O'REILLY®

«Все, что должен знать разработчик-практик, чтобы приступить к применению глубокого обучения для решения реальных задач.»

*— **Грант Ингерсолл,** технический директор, Lucidworks*

Интерес к машинному обучению зашкаливает, но завышенные ожидания нередко губят проекты еще на ранней стадии. Как машинное обучение — и особенно глубокие нейронные сети — может изменить вашу организацию? Эта книга не только содержит практически полезную информацию о предмете, но и поможет приступить к созданию эффективных сетей глубокого обучения.

Авторы сначала раскрывают фундаментальные вопросы глубокого обучения — настройка, распараллеливание, векторизация, конвейеры операций — актуальные для любой библиотеки, а затем переходят к библиотеке Deeplearning4j (DL4J), предназначенной для разработки технологических процессов профессионального уровня. На реальных примерах читатель познакомится с методами и стратегиями обучения глубоких сетей с различной архитектурой и их распараллеливания в кластерах Hadoop и Spark.

Рассматриваются следующие темы:

- концепции машинного обучения вообще и глубокого обучения в частности;
- эволюция глубоких сетей из нейронных;
- основные архитектуры глубоких сетей, в т. ч. сверточные и рекуррентные нейронные сети;
- как выбрать сеть, отвечающую поставленной задаче;
- основы настройки нейронных сетей вообще и конкретных глубоких архитектур;
- применение методов векторизации к данным различных типов с помощью библиотеки DataVec;
- использование DL4J на платформах Hadoop и Spark.

***Джош Паттерсон (Josh Patterson)** руководит отделом поддержки проектов в компании Skymind. Раньше работал главным архитектором решений в компании Cloudera и инженером по машинному обучению и распределенным системам в Управлении ресурсами бассейна реки Теннесси.*

***Адам Гибсон (Adam Gibson)** — технический директор Skymind. Работал с компаниями из списка Fortune 500, хэджевыми фондами, компаниями в области связей с общественностью и акселераторами стартапов, помогая в разработке проектов машинного обучения. Имеет большой опыт консультирования компаний в области обработки и интерпретации больших данных.*

ISBN 978-5-97060-481-6

9 785970 604816 >

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
тел.(499)782-38-89
books@alians-kniga.ru

www.дмк.рф

Глубокое обучение с точки зрения практика

O'REILLY®

Глубокое обучение

с точки зрения практика

Джош Паттерсон,
Адам Гибсон

Джош Паттерсон, Адам Гибсон

Глубокое обучение с точки зрения практика

Josh Patterson and Adam Gibson

Deep Learning

A Practitioner's Approach

Beijing • Boston • Farnham • Sebastopol • Tokyo

Джош Паттерсон, Адам Гибсон

Глубокое обучение с точки зрения практика

Москва, 2018

УДК 004.85Deeplearning4j
ББК 32.813
П20

Паттерсон Дж., Гибсон А.
П20 Глубокое обучение с точки зрения практика / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2018. – 418 с.: ил.

ISBN 978-5-97060-481-6

Интерес к машинному обучению зашкаливает, но завышенные ожидания нередко губят проекты еще на ранней стадии. Как машинное обучение — и особенно глубокие нейронные сети — может изменить вашу организацию? Эта книга не только содержит практически полезную информацию о предмете, но и поможет приступить к созданию эффективных сетей глубокого обучения.

Авторы сначала раскрывают фундаментальные вопросы глубокого обучения — настройка, распараллеливание, векторизация, конвейеры операций, а затем переходят к библиотеке Deeplearning4j (DL4J), предназначенной для разработки технологических процессов профессионального уровня. На реальных примерах читатель познакомится с методами и стратегиями обучения глубоких сетей с различной архитектурой и их распараллеливания в кластерах Hadoop и Spark.

Издание предназначено для специалистов по анализу данных, находящихся в поисках более широкого и практического понимания принципов глубокого обучения.

УДК 004.85Deeplearning4j
ББК 32.813

Authorized Russian translation of the English edition of Deep Learning, ISBN 9781491914250. © 2017 Josh Patterson, Adam Gibson.

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

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-491-91425-0 (анг.)
ISBN 978-5-97060-481-6 (рус.)

Copyright © 2017 Josh Patterson, Adam Gibson
© Оформление, издание, перевод, ДМК Пресс, 2018

Содержание

Предисловие 14

Глава 1. Обзор машинного обучения 20
Обучающиеся машины 20
Как машины могут обучаться? 21
Биологические корни 23
Что такое глубокое обучение? 24
Вниз по кроличьей норе 25
Формулировка вопросов 26
Математические основания машинного обучения: линейная алгебра 26
Скаляры 26
Векторы 26
Матрицы 27
Тензоры 27
Гиперплоскости 28
Математические операции 28
Преобразование данных в векторы 28
Решение систем уравнений 30
Математические основания машинного обучения: статистика 32
Вероятность 32
Условные вероятности 34
Апостериорная вероятность 34
Распределения вероятности 35
Выборка и генеральная совокупность 37
Методы с перевыборкой 37
Смещение выборки 37
Правдоподобие 38
Как работает машинное обучение? 38
Регрессия 38
Классификация 40
Кластеризация 40
Недообучение и переобучение 41
Оптимизация 41
Выпуклая оптимизация 42
Градиентный спуск 43
Стохастический градиентный спуск 45
Квазиньютоновские методы оптимизации 46
Порождающие и дискриминантные модели 46
Логистическая регрессия 47
Логистическая функция 48
Интерпретация результата логистической регрессии 48
Оценивание модели 49
Матрица неточностей 49
Итоги 52

Глава 2. Основы нейронных сетей и глубокого обучения 53
Нейронные сети 53
Биологический нейрон 55
Перцептрон 57
Многослойные сети прямого распространения 60
Обучение нейронных сетей 64
Обучение методом обратного распространения 65
Функции активации 71
Линейная функция 71
Сигмоида 72
Функция tanh 73
Функция hardtanh 73
Функция softmax 73
Линейная ректификация 74
Функции потерь 76
Применяемые обозначения 76
Функции потерь для регрессии 77
Функции потерь для классификации 78
Функции потерь для реконструкции 80
Гиперпараметры 81
Скорость обучения 81
Регуляризация 82
Импульс 82
Разреженность 82

Глава 3. Основания глубоких сетей 83
Определение глубокого обучения 83
Что такое глубокое обучение? 83
Как организована эта глава 92
Общие архитектурные принципы глубоких сетей 92
Параметры 93
Слои 93
Функции активации 94
Функции потерь 95
Алгоритмы оптимизации 96
Гиперпараметры 98
Итоги 102
Строительные блоки глубоких сетей 102
Ограниченные машины Больцмана 103
Автокодировщики 108
Вариационные автокодировщики 109

Глава 4. Основные архитектуры глубоких сетей 111
Сети, предобученные без учителя 111
Глубокие сети доверия 111
Порождающие состязательные сети 114
Сверточные нейронные сети (СНС) 117
Биологические корни 118
Интуитивное описание 119
Общий взгляд на архитектуру СНС 120
Входной слой 121

Пулинговые слои ... 128
Полносвязные слои ... 129
Другие применения СНС ... 129
Самые известные СНС ... 130
Итоги ... 130
Рекуррентные нейронные сети ... 131
Моделирование времени ... 131
Трехмерный вход ... 133
Почему не марковские модели? ... 135
Общая архитектура рекуррентной нейронной сети ... 136
LSTM-сети ... 136
Предметно-ориентированные приложения и гибридные сети ... 143
Рекурсивные нейронные сети ... 144
Архитектура сети ... 144
Разновидности рекурсивных нейронных сетей ... 145
Применение рекурсивных нейронных сетей ... 145
Итоги и обсуждение ... 145
Приведет ли глубокое обучение к отмиранию всех прочих алгоритмов? ... 146
Оптимальный метод зависит от задачи ... 146
Когда мне может понадобиться глубокое обучение? ... 147

Глава 5. Построение глубоких сетей ... 148
Выбор глубокой сети для решения задачи ... 148
Табличные данные и многослойные перцептроны ... 148
Изображения и сверточные нейронные сети ... 149
Последовательности, временные ряды и рекуррентные нейронные сети ... 150
Применение гибридных сетей ... 151
Инструментарий DL4J ... 151
Векторизация и DataVec ... 152
Среды выполнения и ND4J ... 152
Основные концепции DL4J API ... 153
Загрузка и сохранение моделей ... 153
Получение входных данных для модели ... 154
Задание архитектуры модели ... 154
Обучение и оценивание ... 155
Моделирование CSV-данных с помощью многослойных перцептронов ... 156
Подготовка входных данных ... 158
Задание архитектуры сети ... 159
Обучение модели ... 161
Оценивание модели ... 161
Моделирование рукописных цифр с помощью СНС ... 162
Реализация СНС LeNet на Java ... 163
Загрузка и векторизация входных изображений ... 165
Архитектура сети LeNet в DL4J ... 165
Обучение СНС ... 168
Моделирование последовательных данных с помощью рекуррентной нейронной сети ... 169
Порождение текста в стиле Шекспира с помощью LSTM-сети ... 169
Классификация временных рядов, содержащих показания датчика, с помощью LSTM-сети ... 177
Применение автокодировщиков для обнаружения аномалий ... 183

