

# Содержание

Об авторах	15
Посвящение Джона	16
Посвящение Луки	16
Благодарности Джона	17
Благодарности Луки	17
<b>Введение</b>	<b>18</b>
О книге	18
Соглашения, принятые в книге	19
Глупые предположения	20
Источники дополнительной информации	21
Что дальше	22
Ждем ваших отзывов!	23
<b>Часть 1. Появление глубокого обучения</b>	<b>25</b>
<b>Глава 1. Введение в глубокое обучение</b>	<b>27</b>
Определение смысла глубокого обучения	28
Начнем с искусственного интеллекта	28
Роль искусственного интеллекта	30
Сосредоточимся на машинном обучении	34
Переход от машинного обучения к глубокому обучению	36
Использование глубокого обучения в реальном мире	38
Концепция обучения	39
Решение задач глубокого обучения	39
Использование глубокого обучения в приложениях	39
Среда программирования для глубокого обучения	40
Преодоление заблуждений, связанных с глубоким обучением	42
Экосистема запуска	43
Когда глубокое обучение неприменимо	43
<b>Глава 2. Знакомство с принципами машинного обучения</b>	<b>45</b>
Определение машинного обучения	46
Как работает машинное обучение	46
Это все чистая математика	48

Разные стратегии обучения	48
Обучение, проверка и тестирование данных	51
Поиск обобщения	52
Знакомство с предубеждением	53
Сложность модели	54
Разнообразие способов обучения	54
Бесплатных обедов не бывает	55
Пять основных подходов	55
Несколько разных подходов	58
В ожидании следующего прорыва	62
Размышления об истинном использовании машинного обучения	62
Преимущества машинного обучения	63
Границы машинного обучения	65
<b>Глава 3. Получение и использование языка Python</b>	<b>69</b>
Работа с языком Python в этой книге	70
Получение собственного экземпляра Anaconda	71
Получение пакета Anaconda от Continuum Analytics	71
Установка Anaconda на Linux	72
Установка Anaconda на MacOS	73
Установка Anaconda на Windows	74
Загрузка наборов данных и примеров кода	78
Использование Jupyter Notebook	79
Определение хранилища кода	81
Получение и использование наборов данных	86
Создание приложения	87
Понятие ячеек	88
Добавление ячеек документации	89
Использование ячеек других типов	90
Использование отступов	90
Добавление комментариев	91
Понятие комментариев	92
Применение комментариев для напоминания	93
Применение комментариев для предотвращения выполнения кода	95
Получение справки по языку Python	95
Работа в облаке	96
Использование наборов данных Kaggle и ядер	96
Использование Google Colaboratory	97

<b>Глава 4. Использование инфраструктуры глубокого обучения</b>	99
Знакомство с инфраструктурой	100
Определение различий	100
Популярность инфраструктур	101
Определение инфраструктуры глубокого обучения	104
Выбор конкретной инфраструктуры	105
Работа с бюджетными инфраструктурами	106
Caffe2	106
Chainer	107
PyTorch	108
MXNet	108
Microsoft Cognitive Toolkit/CNTK	109
Инфраструктура TensorFlow	109
Понятно, почему TensorFlow так хорош	109
Упрощение TensorFlow с помощью TFLearn	112
Использование Keras для большего упрощения	113
Получение экземпляра TensorFlow и Keras	114
Исправление ошибки инструментов сборки C++ в Windows	116
Доступ к новой среде в Notebook	117
<b>Часть 2. Основы глубокого обучения</b>	119
<b>Глава 5. Обзор матричной математики и оптимизации</b>	121
В какой математике вы действительно нуждаетесь	122
Работа с данными	123
Создание и работа с матрицей	124
Скалярные, векторные и матричные операции	125
Создание матрицы	126
Умножение матриц	127
Расширенные матричные операции	129
Расширение анализа до тензоров	131
Эффективное использование векторизации	133
Интерпретация обучения как оптимизации	134
Функции стоимости	135
Нисходящая кривая ошибок	136
Обучение в правильном направлении	137
Обновление	139

<b>Глава 6. Основы линейной регрессии</b>	141
Объединение переменных	142
Простая линейная регрессия	143
Переход к множественной линейной регрессии	144
Включение градиентного спуска	145
Линейная регрессия в действии	146
Смешивание типов переменных	148
Моделирование ответов	148
Моделирование признаков	149
Работа со сложными отношениями	150
Переход на вероятности	152
Обеспечение бинарного ответа	152
Преобразование числовых оценок в вероятности	153
Прогноз правильных признаков	155
Определение результата несовместимых признаков	155
Избежание переобучения с использованием выборки и регуляризации	156
Изучение по одному примеру за раз	158
Использование градиентного спуска	158
Чем отличается SGD	158
<b>Глава 7. Введение в нейронные сети</b>	163
Знакомство с невероятным перцептроном	164
Функциональность перцептрона	164
Достижение предела неразделяемости	166
Уменьшение сложности нейронными сетями	169
Нейрон	169
Прямая передача данных	171
Еще глубже в кроличью нору	173
Использование обратного распространения для настройки обучения	176
Борьба с переобучением	180
Понятие проблемы	180
Открываем черный ящик	181
<b>Глава 8. Построение простой нейронной сети</b>	183
Понятие нейронных сетей	184
Базовая архитектура	185
Документирование основных модулей	187
Решение простой задачи	190

