O'REILLY®

Знакомство с PyTorch

ГЛУБОКОЕ ОБУЧЕНИЕ ПРИ ОБРАБОТКЕ ЕСТЕСТВЕННОГО ЯЗЫКА

Брайан Макмахан
Делип Рао

ПИТЕР®

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

ISBN 978-1491978238 англ.

ISBN 978-5-4461-1241-8

Краткое содержание

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[1] 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*

[1] 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[1], а наблюдаемой величиной — сам документ. В сфере машинного перевода наблюдаемой величи-

[1] 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.

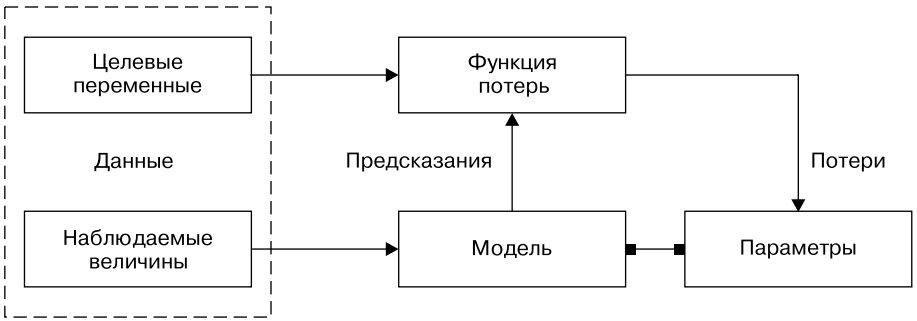

Рис. 1.1. Парадигма обучения с учителем — концептуальная база для обучения на маркированных входных данных

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[1]. 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 представлениям. Все эти представления на основе подсчетов начинаются с вектора фиксированной размерности.

[1] 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.

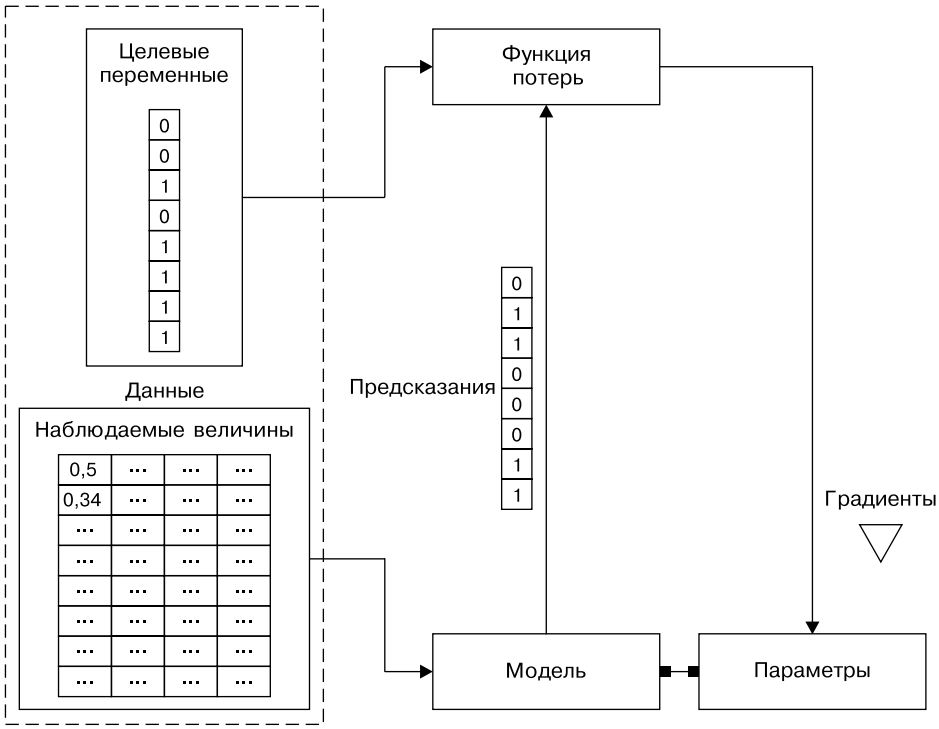

Рис. 1.2. Кодирование наблюдаемых величин и целевых переменных: целевые переменные и наблюдаемые величины с рис. 1.1 представлены здесь численно, в виде векторов или тензоров. Все это в совокупности называется кодированием входных данных