Java-программа автокодировщика 183
Подготовка входных данных 187
Архитектура и обучение сети автокодировщика 187
Оценивание модели 188
Использование вариационных автокодировщиков для реконструкции цифр из набора MNIST 189
Программа реконструкции цифр для набора MNIST 190
Изучение модели VAE 192
Применение глубокого обучения в обработке естественного языка 195
Обучение погружениям слов с помощью Word2Vec 196
Распределенные представления предложений с помощью векторов абзацев 201
Применение векторов абзацев для классификации документов 204

Глава 6. Настройка глубоких сетей 209
Основные концепции настройки глубоких сетей 209
Интуитивное описание построения глубоких сетей 209
Преобразование интуитивных представлений в пошаговую процедуру 210
Подбор сетевой архитектуры, соответствующей входным данным 211
Итоги 212
Соотнесение назначения модели с выходным слоем 213
Выходной слой регрессионной модели 213
Выходной слой модели классификации 213
Количество слоев, количество параметров и объем памяти 216
Многослойные нейронные сети прямого распространения 216
Управление количеством слоев и параметров 217
Оценка требований к объему памяти 219
Стратегии инициализации весов 220
Ортогональная инициализация весов в РНС 221
Выбор функции активации 221
Сводная таблица функций активации 223
Применение функций потерь 223
Скорость обучения 225
Использование отношения обновлений к параметрам 226
Конкретные рекомендации по заданию скорости обучения 227
Как разреженность влияет на обучение 228
Применение методов оптимизации 229
Рекомендации по применению СГС 230
Применение распараллеливания и GPU для ускорения обучения 231
Онлайновое обучение и параллельные итеративные алгоритмы 232
Распараллеливание СГС в DL4J 234
Графические процессоры 236
Управление периодами и размером мини-пакета 237
Компромиссы при определении размера мини-пакета 238
О применении регуляризации 239
Априорная функция как регуляризатор 239
Регуляризация по максимальной норме 240
Прореживание 240
Другие вопросы, связанные с регуляризацией 242
Дисбаланс классов 242
Методы выборки из классов 244
Взвешенные функции потерь 244

Борьба с переобучением 245
Использование статистики сети из интерфейса настройки 246
Обнаружение неудачной инициализации весов 248
Обнаружение неперемешанных данных 249
Обнаружение проблем, относящихся к регуляризации 251

Глава 7. Настройка глубоких сетей с конкретной архитектурой 253
Сверточные нейронные сети (СНС) 253
Общие архитектурные паттерны сверточных сетей 254
Конфигурирование сверточных слоев 257
Конфигурирование пулинговых слоев 261
Перенос обучения 262
Рекуррентные нейронные сети 263
Входные данные и входной слой сети 264
Выходные слои и RnnOutputLayer 264
Обучение сети 265
Отладка типичных проблем в LSTM 267
Дополнение и маскирование 267
Применение маскирования для оценивания и скоринга 268
Варианты архитектуры рекуррентных сетей 269
Ограниченные машины Больцмана 269
Скрытые блоки и моделирование доступной информации 270
Типы блоков 271
Регуляризация в ОМБ 271
Глубокие сети доверия 272
Применение импульса 272
Применение регуляризации 273
Задание числа скрытых блоков 273

Глава 8. Векторизация 274
Введение в векторизацию в машинном обучении 274
Зачем нужно векторизовать данные? 275
Стратегии обработки табличных исходных данных 277
Конструирование признаков и методы нормировки 279
Применение библиотеки DataVec для ETL и векторизации 285
Векторизация изображений 286
Представление изображений в DL4J 286
Нормировка данных изображения с помощью DataVec 288
Векторизация последовательных данных 289
Основные виды источников последовательных данных 289
Векторизация последовательных данных с помощью DataVec 290
Векторизация текста 294
Мешок слов 295
TF-IDF 296
Сравнение Word2Vec и векторной модели 299
Работа с графами 300

Глава 9. Глубокое обучение и DL4J на платформе Spark 301
Введение в использование DL4J совместно с Spark и Hadoop 301
Запуск Spark из командной строки 303
Конфигурирование и настройка Spark 305

Выполнение Spark в кластере Mesos ... 306
Выполнение Spark поверх YARN ... 307
Общее руководство по настройке Spark ... 309
Настройка задач DL4J для Spark ... 311
Подготовка проекта Maven для Spark и DL4J ... 312
Шаблон секции зависимостей в файле pom.xml ... 314
Настройка POM-файла для CDH 5.X ... 317
Настройка POM-файла для HDP 2.4 ... 317
Отладка Spark и Hadoop ... 318
Типичные проблемы при работе с ND4J ... 318
Параллельное выполнение DL4J на платформе Spark ... 319
Минимальный пример обучения на платформе Spark ... 320
Рекомендации по использованию DL4J API для Spark ... 322
Пример многослойного перцептрона на платформе Spark ... 323
Конфигурирование архитектуры МСП для Spark ... 326
Распределенное обучение и оценивание модели ... 327
Создание и выполнение задачи Spark ... 328
Порождение текстов в стиле Шекспира с помощью Spark и LSTM-сети ... 328
Конфигурирование архитектуры LSTM-сети ... 330
Обучение, наблюдение за ходом работы и интерпретация результатов ... 331
Моделирование набора MNIST с помощью сверточной нейронной сети в кластере Spark ... 332
Конфигурирование задачи Spark и загрузка набора данных MNIST ... 334
Конфигурирование и обучение СНС LeNet ... 335

Приложение А. Что такое искусственный интеллект? ... 337
Положение на данный момент ... 338
Определение глубокого обучения ... 338
Определение искусственного интеллекта ... 338
Зима не за горами ... 345

Приложение В. RL4J и обучение с подкреплением ... 347
Введение ... 347
Марковский процесс принятия решений ... 347
Терминология ... 348
Различные варианты ... 349
Безмодельное обучение ... 349
Наблюдаемое состояние ... 349
Однопользовательские и состязательные игры ... 349
Q-обучение ... 350
От политики к нейронным сетям ... 350
Перебор политик ... 352
Исследование и использование ... 355
Уравнение Беллмана ... 356
Выборка начальных состояний ... 357
Реализация Q-обучения ... 358
Моделирование Q(s,a) ... 359
Воспроизведение опыта ... 359
Сверточные слои и предварительная обработка изображений ... 360
Обработка истории ... 361
Двойное Q-обучение ... 361

Ограничение ... 362
Масштабирование вознаграждений ... 362
Приоритетное воспроизведение ... 362
График, визуализация и среднее значение Q ... 362
RL4J ... 365
Заключение ... 366

Приложение C. Числа, которые должен знать каждый ... 367

Приложение D. Нейронные сети и обратное распространение: математическое описание ... 368
Введение ... 368
Обратное распространение в многослойном перцептроне ... 369

Приложение E. ND4J API ... 372
Дизайн и основы использования ... 372
Что такое NDArray? ... 373
Общий синтаксис ND4J ... 374
Основы работы с массивами NDArray ... 375
Класс Dataset ... 377
Создание входных векторов ... 378
Основы создания векторов ... 378
Класс MLLibUtil ... 379
Преобразование INDArray в вектор MLLib ... 379
Преобразование вектора MLLib в INDArray ... 379
Получение предсказаний от модели в DL4J ... 380
Совместное использование DL4J и ND4J ... 380

Приложение F. Библиотека DataVec ... 382
Загрузка данных для машинного обучения ... 382
Загрузка CSV-данных для многослойного перцептрона ... 384
Загрузка изображений для сверточной нейронной сети ... 385
Загрузка последовательных данных для рекуррентных нейронных сетей ... 386
Подготовка данных средствами DataVec ... 387
Преобразования DataVec: основные понятия ... 388
Преобразования DataVec: пример ... 389

Приложение G. Работа с DL4J на уровне исходного кода ... 391
Проверка, установлен ли Git ... 391
Клонирование основных проектов, связанных с DL4J ... 391
Скачивание исходного кода в виде zip-файла ... 392
Сборка библиотеки из исходного кода с помощью Maven ... 392

