных идей в виде модели, обучаемой путем обратного распространения ошибок (backpropagation). В качестве вычислительной платформы выбрали фреймворк PyTorch. В главе 2 мы познакомили вас с основными понятиями NLP и лингвистики, заложив фундамент для остальной книги. Для последующих глав вам также пригодились такие базовые понятия из главы 3, как функции активации, функции потерь, градиентная оптимизация для обучения с учителем и цикл обучения-оценки. Мы разобрали два примера упреждающих нейронных сетей — многослойный перцептрон и сверточную нейронную сеть. Научились использовать механизмы регуляризации, такие как L1/L2-нормализация и дропаут для повышения ошибкоустойчивости сетей. Многослойные перцептроны могут захватывать в скрытых слоях  $n$ -граммподобные связи, но делают это не слишком эффективно. Сверточные нейронные сети, с другой стороны, усваивают подобные субструктуры достаточно эффективно с вычислительной точки зрения благодаря совместному использованию параметров.

В главе 6 мы увидели, как рекуррентные нейронные сети могут захватывать «далекие» по времени зависимости при небольшом количестве параметров. Можно сказать, что сверточные нейронные сети совместно используют параметры в пространстве, а рекуррентные нейронные сети — во времени. Мы рассмотрели три варианта рекуррентных нейронных сетей, начиная с RNN Элмана и заканчивая шлюзовыми вариантами, такими как сети с долгой краткосрочной памятью (LSTM) и шлюзовые рекуррентные блоки (GRU). Мы также продемонстрировали использование рекуррентных сетей для предсказания и маркирования последовательностей, где на каждом временном шаге входных данных предсказываются выходные данные. Наконец, мы познакомили вас с таким классом моделей, как модели типа «кодировщик-декодировщик», и в качестве их примера рассмотрели модель преобразования последовательностей в последовательности (S2S), предназначенную для решения задач контекстно обусловленной генерации, например машинного перевода. В большинстве случаев мы привели комплексные примеры на PyTorch.

## Вечные вопросы NLP

NLP — куда более обширная тема, чем можно охватить в рамках одной книги, и данная книга не исключение. В главе 2 мы описали некоторые базовые термины и задачи NLP. В оставшихся главах мы постарались рассмотреть множество задач NLP, но далее мы вкратце упомянем кое-какие важные вопросы, которые не вошли в предыдущие главы из-за ограничений сферы охвата книги.

## Диалоговые и интерактивные системы

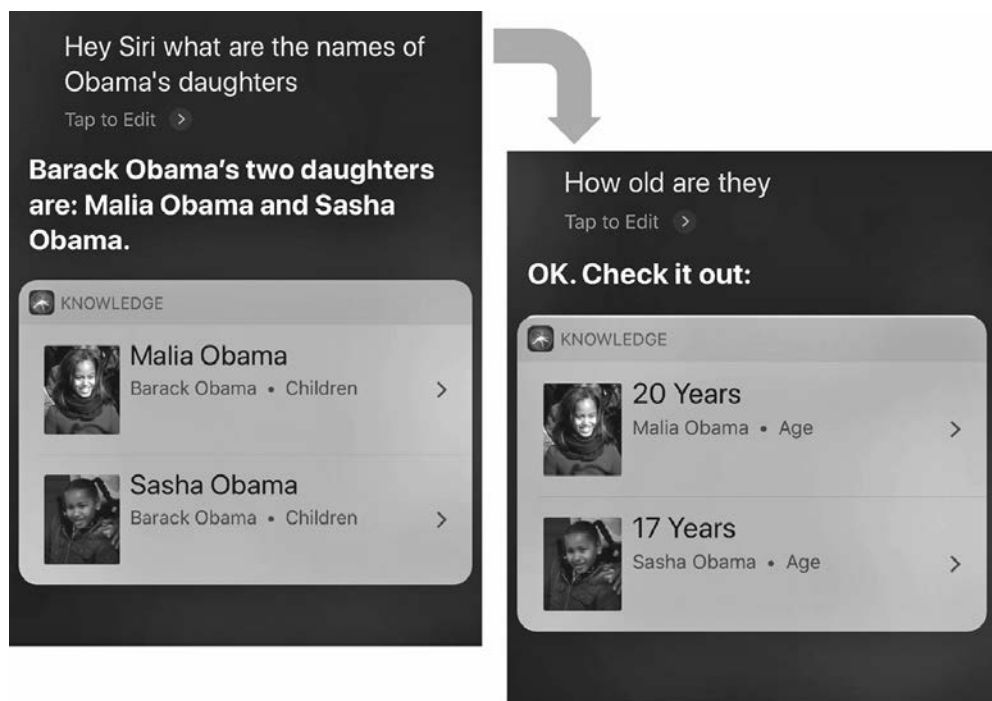
Организация беспрепятственного диалога человека с компьютером — заветная цель вычислительной техники, вдохновившая на создание теста Тьюринга и премии Лебнера. С первых дней работы над искусственным интеллектом NLP связывалась с диалоговыми системами и встречалась в массовой культуре под видом таких вымышленных систем, как главный компьютер на борту звездолета «Энтерпрайз» в сериале «Звездный путь» и ЭАЛ 9000 в фильме «2001 год: Космическая одиссея»<sup>1</sup>. Проектирование диалоговых и вообще интерактивных систем — плодотворная сфера исследований, о чем свидетельствует успех таких недавно вышедших программных продуктов, как Alexa от компании Amazon, Siri от Apple и Assistant от Google. Диалоговые системы могут быть без ограничения

