БИБЛИОТЕКИ ЯЗЫКА PYTHON ДЛЯ КОМПЬЮТЕРНЫХ ВЫЧИСЛЕНИЙ И МОДЕЛИРОВАНИЯ

Библиотеки **Python** для машинного обучение (Ч. 1)

Лобанов Алексей Владимирович
Главный специалист отдела разработки и внедрения АИС,
Ассистент кафедры информационных компьютерных технологий РХТУ им. Д.И. Менделеева

ПОЛЕЗНЫЕ РЕСУРСЫ

- Библиотека TensorFlow
- Библиотека Keras
- Библиотека Theano
- Библиотека Scikit-learn
- Библиотека PyTorch
- Библиотека NumPy
- Библиотека Pandas

ТЕМЫ

- Библиотеки ML
- Основы глубокого обучения
- Основы нейросетей

МАШИННОЕ ОБУЧЕНИЕ (MACHINE LEARNING, ML)

Машинное обучение (Machine Learning, ML) и в целом искусственный интеллект (Artificial Intelligence, AI) все шире распространяются в различных сферах деятельности, и многие предприятия начинают активно инвестировать в эти технологии.

ML проекты сильно отличаются от обычных проектов разработки программного обеспечения. При работе над ними используется другой технологический стек, нужны навыки машинного обучения и готовность заниматься глубокими исследованиями. Python как раз один из таких языков, и не удивительно, что на нем ведется большое количество ML проектов.

+ встроенные библиотеки;
+ пологая кривая изучения;
+ простота интеграции;
+ легкость в создании прототипов;
+ открытый код;
+ объектно-ориентированная парадигма;
+ переносимость;
+ высокая производительность;
+ платформонезависимость.

БИБЛИОТЕКА TENSORFLOW

TensorFlow — библиотека сквозного машинного обучения Python для выполнения высококачественных численных вычислений. С помощью TensorFlow можно построить глубокие нейронные сети для распознавания образов и рукописного текста и рекуррентные нейронные сети для NLP (Natural Language Processing - обработки естественных языков). Также есть модули для векторизации слов (embedding) и решения дифференциальных уравнений в частных производных (partial differential equations - PDE). Этот фреймворк имеет отличную архитектурную поддержку, позволяющую с легкостью производить вычисления на самых разных платформах, в том числе на десктопах, серверах и мобильных устройствах.

Основной козырь TensorFlow - это абстракции. Они позволяют разработчикам сфокусироваться на общей логике приложения, а не на мелких деталях реализации тех или иных алгоритмов. С помощью этой библиотеки разработчики Python могут легко использовать AI и ML для создания уникальных адаптивных приложений, гибко реагирующих на пользовательские данные, например на выражение лица или интонацию голоса.

БИБЛИОТЕКА KERAS

Keras — одна из основных библиотек Python с открытым исходным кодом, написанная для построения нейронных сетей и проектов машинного обучения. Keras может работать совместно с такими инструментами, как: Deeplearning4j, MXNet, Microsoft Cognitive Toolkit (CNTK), Theano или TensorFlow. В этой библиотеке реализованы практически все автономные модули нейронной сети, включая оптимизаторы, нейронные слои, функции активации слоев, схемы инициализации, функции затрат и модели регуляризации. Это позволяет строить новые модули нейросети, просто добавляя функции или классы. И поскольку модель уже определена в коде, разработчику не приходится создавать для нее отдельные конфигурационные файлы.

Keras особенно удобна для начинающих разработчиков, которые хотят проектировать и разрабатывать собственные нейронные сети. Также Keras можно использовать при работе со сверхточными нейронными сетями. В нем реализованы алгоритмы нормализации, оптимизации и активации слоев. Keras не является ML-библиотекой полного цикла (то есть, исчерпывающей все возможные варианты построения нейронных сетей). Вместо этого она функционирует как очень дружелюбный, расширяемый интерфейс, увеличивающий модульность и выразительность (в том числе других библиотек).

БИБЛИОТЕКА THEANO

Theano привлекла разработчиков Python и инженеров ML и AI сразу с момента появления в 2007 году.

По своей сути, это научная математическая библиотека, которая позволяет вам определять, оптимизировать и вычислять математические выражения, в том числе и в виде многомерных массивов. Основой большинства ML и AI приложений является многократное вычисление заковыристых математических выражений. Theano позволяет вам проводить подобные вычисления в сотни раз быстрее, вдобавок она отлично оптимизирована под GPU, имеет модуль для символьного дифференцирования, а также предлагает широкие возможности для тестирования кода.

Когда речь идет о производительности, Theano — отличная библиотека ML и AI, поскольку она может работать с очень большими нейронными сетями. Ее целью является снижение времени разработки и увеличение скорости выполнения приложений, в частности, основанных на алгоритмах глубоких нейронных сетей. Ее единственный недостаток — не слишком простой синтаксис (по сравнению с TensorFlow), особенно для новичков.

БИБЛИОТЕКА SCIKIT-LEARN

Scikit-learn — еще одна известная опенсорсная библиотека машинного обучения Python, с широким спектром алгоритмов кластеризации, регрессии и классификации. DBSCAN, градиентный бустинг, случайный лес, SVM и k-means — вот только несколько примеров. Она также отлично взаимодействует с другими научными библиотеками Python, такими как NumPy и SciPy.

Эта библиотека поддерживает алгоритмы обучения как с учителем, так и без учителя. Вот список основных преимуществ данной библиотеки, делающих ее одной из самых предпочтительных библиотек Python для ML:

+ снижение размерности;
+ алгоритмы, построенные на решающих деревьях (в том числе стрижка и индукция);
+ построение решающих поверхностей;
+ анализ и выбор признаков;
+ обнаружение и удаление выбросов;
+ продвинутое вероятностное моделирование;
+ классификация и кластеризация без учителя.

БИБЛИОТЕКА PYTORCH

PyTorch — это полностью готовая к работе библиотека машинного обучения Python с отличными примерами, приложениями и вариантами использования, поддерживаемая сильным сообществом. PyTorch отлично адаптирована к графическому процессору (GPU), что позволяет использовать его, например в приложениях NLP (обработка естественных языков). Вообще, поддержка вычислений на GPU и CPU обеспечивает оптимизацию и масштабирование распределенных задач обучения как в области исследований, так и в области создания ПО. Глубокие нейронные сети и тензорные вычисления с ускорением на GPU — две основные фишки PyTorch. Библиотека также включает в себя компилятор машинного обучения под названием Glow, который серьезно повышает производительность фреймворков глубокого обучения.