Внутренняя работа нейронных сетей	192
Выбор правильной функции активации	193
Полагаясь на умный оптимизатор	195
Установка рабочей скорости обучения	196
<b>Глава 9. Переход к глубокому обучению</b>	<b>197</b>
Данные везде	198
Учет влияния структуры	199
Следствия закона Мура	200
Что меняет закон Мура	201
Преимущества дополнительных данных	202
Последствия больших данных	203
Своевременность и качество данных	203
Ускорение обработки	205
Использование мощного оборудования	205
Другие инвестиции	206
Отличие глубокого обучения от других форм искусственного интеллекта	207
Добавление большего количества слоев	207
Изменение активаций	210
Добавление регуляризации в ходе отсева	211
Поиск еще более разумных решений	212
Использование дистанционного обучения	213
Перенос обучения	213
Обучение от начала до конца	214
<b>Глава 10. Сверточные нейронные сети</b>	<b>215</b>
Начнем обзор CNN с распознавания символов	216
Основы изображения	216
Как работает свертка	219
Свертка	220
Упрощение использования подвыборки	224
Архитектура LeNet	226
Обнаружение краев и фигур в изображениях	231
Визуализация сверток	231
Успешные архитектуры	233
Перенос обучения	235

<b>Глава 11. Введение в рекуррентные нейронные сети</b>	239
Рекуррентные сети	240
Моделирование последовательностей с использованием памяти	241
Распознавание и перевод речи	242
Размещение правильной подписи на картинках	245
Долгая краткосрочная память	246
Определение различий в памяти	247
Архитектура LSTM	248
Открывая интересные варианты	250
Получение необходимого внимания	252
<b>Часть 3. Взаимодействие с глубоким обучением</b>	255
<b>Глава 12. Классификация изображений</b>	257
Конкурсы по классификации изображений	258
ImageNet и MS COCO	259
Магия приращения данных	261
Распознавание дорожных знаков	265
Подготовка данных изображения	266
Выполнение задачи классификации	269
<b>Глава 13. Передовые CNN</b>	275
Различные задачи классификации	276
Локализация	278
Классификация нескольких объектов	278
Аннотирование нескольких объектов на изображениях	279
Сегментирование изображений	280
Восприятие объектов в их окружении	282
Как работает RetinaNet	283
Использование кода Keras-RetinaNet	284
Предотвращение преднамеренных атак на приложения глубокого обучения	289
Обманные пиксели	290
Взлом с помощью наклеек и других артефактов	292
<b>Глава 14. Обработка текстов на естественном языке</b>	295
Обработка языка	296
Определение понимания как лексического анализа	297
Помещение всех документов в набор	299

Запоминание имеющих значение последовательностей	301
Понимание семантики по векторным представлениям слов	302
Использование искусственного интеллекта для анализа настроений	307
<b>Глава 15. Создание произведений изобразительного искусства и музыки</b>	<b>315</b>
Учимся подражать искусству и жизни	316
Передача художественного стиля	317
Сведение проблемы к статистике	319
Что глубокое обучение создать не может	320
Подражание художнику	321
Определение нового произведения на основе одного примера	321
Объединение стилей для создания нового произведения искусства	323
Визуализация мечты нейронных сетей	324
Использование сети для сочинения музыки	324
<b>Глава 16. Построение генеративно-согласительных сетей</b>	<b>327</b>
Создание состоящих сетей	328
Поиск ключа в состязании	329
Достижение более реалистичных результатов	331
Рост поля деятельности	338
Создание реалистичных фотографий знаменитостей	338
Улучшение деталей и преобразование изображений	339
<b>Глава 17. Глубокое обучение с подкреплением</b>	<b>341</b>
Играя в игру с нейронными сетями	342
Знакомство с обучением с подкреплением	343
Имитация игровой среды	345
Q-обучение	348
Объяснение Alpha-Go	351
Если вы собираетесь выиграть	352
Применение самообучения в масштабе	354
<b>Часть 4. Великолепные десятки</b>	<b>357</b>
<b>Глава 18. Десять приложений, требующих глубокого обучения</b>	<b>359</b>
Восстановление цвета на черно-белых видео и изображениях	360
Аппроксимация позы человека в реальном времени	361
Анализ поведения в реальном времени	361