<sup>1</sup> Вообще говоря, ЭАЛ — это не просто диалоговая система с эмоциями, обладающая самосознанием, но в данном случае нас интересует именно диалоговая составляющая. См. эпизод 9 сезона 2 сериала «Звездный путь: Следующее поколение» о сознании ботов.

предметной области (спрашивай, что хочешь) или с ограниченной предметной областью (например, для бронирования авиабилетов или управления автомобилем).

В числе важных тем исследования в этой сфере можно назвать следующие.

- ❑ Моделирование диалоговых действий, контекста (рис. 9.1) и состояния диалога.
- ❑ Моделирование комбинированных диалоговых систем (скажем, с входными данными в виде речевого и зрительного сигналов или текста и зрительного сигнала).
- ❑ Распознавание системой намерений пользователя.
- ❑ Моделирование предпочтений пользователя и генерация индивидуальных ответов для конкретного пользователя.
- ❑ Очеловечивание ответов системы. Например, современные промышленные диалоговые системы начали вставлять в свои ответы слова-паразиты вроде «кхм» и «э-э», чтобы казаться менее роботоподобными.



**Рис. 9.1.** Диалоговая система в действии (Siri от компании Apple). Обратите внимание, как система учитывает контекст при ответе на последующие вопросы: а именно, понимает, что they относится к дочерям Барака Обамы

# Дискурс

Дискурс подразумевает понимание соотношений «часть — целое» в текстовых документах. Например, задача анализа дискурса включает выяснение взаимосвязи двух предложений в контексте. Для иллюстрации приведем в табл. 9.1 несколько примеров из банка синтаксических деревьев Пенсильванского университета (Penn Discourse Treebank, PDTB).

Таблица 9.1. Примеры из задачи поверхностного анализа дискурса (CoNLL 2015)

Пример	Дискурсивное соотношение
Официальные представители GM хотели воплотить стратегию уменьшения производственных мощностей и персонала еще <b>до начала этих разговоров</b>	Временное.Асинхронное. Приоритет
Но этому призраку недостаточно слов, ему нужны деньги и люди, причем много. <b>Так что господин Картер сформировал три новых армейских подразделения и передал их в подчинение новым бюрократам из Тампы — Силам быстрого развертывания</b>	Непредвиденные_обстоятельства. Причина.Следствие
У арабов была всего лишь нефть. Неявное <sup>1</sup> =в то время <b>как</b> Эти земледельцы контролировали самое сердце мира	Сравнение.Противопоставление

Для понимания дискурса необходимо решить и другие задачи вроде *разрешения анафор* и *обнаружения метонимии*. При разрешении анафор нужно сопоставить вхождения местоимений и сущности, к которым они относятся. Это может оказаться непростой задачей, как видно из рис. 9.2<sup>2</sup>.

- (a) The dog chewed the bone. It was delicious.
- (b) The dog chewed the bone. It was a hot day.
- (c) Nia drank a tall glass of beer. It was chipped.
- (d) Nia drank a tall glass of beer. It was bubbly.

**Рис. 9.2.** Некоторые проблемы разрешения анафор. К чему относится It в примере (a) — к dog или bone? В примере (b) It не относится ни к той ни к другой сущности. В примерах (c) и (d) It относится к glass и beer соответственно. Знание о том, что beer (пиво) бывает bubbly (пенистым), просто необходимо при разрешении подобных референтов (предпочтений сочетаемости)

<sup>1</sup> Соединительное слово. В оригинале Implicit=while. — Примеч. пер.

<sup>2</sup> Более подробное обсуждение подобных задач читатель может найти в статье The Winograd Schema Challenge (<https://cs.nyu.edu/faculty/davise/papers/WinogradSchemas/WS.html>).

Референты могут также быть метонимами, как видно из следующего примера:

*Пекин ввел торговые пошлины в ответ на пошлины на китайские товары.*

Под Пекином здесь понимается не город, а китайское правительство. Иногда для успешного разрешения референтов требуется определенная работа с базой знаний.