Приложение H. Подготовка проектов на базе DL4J ... 393
Создание нового проекта на базе DL4J ... 393
Java ... 393
Работа с Maven ... 394
Интегрированные среды разработки (IDE) ... 395
Настройка других POM-файлов Maven ... 396
ND4J и Maven ... 396

Приложение I. Подготовка проектов на базе DL4J к работе с GPU ... 397
Переключение библиотек в режим работы с GPU ... 397
Выбор GPU ... 397
Обучение на системе с несколькими GPU ... 398
CUDA на разных платформах ... 398
Мониторинг производительности GPU ... 399

Приложение J. Отладка проблем с установкой DL4J ... 400
Предыдущая установка ... 400
Ошибки нехватки памяти при сборке из исходного кода ... 400
Старые версии Maven ... 400
Maven и переменная среды PATH ... 400
Недопустимые версии JDK ... 401
C++ и другие средства разработки ... 401
Windows и путь к включаемым файлам ... 401
Мониторинг GPU ... 401
Использование JVisualVM ... 401
Работа с Clojure ... 402
Поддержка чисел с плавающей точкой в OS X ... 402
Ошибка разветвления-соединения в Java 7 ... 402
Предостережения ... 402
Клонирование других репозиториев ... 402
Проверьте зависимости Maven ... 403
Переустановка зависимостей ... 403
Если ничего не помогает ... 403
Различные платформы ... 403
OS X ... 403
Windows ... 403
Linux ... 404

Предметный указатель ... 405

Об авторах ... 416

Об иллюстрации на обложке ... 417

Посвящается моим сыновьям Этану, Гриффину и Дэйну:
идите вперед, будьте упорными и храбрыми.

– Дж. Гибсон

Предисловие

О структуре книги

В первых четырех главах излагаются теория и фундаментальные факты в объеме, достаточном для того, чтобы специалист-практик мог двигаться дальше. А в последних пяти главах на этой основе исследуются различные области глубокого обучения с помощью библиотеки DL4J:

- построение глубоких сетей;
- методы настройки сетей;
- векторизация для различных типов данных;
- реализация процессов глубокого обучения на платформе Spark.

DL4J – сокращение для Deeplearning4j
В этой книге названия DL4J и Deeplearning4j считаются синонимами. Оба они относятся к набору инструментов в библиотеке Deeplearning4j.

Мы организовали материал именно таким образом, потому что ощущали потребность в книге, которая содержала бы «достаточно теории» и вместе с тем была бы достаточно практичной для построения процессов глубокого обучения производственного уровня. Мы полагаем, что выбранный гибридный подход хорошо отвечает поставленной цели.

Глава 1 содержит обзор концепций машинного обучения вообще и глубокого обучения в частности с целью ввести читателя в курс дела и сообщить достаточно сведений для понимания остальной части книги. Мы включили эту главу, чтобы начинающие могли освежить подзабытое или познакомиться с новыми для себя идеями, и тем самым хотели сделать книгу доступной максимально широкой аудитории.

В главе 2, основанной на материале главы 1, излагаются основы нейронных сетей. По сути своей она посвящена теории нейронных сетей, но мы старались представить информацию в доступной форме. В главе 3 описывается, как из нейронных сетей вырастают глубокие сети. В главе 4 представлены четыре основные архитектуры глубоких сетей, это фундамент для понимания последующих глав.

Глава 5 содержит ряд примеров Java-кода, в которых используются методы из первой части книги. В главах 6 и 7 рассматриваются фундаментальные основы настройки нейронных сетей общего вида, а затем эти знания применяются к настройке специальных архитектур глубоких сетей. Эти главы платформенно-независимы и в равной мере относятся к любой библиотеке глубокого обучения. В главе 8 приводятся обзор методов векторизации и основы работы с библиотекой DataVec (средством ETL и векторизации, входящим в состав DL4J). Глава 9 завершает основной текст книги обзором применения DL4J в системах Spark и Hadoop – здесь вы найдете реальные примеры, которые сможете выполнить в собственном кластере Spark.

В книге много приложений, в которых рассматриваются важные темы, не нашедшие отражения в основных главах, а именно:

- искусственный интеллект;
- использование Maven в проектах на основе DL4J;
- работа с GPU;
- использование ND4J API
- и прочее.

Кто такой «практик»?

В настоящее время термин «наука о данных» (data science) не имеет четкого определения и зачастую означает разные вещи. Мир науки о данных и искусственного интеллекта (ИИ) широк и расплывчат, как и многие термины в современной информатике. Связано это с тем, что машинное обучение проникло почти во все дисциплины.

У этого широкого проникновения есть исторические параллели с тем, как в 1990-х годах Всемирная паутина проникла во все дисциплины и привела много новых людей в область технологии. Так и теперь самые разные специалисты – инженеры, статистики, аналитики, люди творческих профессий – каждодневно пополняют ряды поборников машинного обучения. Эта книга призвана сделать глубокое (и машинное) обучение доступным самой широкой аудитории.

Если эта тематика вам интересна и вы читаете это предисловие – значит, *вы практик, и эта книга для вас*.

Кому стоит прочитать эту книгу?

Мы решили начать книгу не с игрушечных примеров, которые затем постепенно обрастают деталями, а с фундаментальных основ, которые позволят в полной мере погрузиться в глубокое обучение.

Нам кажется, что во многих книгах опущены базовые вопросы, с которыми каждый практик должен хотя бы бегло познакомиться. Опираясь на свой опыт работы в области машинного обучения, мы решили включить материал, который необходим начинающему практику для успешной работы над своими проектами в сфере глубокого обучения.

Возможно, вы захотите пропустить первые две главы и сразу перейти к основам глубокого обучения. Но мы полагаем, что многие оценят наличие вводных сведений, поскольку это делает гладким переход к основанным на них более трудным темам. Ниже мы предлагаем различные подходы к чтению книги в зависимости от подготовки читателя.

Практический специалист по машинному обучению в коммерческой компании

В этой категории мы выделяем две подгруппы:

- специалист по анализу данных (data scientist);
- Java-программист.

Специалист по анализу данных

Эти люди уже строили модели раньше и свободно ориентируются в науке о данных. Если вы из их числа, то главу 1 можете пропустить, а главу 2 бегло просмот-

реть. Мы рекомендуем сразу перейти к главе 3, поскольку вы, скорее всего, достаточно подготовлены к знакомству с основаниями глубоких сетей.

Java-программист

Java-программистам обычно поручают интегрировать код машинного обучения в производственные системы. Если это как раз к вам и относится, то вас, наверное, заинтересует глава 1, поскольку она позволит познакомиться с терминологией, применяемой в науке о данных. Вам также будет интересно приложение E, поскольку код интеграции модельных оценок обычно не обходится без ND4J API.

Руководитель предприятия

Среди рецензентов нашей книги были и руководители компаний из списка Fortune 500, им материал понравился, т. к. позволил лучше понять, что вообще происходит в области глубокого обучения. Один из них отметил, что после окончания колледжа уже «прошло какое-то время», так что глава 1 пришлась весьма кстати для освежения памяти. Если вы руководитель, то рекомендуем начать с беглого прочтения главы 1, чтобы вспомнить кое-какую терминологию. Но, вероятно, вам стоит пропустить главы, посвященные API и конкретным примерам.

Ученый

Если вы работаете в академическом учреждении, то главы 1 и 2 вряд ли будут вам интересны, т. к. в высшей школе вы это уже проходили. Зато вас наверняка заинтересуют главы о настройке нейронных сетей вообще и для конкретной архитектуры, поскольку эта информация основана на результатах исследований и применима ко всем реализациям глубокого обучения. Тем, кто предпочитает высокопроизводительную линейную алгебру на виртуальной машине Java (JVM), будет также интересно обсуждение библиотеки ND4J.

Графические выделения

В книге применяются следующие графические выделения.

Курсив

Новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

`Моноширинный`

Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка. Также применяется для набора имен модулей и пакетов, команд или иного текста, которые следует вводить буквально, и результатов работы команд.

`Моноширинный курсив`

Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.

Так обозначается примечание общего характера.

Так обозначается предупреждение или предостережение.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Программирование на языке Rust» Джима Блэнди, Джейсона Орендорф (O'Reilly, ДМК Пресс). Copyright © 2018 Jim Blandy and Jason Orendorff, 978-1-491-92728-1 (англ.), 978-5-97060-236-2 (рус.).

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу dmkpress@gmail.com.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Благодарности

От Джоша

В отборе материала для этой книги и ее рецензировании я опирался на помощь людей куда умнее меня. Проект такого размера, как DL4J, не может существовать в вакууме, поэтому в формулировании многих идей и рекомендаций я полагался на мнение экспертов из сообщества и инженеров компании Skymind.