БИБЛИОТЕКА NUMPY

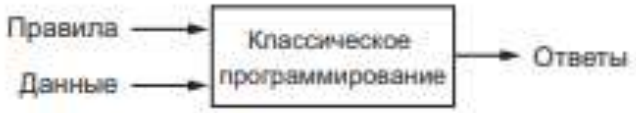
NumPy — это библиотека линейной алгебры, разработанная на Python. Практически все пакеты Python, использующиеся в машинном обучении, так или иначе опираются на NumPy. В библиотеку входят функции для работы со сложными математическими операциями линейной алгебры, алгоритмы преобразования Фурье и генерации случайных чисел, методы для работы с матрицами и n-мерными массивами. Модуль NumPy также применяется в научных вычислениях. В частности, он широко используется для работы со звуковыми волнами и изображениями.

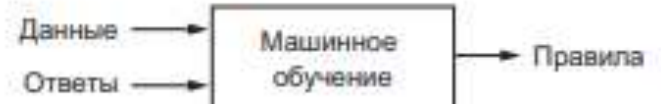
БИБЛИОТЕКА PANDAS

Pandas — это библиотека с открытым исходным кодом, которая предлагает широкий спектр инструментов для обработки и анализа данных. С ее помощью вы можете читать данные из широкого спектра источников, таких как CSV, базы данных SQL, файлы JSON и Excel.

В проектах по машинному обучению значительное время уходит на подготовку данных, а также на анализ основных тенденций и моделей. Именно здесь Pandas привлекает внимание специалистов по машинному обучению. Эта библиотека позволяет производить сложные операции с данными помощью всего одной команды. Python Pandas поставляется с несколькими встроенными методами для объединения, группировки и фильтрации данных и временных рядов. Но Pandas не ограничивается только решением задач, связанных с данными; он служит лучшей отправной точкой для создания более сфокусированных и мощных инструментов обработки данных.

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ, МАШИННОЕ И ГЛУБОКОЕ ОБУЧЕНИЕ





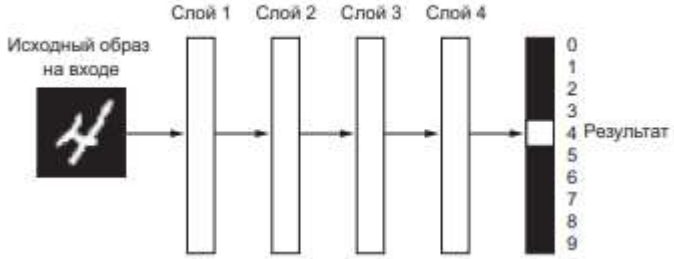
ИИ — это область, охватывающая машинное обучение и глубокое обучение, а также включающая в себя многие подходы, не связанные с обучением.

В машинном обучении система обучается, а не программируется явно. Ей передаются многочисленные примеры, имеющие отношение к решаемой задаче, а она находит в этих примерах статистическую структуру, которая позволяет системе выработать правила для автоматического решения задачи.

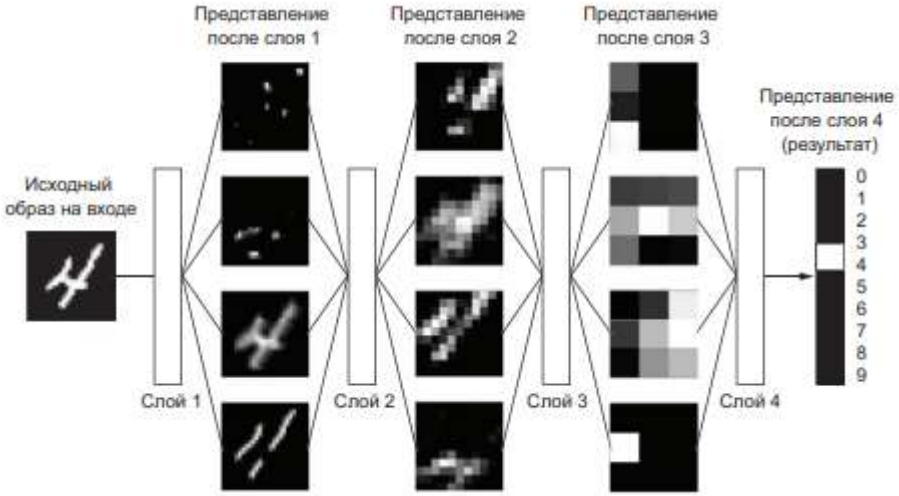
Глубокое обучение — это особый раздел машинного обучения: новый подход к поиску представления данных, делающий упор на изучение последовательных слоев (или уровней) все более значимых представлений

ГЛУБОКОЕ ОБУЧЕНИЕ

В глубоком обучении такие многослойные представления изучаются (почти всегда) с использованием моделей, так называемых нейронных сетей, структурированных в виде слоев, наложенных друг на друга. Термин нейронная сеть заимствован из нейробиологии, тем не менее, хотя некоторые основополагающие идеи глубокого обучения отчасти заимствованы из науки о мозге, модели глубокого обучения не являются моделями мозга. Нет никаких доказательств, что мозг реализует механизмы, подобные механизмам, используемым в современных моделях глубокого обучения. Вам могут встретиться научно-популярные статьи, в которых утверждается, что глубокое обучение работает подобно мозгу или моделирует работу мозга, но в действительности это не так.

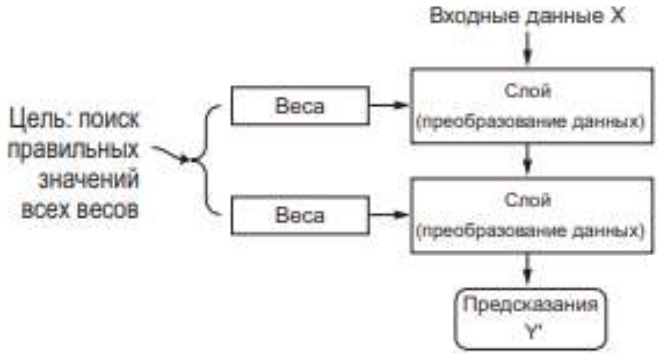


Глубокая нейронная сеть для классификации цифр



Глубокие представления, получаемые моделью классификации цифр

ПРИНЦИП ДЕЙСТВИЯ ГЛУБОКОГО ОБУЧЕНИЯ



Нейронная сеть параметризуется ее весами

Оценка потерь используется как обратная связь для корректировки весов

Функция потерь оценивает качество результатов, производимых нейронной сетью

ПРОРЫВЫ ГЛУБОКОГО ОБУЧЕНИЯ

- классификация изображений на уровне человека;
- распознавание речи на уровне человека;
- распознавание рукописного текста на уровне человека;
- улучшение качества машинного перевода с одного языка на другой;
- улучшение качества машинного чтения текста вслух;
- появление цифровых помощников, таких как Google Now и Amazon Alexa;
- управление автомобилем на уровне человека;
- повышение точности целевой рекламы, используемой компаниями Google, Baidu и Bing;
- повышение релевантности поиска в интернете;
- появление возможности отвечать на вопросы, заданные вслух;

ПЕРВОЕ ЗНАКОМСТВО С НЕЙРОННОЙ СЕТЬЮ

ПРИМЕЧАНИЕ О КЛАССАХ И МЕТКАХ

В машинном обучении *категория* в задаче классификации называется *классом*. Элементы исходных данных называются *образцами*. Класс, связанный с конкретным образцом, называется *меткой*.