## Извлечение информации и интеллектуальный анализ текста

Одна из часто встречающихся в промышленности категорий задач связана с извлечением информации. Как извлечь из текста сущности (имена людей, названия товаров и т. д.), события и связи? Как соотнести упоминания сущностей в тексте с записями из базы знаний (так называемое обнаружение сущностей (entity discovery), связывание сущностей (entity linking) или заполнение слотов (slot-filling))<sup>1</sup>? Как для начала создать и поддерживать подобную базу знаний (как ее наполнить)? Лишь на немногие из этих вопросов были даны исчерпывающие ответы в исследовательских статьях, посвященных извлечению информации в различных контекстах.

## Анализ и информационный поиск документов

Еще одна обширная категория промышленных задач NLP включает анализ объемных подборок документов. Как извлечь темы из документа (моделирование тем)? Как более интеллектуально выполнить индексацию и поиск документов? Как понять смысл поискового запроса (синтаксический разбор запросов)? Как генерировать краткие резюме для больших наборов документов?

Область применимости методик NLP весьма широка, и на самом деле их можно применять везде, где речь идет о неструктурированных или частично структурированных данных. В качестве примера мы отсылаем вас к статье Дилл и др. (Dill et al., 2007), где методики синтаксического разбора естественного языка применяются для анализа сворачивания белка.

## Перспективные направления в NLP

Писать раздел «Перспективные направления в NLP», когда эта сфера так быстро развивается, может показаться бесполезной затеей. Впрочем, хотелось бы дать вам хотя бы одним глазком взглянуть на современные тенденции по состоянию на осень 2018 года.

- *Анализ классической литературы по NLP с точки зрения парадигмы дифференцируемого обучения.* NLP исполнилось уже несколько десятилетий, а глубокое

<sup>1</sup> Больше подробностей об этих задачах можно найти по адресу <https://tac.nist.gov/2018/SM-KBP/>.



обучение существует всего несколько лет. Множество новых статей посвящено изучению традиционных методик и задач с точки зрения новой парадигмы глубокого (дифференцируемого) обучения. При чтении классических статей (а мы рекомендуем вам их читать!) по NLP имеет смысл задаться вопросом: чего авторы статьи хотят достичь? Каковы входные/выходные представления? Как можно упростить решение задачи, применяя обсуждавшиеся в предыдущих главах методики?

- ❑ *Композиционность моделей.* В этой книге мы обсуждали различные виды архитектур глубокого обучения для NLP: MLP, CNN, модели последовательностей, модели преобразования последовательностей в последовательности и модели на основе механизма внимания. Важно отметить, что хотя мы изучали все эти модели по отдельности, но делали это исключительно в педагогических целях. Одна из прослеживающихся сейчас в литературе тенденций — композиция различных архитектур для решения имеющейся задачи. Например, можно написать сверточную нейронную сеть, работающую с символами слов, с последующей LSTM поверх этого представления, а завершающую классификацию кодировки LSTM выполнить с помощью MLP. Возможность сочетания различных архитектур в зависимости от того, что нужно для решения конкретной задачи, — одна из самых замечательных идей глубокого обучения.
- ❑ *Использование операций свертки для последовательностей.* Одна из недавних тенденций в моделировании последовательностей — моделирование последовательности целиком с помощью операций свертки. В качестве примера полностью сверточной модели машинного перевода см. статью Геринга и др. (Gehring et al., 2018). Шаг декодирования включает операцию обращения свертки. Преимущество такого решения состоит в значительном ускорении обучения с помощью полностью сверточной модели.
- ❑ *Вам нужно лишь внимание.* Еще одна тенденция — замена операций свертки механизмом внимания (см. статью Васвани и др. [Vaswani et al., 2017]). При использовании механизма внимания, особенно таких его вариантов, как аутовнимание и мультивнимание, можно успешно захватывать «далекие» зависимости, обычно моделируемые с помощью RNN или CNN.
- ❑ *Перенос обучения.* Перенос обучения представляет собой обучение представлением для одной задачи и использование этих представлений для повышения эффективности обучения в другой задаче. Благодаря новой волне популярности нейронных сетей и глубокого обучения в NLP методики переноса обучения с помощью предобученных векторов слов стали применяться повсеместно. Недавние работы Рэдфорда, Петерса и др. (Radford et al., 2018; Peters et al., 2018) демонстрируют полезность представлений без учителя, усвоенных для задач моделирования языков, в различных задачах NLP. Сюда можно отнести составление ответов на вопросы, классификацию, определение сходства предложений и формирование рассуждений на естественном языке.

Кроме того, *обучение с подкреплением* (reinforcement learning) в последнее время успешно применяется в задачах, связанных с диалогами, а моделирование

с применением баз памяти и знаний для сложных задач на логическое мышление на естественном языке вызывает особенный интерес у исследователей, как теоретиков, так и практиков. В следующем разделе мы перейдем от классических тематик и перспективных направлений к более насущной задаче — проектированию промышленных NLP-систем.