Я никогда не думал, что работа над тем, что впоследствии превратилось в DL4J, вместе с Адамом (после случайной встречи на конференции MLConf) когда-ни-

будь воплотится в книгу. Честно говоря, хотя я стоял у истоков DL4J, Адам сделал куда больше записей в репозиторий, чем я. Поэтому я чрезвычайно признателен Адаму за его преданность проекту, самой идее глубокого обучения на JVM и за то, что он не отступился на начальном этапе, когда все было зыбко и неясно. И да, ты был прав: ND4J оказалась отличной идеей.

Написание книги может стать долгим путешествием в одиночестве, поэтому я особенно благодарен Алексу Блэку, который не только потратил массу времени на рецензирование, но и предложил материал для приложений. Энциклопедические познания Алекса касательно опубликованной литературы по нейронным сетям были чрезвычайно важны при шлифовке многих мелких деталей и стали гарантией правильности изложения в целом и в частности. Без Алекса главы 6 и 7 и наполовину не были бы тем, чем являются теперь.

Сьюзан Эрали (Susan Eraly) оказала ценнейшую помощь в написании раздела о функциях потерь, предоставила материал для приложений (многие уравнения, приведенные в этой книге, были выправлены Сьюзан) и сделала много замечаний в процессе рецензирования. Мелани Уоррик (Melanie Warrick) сыграла основную роль в рецензировании ранних вариантов книги, делилась своими замечаниями и рассказала немало интересного о внутренних механизмах работы сверточных нейронных сетей (CNN).

Дэвид Кэйл (David Kale) частенько оставлял замечания, не пропуская небрежностей в отношении деталей работы сетей и библиографических ссылок. Дэйв всегда представлял академический взгляд на необходимый уровень строгости с учетом того, для какой аудитории мы пишем.

Джеймс Лонг (James Long) критически внимал моим тирадам на тему того, что включать в книгу, а что опустить, и представлял точку зрения статистика-практика. Нередко, когда было не ясно, как лучше изложить какой-то сложный вопрос, Джеймс выступал в роли слушателя, на котором я опробовал обсуждение с разных сторон. Если Дэвид Кэйл и Алекс Блэк зачастую напоминали мне о необходимости математической строгости, то Джеймс играл роль рационального адвоката дьявола, не допускающего, чтобы читатель «утонул в математических деталях».

Вячеслав Кокорин (Raver) привнес свои знания в разработку примеров из области обработки естественного языка и технологии Word2Vec.

Хотел бы отметить поддержку со стороны Криса Николсона (Chris Nicholson), генерального директора компании Skymind. Крис оказывал поддержку этой книге на каждом этапе, и в значительной степени благодаря ему у нас были время и ресурсы для завершения работы над ней.

Я выражаю благодарность всем, кто предлагал материалы для приложений: Алексу Блэку (обратное распространение, DataVec), Вячеславу Кокорину (GPU), Сьюзан Эрали (GPU) и Рубену Фижелю (Ruben Fiszel) (обучение с подкреплением). На различных этапах в рецензировании книги принимали также участие Гран Ингерсол (Grant Ingersol), Дин Уэмплер (Dean Wampler), Роберт Чонг (Robert Chong), Тэд Маласка (Ted Malaska), Райан Джено (Ryan Geno), Ларс Джордж (Lars George), Сунеель Мартхи (Suneel Marthi), Франсу Гарилло (Francois Garillot) и Дон Браун (Don Brown). Все ошибки, которые могли остаться в книге, – целиком и полностью моя вина.

Я благодарен нашему уважаемому редактору Тиму Макговерну (Tim McGovern) за отзывы, замечания и просто терпение – ведь проект растянулся на годы и стал

больше на три главы. Мы высоко ценим, что он позволил нам сорвать сроки, но сделать все как надо.

Ниже перечислены другие люди, которые оказали влияние на мою карьеру, приведшую к появлению этой книги: мои родители (Льюис и Конни), д-р Энди Новобилиски (Andy Novobiliski) (аспирантура), д-р Мина Сартипи (Dr. Mina Sartipi) (научный руководитель), д-р Билли Харрис (Billy Harris) (курс алгоритмов для аспирантов), д-р Джо Дьюмас (Joe Dumas) (аспирантура), Ричи Кэрролл (Ritchie Carroll) (создатель openPDC), Пол Трачан (Paul Trachian), Кристоф Бисиглиа (Christophe Bisciglia) и Майк Олсон (Mike Olson) (затащившие меня в компанию Cloudera), Малком Рэйми (Malcom Ramey) (за мою первую работу в качестве программиста), университет Теннесси в Чаттануге и пиццерию Лупи (где я питался во время учебы в аспирантуре).

И последними по порядку, но отнюдь не по значимости я хочу поблагодарить свою жену Лесли и сыновей Этана, Гриффина и Дэйна за терпение, с которым они переносили мою работу по ночам, а иногда и во время отпуска.

От Адама

Я благодарен своей команде в компании Skymind за ту огромную работу, которую они проделали, рецензируя бесчисленные варианты книги. Особое спасибо Крису, который мирился с моей безумной идеей писать книгу, одновременно пытаясь основать стартап.

Библиотека родилась в 2013 году со случайной встречи с Джошем на конференции MLConf и выросла в проект, который ныне используется во всем мире. Благодаря DL4J я поездил по миру и открыл для себя целый мир новых впечатлений.

Прежде всего я хочу поблагодарить своего соавтора Джоша Паттерсона, который взял на себя львиную долю работы над книгой и заслуживает всяческой признательности. Он тратил вечера и выходные, торопясь выпустить книгу в свет, пока я продолжал работать над кодом и включал в него все новые возможности.

Вслед за Джошем отдаю должное многим нашим коллегам и соавторам – тем, кто был с нами с самого начала, как Алекс, Вячеслав Кокорин (Raver), и тем, кто, как Дэйв, присоединился позже и зорко следил за правильностью математики.

Тим Макговерн оказался прекрасным слушателем, на котором я опробовал некоторые свои завиральные идеи относительно книг для издательства O’Reilly, а также любезно позволил мне самому выбрать название книги.

Глава 1

Обзор машинного обучения

Сконденсировать факт из туманности нюансов.
– *Нил Стивенсон «Лавина»*

Обучающиеся машины

Интерес к машинному обучению резко возрос в последние десять лет. Машинное обучение включают в программы факультетов информатики, по нему проводят конференции, а в заголовках «Wall Street Journal» оно встречается чуть ли не ежедневно. Говоря о машинном обучении, многие путают два вопроса: что оно действительно может и чего от него хотели бы. По большому счету, *машинное обучение* – это применение алгоритмов, которые извлекают информацию из исходных данных и представляют ее в виде той или иной *модели*. Эта модель используется, чтобы вывести другие данные, которые в модели отсутствуют.

Нейронные сети – один из типов моделей машинного обучения, они существуют уже по меньшей мере 50 лет. Фундаментальной единицей нейронной сети является *блок*, или *узел*, – приблизительный аналог биологического нейрона в мозге млекопитающих. Модель связей между нейронами также основана на работе биологического мозга, равно как и эволюция связей с течением времени (в результате «обучения»). В следующих двух главах мы будем подробно рассматривать функционирование таких моделей.

В середине 1980-х и в начале 1990-х годов произошли важные подвижки в области архитектуры нейронных сетей. Но для получения хороших результатов требовалось очень много данных и времени, что замедляло их практическое внедрение, так что постепенно интерес угас. В начале 2000-х годов наблюдался экспоненциальный рост вычислительной мощности, и промышленность стала свидетелем «кембрийского взрыва» методов вычислений – ничего подобного раньше не было и в помине. Глубокое обучение, появившееся в результате взрывного роста вычислительной мощности в этом десятилетии, стало серьезным игроком и победителем многих важных состязаний по машинному обучению. В 2017 году интерес не спадает, сегодня глубокое обучение можно встретить в любом закоулке машинного обучения.

Ниже мы еще вернемся к нашему определению глубокого обучения. Эта книга построена так, чтобы практик, например вы, мог снять ее с полки и сделать следующее:

- получить общие сведения об основах линейной алгебры и машинного обучения;

- получить общие сведения об основах нейронных сетей;
- изучить четыре основные архитектуры глубоких сетей;
- воспользоваться примерами кода для исследования вариантов глубоких сетей на практике.

Мы надеемся, что материал покажется вам практичным и не слишком сложным. И давайте начнем с вопроса о том, что же такое машинное обучение, а также с базовых понятий, которые помогут лучше понять, о чем говорится в книге.

Как машины могут обучаться?