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 × 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_{time}	1	0	0	0	0	0	0	0
1_{fruit}	0	1	0	0	0	0	0	0
1_{flies}	0	0	1	0	0	0	0	0
1_{like}	0	0	0	1	0	0	0	0
1_{a}	0	0	0	0	1	0	0	0
1_{an}	0	0	0	0	0	1	0	0
1_{arrow}	0	0	0	0	0	0	1	0
1_{banana}	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 проницательный 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 2'])
```

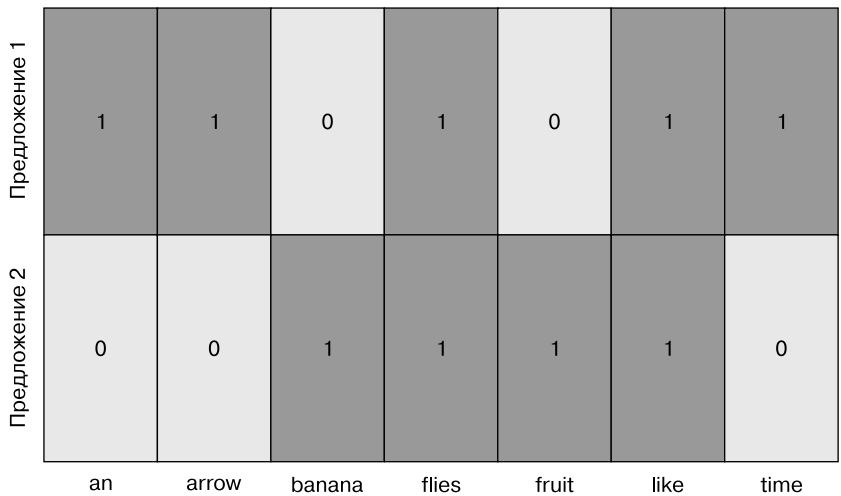

Рис. 1.4. Свернутое унитарное представление, сгенерированное в примере 1.1

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 нескольку 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») попадается не так часто, но, вероятно, ясно свидетельствует о сущности патентного документа, то имеет смысл присвоить ему больший вес в представлении. Эвристический алго-

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 терма 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[1]. 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.

[1] «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[1]. 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.

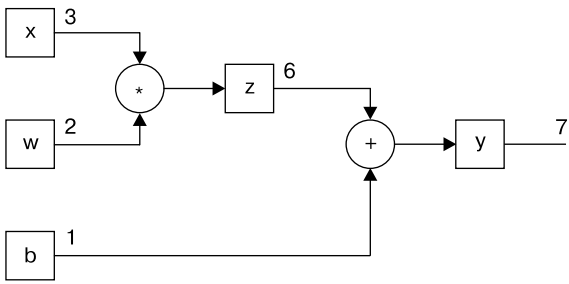

Рис. 1.6. Представление выражения y = wx + b с помощью графа вычислений

[1] Сеппо Линнайнмаа (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.

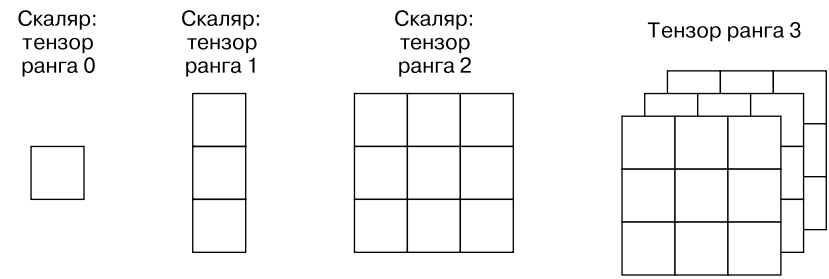

Рис. 1.7. Тензоры как обобщение многомерных массивов

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[1]. 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 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.

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))
```

[1] 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[1], 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

[1] 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
npy = np.random.rand(2, 3)
describe(torch.from_numpy(npy))
```

```
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 свойство `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[1]

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

[1] 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))
```