## Паттерны проектирования для промышленных NLP-систем

Промышленные NLP-системы могут быть очень сложными. При создании NLP-системы важно помнить, что она представляет собой просто средство для решения какой-либо задачи и не более. При создании системы у инженеров, исследователей, архитекторов и ответственных за выпуск программного продукта есть несколько альтернатив. Хотя наша книга посвящена в основном конкретным методам и базовым строительным блокам, чтобы собрать их вместе в сложную структуру, удовлетворяющую вашим запросам, необходимо мыслить в терминах паттернов. Кроме того, нужен язык для описания этих паттернов<sup>1</sup>. Эта методология популярна во многих дисциплинах (см. книгу Александра [Alexander, 1979]), включая инженерию разработки ПО.

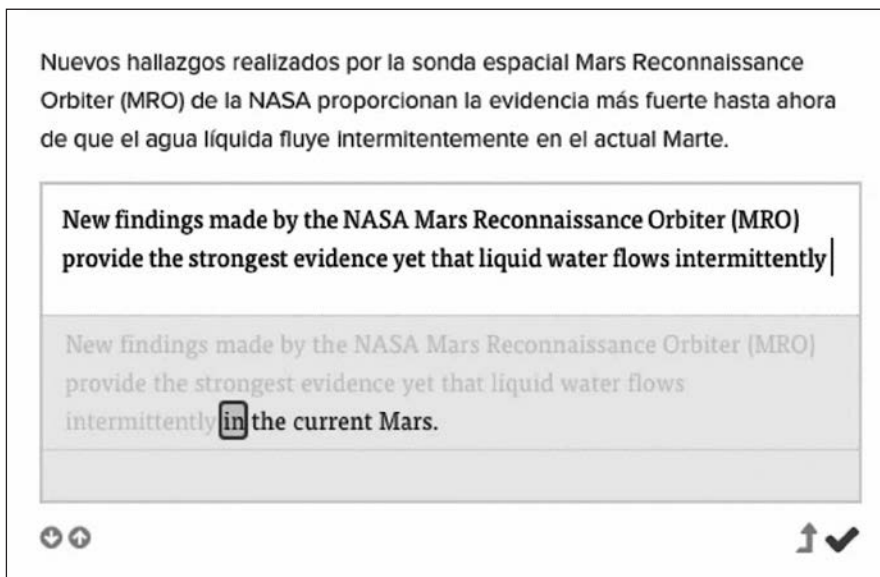
В данном разделе мы опишем несколько распространенных паттернов проектирования и развертывания промышленных NLP-систем. Разработчикам часто приходится выбирать одну из альтернатив (идти на компромиссы) для согласования разработки продукта с техническими, бизнес-, стратегическими и эксплуатационными целями. Мы изучим эти архитектурные альтернативы в шести измерениях.

- *Онлайновые и офлайновые системы.* Онлайн-системы (online systems) — такие, в которых предсказания модели производятся в режиме реального (или почти реального) времени. Для некоторых задач, таких как борьба со спамом и модерация контента, по своей природе необходимы онлайн-системы. Офлайн-системы (offline systems), с другой стороны, не требуют работы в реальном времени. Им достаточно эффективно работать при пакетных входных данных, пользуясь преимуществами таких подходов, как *трансдуктивное обучение* (transductive learning). Некоторые онлайн-системы могут быть реактивными и даже выполнять обучение в онлайн-стиле (так называемое *онлайновое обучение*), но многие из онлайн-систем создаются и разворачиваются на основе офлайн-модели, которую пришлось отправить в промышленную эксплуатацию. Системы на основе онлайн-обучения особенно чувствительны к враждебной среде. Недавний пример — (печально) известный чат-бот Tay ([https://en.wikipedia.org/wiki/Tay\\_\(bot\)](https://en.wikipedia.org/wiki/Tay_(bot))) в соцсети Twitter, выпешедший из-под контроля и начавший обучаться

<sup>1</sup> Язык описания паттернов ([https://en.wikipedia.org/wiki/Pattern\\_language](https://en.wikipedia.org/wiki/Pattern_language)) — это «метод описания рекомендуемых проектных решений (паттернов) в рамках своей области знаний».