Чтобы определить, как машины могут обучаться, нужно определить, что мы понимаем под «обучением». В быту мы имеем в виду «приобретение знаний посредством самостоятельного изучения, на опыте или в процессе общения с преподавателем». Немного сместив фокус, можно считать, что машинное обучение – это применение алгоритмов для получения структурных описаний из примеров данных. Компьютер обучается чему-то, относящемуся к информационным структурам, содержащимся в исходных данных. Структурные описания – другое название моделей, которые служат для представления информации, извлеченной из данных. Эти структуры, или модели, можно использовать для предсказания неизвестных данных. Структурные описания (или модели) могут принимать разную форму, в том числе:

- решающие деревья;
- линейная регрессия;
- веса связей в нейронных сетях.

Модель каждого типа определяет свой способ применения правил к известным данным для предсказания неизвестных. В случае решающих деревьев создается древовидный набор правил, а в случае линейных моделей – набор параметров, представляющих входные данные.

В нейронных сетях имеется так называемый *вектор параметров*, представляющий веса связей между узлами сети. Ниже в этой главе мы опишем этот тип моделей подробнее.

Машинное обучение и добыча данных

Термин *добыча данных*[1] существует уже много десятилетий и, как многие термины машинного обучения, понимается или используется неправильно. В этой книге под «добычей данных» мы будем понимать «извлечение информации из данных». Отличие машинного обучения состоит в том, что оно относится к алгоритмам, которые применяются в процессе добычи данных для получения структурных описаний из исходных данных. Вот простая интерпретация добычи данных:

- для обучения концепциям нам нужны примеры исходных данных;
- примеры состоят из строк или экземпляров данных, из которых видны определенные закономерности в данных;

[1] В отечественной литературе устоялся термин «добыча данных», хотя он совершенно не отражает смысла понятия. Английское слово «mining» означает «добыча полезных ископаемых» с ударением на слове «полезных», т. е. «извлечение руды из породы». Правильнее было бы говорить о «добыче информации» или прибегнуть к описательному переводу типа «глубокий анализ данных», но мы будем придерживаться сложившейся терминологии. – *Прим. перев.*

- машина обучается концепциям на этих закономерностях посредством алгоритмов машинного обучения.

В целом этот процесс можно считать «добычей данных».

Артур Сэмюэл (Arthur Samuel), пионер в области искусственного интеллекта (ИИ), работавший в компании IBM и в Стэнфордском университете, определял машинное обучение следующим образом:

> Область исследований, посвященная наделению компьютеров способностью обучаться без явного программирования.

Сэмюэл разработал программу, которая могла играть в шашки и адаптировала свою стратегию по мере того, как обучалась ассоциировать вероятность выигрыша или проигрыша с определенными позициями на доске. Эта фундаментальная схема поиска паттернов, ведущих к победе или к поражению, с последующим распознаванием и подкреплением успешных паттернов и по сей день лежит в основе машинного обучения и ИИ.

Концепция машин, способных самостоятельно обучаться для достижения целей, поражала наше воображение в течение многих десятилетий. Наверное, наилучшим образом она была выражена дедушками современного ИИ, Стюартом Расселом (Stuart Russell[2]) и Питером Норвигом (Peter Norvig[3]), в книге «Artificial Intelligence: A Modern Approach»[4]:

> Как может медленный, крохотный мозг, не важно, биологический или электронный, воспринимать, понимать, предсказывать и манипулировать миром, который гораздо больше и сложнее его самого?

Эта цитата отсылает нас к идее о том, что концепции обучения были заимствованы у процессов и алгоритмов, подсмотренных в природе. На рис. 1.1 наглядно изображено наше представление о связи между ИИ, машинным обучением и глубоким обучением.

Рис. 1.1 ❖ Связь между ИИ и глубоким обучением

---

2 https://www2.eecs.berkeley.edu/Faculty/Homepages/russell.html.

3 http://norvig.com/.

4 *Стюарт Р., Норвиг П.* Искусственный интеллект. Современный подход. М.: Вильямс, 2017.

Область ИИ широка и давно уже является предметом исследований. Глубокое обучение – это часть машинного обучения, которая, в свою очередь, является частью ИИ. Теперь кратко рассмотрим другой источник глубокого обучения: связь искусственных нейронных сетей с биологией.

Биологические корни

Биологическая нейронная сеть (мозг) содержит приблизительно 86 миллиардов нейронов, каждый из которых соединен с большим числом других.

Общее число связей в мозге человека

По осторожной оценке исследователей, между нейронами в мозге человека существуют более 500 триллионов связей. Самые крупные современные искусственные нейронные сети даже отдаленно не приближаются к этой величине.

С точки зрения обработки информации, биологический нейрон представляет собой возбуждаемый блок, который может обрабатывать и передавать информацию с помощью электрических и химических сигналов. Нейрон считается основным компонентом головного мозга, спинного мозга и ганглий периферийной нервной системы. Как мы увидим ниже, искусственные нейронные сети структурно гораздо проще.

Сравнение биологической и искусственной нейронных сетей

Биологические нейронные сети гораздо (на несколько порядков) сложнее искусственных!

У искусственных нейронных сетей есть два свойства, отражающих общие представления о работе мозга. Во-первых, самой мелкой единицей нейронной сети является *искусственный нейрон* (или, для краткости, *блок*). Образцом для искусственного нейрона служит биологический нейрон мозга: как и биологический нейрон, он возбуждается в результате поступления входных сигналов. Искусственные нейроны передают часть полученной информации (но не всю) другим искусственным нейронам, часто в преобразованном виде. Далее в этой главе мы детально рассмотрим эти преобразования в контексте нейронных сетей.

Во-вторых, как биологические нейроны можно научить передавать дальше только сигналы, полезные для достижения более общих целей мозга, так и искусственные нейроны можно обучить передавать только полезные сигналы. Далее мы увидим, как эти идеи позволяют искусственной нейронной сети моделировать биологическую посредством битов и функций.

Биологические идеи в информатике

Применение биологических идей в информатике не ограничивается искусственными нейронными сетями. За последние 50 лет предметом исследований, нашедших отражение в информатике, стали и другие природные объекты, например:

- муравьи:
- термиты[5];
- пчелы;
- генетические алгоритмы.

[5] Patterson, 2008. TinyTermite: A Secure Routing Algorithm; Sartipi and Patterson, 2009. TinyTermite: A Secure Routing Algorithm on Intel Mote 2 Sensor Network Platform.

Например, муравейник рассматривался учеными как мощный децентрализованный компьютер, в котором ни один отдельный муравей не является точкой общего отказа[6]. Муравьи постоянно переключаются с одной задачи на другую в поисках близких к оптимальному решений задачи балансировки нагрузки с помощью таких метаэвристик, как количественная стигмергия. Колонии муравьев способны убирать территорию, обороняться, строить гнездо и добывать пищу, выделяя для каждой задачи почти оптимальное число работников, основанное на относительной потребности, при этом не существует какого-то одного муравья, координирующего работу.

Что такое глубокое обучение?

Дать определение глубокому обучению не так-то просто, потому что его формы медленно изменялись на протяжении последнего десятилетия. Одно из полезных определений гласит, что глубокое обучение имеет дело с «нейронными сетями, имеющими более двух слоев». Проблема в том, что такие сети считались глубокими где-то в 1980-х годах. Но нам кажется, что для демонстрации впечатляющих результатов, наблюдаемых в последние годы, нейронные сети архитектурно должны были перерасти ранние формы (что стало возможно благодаря значительному увеличению вычислительной мощности). Перечислим некоторые аспекты эволюции нейронных сетей:

- больше нейронов, чем в предшествующих сетях;
- более сложные способы соединения нейронов и слоев;
- резкий рост вычислительной мощности, доступной для обучения;
- автоматическое выделение признаков.

В этой книге мы будем определять глубокую сеть как нейронную сеть с большим числом параметров и слоев, имеющую одну из четырех фундаментальных архитектур:

- сети, предобученные без учителя;
- сверточные нейронные сети;
- рекуррентные нейронные сети;
- рекурсивные нейронные сети.

Существует ряд вариаций этих архитектур, например гибридная сверточно-рекуррентная нейронная сеть. Но в этой книге мы сосредоточимся только на четырех вышеперечисленных архитектурах.

Автоматическое выделение признаков – еще одно крупное преимущество глубокого обучения над традиционными алгоритмами машинного обучения. Под этим мы понимаем процесс определения сетью тех характеристик набора данных, которые можно надежно использовать для пометки данных. Исторически специалисты по машинному обучению тратили месяцы, годы, а иногда и десятки лет жизни на то, чтобы вручную создать исчерпывающий набор признаков для классификации данных. Со времени Большого взрыва глубокого обучения, начавшегося в 2006 году, необходимость тратить годы ручного труда исчезла, поскольку новейшие алгоритмы машинного обучения способны сами находить признаки, по которым можно классифицировать входные данные. По точности глубокое обучение превзошло традиционные алгоритмы почти для всех типов данных, требуя

[6] https://mitpress.mit.edu/books/ant-colony-optimization.