Перевод	362
Оценка потенциала солнечной энергии	363
Победа над людьми в компьютерных играх	363
Генерация голоса	364
Прогнозирование демографии	365
Создание произведений искусства из реальных фотографий	366
Прогнозирование природных катастроф	367
<b>Глава 19. Десять инструментов глубокого обучения</b>	<b>369</b>
Компиляция математических выражений с использованием Theano	369
Дополнение TensorFlow с помощью Keras	370
Динамическое вычисление графиков с помощью Chainer	371
Создание среды, похожей на MATLAB, с помощью Torch	372
Динамическое выполнение задач с помощью PyTorch	373
Ускорение исследований в области глубокого обучения с использованием CUDA	373
Поддержка бизнес-потребностей с Deeplearning4j	375
Получение данных с использованием Neural Designer	376
Алгоритмы обучения с использованием Microsoft Cognitive Toolkit (CNTK)	377
Полное использование возможностей графического процессора с помощью MXNet	377
<b>Глава 20. Десять профессий, использующих глубокое обучение</b>	<b>379</b>
Управление людьми	380
Улучшение медицинского обслуживания	380
Разработка новых устройств	381
Обеспечение поддержки клиентов	382
Просмотр данных новыми способами	383
Ускорение выполнения анализа	383
Создание лучшей рабочей среды	384
Исследование неясной или подробной информации	385
Проектирование зданий	386
Повышение безопасности	387





## Глава 8

# Построение простой нейронной сети

### В ЭТОЙ ГЛАВЕ...

- » Базовая архитектура
- » Определение задачи
- » Процесс решения

**В** главе 7 представлены нейронные сети в самой простой базовой форме — в виде перцептрона. Однако нейронные сети бывают разных форм, каждая из которых имеет свои преимущества. К счастью, для достижения своей цели все формы нейронных сетей следуют базовой архитектуре и полагаются на определенные стратегии. Если вы узнаете, как работает базовая нейронная сеть, вы сможете понять, как работают более сложные архитектуры. В первой части этой главы рассматриваются основы работы нейронной сети — то есть то, что вам нужно знать, чтобы понять, как нейронная сеть выполняет свою работу. Далее объясняются функции нейронной сети на примере базовой нейронной сети, которую вы можете построить с нуля, используя язык Python.

Вторая часть главы посвящена некоторым различиям между нейронными сетями. Например, в главе 7 упоминалось, что отдельные нейроны срабатывают после достижения определенного порога. Функция активации определяет, когда входной сигнал достаточен для срабатывания нейрона, поэтому знание, какие функции активатора доступны, важно для различения нейронных сетей. Кроме того, вам необходимо знать об оптимизаторе, используемом для быстрого получения результатов, которые фактически моделируют решаемую задачу. Наконец, вам нужно решить, как быстро учится ваша нейронная сеть.



ЗАПОМНИ!

Вам не нужно вводить исходный код вручную. Намного проще использовать загружаемый исходный код (подробности о загрузке исходного кода см. во введении). Исходный код примеров из этой главы представлен в файле `DL4D_08_NN_From_Scratch.ipynb`.

## Понятие нейронных сетей

В Интернете можно найти много дискуссий об архитектуре нейронной сети (например, на <https://www.kdnuggets.com/2018/02/8-neural-network-architectures-machine-learning-researchers-need-learn.html>). Проблема, однако, в том, что все они быстро становятся безумно сложными. Некоторые неписанные законы, казалось бы, гласят, что математика мгновенно должна стать абстрактной и настолько сложной, что ни один простой смертный не сможет ее понять, однако нейронную сеть сможет понять любой. Хорошее начало дает материал главы 7. Несмотря на то, что глава 7 немного полагается на математику, чтобы донести свою мысль, эта математика относительно проста. Теперь, в этой главе вы узнаете, как внедрить в код Python все основные функции нейронной сети.

То, что на самом деле представляет собой нейронная сеть, является своего рода фильтром. Вы заливаете данные в верхнюю часть, эти данные просачиваются через различные слои, которые вы создаете, а выходные данные вытекают снизу. То, что отличает нейронные сети, — это те же самые элементы, которые вы можете найти в фильтре. Например, тип выбранного вами алгоритма определяет тип фильтрации, который будет выполнять нейронная сеть. Возможно, вы захотите отфильтровать из воды свинец, но оставить при этом кальций и другие полезные минералы, что означает выбор такого типа фильтра, который подходит именно для этого.

Однако фильтры могут быть контролируемы. Например, вы можете выбрать фильтрацию частиц одного размера, но пропускать частицы другого размера. Использование весов и смещений в нейронной сети — это просто своего рода контроль. Вы настраиваете его так, чтобы обеспечить точную фильтрацию, которая вам необходима. В этом случае, поскольку вы используете электрические сигналы, смоделированные по аналогии с сигналами, обнаруженными в мозге, сигнал может пройти, когда он удовлетворяет некому условию — порогу, определяемому функцией активации. Для простоты считайте пока это как бы настройкой работы любого фильтра.

Вы можете отслеживать активность вашего фильтра. Но если вы не хотите стоять там весь день, глядя на работу фильтра, вы, вероятно, полагаетесь на какую-то автоматизацию, чтобы гарантировать, что вывод фильтра остается

постоянным. Вот где в игру вступает оптимизатор. Оптимизируя вывод нейронной сети, вы получаете нужные результаты, избежав настройки вручную.