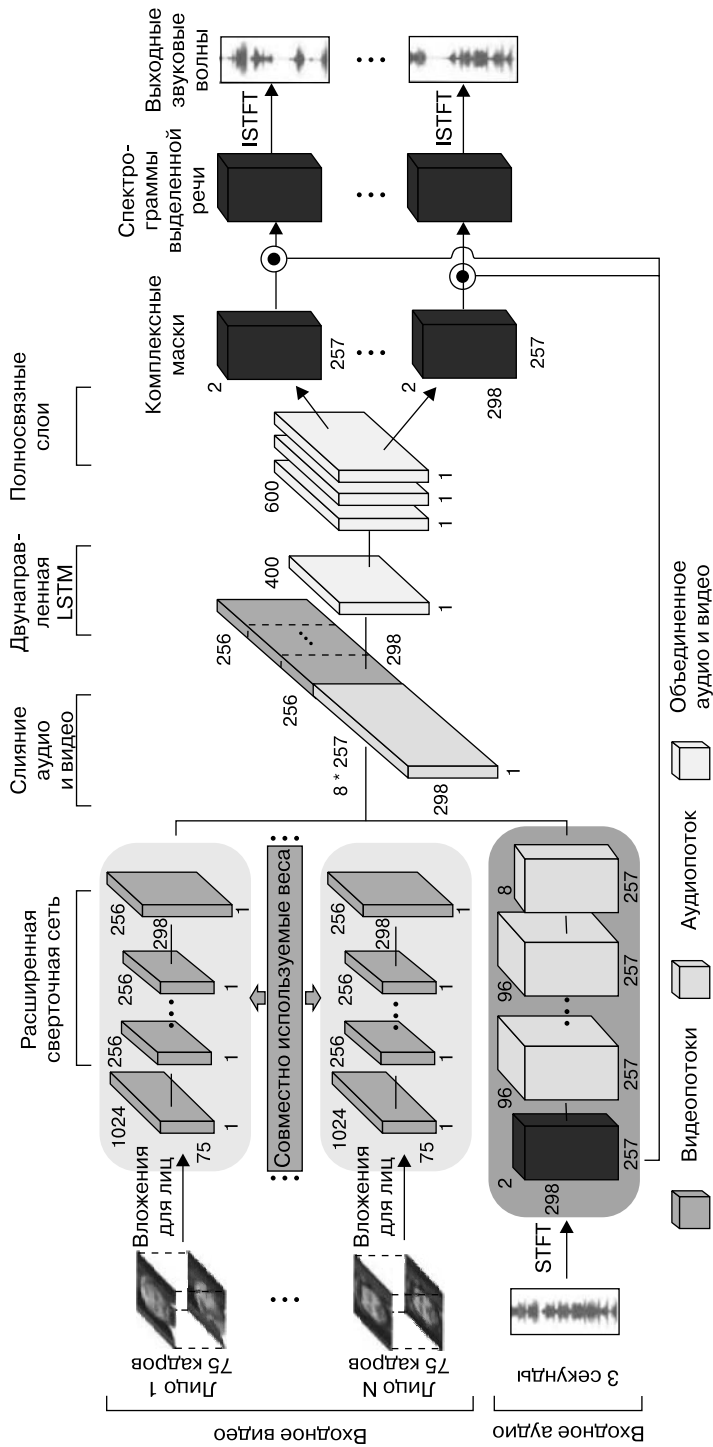


полезным человек. Рисунок 9.4 демонстрирует пример интерактивного интерфейса машинного перевода от компании Lilt Inc., где в предсказаниях в рамках так называемых моделей со смешанной инициативой (см. статью Грина [Green, 2014]) участвуют как люди, так и модели. Разработка интерактивных систем не проста, но за счет включения в цикл человека может достигаться очень высокая точность.



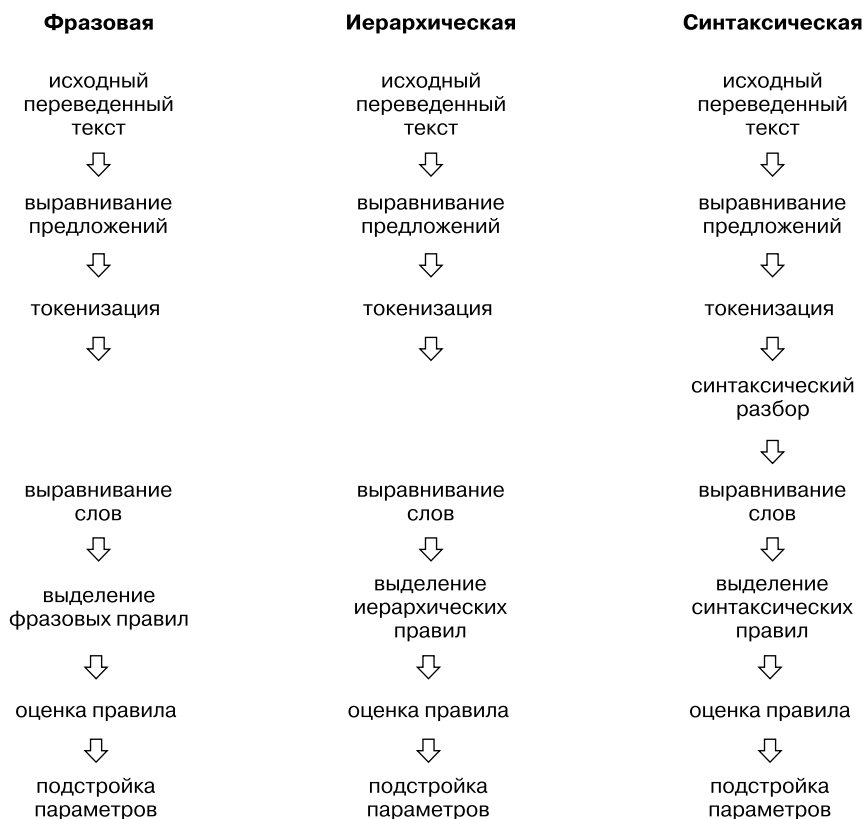
**Рис. 9.4.** Работа модели машинного перевода с участием в цикле человека: человек может исправлять или перефразировать предлагаемые системой машинного перевода варианты, в результате чего получаются высококачественные переводы (рисунок любезно предоставлен компанией Lilt Inc.)