Загрузка набора данных MNIST в Keras

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Здесь `train_images` и `train_labels` — это *тренировочный набор*, то есть данные, необходимые для обучения. После обучения модель будет проверяться тестовым (или контрольным) набором, `test_images` и `test_labels`.

Изображения хранятся в массивах Numpy, а метки — в массиве цифр от 0 до 9. Изображения и метки находятся в прямом соответствии, один к одному.

Рассмотрим обучающие данные:

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

И контрольные данные:

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Вот как мы будем действовать дальше: сначала передадим нейронной сети обучающие данные, `train_images` и `train_labels`. В результате этого сеть обучится сопоставлять изображения с метками. Затем мы предложим сети классифицировать изображения в `test_images` и проверим точность классификации по меткам из `test_labels`.

ПЕРВОЕ ЗНАКОМСТВО С НЕЙРОННОЙ СЕТЬЮ

Архитектура сети

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Основным строительным блоком нейронных сетей является *слой* (или *уровень*), модуль обработки данных, который можно рассматривать как фильтр для данных. Он принимает некоторые данные и выводит их в более полезной форме. В частности, слои извлекают *представления* из подаваемых в них данных, которые, как мы надеемся, будут иметь больше смысла для решаемой задачи. Фактически методика глубокого обучения заключается в объединении простых слоев, реализующих некоторую форму поэтапной *очистки данных*. Модель глубокого обучения можно сравнить с ситом, состоящим из последовательности фильтров все более тонкой очистки данных — слоев.

В данном случае наша сеть состоит из последовательности двух слоев `Dense`, которые являются тесно связанными (их еще называют *полносвязными*) нейронными

В данном случае наша сеть состоит из последовательности двух слоев `Dense`, которые являются тесно связанными (их еще называют *полносвязными*) нейронными
слоями. Второй (и последний) слой — это 10-переменный слой потерь (softmax layer), возвращающий массив с 10 оценками вероятностей (в сумме дающих 1). Каждая оценка определяет вероятность принадлежности текущего изображения к одному из 10 классов цифр.

Чтобы подготовить сеть к обучению, нужно настроить еще три параметра для этапа *компиляции*:

- *функцию потерь*, которая определяет, как сеть должна оценивать качество своей работы на обучающих данных и, соответственно, как корректировать ее в правильном направлении;
- *оптимизатор* — механизм, с помощью которого сеть будет обновлять себя, опираясь на наблюдаемые данные и функцию потерь;
- *метрики для мониторинга на этапах обучения и тестирования* — здесь нас будет интересовать только точность (доля правильно классифицированных изображений).

ПЕРВОЕ ЗНАКОМСТВО С НЕЙРОННОЙ СЕТЬЮ

Этап компиляции

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Перед обучением мы выполним предварительную обработку данных, преобразовав их в форму, которую ожидает получить нейронная сеть, и масштабируем их так, чтобы все значения оказались в интервале `[0, 1]`. Исходные данные — обучающие изображения — хранятся в трехмерном массиве `(60000, 28, 28)` типа `uint8`, значениями в котором являются числа в интервале `[0, 255]`. Мы преобразуем его в массив `(60000, 28 * 28)` типа `float32` со значениями в интервале `[0, 1]`.

Подготовка исходных данных

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Подготовка меток

```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Теперь можно начинать обучение сети, для чего в случае использования библиотеки Keras достаточно вызвать метод `fit` сети — он пытается *адаптировать* (fit) модель под обучающие данные:

```
>>> network.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [==============================] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [========================>.....] - ETA: 1s - loss: 0.1035 - acc:
0.9692
```

В процессе обучения отображаются две величины: потери сети на обучающих данных и точность сети на обучающих данных.

В данном случае мы достигли точности 0,989 (98,9%) на обучающих данных. Теперь проверим, как модель распознает контрольный набор:

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

Точность на контрольном наборе составила 97,8 % — немного меньше, чем на тренировочном наборе. Эта разница между точностью на тренировочном и контрольном наборах демонстрирует пример *переобучения* (overfitting), когда модели машинного обучения показывают худшую точность на новом наборе данных по сравнению с тренировочным.

ПРЕДСТАВЛЕНИЕ ДАННЫХ ДЛЯ НЕЙРОННЫХ СЕТЕЙ

2.2.1. Скаляры (тензоры нулевого ранга)

Тензор, содержащий единственное число, называется *скаляром* (скалярным, или тензором нулевого ранга). В Numpy число типа `float32` или `float64` — это скалярный тензор (или скалярный массив). Определить количество осей тензора Numpy можно с помощью атрибута `ndim`; скалярный тензор имеет 0 осей (`ndim == 0`). Количество осей тензора также называют его *рангом*. Вот пример скаляра в Numpy:

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

2.2.2. Векторы (тензоры первого ранга)

Одномерный массив чисел называют *вектором*, или тензором первого ранга. Тензор первого ранга имеет единственную ось. Далее приводится пример вектора в Numpy:

```
>>> x = np.array([12, 3, 6, 14])
>>> x
array([12, 3, 6, 14])
>>> x.ndim
1
```

Этот вектор содержит пять элементов и поэтому называется *пятимерным вектором*. Не путайте пятимерные векторы с пятимерными тензорами! Пятимерный вектор имеет только одну ось и пять значений на этой оси, тогда как пятимерный тензор имеет пять осей (и может иметь любое количество значений на каждой из них). *Мерность* может обозначать или количество элементов на данной оси (как в случае с пятимерным вектором), или количество осей в тензоре (как в пятимерном тензоре), что иногда может вызывать путаницу. В последнем случае технически более корректно говорить о *тензоре пятого ранга* (ранг тензора совпадает с количеством осей), но, как бы то ни было, для тензоров используется неоднозначное обозначение: *пятимерный тензор*.

2.2.3. Матрицы (тензоры второго ранга)

Массив векторов — это *матрица*, или двумерный тензор. Матрица имеет две оси (часто их называют *строками* и *столбцами*). Матрицу можно представить как прямоугольную таблицу с числами. Вот пример матрицы в Numpy:

```
>>> x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])
>>> x.ndim
2
```

Элементы на первой оси называют *строками*, а на второй — *столбцами*. В предыдущем примере `[5, 78, 2, 34, 0]` — это первая строка матрицы `x`, а `[5, 6, 7]` — ее первый столбец.

2.2.4. Тензоры третьего и высшего рангов

Если упаковать такие матрицы в новый массив, получится трехмерный тензор, который можно представить как числовой куб. Ниже приводится пример трехмерного тензора в Numpy:

```
>>> x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

Упаковав трехмерные тензоры в массив, вы получите четырехмерный тензор и т. д. В глубоком обучении чаще всего используются тензоры от нулевого ранга до четырехмерных, но иногда, например при обработке видеоданных, дело может дойти и до пятимерных тензоров.