при этом минимальной настройки и человеческого труда. Глубокие сети помогают коллективам ученых расходовать кровь, пот и слезы на более важные дела.

Вниз по кроличьей норе

Глубокое обучение переплелось со всеми отраслями информатики теснее, чем все прочие методы в недавней истории. С одной стороны, это объясняется высочайшей точностью моделирования, а с другой – порождающими механизмами, очаровывающими даже непрофессионалов. Например, глубокая сеть, обученная на картинах знаменитого художника, оказалась способна генерировать новые картины в его стиле – смотрите рис. 1.2.

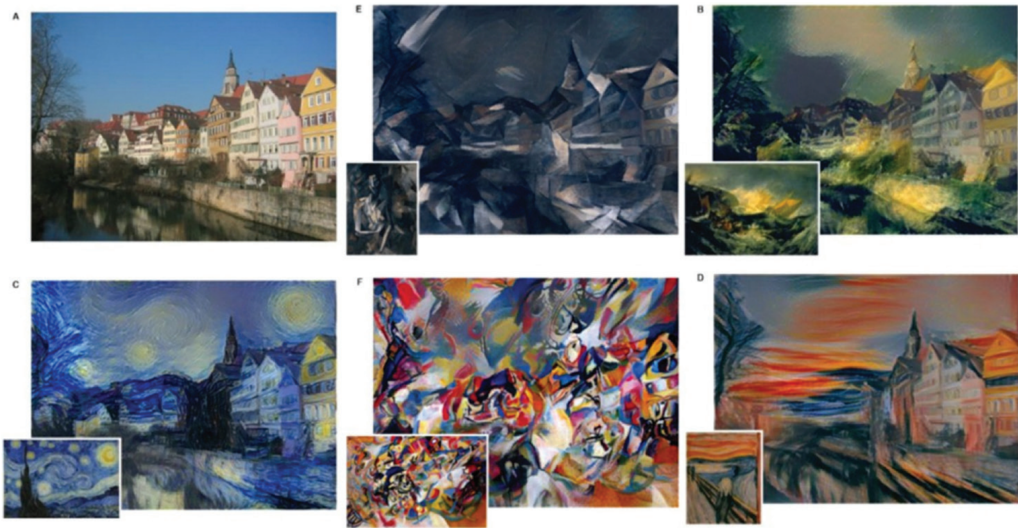


Рис. 1.2 ❖ Стилизованные изображения, взятые из работы Gatys et al., 2015[7]

Это открывает простор для разного рода философских дискуссий, например: «Способна ли машина к творчеству?» и тогда уж «Что такое творчество?». Оставляем эти вопросы вам – поразмыслите на досуге. Машинное обучение развивалось медленно, как смена времен года: потихоньку-полегоньку – и вдруг в один прекрасный день машина становится чемпионом в игре *Jeopardy* или обыгрывает гроссмейстера в го.

Могут ли машины быть разумными или обрести интеллект человеческого уровня? Что такое ИИ и насколько мощным он мог бы стать? На эти вопросы еще предстоит ответить, но не в этой книге. Мы просто хотим проиллюстрировать некоторые фрагменты машинного интеллекта, которые уже сегодня пропитывают наше окружение благодаря практическому применению глубокого обучения.

Более полное обсуждение ИИ

Если вы хотите еще почитать об ИИ, обратитесь к приложению А.

[7] Gatys et. al, 2015. A Neural Algorithm of Artistic Style // https://arxiv.org/pdf/1508.06576v1.pdf.

Формулировка вопросов

Понять, как применяется машинное обучение, проще всего, задавая правильные вопросы. Вот что нам нужно определить:

- что представляют собой исходные данные, из которых мы хотим извлечь информацию (модель)?
- модель какого вида лучше всего отвечает этим данным?
- какого рода ответ мы хотели бы получить от новых данных, основываясь на этой модели?

Если мы сумеем ответить на эти три вопроса, то сможем организовать процесс машинного обучения, в ходе которого будет построена модель и получены желаемые ответы. А для этого вспомним базовые понятия, необходимые для практического применения машинного обучения. Позже мы увидим, как эти понятия соединяются вместе в машинном обучении, и воспользуемся этой информацией, чтобы подвести фундамент под наше понимание нейронных сетей и глубокого обучения.

Математические основания машинного обучения: линейная алгебра

Линейная алгебра – краеугольный камень машинного и глубокого обучения. Она дает математический аппарат для решения уравнений, используемых при построении моделей.

Прекрасным учебником по линейной алгебре является книга James E. Gentle «Matrix Algebra: Theory, Computations, and Applications in Statistics» (http://mason.gmu.edu/~jgentle/books/matbk.htm).

Прежде чем двигаться дальше, рассмотрим некоторые базовые понятия из этой области и начнем со *скаляра*.

Скаляры

В математике термин «скаляр» встречается в основном в контексте элементов вектора. Скаляр – это вещественное число или, в более общем случае, элемент поля, над которым определено векторное пространство.

В информатике скаляр – синоним «переменной», т. е. область памяти вместе с сопоставленным ей символическим именем. В этой области хранится неизвестная величина, называемая *значением*.

Векторы

Нас устроит следующее определение вектора:

> Если n – целое положительное число, то вектором называется n-кортеж, т. е. упорядоченное мультимножество или массив n чисел, называемых элементами, или скалярами.

Сказанное означает, что мы хотим создавать структуру данных, именуемую вектором, посредством процесса *векторизации*. Количество элементов вектора

называется его «порядком», или «длиной». Векторы могут также представлять точки в n-мерном пространстве. При такой интерпретации длина вектора равна евклидову расстоянию между представленной им точкой и началом координат.

В математических текстах векторы часто записываются в виде:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix}$$

или

$$x = [x_1, x_2, x_3, \dots, x_n].$$

Существует много способов векторизации. Кроме того, можно применять различные шаги предварительной обработки, получая на выходе модели разной эффективности. Вопрос о преобразовании исходных данных в векторы мы рассмотрим ниже в этой главе, а затем более полно в главе 5.

Матрицы

Будем называть матрицей группу векторов одинаковой размерности (с одинаковым числом столбцов). Таким образом, матрица – это двумерный массив, состоящий из строк и столбцов.

Матрицей размера $n \times m$ называют матрицу, имеющую n строк и m столбцов. На рис. 1.3 изображена матрица 3×3. Матрицы – основная структура в линейной алгебре и машинном обучении, в чем мы убедимся далее в этой главе.

Рис. 1.3 ❖ Матрица 3 × 3

Тензоры

Тензором называется многомерный массив. Вектор можно считать частным случаем тензора.

В тензорах строки расположены вдоль оси y, а столбцы – вдоль оси x. Каждая ось является измерением, и у тензоров имеются дополнительные измерения, помимо

x и y. Тензор также характеризуется рангом. Так, скаляр имеет ранг 0, вектор – ранг 1, а матрица – ранг 2. Структуры ранга 3 и выше называются тензорами.

Гиперплоскости

Еще один полезный объект из линейной алгебры – *гиперплоскость*. В геометрии гиперплоскостью называется подпространство, размерность которого на единицу меньше размерности объемлющего пространства. В трехмерном случае размерность гиперплоскости равна 2, а в двумерном гиперплоскостью считается одномерная прямая.

Гиперплоскость делит *n*-мерное пространство на две части и потому имеет полезные применения в приложениях классификации. Оптимизация параметров гиперплоскости – ключевая идея линейного моделирования, как мы увидим в этой главе ниже.

Математические операции

В этом разделе мы кратко рассмотрим наиболее распространенные в линейной алгебре операции.

Скалярное произведение

В машинном обучении часто встречается операция *скалярного произведения* (иногда ее еще называют «внутренним произведением»). Скалярное произведение применяется к двум векторам одинаковой длины и возвращает одно число. Для этого соответственные элементы векторов перемножаются, а произведения суммируются. Не вдаваясь пока в математические детали, отметим, что в этом числе закодировано много информации.

Прежде всего скалярное произведение показывает, насколько велики отдельные элементы обоих векторов. Произведение двух векторов с большими элементами будет большим, а с малыми – малым. Если применить к значениям элементов математическую процедуру *нормировки*, т. е. принимать во внимание не абсолютные, а относительные значения, то скалярное произведение векторов будет мерой их похожести. Скалярное произведение нормированных векторов называется *косинусным коэффициентом*, или *коэффициентом Отиаи*.