- ❑ **Однородные и комбинированные системы.** Во многих случаях в процессе обучения и предсказания полезно использовать несколько типов входных данных. Например, для системы текстовой расшифровки новостей на входе будет полезен не только аудиопоток, но и видеоряд. Например, в недавней статье от Google под названием «Смотреть, чтобы слышать» Эфрата и др. (Ephrat et al., 2018) используются комбинированные входные данные для решения непростой задачи выделения источника голоса (она же «задача вечеринки») (рис. 9.5). Создание и развертывание комбинированных систем — недешевое удовольствие, но в сложных задачах сочетание нескольких типов входных данных обеспечивает информацию, которую невозможно получить из какого-либо одного типа входных данных. В NLP также встречаются подобные примеры. Например, при комбинированном переводе его качество можно улучшить за счет входных данных на нескольких исходных языках, если они есть в наличии. При генерации тем для веб-страниц (тематическое моделирование) можно использовать признаки, извлеченные из содержащихся в них изображений вдобавок к тексту на веб-странице.



**Рис. 9.5.** Комбинированная система, в которой совместно используются как аудио-, так и видеопризнаки для решения сложной задачи, например «задачи вечеринки» (рисунок из статьи Эфрата и др. [Ephrat et al., 2018])

- *Комплексные и фрагментарные системы.* С момента появления глубокого обучения при создании сложных систем NLP у исследователей и инженеров появился выбор между еще двумя вариантами: конвейером из различных блоков или монолитной комплексной системой. Комплексная архитектура предпочтительнее во многих сферах (таких как машинный перевод, автоматическое реферирование и распознавание речи), где тщательно спроектированная комплексная система может существенно снизить сложность реализации и развертывания и уж наверняка уменьшить количество строк кода. Фрагментарные системы (рис. 9.6) разбивают сложную задачу NLP на подзадачи, оптимизируемые по отдельности, независимо от цели итоговой задачи. Разбиение на подзадачи делает фрагментарные системы высокомодульными и обеспечивает удобство исправления конкретных ошибок в промышленной эксплуатации, но обычно для него характерен некоторый технический долг.



**Рис. 9.6.** Конвейеры обучения традиционных систем машинного перевода рассматривают задачу как последовательность подзадач, каждую со своей моделью. Сравните это с моделью нейронного машинного перевода, обсуждавшейся в разделе «Пример: нейронный машинный перевод» на с. 215 (рисунок любезно предоставлен Х. Хоангом [Hoang et al., 2009])

- ❑ *Системы с ограниченной и неограниченной предметной областью.* Системы с ограниченной предметной областью оптимизированы явным образом для одной цели: демонстрации хороших результатов в конкретной предметной области. Например, система машинного перевода может быть оптимизирована под работу с биомедицинскими журналами — для этого потребуется не просто обучение на корпусе биомедицинских параллельных текстов. Системы без ограничения предметной области, с другой стороны, носят универсальный характер (например, Google Translate). Еще один пример: система маркировки документов. Если система выполняет предсказания только в рамках одного из заранее заданных классов (типичный случай), значит, ее предметная область ограничена. Но если система спроектирована так, чтобы выявлять новые классы во время работы, то это система с неограниченной предметной областью. В контексте перевода и систем распознавания речи системы с ограниченной предметной областью называются также системами «с ограниченным словарем».
- ❑ *Одноязычные и многоязычные системы.* Системы NLP, нацеленные на работу с одним языком, называются *одноязычными* системами (monolingual systems). Создание и оптимизация одноязычной системы — задача несложная. *Многоязычные* же системы (multilingual systems), напротив, способны работать с несколькими языками. Ожидается, что такая система будет сразу готова к работе на новом языке после обучения на соответствующем наборе данных. Хотя создать многоязычную систему заманчиво, у одноязычной версии есть свои преимущества. У исследователей и инженеров появляется возможность воспользоваться доступными ресурсами и знаниями предметной области конкретного языка для создания высококачественной системы, достигая при этом невозможных для универсальной многоязычной системы результатов. Поэтому многоязычные системы часто реализуются в виде набора оптимизированных по отдельности одноязычных систем и компонента, выполняющего распознавание языка и переадресацию входных данных соответствующей системе.