ПРЕДСТАВЛЕНИЕ ДАННЫХ ДЛЯ НЕЙРОННЫХ СЕТЕЙ

Ключевые атрибуты

Тензор определяется тремя ключевыми атрибутами:

- *Количество осей (ранг)* — например, трехмерный тензор имеет три оси, а матрица — две. В библиотеках для Python, таких как Numpy, этот атрибут тензоров имеет имя `ndim`.
- *Форма* — кортеж целых чисел, описывающих количество измерений на каждой оси тензора. Например, матрица в предыдущем примере имеет форму `(3, 5)`, а трехмерный тензор имеет форму `(3, 3, 5)`. Вектор имеет форму с единственным элементом, например `(5,)`, тогда как скаляр имеет пустую форму `()`.
- *Тип данных* (обычно в библиотеках для Python ему дается имя `dtype`) — это тип данных, содержащихся в тензоре; например, тензор может иметь тип `float32`, `uint8`, `float64` и др. В редких случаях можно встретить тензоры типа `char`. Обратите внимание, что в Numpy (и в большинстве других библиотек) отсутствуют строковые тензоры, потому что тензоры хранятся в заранее выделенных, непрерывных сегментах памяти и строки, будучи сущностями с изменяющейся длиной, препятствуют использованию такой реализации.

Чтобы добавить конкретики, вернемся к данным из MNIST, которые мы обрабатывали в первом примере. Сначала загрузим набор данных MNIST:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Узнаем количество осей тензора `train_images`, обратившись к его атрибуту `ndim`:

```
>>> print(train_images.ndim)
3
```

его форму:

```
>>> print(train_images.shape)
(60000, 28, 28)
```

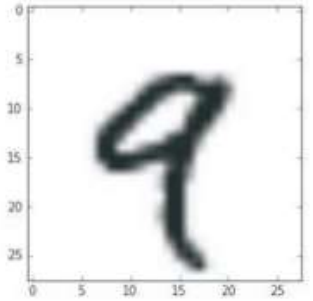
и тип данных, заглянув в атрибут `dtype`:

```
>>> print(train_images.dtype)
uint8
```

Итак, теперь мы знаем, что это трехмерный тензор с 8-разрядными целыми числами. Точнее, это массив с 60 000 матриц целых чисел размером 28 × 28. Каждая матрица представляет собой черно-белое изображение, где каждый элемент представляет пиксел с плотностью серого цвета в диапазоне от 0 до 255.

Вывод четвертой цифры на экран

```
digit = train_images[4]

import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



Четвертый образец из нашего набора данных

МАНИПУЛИРОВАНИЕ И ПАКЕТЫ ДАННЫХ

Манипулирование тензорами с помощью Numpy

В предыдущем примере мы *выбрали* конкретную цифру на первой оси, использовав синтаксис `train_images[i]`. Операция выбора конкретного элемента в тензоре называется *получением среза тензора*. Давайте посмотрим, какие операции получения среза тензора можно использовать с массивами Numpy.

Следующий пример извлекает цифры с 10-й до 100-й (100-я цифра не включается в срез) и помещает их в массив, имеющий форму `(90, 28, 28)`:

```
>>> my_slice = train_images[10:100]
>>> print(my_slice.shape)
(90, 28, 28)
```

Это эквивалентно более подробной форме записи, в которой определяются начальный и конечный индексы среза для каждой оси тензора. Обратите внимание, что `:` эквивалентно выбору всех элементов на оси:

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

Эквивалентно предыдущему примеру

Также эквивалентно предыдущему примеру

В общем случае можно получить срез между любыми двумя индексами по каждой оси тензора. Например, вот как можно выбрать пикселы из области 14 × 14 в правом нижнем углу каждого изображения:

```
my_slice = train_images[:, 14:, 14:]
```

Допускается использовать и отрицательные индексы. Так же как отрицательные индексы в списках на Python, они будут откладываться от конца текущей оси. Например, вот так можно обрезать все изображения, оставив только квадрат 14 × 14 пикселов в центре:

```
my_slice = train_images[:, 7:-7, 7:-7]
```

Пакеты данных

В общем случае первая ось (ось с индексом 0, потому что нумерация начинается с 0) во всех тензорах, с которыми вам придется столкнуться в глубоком обучении, будет *осью образцов* (иногда ее называют *измерением образцов*). В примере MNIST образцы — это изображения цифр.

Кроме того, модели глубокого обучения не обрабатывают весь набор данных целиком; они разбивают его на небольшие пакеты. Если говорить конкретнее, вот один пакет из примера с изображениями цифр MNIST, имеющий размер 128:

```
batch = train_images[:128]
```

А вот следующий пакет:

```
batch = train_images[128:256]
```

А вот *n*-й пакет:

```
batch = train_images[128 * n:128 * (n + 1)]
```

При рассмотрении таких пакетных тензоров первую ось (ось с индексом 0) называют *осью пакетов*, или *измерением пакетов*. Эта терминология часто будет встречаться вам при работе с Keras и другими библиотеками глубокого обучения.