Наконец, вы хотите, чтобы фильтр работал с той скоростью и емкостью, которые позволят ему правильно выполнять свои задачи. Слишком быстрая заливка воды или другого вещества в фильтр может привести к его переполнению. Если вы наливаете не достаточно быстро, фильтр может засориться или работать беспорядочно. Регулировка скорости обучения оптимизатора нейронной сети позволяет вам гарантировать, что нейронная сеть выдаст желаемый результат. Это похоже на регулировку скорости заливки фильтра.

Нейронные сети могут показаться трудными для понимания. Тот факт, что многое из того, что они делают, объясняет довольно сложная математика, отнюдь не поможет. Но вам не нужно быть ученым, чтобы понять, что такое нейронные сети. Все, что вам действительно нужно сделать, это разделить их на управляемые части и использовать правильную точку зрения, чтобы взглянуть на них. В следующих разделах показано, как создать код каждой части простой нейронной сети с самого начала.

## Базовая архитектура

Нейронная сеть опирается на многочисленные вычислительные блоки, *нейроны*, организованные в иерархические слои. Каждый нейрон получает входные данные от всех своих предшественников и предоставляет выходные данные своим преемникам, пока вся нейронная сеть в целом не станет удовлетворять требованиям. В этот момент работа сети заканчивается, и вы получаете вывод.

Все эти вычисления происходят исключительно в нейронной сети. Сеть проходит по каждому из них, используя для итераций циклы. Вы также можете использовать тот факт, что большинство из этих операций представляют собой простые умножения с последующим сложением и использовать преимущества матричных вычислений, показанных в главе 5.

В примере этого раздела создается сеть с входным слоем (размеры которого определяются входными данными), скрытым слоем с тремя нейронами и одним выходным слоем, который сообщает, принадлежит ли входной элемент некому классу (проще говоря, ответ двоичный 0 или 1). Эта архитектура подразумевает создание двух наборов весов, представленных двумя матрицами (вот когда вы фактически используете матрицы).

- » Первая матрица имеет размер, определяемый количеством входов  $\times 3$ , и представляет веса, на которые умножаются вводы, и суммирует их в три нейрона.
- » Вторая матрица имеет размер  $3 \times 1$ , она собирает все выходные данные со скрытого слоя и объединяет этот слой в выходные данные.

Вот необходимый код на языке Python (выполнение которого может занять некоторое время, в зависимости от скорости вашей системы).

```
import numpy as np
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline

def init(inp, out):
    return np.random.randn(inp, out) / np.sqrt(inp)

def create_architecture(input_layer, first_layer,
                        output_layer, random_seed=0):
    np.random.seed(random_seed)
    layers = X.shape[1], 3, 1
    arch = list(zip(layers[:-1], layers[1:]))
    weights = [init(inp, out) for inp, out in arch]
    return weights
```

Интересным моментом этой инициализации является то, что для автоматизации сетевых расчетов она использует последовательность матриц. То, как код инициализирует их, имеет значение, поскольку вы не можете использовать слишком малые числа — сигнал будет слишком слабым для работы сети. Однако вы также должны избегать слишком больших чисел, поскольку вычисления становятся слишком громоздкими для обработки. Иногда они терпят неудачу, что приводит к *проблеме взрывающегося градиента* (exploding gradient problem) или иначе *насыщение нейронов* (saturation of the neuron), что означает, что вы не можете правильно обучить сеть, поскольку все нейроны всегда активированы.



ЗАПОМНИ!

Инициализация сети с использованием всех нулей всегда плохая идея, поскольку если все нейроны имеют одинаковое значение, они будут одинаково реагировать на ввод обучения. Независимо от того, сколько нейронов содержится в архитектуре, они работают как один нейрон.

Более простое решение заключается в том, чтобы начать со случайных весов, находящихся в диапазоне, необходимом для *функций активации*, которые, по сути, являются функциями преобразования, что добавляет гибкость к решению задач с использованием сети. Возможное простое решение заключается в том, чтобы установить для весов нулевое среднее и единичное стандартное отклонение, которое в статистике называется *стандартным нормальным распределением* (standard normal distribution), а в коде отображается как команда `np.random.randn`.



СОВЕТ

Тем не менее, есть более разумные инициализации весов для более сложных сетей, таких как описаны в этой статье: <https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78>.

Более того, поскольку каждый нейрон получает входные данные всех предыдущих нейронов, код изменяет масштаб случайных нормально распределенных весов, используя квадратный корень из количества входных данных. Следовательно, нейроны и их функции активации всегда вычисляют правильный размер, чтобы все работало гладко.

## Документирование основных модулей

Архитектура является лишь частью нейронной сети. Вы можете представить ее как структуру сети. Архитектура объясняет, как сеть обрабатывает данные и предоставляет результаты. Тем не менее, для любой обработки, вам также необходимо реализовать в коде основные функции нейронной сети.