## Библиография

1. Alexander C. The Timeless Way of Building. Oxford University Press, 1979.
2. Dill K. A. et al. Computational linguistics: A New Tool for Exploring Biopolymer Structures and Statistical Mechanics // Polymer, 48. 2007.
3. Hoang H., Koehn P., Lopez A. A Unified Framework for Phrase-Based, Hierarchical, and Syntax-Based Statistical Machine Translation // Proceedings of IWSLT. 2009.
4. Stein T., Chen E., Mangla K. Facebook Immune System // SNS. 2011.

5. *Green S.* Mixed-Initiative Language Translation // PhD thesis. Stanford University. 2014.
6. *Vaswani A. et al.* Attention Is All You Need // Proceedings of NIPS. 2017.
7. *Ephrat A. et al.* Looking to Listen: A Speaker-Independent Audio-Visual Model for Speech Separation // SIGGRAPH. 2018.
8. *Peters M. E. et al.* Deep Contextualized Word Representations // Proceedings of NAACL-HLT. 2018.



# Что читать дальше

Работа с таким перспективным фреймворком, как PyTorch, в такой быстро меняющейся сфере, как глубокое обучение, напоминает строительство дома на сдвигающемся грунте. В этом разделе мы перечислим некоторые ресурсы, посвященные глубокому обучению, PyTorch и NLP, — они помогут вам укрепить заложенный в этой книге фундамент.

В этой книге мы не охватили, конечно, все возможности фреймворка PyTorch. Рекомендуем вам обратиться к великолепной документации PyTorch и заглянуть на его форум для продолжения практической работы с ним.

- ❑ Документация PyTorch (<https://pytorch.org/docs>).
- ❑ Форум PyTorch (<https://discuss.pytorch.org/>).

В самой сфере глубокого обучения также наблюдается активная деятельность, как с промышленной стороны, так и в научных кругах. Большинство работ по глубокому обучению можно найти в различных разделах arXiv.

- ❑ Машинное обучение (<https://arxiv.org/list/cs.LG/recent>).
- ❑ Язык и вычисления (<https://arxiv.org/list/cs.CL/recent>).
- ❑ Искусственный интеллект (<https://arxiv.org/list/cs.AI/recent>).

Лучший способ не отстать от последних разработок в NLP — посещать научные конференции.

- ❑ Ассоциация вычислительной лингвистики (Association of Computational Linguistics, ACL).
- ❑ Вычислительные методы обработки данных на естественных языках (Empirical Methods in Natural Language Processing, EMNLP).
- ❑ Ассоциация вычислительной лингвистики Северной Америки (North American Association for Computational Linguistics, NAACL).
- ❑ Европейский филиал ACL (EACL).

- ❑ Конференция по вычислительным методам машинного обучения естественным языкам (Conference on Computational Natural Language Learning, CoNLL).

Рекомендуем вам следить на сайте [aclweb.org](http://aclweb.org) за актуальными трудами этих и других конференций, семинаров и важными новостями в сфере NLP.

Когда будете готовы перейти от основ к более продвинутым вопросам, вы, возможно, начнете читать научные статьи. Это своего рода искусство. Советы относительно чтения статей по NLP вы можете найти в статье Джейсона Эйснера (Jason Eisner): <https://www.cs.jhu.edu/~jason/advice/how-to-read-a-paper.html>.

И наконец, учебные материалы, дополняющие содержимое этой книги, вы можете найти в нашем репозитории GitHub (<https://nlproc.info/PyTorchNLPBook/repo/>).

## Об авторах

**Делип Рао** — основатель консалтинговой компании Joostware из Сан-Франциско, специализирующейся на машинном обучении и исследованиях в области NLP. Один из соучредителей Fake News Challenge — инициативы, призванной объединить хакеров и исследователей в области ИИ над задачами проверки фактов в СМИ. Ранее Делип занимался связанными с NLP исследованиями и программными продуктами в компаниях Twitter и Amazon (Alexa).

**Брайан Макмахан** — научный сотрудник в компании Wells Fargo, занимающийся преимущественно NLP. Ранее работал в компании Joostware.

## Об иллюстрации на обложке

Изображенная на обложке птица — тайваньский королек (*Regulus goodfellowi*), живущий в хвойных лесах высокогорий Тайваня. Это самая маленькая из местных тайваньских птиц. Как по манере пения, так и по внешнему виду и повадкам она является близким родственником аналогичных видов рода королек из материковой Азии. Своим видовым названием, *goodfellowi*, она обязана британскому коллекционеру дикой фауны и орнитологу Волтеру Гудфеллоу (Walter Goodfellow), первым описавшему ее.