```
y_slice = train_images[:, 7:-7, 7:-7]
```

ПРИМЕРЫ ТЕНЗОРОВ

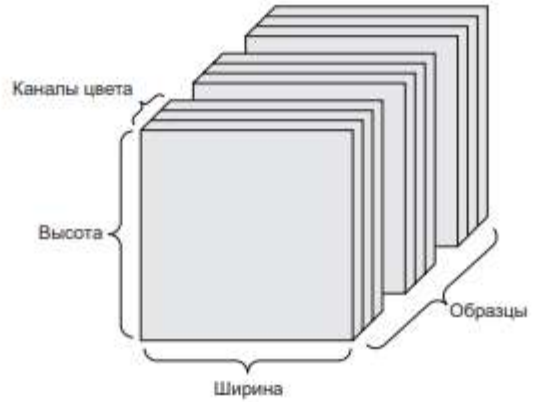
- *векторные данные* — двумерные тензоры с формой (образцы, признаки);
- *временные ряды или последовательности* — трехмерные тензоры с формой (образцы, метки_времени, признаки);
- *изображения* — четырехмерные тензоры с формой (образцы, высота, ширина, цвет) или с формой (образцы, цвет, высота, ширина);
- *видео* — пятимерные тензоры с формой (образцы, кадры, высота, ширина, цвет) или с формой (образцы, кадры, цвет, высота, ширина).

Всякий раз, когда время (или понятие последовательной упорядоченности) играет важную роль в ваших данных, такие данные предпочтительнее сохранять в трехмерном тензоре с явной осью времени. Каждый образец может быть представлен как последовательность векторов (двумерных тензоров), а сам пакет данных — как трехмерный тензор

Трехмерный тензор с временным рядом

Изображения

Обычно изображения имеют три измерения: высоту, ширину и цвет. Даже при том, что черно-белые изображения (как в наборе данных MNIST) имеют только один канал цвета и могли бы храниться в двумерных тензорах, по соглашениям тензоры с изображениями всегда имеют три измерения, где для черно-белых изображений отводится только один канал цвета. Соответственно, пакет со 128 черно-белыми изображениями, имеющими размер 256 × 256, можно сохранить в тензоре с формой (128, 256, 256, 1), а пакет со 128 цветными изображениями — в тензоре с формой (128, 256, 256, 3) (рис. 2.4).



Четырехмерный тензор с изображениями
(в соответствии с соглашением «канал следует первым»)

ОПЕРАЦИИ С ТЕНЗОРАМИ

В нашем первом примере мы создали сеть, наложив друг на друга два слоя `Dense`. В библиотеке Keras экземпляр слоя выглядит так:

```
keras.layers.Dense(512, activation='relu')
```

Этот слой можно интерпретировать как функцию, которая принимает двумерный тензор и возвращает другой двумерный тензор — новое представление исходного тензора. В данном случае функция имеет следующий вид (где `W` — это двумерный тензор, а `b` — вектор, оба значения являются атрибутами слоя):

```
output = relu(dot(W, input) + b)
```

Давайте развернем ее. Здесь у нас имеется три операции с тензорами: скалярное произведение (`dot`) исходного тензора `input` и тензора с именем `W`; сложение (`+`) получившегося двумерного тензора и вектора `b`; и, наконец, операция `relu`. `relu(x)` эквивалентна операции `max(x, 0)`.

Иными словами, при использовании Numpy поэлементные операции можно записывать, как показано ниже, и они будут выполняться почти мгновенно:

```
import numpy as np

z = x + y  <-------  Поэлементное сложение

z = np.maximum(z, 0.)  <-------  Поэлементная операция relu
```

Поэлементные операции

Операция `relu` и сложение — это *поэлементные операции*: операции, которые применяются к каждому элементу в тензоре по отдельности. То есть эти операции поддаются массовому распараллеливанию (*векторизации*, термин пришел из архитектуры *векторного процессора* суперкомпьютера периода 1970–1990-х). Для реализации поэлементных операций на Python можно использовать цикл `for`, как в следующем примере реализации операции `relu`:

```
def naive_relu(x):
    assert len(x.shape) == 2  <-------  Убедиться, что x — двумерный тензор Numpy

    x = x.copy()  <-------  Исключить затирание исходного тензора
    for i in range(x.shape[0]):

        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

Точно так же реализуется сложение:

```
def naive_add(x, y):
    assert len(x.shape) == 2  <-------  Убедиться, что x и y — двумерные тензоры Numpy
    assert x.shape == y.shape
    x = x.copy()  <-------  Исключить затирание исходного тензора
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

ОПЕРАЦИИ С ТЕНЗОРАМИ

Расширение

Наша предыдущая реализация `naive_add` поддерживает только сложение двумерных тензоров с идентичными формами. Но в слое `Dense`, представленном выше, мы складывали двумерный тензор с вектором. Что происходит при сложении, когда формы складываемых тензоров различаются?

Когда это возможно и не вызывает неоднозначности, меньший тензор *расширяется* так, чтобы его новая форма соответствовала форме большего тензора. Расширение выполняется в два этапа:

1. В меньший тензор добавляются оси (называются *осями расширения*), чтобы значение его атрибута `ndim` соответствовало значению этого же атрибута большего тензора.
2. Меньший тензор копируется в эти новые оси до полного совпадения с формой большего тензора.

Рассмотрим конкретный пример. Пусть имеются тензоры `X` с формой `(32, 10)` и `y` с формой `(10,)`. Чтобы привести их в соответствие, сначала нужно добавить в тензор `y` первую пустую ось, чтобы он приобрел форму `(1, 10)`, а затем скопировать вторую ось 32 раза, чтобы в результате получился тензор `Y` с формой `(32, 10)`, где `Y[i, :] == y` для `i` в диапазоне `range(0, 32)`. После этого можно сложить `X` и `Y`, которые имеют одинаковую форму.

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2.          <-- Убедиться, что x — двумерный тензор Numpy.
    assert len(y.shape) == 1.          <-- Убедиться, что y — вектор Numpy.
    assert x.shape[1] == y.shape[0]

    x = x.copy()                       <-- Исключить затирание исходного тензора
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

```
import numpy as np

x = np.random.random((64, 3, 32, 10))   <-- x — тензор случайных чисел, имеющий форму (64, 3, 32, 10)
y = np.random.random((32, 10))          <-- y — тензор случайных чисел, имеющий форму (32, 10)

z = np.maximum(x, y)                    <-- Получившийся тензор z имеет форму (64, 3, 32, 10) аналогично x
```

ОПЕРАЦИИ С ТЕНЗОРАМИ

Скалярное произведение двух векторов, x и y

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]

    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

Убедиться, что x и y — векторы Numpy

Скалярное произведение строк x на y

```
import numpy as np

def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]

    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

Убедиться, что x — матрица Numpy

Убедиться, что y — вектор Numpy

Первое измерение x должно совпадать с нулевым измерением y!

Эта операция вернет вектор с нулевыми элементами, имеющий ту же форму, что и y

Также можно было бы повторно использовать код, написанный прежде, подчеркнув общность произведений матрицы на вектор и вектора на вектор:

```
def naive_matrix_vector_dot(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_dot(x[i, :], y)
    return z
```

Разумеется, скалярное произведение можно распространить на тензоры с произвольным количеством осей. Наиболее часто на практике применяется скалярное произведение двух матриц. Получить скалярное произведение двух матриц, x и y (`dot(x, y)`), можно, только если `x.shape[1] == y.shape[0]`. В результате получится матрица с формой `(x.shape[0], y.shape[1])`, элементами которой являются скалярные произведения строк x на столбцы y. Вот как могла бы выглядеть простейшая реализация:

```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]

    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

Убедиться, что x и y — матрицы Numpy

Первое измерение x должно совпадать с нулевым измерением y!

Эта операция вернет матрицу заданной формы с нулевыми элементами

Обход строк в x...

... и столбцов в y

В общем случае скалярное произведение тензоров с бóльшим числом измерений выполняется в соответствии с теми же правилами совместимости форм, как описывалось выше для случая двумерных матриц:

```
(a, b, c, d) . (d,) -> (a, b, c)
(a, b, c, d) . (d, e) -> (a, b, c, e)
```

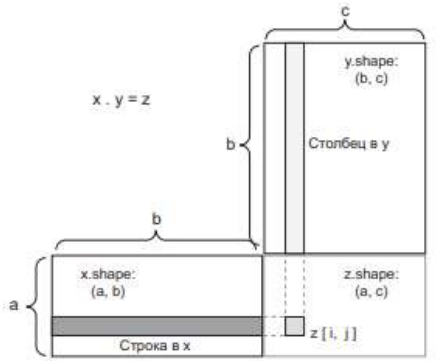


Диаграмма скалярного произведения матриц

ОПЕРАЦИИ С ТЕНЗОРАМИ

Третий вид операций с тензорами, который мы должны рассмотреть, — это *изменение формы тензора*. Эта операция не используется в слоях `Dense` нашей нейронной сети, но мы использовали ее, когда готовили исходные данные для передачи в сеть:

```
train_images = train_images.reshape((60000, 28 * 28))
```

Изменение формы тензора предполагает такое переупорядочение строк и столбцов, чтобы привести его форму к заданной. Разумеется, тензор с измененной формой имеет такое же количество элементов, что и исходный тензор. Чтобы было понятнее, рассмотрим несколько простых примеров:

```
>>> x = np.array([[0., 1.],
                  [2., 3.],
                  [4., 5.]])
>>> print(x.shape)
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
>>> x = x.reshape((2, 3))
>>> x
array([[ 0., 1., 2.],
       [ 3., 4., 5.]])
```

Особый случай изменения формы, который часто встречается в практике, — это *транспонирование*. *Транспонирование* — это такое преобразование матрицы, когда строки становятся столбцами, а столбцы — строками, то есть `x[i, :]` превращается в `x[:, i]`:

```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> print(x.shape)
(20, 300)
```

Разглаживание смятого комка исходных данных

МЕХАНИЗМ НЕЙРОННЫХ СЕТЕЙ: ОПТИМИЗАЦИЯ НА ОСНОВЕ ГРАДИЕНТА

```
output = relu(dot(W, input) + b)
```

В этом выражении `W` и `b` — тензоры, являющиеся атрибутами слоя. Они называются *весами*, или *обучаемыми параметрами* слоя (атрибуты `kernel` и `bias` соответственно). Эти веса содержат информацию, извлеченную сетью из обучающих данных.

Первоначально эти весовые матрицы заполняются небольшими случайными значениями (этот шаг называется *случайной инициализацией*). Конечно, бессмысленно было бы ожидать, что `relu(dot(W, input) + b)` вернет хоть сколько-нибудь полезное представление для случайных `W` и `b`. Начальные представления не несут никакого смысла, но они служат начальной точкой. Далее, на основе сигнала обратной связи, происходит постепенная корректировка весов. Эта постепенная корректировка, которая также называется *обучением*, составляет суть машинного обучения.

Ниже перечислены шаги, выполняемые в так называемом *цикле обучения*, который повторяется столько раз, сколько потребуется:

1. Извлекается пакет обучающих экземпляров `x` и соответствующих целей `y`.
2. Сеть обрабатывает пакет `x` (этот шаг называется *прямым проходом*) и получает пакет предсказаний `y_pred`.
3. Вычисляются потери сети на пакете, дающие оценку несовпадения между `y_pred` и `y`.
4. Корректируются веса сети так, чтобы немного уменьшить потери на этом пакете.

Производная операций с тензорами: градиент

Градиент — это производная операции с тензором, обобщение понятия производной на функции с многомерными входными данными, то есть на функции, принимающие на входе тензоры.

Рассмотрим входной вектор `x`, матрицу `W`, цель `y` и функцию потерь `loss`. Вы можете с помощью `W` вычислить приближение к цели `y_pred` и определить потери или несоответствие, между кандидатом `y_pred` и целью `y`:

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y)
```

Если входные данные `x` и `y` зафиксированы, тогда это можно интерпретировать как функцию, отображающую значения `W` в значения потерь:

```
loss_value = f(W)
```

Допустим, что `W0` — текущее значение `W`. Тогда производной функции `f` в точке `W0` будет тензор `gradient(f)(W0)` с той же формой, что и `W`, в котором каждый элемент `gradient(f)(W0)[i, j]` определяет направление и величину изменения в `loss_value`, наблюдаемого при изменении `W0[i, j]`. Тензор `gradient(f)(W0)` — это градиент функции `f(W) = loss_value` в `W0`.

МЕХАНИЗМ НЕЙРОННЫХ СЕТЕЙ: ОПТИМИЗАЦИЯ НА ОСНОВЕ ГРАДИЕНТА

Давайте вернемся назад, к первому примеру, и рассмотрим каждую его часть в свете новых знаний, полученных в трех предыдущих разделах.

Вот наши входные данные:

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Теперь вы уже знаете, что входные данные типа `float32` хранятся в тензорах Numpy, имеющих форму `(60000, 784)` (обучающие данные) и `(10000, 784)` (контрольные данные) соответственно.

Вот наша сеть:

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Вот как выглядел этап компиляции:

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Наконец, вот как выглядел цикл обучения:

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

СЛОИ: СТРОИТЕЛЬНЫЕ БЛОКИ ГЛУБОКОГО ОБУЧЕНИЯ

Слой — это модуль обработки данных, принимающий на входе и возвращающий на выходе один или несколько тензоров. Некоторые слои не сохраняют состояния, но чаще это не так: веса слоя, один или несколько тензоров, обучаемых с применением алгоритма стохастического градиентного спуска, которые вместе хранят знание сети.

Разным слоям соответствуют тензоры разных форматов и разные виды обработки данных. Например, простые векторные данные, хранящиеся в двумерных тензорах с формой (`образцы, признаки`), часто обрабатываются *плотно связанными* слоями, которые также называют *полносвязными*, или *плотными*, слоями (класс `Dense` в Keras). Ряды данных хранятся в трехмерных тензорах с формой (`образцы, метки_времени, признаки`) и обычно обрабатываются *рекуррентными* слоями, такими как `LSTM`. Изображения хранятся в четырехмерных тензорах и обычно обрабатываются двумерными сверточными слоями (`Conv2D`).

```
from keras import layers

layer = layers.Dense(32, input_shape=(784,))
```

Полносвязный слой с 32 выходными нейронами

Здесь создается слой, принимающий только двумерные тензоры, первое измерение которых равно 784 (ось 0 — измерение пакетов — не задана, поэтому допустимо любое значение). Этот слой возвращает тензор, первое измерение которого равно 32.