Первым строительным блоком сети является функция активации. В главе 7 подробно описаны некоторые функции активации, используемые в нейронных сетях, без подробного их объяснения. Пример в этом разделе предоставляет код для сигмоидной функции, одной из основных функций активации нейронной сети. *Сигмоидная функция* (sigmoid function) — это следующий шаг по сравнению со *ступенчатой функцией Хевисайда* (Heaviside step function), которая действует как переключатель, активирующийся при определенном пороге. Ступенчатая функция Хевисайда возвращает значение 1 для вводов превышающих порог и 0 для вводов ниже него.

Сигмоидные функции выводят 0 или 1 соответственно для малых входных значений ниже нуля или высоких значений выше нуля. Для входных значений в диапазоне от  $-5$  до  $+5$  функция выводит значения в диапазоне  $0-1$ , медленно увеличивая выводимые значения до тех пор, пока оно не достигнет примерно  $0,2$ , а затем линейно увеличивая его до значения  $0,8$ . Затем она снова уменьшает вывод по мере приближения к 1. Такое поведение представляет собой логистическую кривую, применимую при описании многих природных явлений, таких как рост популяции, которая начинает рост медленно, а затем линейно развивается, пока ее рост не замедлится при почти полном исчерпании ресурсов (например, доступное жилое пространство или еда).

В нейронных сетях сигмоидная функция особенно полезна для моделирования входных данных, которые похожи на вероятности, и она *дифференцируема*, что является математическим аспектом, помогающим обратить вспять ее эффекты и выработать лучшую фазу обратного распространения, упомянутую в главе 7.

```
def sigmoid(z):
    return 1/(1 + np.exp(-z))

def sigmoid_prime(s):
    return s * (1 -s)
```

Получив функцию активации, вы можете создать *процедуру передачи* (forward procedure), представляющую собой матричное умножение между входными данными для каждого слоя и весами соединения. После завершения умножения код применяет к результатам функцию активации, чтобы преобразовать их нелинейным способом. Следующий код встраивает сигмоидную функцию в код прямой передачи по сети. Конечно, при желании вы можете использовать любую другую функцию активации.

```
def feed_forward(X, weights):
    a = X.copy()
    out = list()
    for W in weights:
        z = np.dot(a, W)
        a = sigmoid(z)
        out.append(a)
    return out
```

Применяя прямую передачу ко всей сети, вы, наконец, получите результат в выходном слое. Теперь вы можете сравнить выходные данные с реальными значениями, которые необходимо получить в сети. Сравнивая количество правильных предположений с общим количеством предоставленных прогнозов, функция точности определяет, хорошо ли нейронная сеть осуществляет прогнозирование.

```
def accuracy(true_label, predicted):
    correct_preds = np.ravel(predicted)==true_label
    return np.sum(correct_preds) / len(true_label)
```

Далее следует функция обратного распространения, поскольку сеть работает, но все или некоторые прогнозы неверны. Исправление прогнозов во время обучения позволяет создать нейронную сеть, способную получать новые примеры и предоставлять хорошие прогнозы. Обучение включается в весовые коэффициенты соединения в виде шаблонов, представленных в данных, и способные помочь правильно предсказать результаты.

Для выполнения обратного распространения, вы сначала вычисляете ошибку в конце каждого слоя (в этой архитектуре их два). Вы умножаете ошибку на производную функции активации. Результат дает вам градиент, то есть изменение весов, необходимое для более правильного вычисления прогнозов. Код начинается со сравнения выходных данных с правильными ответами (`l2_error`),

а затем вычисляет градиенты, являющиеся необходимыми поправками веса (`l2_delta`). Затем код переходит к умножению градиентов на веса, которые код должен исправить. Операция распространяет ошибку из выходного слоя на промежуточный (`l1_error`). Новое вычисление градиента (`l1_delta`) также предоставляет поправки веса для применения к входному слою, который завершает процесс для сети с входным слоем, скрытым слоем и выходным слоем.

```
def backpropagation(l1, l2, weights, y):
    l2_error = y.reshape(-1, 1) - l2
    l2_delta = l2_error * sigmoid_prime(l2)
    l1_error = l2_delta.dot(weights[1].T)
    l1_delta = l1_error * sigmoid_prime(l1)
    return l2_error, l1_delta, l2_delta
```



ЗАПОМНИ!

Это перевод в код Python, в упрощенной форме, формул главы 7. Функция стоимости — это разница между выводом сети и правильными ответами. В этом примере не добавляются смещения фазы прямой передачи, что уменьшает сложность процесса обратного распространения и облегчает понимание.