Тайваньский королек — насекомоядная птица. Всего 7–12 сантиметров в длину и весом около семи граммов, он неустанно порхает между деревьями, перепрыгивая с ветки на ветку и добывая мелких насекомых. В верхней части головы тайваньского королька — черная кайма с оранжево-желтым пятном на макушке; вокруг глаз — белая маска; по бокам корольки желтые, с желтовато-зелеными крыльями. Пятно оранжевых перьев у самцов больше, и, когда они волнуются при спорах за территорию или самку, эти перья становятся дыбом, превращаясь в необычный огненный хохолок, по которому птица и получила свое английское название *flamecrest*. Тайваньский королек не мигрирует, оставаясь на Тайване круглый год, только перебирается с одной горы на другую, в зависимости от поры года. Брачные повадки этих птиц пока плохо изучены, вероятно, из-за удаленности их горных мест обитания.

Хотя тайваньский королек — распространенная в своем ареале птица, которая не находится под угрозой исчезновения, ареал ее обитания ограничивается горами Тайваня, где защищается местными законами. Охраняемые законом области в горах (одни тайваньские национальные парки занимают около 7500 квадратных километров) — излюбленное место не только птиц, но и пеших туристов и альпинистов.

Многие из животных на обложках O'Reilly находятся под угрозой исчезновения; все они важны для нашего мира. Чтобы узнать, каким может быть ваш личный вклад в их спасение, зайдите на сайт [animals.oreilly.com](http://animals.oreilly.com).

Рисунок на обложке выполнен Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из *British Birds*.

*Брайан Макмахан, Делип Рао*

**Знакомство с PyTorch: глубокое обучение при обработке  
естественного языка**

Перевела с английского *И. Пальти*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>С. Давид</i>
Ведущий редактор	<i>Н. Гринчик</i>
Научный редактор	<i>Е. Черских</i>
Художественный редактор	<i>А. Михеева</i>
Корректоры	<i>О. Андриевич, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,  
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные  
профессиональные, технические и научные.

Импортёр в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 03.10.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: [www.chpk.ru](http://www.chpk.ru). E-mail: [marketing@chpk.ru](mailto:marketing@chpk.ru)

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87



**ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу**

**Заказать книги оптом можно в наших представительствах**

### **РОССИЯ**

**Санкт-Петербург:** м. «Выборгская», Б. Сампсониевский пр., д. 29а  
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

**Москва:** м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж  
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

**Воронеж:** тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

**Екатеринбург:** ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;  
e-mail: office@ekat.piter.com; skype: ekat.manager2

**Нижний Новгород:** тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

**Ростов-на-Дону:** ул. Ульяновская, д. 26  
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

**Самара:** ул. Молодогвардейская, д. 33а, офис 223  
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,  
pitvolga@samara-ttk.ru

### **БЕЛАРУСЬ**

**Минск:** ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;  
e-mail: og@minsk.piter.com

**Издательский дом «Питер» приглашает к сотрудничеству авторов:**  
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com  
Подробная информация здесь: <http://www.piter.com/page/avtoru>

**Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок:** тел./факс: (812) 703-73-73; e-mail: sales@piter.com

---

**Заказ книг для вузов и библиотек:**  
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

**Заказ книг по почте:** на сайте [www.piter.com](http://www.piter.com); тел.: (812) 703-73-74, доб. 6216;  
e-mail: books@piter.com

---





**Вопросы по продаже электронных книг:** тел.: (812) 703-73-74, доб. 6217;  
e-mail: kuznetsov@piter.com



**ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»  
МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:**

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

**ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:**

-  Наложением платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

**ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:**

- Псылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте [www.piter.com](http://www.piter.com)).
- Можно оформить доставку заказа через почтоматы, (адреса почтоматов можно узнать на нашем сайте [www.piter.com](http://www.piter.com)).

**ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:**

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

**БЕСПЛАТНАЯ ДОСТАВКА:**

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

## ВАША УНИКАЛЬНАЯ КНИГА

*Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.*

### МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

### **Почему надо выбрать именно нас:**

*Издательству «Питер» более 20 лет. Наш опыт — гарантия высокого качества.*

### **Мы предлагаем:**

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

### **Обеспечим продвижение вашей книги:**

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

*Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.*

*Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.*

*Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.*

*Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF — самых популярных и надежных форматах на сегодняшний день.*

### **Свяжитесь с нами прямо сейчас:**

**Санкт-Петербург** – Анна Титова, (812) 703-73-73, [titova@piter.com](mailto:titova@piter.com)  
**Москва** – Сергей Клебанов, (495) 234-38-15, [klebanov@piter.com](mailto:klebanov@piter.com)