Другими словами, этот слой можно связать со слоем ниже, только если тот принимает двумерные векторы. Фреймворк Keras избавляет от необходимости беспокоиться о совместимости, потому что слои, добавляемые в модели, автоматически конструируются так, чтобы соответствовать форме входного слоя. Например, представьте, что вы написали следующий код:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(32))
```

Второй слой создается без явного значения для аргумента `input_shape`, поэтому форма входных данных будет автоматически выведена из формы выходных данных предыдущего слоя.

РАЗРАБОТКА С ИСПОЛЬЗОВАНИЕМ KERAS: КРАТКИЙ ОБЗОР

Вы уже видели один пример модели Keras: в примере с рукописными цифрами из набора MNIST. Вот как выглядит типичный процесс использования Keras:

1. Определяются обучающие данные: входные и целевые тензоры.
2. Определяются слои сети (*модель*), отображающие входные данные в целевые.
3. Настраивается процесс обучения выбором функции потерь, оптимизатора и некоторых параметров для мониторинга.
4. Выполняются итерации по обучающим данным вызовом метода `fit()` модели.

Модель можно определить двумя способами: с использованием класса `Sequential` (только для линейного стека слоев — наиболее популярная архитектура сетей в настоящее время) или *функционального API* (для ориентированного ациклического графа слоев, позволяющего конструировать произвольные архитектуры).

Для напоминания ниже приводится определение двухслойной модели с использованием класса `Sequential` (обратите внимание, что первому слою передается ожидаемая форма входных данных):

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

А вот та же модель, но сконструированная с применением функционального API:

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

Функциональный API позволяет манипулировать данными в тензорах, которые обрабатывает модель, и применять слои к этим тензорам, как если бы они были функциями.

После определения архитектуры уже неважно, используете вы модель `Sequential` или функциональный API. В обоих случаях далее выполняются одинаковые шаги.

Настройка процесса обучения производится на этапе компиляции. При этом задаются оптимизатор и функция(-и) потерь, которые должны использоваться моделью, а также все метрики для мониторинга во время обучения. Вот пример с единственной функцией потерь, которая используется чаще всего:

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='mse',
              metrics=['accuracy'])
```

Наконец, сам процесс обучения состоит в передаче массивов Numpy с входными данными (и соответствующими целевыми данными) в метод `fit()` модели, по аналогии с некоторыми другими библиотеками машинного обучения, такими как Scikit-Learn:

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

КЛАССИФИКАЦИЯ ОТЗЫВОВ К ФИЛЬМАМ: ПРИМЕР БИНАРНОЙ КЛАССИФИКАЦИИ

The IMDB dataset

Будем работать с набором данных IMDB: множеством из 50 000 самых разных отзывов к кинолентам в интернет-базе фильмов (Internet Movie Database). Набор разбит на 25 000 обучающих и 25 000 контрольных отзывов, каждый набор на 50 % состоит из отрицательных и на 50 % из положительных отзывов.

Листинг 3.1. Загрузка набора данных IMDB

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

Аргумент `num_words=10000` означает, что в обучающих данных будет сохранено только 10 000 слов, наиболее часто встречающихся в обучающем наборе отзывов.

Редкие слова будут отброшены. Это позволит вам работать с вектором управляемого размера.

Переменные `train_data` и `test_data` — это списки отзывов; каждый отзыв — это список индексов слов (кодированное представление последовательности слов). Переменные `train_labels` и `test_labels` — это списки нулей и единиц, где нули соответствуют *отрицательным* отзывам, а единицы — *положительным*:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]

>>> train_labels[0]
1
```

Поскольку мы ограничили себя 10 000 наиболее употребительных слов, в наборе отсутствуют индексы больше 10 000:

```
>>> max([max(sequence) for sequence in train_data])
9999
```

Чтобы вам было понятнее, ниже показано декодирование одного из отзывов в последовательность слов на английском языке:

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

word_index — это словарь, отображающий слова в целочисленные индексы

Получить обратное представление словаря, отображающее индексы в слова

Декодирование отзыва. Обратите внимание, что индексы смещены на 3, потому что индексы 0, 1 и 2 зарезервированы для слов «padding» (отступ), «start of sequence» (начало последовательности) и «unknown» (неизвестно)

КЛАССИФИКАЦИЯ ОТЗЫВОВ К ФИЛЬМАМ: ПРИМЕР БИНАРНОЙ КЛАССИФИКАЦИИ

Листинг 3.2. Кодирование последовательностей целых чисел в бинарную матрицу

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Создание матрицы с формой (len(sequences), dimension)

Запись единицы в элемент с данным индексом

Векторизованные обучающие данные

Векторизованные контрольные данные

Вот как теперь выглядят образцы:

```
>>> x_train[0]
array([ 0., 1., 1., ..., 0., 0., 0.])
```

Нам также нужно векторизовать метки, что делается очень просто:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Теперь данные готовы к передаче в нейронную сеть

Конструирование сети

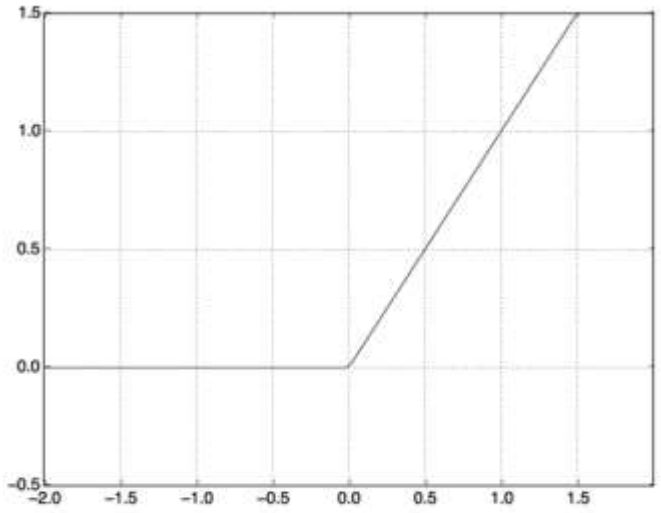
Входные данные представлены векторами, а метки — скалярами (единицами и нулями): это самый простой набор данных, какой можно встретить. С задачами этого вида прекрасно справляются сети, организованные как простой стек полносвязных (`Dense`) слоев с операцией активации `relu`: `Dense(16, activation='relu')`.

Аргумент (16), передаваемый каждому слою `Dense`, — это число скрытых нейронов слоя. *Скрытый нейрон* (hidden unit) — это измерение в пространстве представлений слоя. Как рассказывалось в главе 2, каждый слой `Dense` с операцией активации `relu` реализует следующую цепочку операций с тензорами:

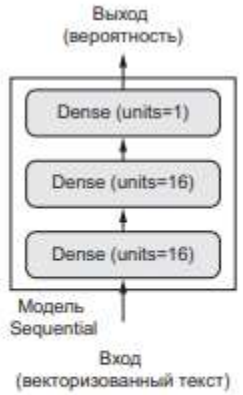
```
output = relu(dot(W, input) + b)
```

Наличие 16 скрытых нейронов означает, что весовая матрица `W` будет иметь форму `(input_dimension, 16)`: скалярное произведение на `W` спроецирует входные данные в 16-мерное пространство представлений (затем будет произведено сложение с вектором смещений `b` и выполнена операция `relu`). Размерность пространства представлений можно интерпретировать как «степень свободы нейронной сети при изучении внутренних представлений». Большее количество скрытых нейронов (бóльшая размерность пространства представлений) позволяет сети обучаться на более сложных представлениях, но при этом увеличивается вычислительная стоимость сети, что может привести к выявлению нежелательных шаблонов (шаблонов, которые могут повысить качество классификации обучающих данных, но не контрольных).

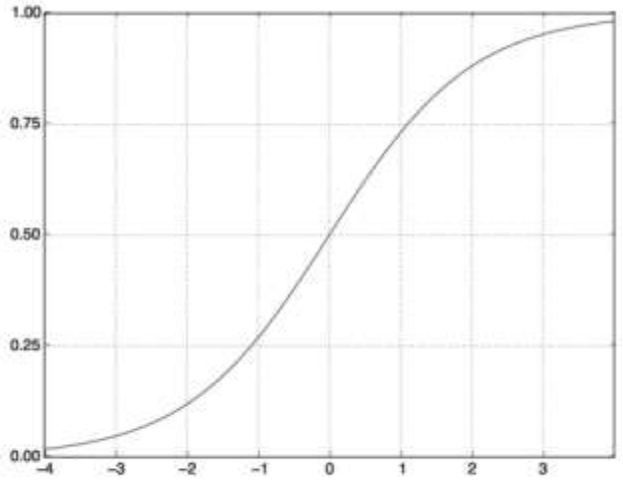
КЛАССИФИКАЦИЯ ОТЗЫВОВ К ФИЛЬМАМ: ПРИМЕР БИНАРНОЙ КЛАССИФИКАЦИИ



Функция блока линейной ректификации



Трехслойная сеть



Сигмоидная функция

Реализация этой сети с использованием Keras напоминает пример MNIST, который мы видели раньше.

Листинг 3.3. Определение модели

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Листинг 3.4. Компиляция модели

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

КЛАССИФИКАЦИЯ ОТЗЫВОВ К ФИЛЬМАМ: ПРИМЕР БИНАРНОЙ КЛАССИФИКАЦИИ

Листинг 3.5. Настройка оптимизатора

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Листинг 3.6. Использование нестандартных функций потерь и метрик

```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

Проверка решения

Чтобы проконтролировать точность модели во время обучения на данных, которые она прежде не видела, создадим проверочный набор, выбрав 10 000 образцов из оригинального набора обучающих данных.

Листинг 3.7. Создание проверочного набора

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Теперь проведем обучение модели в течение 20 эпох (выполнив 20 итераций по всем образцам в тензорах `x_train` и `y_train`) пакетами по 512 образцов. В то же время будем следить за потерями и точностью на 10 000 отложенных образцов. Для этого достаточно передать проверочные данные в аргументе `validation_data`.

Листинг 3.8. Обучение модели

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

При использовании CPU на каждую эпоху будет потрачено менее 2 секунд — а все обучение закончится через 20 секунд. В конце каждой эпохи обучение приостанавливается, потому что модель вычисляет потерю и точность на 10 000 образцах проверочных данных.

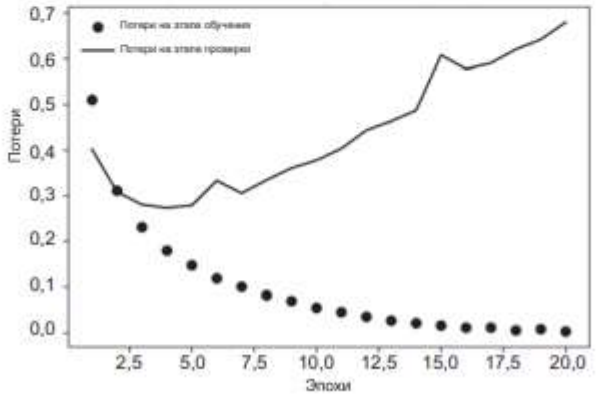
КЛАССИФИКАЦИЯ ОТЗЫВОВ К ФИЛЬМАМ: ПРИМЕР БИНАРНОЙ КЛАССИФИКАЦИИ

Листинг 3.9. Формирование графиков потерь на этапах обучения и проверки

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

«bo» означает «blue dot» — «синяя точка»

«b» означает «solid blue line» — «сплошная синяя линия»

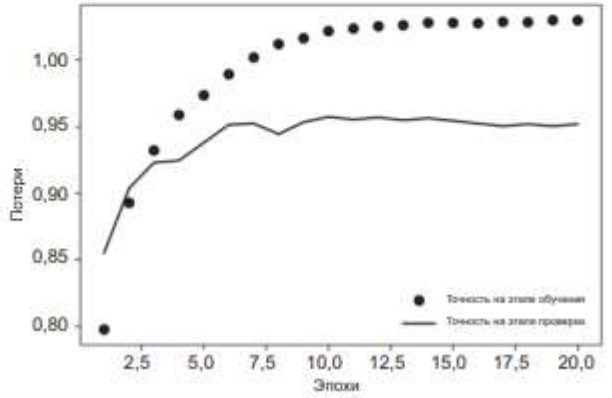


Потери на этапах обучения и проверки

Листинг 3.10. Формирование графиков точности на этапах обучения и проверки

```
plt.clf()
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

Очистить рисунок



Точность на этапах обучения и проверки

КЛАССИФИКАЦИЯ ОТЗЫВОВ К ФИЛЬМАМ: ПРИМЕР БИНАРНОЙ КЛАССИФИКАЦИИ

А теперь обучим новую сеть с нуля в течение четырех эпох и затем оценим получившийся результат на контрольных данных.

Листинг 3.11. Обучение новой модели с нуля

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

Конечные результаты:

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

Это простейшее решение позволило достичь точности 88 %. При использовании же самых современных подходов точность может доходить до 95 %.

Использование обученной сети для предсказаний на новых данных

После обучения сети ее можно использовать для решения практических задач. Например, попробуем предсказать вероятность того, что отзывы будут положительными, с помощью метода `predict`:

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

Как видите, сеть уверена в одних образцах (0,99 или выше или 0,01 или ниже), но не так уверена в других (0,6; 0,4).

ЗАДАНИЕ ДЛЯ САМОКОНТРОЛЯ

Создать тензоры 0, 1, 2 ранга

СПАСИБО ЗА ВНИМАНИЕ!