После того, как обратное распространение назначит каждому соединению свою часть коррекции, которая должна применяться ко всей сети, вы корректируете начальные веса так, чтобы представить обновленную нейронную сеть. Это осуществляется в результате добавления к весам каждого слоя, умножения входных данных для этого слоя и поправок дельта для слоя в целом. Это этап метода градиентного спуска, при котором вы подходите к решению, предпринимая многократные небольшие шаги в правильном направлении, поэтому вам может потребоваться отрегулировать размер шага, используемого для решения задачи. Изменение размера шага позволяют осуществить параметры альфа. Значения 1 не влияют на результат предыдущей коррекции веса, но значения меньше 1 эффективно уменьшат его.

```
def update_weights(X, l1, l1_delta, l2_delta, weights,
                  alpha=1.0):
    weights[1] = weights[1] + (alpha * l1.T.dot(l2_delta))
    weights[0] = weights[0] + (alpha * X.T.dot(l1_delta))
    return weights
```

Нейронная сеть не является полной, если она может только учиться на основе данных, но не прогнозировать. Последняя функция, `predict`, выдает новые данные с использованием прямой связи, читает последний выходной слой и преобразует его значения в прогноз задачи. Поскольку сигмоидная функция активации настолько хороша для моделирования вероятности, код использует значение на полпути между 0 и 1, то есть 0,5, в качестве порога для получения

положительного или отрицательного вывода. Такой двоичный вывод может помочь при классификации двух классов или одного класса по отношению ко всем остальным, если набор данных имеет три или более типов результатов для классификации.

```
def predict(X, weights):  
    _, l2 = feed_forward(X, weights)  
    preds = np.ravel((l2 > 0.5).astype(int))  
    return preds
```

На данный момент в примере есть все части, обеспечивающие работу нейронной сети. Вам просто нужна задача, демонстрирующая работу нейронной сети.

## Решение простой задачи

В этом разделе вы проверите написанный вами код нейронной сети, попробовав его решить простую, но не банальную задачу с данными. В коде для создания двух перемежающихся окружностей точек в форме двух полумесяцев используется функция `make_moons` из пакета `Scikit-learn`. Разделение этих двух окружностей требует алгоритма, способного определять нелинейную функцию разделения, которая обобщает и новые случаи того же рода. Нейронная сеть, такая как представленная ранее в этой главе, может легко справиться с этой задачей.

```
np.random.seed(0)  
  
coord, cl = make_moons(300, noise=0.05)  
X, Xt, y, yt = train_test_split(coord, cl,  
                                test_size=0.30,  
                                random_state=0)  
  
plt.scatter(X[:,0], X[:,1], s=25, c=y, cmap=plt.cm.Set1)  
plt.show()
```

Сначала код устанавливает случайное начальное число для получения одинакового результата при каждом запуске примера. Следующим шагом является создание 300 примеров данных и разделение их на наборы проверочных и тестовых данных. (Набор тестовых данных составляет 30% от общего количества.) Данные состоят из двух переменных, представляющих координаты  $x$  и  $y$  точек на декартовой плоскости. На рис. 8.1 показаны результаты этого процесса.

Поскольку обучение нейронной сети осуществляется в ходе последовательных итераций (называемых *эпохами*), после создания и инициализации набора весов код проводит 30 000 итераций для данных двух полумесяцев (каждый



проход является эпохой). На каждой итерации код вызывает некоторые из ранее подготовленных базовых функций нейронной сети.

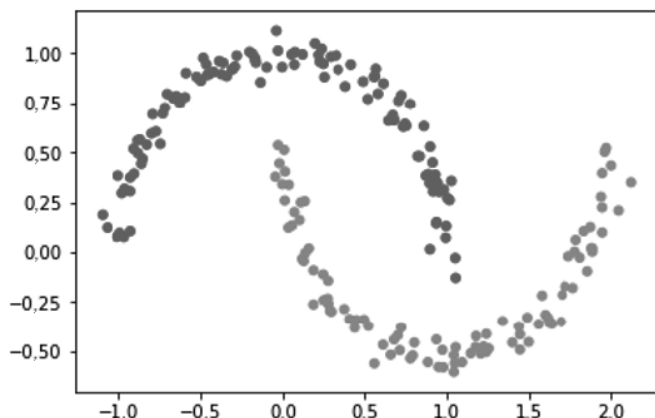


Рис. 8.1. Два чередующихся облака точек данных в форме полумесяца

- » Передача данных через всю сеть.
- » Обратное распространение ошибки по сети.
- » Обновление веса каждого слоя сети, на основании обратного распространения ошибки.
- » Вычисление ошибки обучения и проверки.

Следующий код использует комментарии для детализации работы каждой функции.

```
weights = create_architecture(X, 3, 1)

for j in range(30000 + 1):

    # Сначала прямая передача через скрытый слой
    l1, l2 = feed_forward(X, weights)

    # Затем обратное распространение ошибки от выхода к входу
    l2_error, l1_delta, l2_delta = backpropagation(l1,
                                                    l2, weights, y)

    # Наконец, обновление весов сети
    weights = update_weights(X, l1, l1_delta, l2_delta,
                             weights, alpha=0.05)

    # Время от времени, сообщать о результатах
    if (j % 5000) == 0:
```

```

train_error = np.mean(np.abs(l2_error))
print('Epoch {:5}'.format(j), end=' - ')
print('error: {:.4f}'.format(train_error),
      end= ' - ')
train_accuracy = accuracy(true_label=y,
                          predicted=(l2 > 0.5))
test_preds = predict(Xt, weights)
test_accuracy = accuracy(true_label=yt,
                        predicted=test_preds)
print('acc: train {:.3f}'.format(train_accuracy),
      end= ' | ')
print('test {:.3f}'.format(test_accuracy))

```

Переменная `j` считает итерации. На каждой итерации код пытается разделить `j` на 5000 по модулю и проверяет результат. Когда модуль равен нулю, код понимает, что с момента предыдущей проверки прошло 5000 эпох. А значит, возможно суммирование ошибки нейронной сети в ходе проверки ее точности (сколько раз прогноз является правильным по отношению к общему количеству прогнозов) на учебном и тестовом наборах. Точность учебного набора показывает, насколько хорошо нейронная сеть подбирает данные, адаптируя их параметры в процессе обратного распространения. Точность тестового набора дает представление о том, насколько хорошо решение обобщено для новых данных, а, следовательно, можно ли использовать его повторно.



СОВЕТ

Точность тестового набора должна иметь наибольшее значение, поскольку она демонстрирует потенциальную применимость нейронной сети с другими данными. Точность обучающего набора просто говорит о том, как сеть оценивает используемые данные.

## Внутренняя работа нейронных сетей

Узнав, как нейронные сети работают в основном, вам нужно лучше понять, что их отличает. Существуют различия и помимо архитектур, выбора функций активации, оптимизаторов и скорости обучения нейронной сети. Знания основных операций недостаточно, поскольку вы не получите желаемых результатов. Взгляд изнутри поможет понять, как можно настроить решение нейронной сети для моделирования конкретных задач. Кроме того, понимание различных алгоритмов, используемых для создания нейронной сети, поможет вам быстрее получить лучшие результаты и с меньшими усилиями. Следующие разделы посвящены трем областям отличий нейронных сетей.

## Выбор правильной функции активации

Функция активации просто определяет, когда сработает нейрон. Считайте, что это своего рода переломный момент: ввод определенного значения не вызовет срабатывания нейрона, поскольку этого недостаточно, но чуть большее значение ввода может вызвать срабатывание нейрона. Нейрон в простой форме определяется следующим образом.

$$y = \Sigma (\text{weight} * \text{input}) + \text{bias}$$

Вывод  $y$  может быть любым значением между  $+$  и  $-$  бесконечностью. Таким образом, задача заключается в том, чтобы решить, какое значение  $y$  является значением срабатывания, и где в игру вступает функция активации. Функция активации определяет, какое значение является достаточно высоким или низким, чтобы отразить точку принятия решения в нейронной сети для конкретного нейрона или группы нейронов.

Как и все остальное в нейронных сетях, у вас нет только одной функции активации. Вы используете ту функцию активации, которая лучше всего работает в конкретном случае. С учетом этого, функции активации можно отнести к следующим категориям.

- » **Шаговая.** *Шаговая функция* (step function) (или двоичная функция) полагается на определенный порог для принятия решения об активации. Использование шаговой функции означает, что вы знаете, какое конкретное значение вызовет активацию. Однако шаговые функции ограничены тем, что они либо полностью активированы, либо полностью деактивированы — оттенки отсутствуют. Следовательно, при попытке определить на основании заданного ввода, какой класс, скорее всего, правильный, пошаговая функция не будет работать.
- » **Линейная.** *Линейная функция* (linear function) ( $A = cx$ ) обеспечивает простое линейное определение активации на основе входных данных. Использование линейной функции помогает определить, какой выход активировать, на основании того, какой выход является наиболее правильным (как указано весами). Однако линейные функции работают только как один слой. Если вы хотите собирать несколько слоев линейных функций, результат будет как при использовании одного слоя, что противоречит цели использования нейронных сетей. Следовательно, линейная функция может использовать один слой, но не несколько.
- » **Сигмоидная.** *Сигмоидная функция* (sigmoid function) ( $A = 1 / 1 + e^{-x}$ ), создающая кривую в форме буквы S или S, является нелинейной. Сначала она выглядит как шаговая функция, за исключением того, что значения между двумя точками фактически расположены на

кривой, а значит сигмоидные функции можно располагать слоями для выполнения классификации с несколькими выходами. Диапазон сигмоидной функции распространяется от 0 до 1, а не от  $-\infty$  до  $+\infty$ , как с линейной функцией, поэтому активации ограничены в определенном диапазоне. Однако сигмоидная функция страдает от проблемы *исчезающего градиента* (*vanishing gradient*), когда функция отказывается учиться после определенной точки, поскольку распространяемая ошибка уменьшается до нуля при приближении к удаленным слоям.

- » **Tanh.** Функция  $\tanh$  ( $A = (2 / (1 + e^{-2x})) - 1$ ) на самом деле является масштабной сигмоидной функцией. Она имеет диапазон от  $-1$  до  $1$ , поэтому это точный метод активации нейронов. Основная разница между сигмоидными функциями и функциями  $\tanh$  в том, что градиент функции  $\tanh$  сильнее, а значит обнаружение небольших различий легче, что делает классификацию более чувствительной. Подобно сигмоидной функции, функция  $\tanh$  страдают от исчезающего градиента.
- » **ReLU.** Функция *ReLU* или *линейный выпрямитель* (Rectified Linear Units) ( $A(x) = \max(0, x)$ ) обеспечивает вывод в диапазоне от 0 до бесконечности, поэтому она похожа на линейную функцию, за исключением того, что она также нелинейна, что позволяет наслаивать функции ReLU. Преимущество ReLU в том, что она требует меньше вычислительной мощности, потому что срабатывает меньше нейронов. Отсутствие активности, когда нейрон приближается к нулевой части линии, означает, что слишком мало потенциальных выходов для рассмотрения. Тем не менее, это преимущество также может стать недостатком, когда у вас есть проблема *затухающего ReLU* (*dying ReLU*). Через некоторое время веса нейронной сети уже не дают желаемого эффекта (она просто прекращает обучение), и пораженные нейроны умирают — они не реагируют ни на какие входные данные.
- » **ELU.** *Экспоненциальная линейная функция* (Exponential Linear Unit). Отличается от ReLU, когда вводы отрицательны. В этом случае выходные данные не стремятся к нулю, а постепенно экспоненциально уменьшаются до  $-1$ .
- » **PReLU.** *Параметрический линейный выпрямитель* (Parametric Rectified Linear Unit). Отличается от ReLU, когда вводы отрицательны. В этом случае вывод является линейной функцией, параметры которой изучаются с использованием той же методики, что и любой другой параметр сети.
- » **LeakyReLU.** Аналогичен PReLU, но параметр для линейной стороны фиксирован.

## Полагаясь на умный оптимизатор

Оптимизатор обеспечивает быстрое и правильное моделирование нейронной сетью любой задачи, которую вы хотите решить, изменяя смещения и веса нейронной сети. Оказывается, что эту задачу выполняет алгоритм, но вы должны выбрать правильный алгоритм, чтобы получить ожидаемые результаты. Как и во всех случаях нейронной сети, у вас есть несколько необязательных типов алгоритмов, из которых можно выбрать (см. <https://keras.io/optimizers/>).

- » Стохастический градиентный спуск (SGD).
- » RMSProp.
- » AdaGrad.
- » AdaDelta.
- » AMSGrad.
- » Adam и его варианты, Adamax и Nadam.

Оптимизатор работает минимизируя или максимизируя выходные данные целевой функции (функции ошибок), представленной как  $E(x)$ . Эта функция зависит от внутренних обучаемых параметров модели, используемых для вычисления целевых значений ( $Y$ ) из предикторов ( $X$ ). Двумя внутренними обучаемыми параметрами являются веса ( $W$ ) и смещения ( $b$ ). Различные алгоритмы имеют разные методы работы с целевой функцией.

Вы можете классифицировать функции оптимизатора по тому, как они работают с производной ( $dy/dx$ ), которая представляет собой мгновенное изменение  $y$  относительно  $x$ . Вот два уровня обработки производных.

- » **Первый порядок.** Эти алгоритмы минимизируют или максимизируют целевую функцию, используя значения градиента по отношению к параметрам.
- » **Второй порядок.** Эти алгоритмы минимизируют или максимизируют объектную функцию, используя значения производных второго порядка по параметрам. Производная второго порядка может дать подсказку о том, увеличивается или уменьшается производная первого порядка, что дает информацию о кривизне линии.

Вы обычно используете методы оптимизации первого порядка, такие как Gradient Descent, поскольку они требуют меньше вычислений и имеют тенденцию относительно быстро сходиться к хорошему решению при работе с большими наборами данных.

## Установка рабочей скорости обучения

Каждый оптимизатор имеет совершенно разные параметры для настройки. Одной из констант является фиксация скорости обучения, представляющей собой скорость, с которой код обновляет веса в сети (например, параметр `alpha`, используемый в примере этой главы). Скорость обучения может влиять как на время, необходимое нейронной сети для получения хорошего решения (количество эпох), так и на результат. На самом деле, если скорость обучения слишком низка, сеть будет учиться вечно. Установка слишком высокого значения приводит к нестабильности при обновлении весов, и сеть никогда не сможет найти хорошее решение.

Выбор рабочей скорости обучения является пугающим потому, что вы можете опробовать значения в диапазоне от 0,000001 до 100. Лучшее значение варьируется от оптимизатора к оптимизатору. Значение, которое вы выбираете, зависит от того, какой тип данных у вас есть. Теория может здесь мало помочь; вы должны проверить различные комбинации, прежде чем найдете наиболее подходящую скорость для успешного обучения вашей нейронной сети.



ЗАПОМНИ!

Несмотря на всю окружающую их математику, настройка нейронных сетей и обеспечение их наилучшей работы — это в основном вопрос эмпирических усилий при опробовании различных комбинаций архитектур и параметров.