Поэлементное произведение

На практике также часто встречается операция *поэлементного произведения*, или *произведения Адамара*. Она применяется к двум векторам одинаковой длины и возвращает вектор той же длины, каждый элемент которого равен произведению соответственных элементов векторов-сомножителей.

Внешнее произведение

Оно называется еще *тензорным произведением* двух векторов. Каждый элемент вектора-столбца умножается на все элементы вектора-строки, в результате чего создается новая строка в результирующей матрице.

Преобразование данных в векторы

В машинном обучении и науке о данных требуется анализировать данные самых разных типов. Ключевое требование – возможность представить тип дан-

ных в виде вектора. В машинном обучении рассматривается много типов данных (текст, временные ряды, звук, изображения, видео и т. п.).

Так почему бы не подать на вход алгоритма обучения исходные данные – и пусть обрабатывает? Проблема в том, что машинное обучение основано на линейной алгебре и решении систем уравнений. Решатель уравнений ожидает получить на входе числа с плавающей точкой, поэтому нужно каким-то образом преобразовать исходные данные во множество чисел с плавающей точкой. Мы соединим эти две концепции в следующем разделе, посвященном решению систем уравнений. Примером исходных данных является канонический набор данных Iris об ирисах (http://archive.ics.uci.edu/ml/datasets/Iris):

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
```

Еще один пример – текстовый документ:

```
Go, Dogs. Go!
Go on skates
or go by bike.
```

В этих двух случаях мы имеем данные разного типа, но для каждого необходима та или иная векторизация, чтобы их можно было использовать в машинном обучении. На каком-то этапе входные данные должны быть представлены в виде матрицы, но можно сначала преобразовать их в некий промежуточный формат (например, в формат «svmlight», показанный в примере кода ниже). Мы хотим, чтобы входные данные для алгоритма машинного обучения выглядели как сериализованный разреженный вектор в следующем формате svmlight:

```
1.0 1:0.7500000000000001 2:0.41666666666666663 3:0.702127659574468 4:0.5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.45833333333333326 2:0.3333333333333336 3:0.8085106382978723 4:0.7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.3333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872 4:0.5652173913043479
1.0 1:0.916666666666667 2:0.6666666666666667 3:0.7659574468085107 4:0.5652173913043479
0.0 1:0.08333333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149 4:0.5217391304347826
0.0 2:0.7500000000000002
```

Из данных в таком формате легко получить матрицу и вектор-столбец меток (первое число в каждой строке). После метки идут пары «номер столбца:значение», которые загружаются в соответствующие позиции матрицы и интерпретируются

как «признаки». Полученная матрица готова к различным операциям линейной алгебры в алгоритме машинного обучения. Более подробно процесс векторизации обсуждается в главе 8.

Зададимся стандартным вопросом: «Почему алгоритмам машинного обучения (обычно) требуются данные, представленные в виде (разреженной) матрицы?» Для ответа на него сделаем небольшой экскурс в область решения систем уравнений.

Решение систем уравнений

В линейной алгебре нас интересует решение систем линейных уравнений вида:

$Ax = b$,

где A – матрица, образованная входными векторами-строками, а b – вектор-столбец меток, соответствующих каждой строке матрицы A. Если взять первые три строки сериализованного разреженного представления из предыдущего примера и представить их в линейно-алгебраической форме, то получим:

```
Столбец 1           Столбец 2           Столбец 3           Столбец 4
0.7500000000000001  0.41666666666666663 0.702127659574468   0.5652173913043479
0.6666666666666666  0.5                 0.9148936170212765  0.6956521739130436
0.45833333333333326 0.3333333333333336  0.8085106382978723  0.7391304347826088
```

Эта числовая матрица и есть переменная A в нашем уравнении, а независимые значения в каждой строке считаются признаками входных данных.

Что такое признак?

В машинном обучении признаком называется любой столбец матрицы A, рассматриваемый как независимая переменная. В качестве признаков можно брать непосредственно исходные данные, но обычно производится некоторое преобразование исходных данных в форму, более подходящую для моделирования.

Примером может служить столбец входных данных, который изначально содержал четыре разные текстовые метки. Мы должны просмотреть все исходные данные и присвоить каждой метке числовой индекс. Затем четыре значения (0, 1, 2, 3) следует нормировать, приведя к интервалу от 0.0 до 1.0, учитывая, сколько раз каждый индекс встречается в строках. Такого рода преобразования очень способствуют нахождению хороших решений задач моделирования. В главе 5 мы подробнее поговорим о методах векторизации и преобразованиях.

Мы хотим найти для каждого столбца такие коэффициенты, которые после умножения и суммирования дадут вектор-столбец меток b. В сериализованном разреженном представлении были такие метки:

```
Метки
1.0
2.0
2.0
```

Вышеупомянутые коэффициенты образуют вектор-столбец x (называемый также *вектором параметров*), показанный на рис. 1.4.

	Обучающие записи (A)					Вектор параметров (x)		Выход (b)
Входная запись 1	0.7500	0.4166	0.7021	0.5652		?		1.0
Входная запись 2	0.6666	0.5	0.9148	0.6956	•	?	=	2.0
Входная запись 3	0.4583	0.3333	0.8085	0.7391		?		2.0

Рис. 1.4 ❖ Визуализация уравнения Ax = b

Говорят, что эта система уравнений *совместна*, если существует вектор параметров x – такой, что решение можно записать в виде:

$x = A^{-1}b.$

Важно отличать выражение $x = A^{-1}b$ от метода вычисления решения. Это выражение всего лишь представляет решение. Переменная A^{-1} – это матрица, обратная A, которая вычисляется с помощью процедуры *обращения матрицы*. Учитывая, что не все матрицы обратимы, нам нужен такой метод решения уравнения, который не прибегает к обращению матрицы. Один из таких методов называется *разложением* (или *декомпозицией*) *матрицы*. Примером может служить LU-разложение матрицы A. Рассмотрим также общие методы решения систем линейных уравнений, помимо разложения матриц.

Методы решения систем линейных уравнений

Существуют два подхода к решению систем линейных уравнений. Первый – прямые методы – заключается в применении фиксированных вычислений, объем которых известен заранее. Другой подход – так называемые *итеративные методы* – заключается в нахождении последовательных приближений к вектору параметров x, при этом процесс прекращается, как только будут выполнены заранее заданные условия завершения. Прямые методы особенно эффективны, если все обучающие данные (A и b) помещаются в памяти одного компьютера. Хорошо известными примерами прямых методов являются *метод исключения Гаусса* и *метод нормальных уравнений*.

Итеративные методы

Итеративные методы особенно эффективны, когда данные не помещаются в оперативную память одного компьютера, а загрузка записей с диска в цикле позволяет включить в модель гораздо больший объем данных. Канонический пример итеративного метода, повсеместно встречающийся в современном машинном обучении, – стохастический градиентный спуск (СГС), мы обсудим его ниже в этой главе. Упомянем также *метод сопряженных градиентов* и *метод чередующихся наименьших квадратов* (обсуждаются в главе 3). Итеративные методы также доказали свою эффективность в горизонтально масштабируемых конфигурациях, когда весь набор данных распределен между машинами из одного кластера и вектор параметров периодически усредняется по всем агентам, а затем обновляется на каждом локальном агенте моделирования (подробное объяснение отложим до главы 9).

Итеративные методы и линейная алгебра

Мы хотели бы применить эти алгоритмы к операциям с входным набором данных. Это означает, что исходные данные следует преобразовать во входную матрицу A.

Краткий обзор линейной алгебры отвечает на вопрос, *зачем* нужно подвергать данные векторизации. В этой книге мы на примерах кода продемонстрируем преобразование исходных данных в матрицу A и тем ответим на вопрос – *как*. Механизм векторизации данных также оказывает влияние на результаты процесса обучения. Ниже мы увидим, что этап предварительной обработки данных еще до векторизации может привести к созданию более точных моделей.

Математические основания машинного обучения: статистика

Мы приведем ровно столько сведений из области статистики, сколько необходимо, чтобы двигаться дальше. Необходимо вспомнить некоторые базовые понятия:

- вероятность;
- распределения;
- правдоподобие.

Существуют и другие важные понятия, относящиеся к описательной статистике и индуктивной статистике. К описательной статистике относятся:

- гистограммы;
- коробчатые диаграммы;
- диаграммы рассеяния;
- среднее;
- стандартное отклонение;
- коэффициент корреляции.

С другой стороны, предмет индуктивной статистики – обобщение с выборки на всю генеральную совокупность. Вот несколько примеров индуктивной статистики:

- p-значения;
- интервалы правдоподобности (credibility interval).

Сформулируем соотношение между вероятностью и индуктивной статистикой:

- вероятность подразумевает переход от генеральной совокупности к выборке (дедуктивное рассуждение);
- индуктивная статистика подразумевает переход от выборки к генеральной совокупности.

Чтобы понять, что конкретная выборка говорит нам об исходной генеральной совокупности, нужно сначала оценить недостоверность, связанную с извлечением выборки из заданной генеральной совокупности.

Говоря об общей статистике, мы не будем задерживаться на весьма обширной проблематике, глубоко раскрытой в других книгах. Этот раздел ни в коей мере не является сколько-нибудь полным обзором статистики, мы лишь хотели обозначить темы, которые можно более подробно изучить по другим источникам. Теперь, оградив себя от критики, начнем с определения вероятности в статистике.

Вероятность

По определению, вероятность события E – число от 0 до 1. Вероятность 0 означает, что событие E вообще не может произойти, а вероятность 1 – что оно произойдет наверняка. Часто вероятность выражается в виде числа с плавающей точкой, но

иногда также в виде процентной величины от 0 до 100%: вероятность не может быть меньше 0 и больше 100 %. Например, вероятность 0.35 можно выразить как 35% (0.35 × 100 == 35%).

Канонический пример измерения вероятности – наблюдение за тем, сколько раз выпали орел и решка при подбрасывании правильной монеты (такой, что каждая сторона выпадает с вероятностью 0.5). Вероятность выборочного пространства всегда равна 1, потому что оно представляет все возможные исходы испытания. Как мы видим на примере двух исходов (выпадение орла и решки), 0.5 + 0.5 = 1.0, потому что полная вероятность выборочного пространства должна быть равна 1. Вероятность события записывается в виде:

$P(E) = 0.5.$

Эта формула читается так:

Вероятность события E равна 0.5.

Вероятность и шанс

Начинающие изучать статистику и машинное обучение часто путают смысл слов «вероятность» и «шанс». Давайте разберемся.

Вероятность события E определяется следующим образом:

$P(E)$ = (Число благоприятных для E исходов) / (Общее число исходов).

Так, вероятность вытянуть туза (4 штуки) из колоды карт (52 штуки) равна:

4/52 = 0.077.

С другой стороны, шанс (odds) определяется следующим образом:

(Число благоприятных для E исходов) : (Число неблагоприятных для E исходов).

В примере с картами «шанс вытянуть туза» равен:

4 : (52 – 4) = 1/12 = 0.0833333…

Отличие – в знаменателе: «общее число исходов» и «число неблагоприятных для E исходов».

Понятие вероятности является центральным для нейронных сетей и глубокого обучения в силу своей роли в выделении признаков и классификации – двух основных функций глубоких нейронных сетей. Более полное изложение математической статистики см. в книге Boslaugh and Watters «Statistics in a Nutshell: A Desktop Quick Reference», выпущенной издательством O'Reilly (http://shop.oreilly.com/product/0636920023074.do).

Еще об определении вероятности: частотный и байесовский подходы

В статистике имеются два разных подхода к определению вероятности: *байесовский* и *частотный*.

Сторонники частотного подхода считают, что говорить о вероятности имеет смысл только в контексте повторяющихся измерений. Измеряя некую величину, мы наблюдаем небольшие вариации из-за физических особенностей оборудования, исполь-

зуемого для сбора данных. При многократном повторении измерения частота конкретного значения определяет его вероятность.

При байесовском подходе вероятность связывают с достоверностью утверждений. Вероятность характеризует степень нашей уверенности в том, каким будет результат измерения. Сторонники байесовского подхода считают, что наши знания о событии по сути своей связаны с его вероятностью.

Сторонники частотного подхода полагаются на большое число слепых испытаний и только после этого готовы дать оценку случайной величины. А сторонники байесовского подхода имеют дело с «верой» (математическим «распределением») относительно случайной величины и обновляют эту веру по мере поступления новой информации.

Условные вероятности

Если мы хотим знать, какова вероятность данного события при условии, что произошло некоторое другое событие, то говорим об *условной вероятности*. В литературе условная вероятность записывается в виде:

$P(E|F)$,

где E – событие, вероятность которого нас интересует; F – событие, которое уже произошло.

В качестве примера выразим тот факт, что для человека с нормальной частотой сердечных сокращений вероятность умереть в палате интенсивной терапии ниже:

P(смерть в ПИТ | плохой пульс) > P(смерть в ПИТ | хороший пульс).

Иногда второе событие F называют «условием». Условная вероятность представляет интерес для машинного и глубокого обучений, потому что часто нам важно знать, как взаимодействуют различные факторы. В контексте машинного обучения мы обучаем классификатор путем определения условной вероятности

$P(E|F)$,

где E – метка, а F – ряд атрибутов сущности, для которой мы хотим предсказать E. Примером может служить предсказание вероятности смерти (E), если известны результаты измерений, сделанных в палате интенсивной терапии для каждого пациента (F).

Теорема Байеса

Одно из основных применений условных вероятностей дает теорема (или формула) Байеса. В медицине она применяется для вычисления вероятности того, что пациент, сдавший положительный анализ на некоторое заболевание, действительно болен им. Формула Байеса для двух событий A и B имеет вид:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}.$$

Апостериорная вероятность

В байесовской статистике апостериорной вероятностью случайного события называется условная вероятность после наблюдения факта. Апостериорное распре-

деление вероятности определяется как распределение вероятности неизвестной величины, обусловленное фактическими данными, собранными в результате эксперимента и рассматриваемыми как случайная величина. Как эта идея применяется на практике, мы увидим ниже на примере функции активации softmax, которая преобразует входные значения в апостериорные вероятности.

Распределения вероятности

Распределение вероятности – это описание стохастической структуры случайных величин. В статистике мы высказываем предположения о распределении данных, чтобы делать о них какие-то выводы. Нам нужна формула, описывающая частоту наблюдаемых значений величины с данным распределением. Часто встречается *нормальное распределение* (называемое также *гауссовым*, или *колоколообразной кривой*). Мы хотим аппроксимировать набор данных распределением, поскольку если набор данных действительно близок к распределению, то о данных можно высказывать гипотезы, исходя из теоретических свойств распределения.

Выделяют *непрерывные* и *дискретные* распределения. Дискретная случайная величина может принимать значения из дискретного множества, а непрерывная – значения из некоторого диапазона. Нормальное распределение – пример непрерывного распределения. Примером дискретного может служить биномиальное распределение.

Нормальное распределение (см. рис. 5.1) было открыто Карлом Гауссом, математиком и физиком, жившим в XVIII веке. Оно полностью определяется своим средним и стандартным отклонением и имеет одинаковую характерную форму для любых параметров.

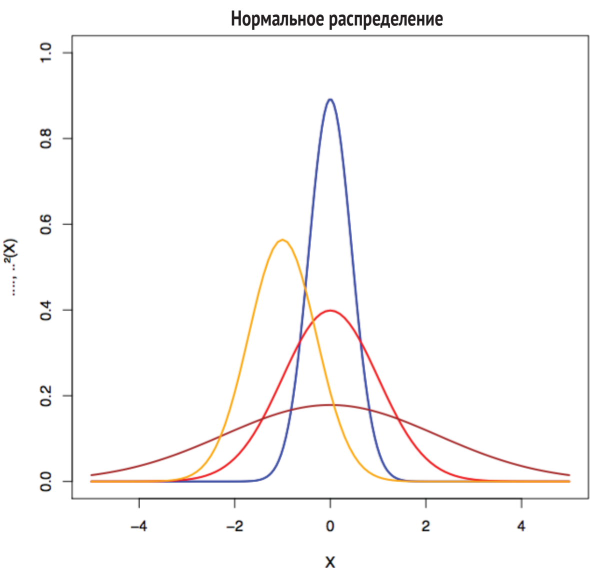


Рис. 1.5 ❖ Примеры нормальных распределений

Из других распределений, встречающих в машинном обучении, отметим:

- биномиальное распределение;

- обратное нормальное распределение;
- логнормальное распределение.

Понимать, как распределены обучающие данные, важно, чтобы правильно векторизовать данные для моделирования.

Центральная предельная теорема

Если размер выборки достаточно велик, то выборочное распределение выборочных средних аппроксимируется нормальным распределением. Это справедливо независимо от распределения генеральной совокупности, из которой производилась выборка.

Это позволяет делать статистические выводы, применяя критерии, основанные на приближенной нормальности среднего. И выводы остаются справедливыми, даже если выборка произведена не из нормально распределенной генеральной совокупности.

В информатике мы столкнемся с этим в алгоритмах, которые многократно производят выборку заданного размера из ненормальной генеральной совокупности. Построив гистограмму выборочной совокупности выборок из нормального распределения, мы увидим этот эффект в действии.

Распределения с длинным хвостом (например, Zipf, степенные законы и распределение Парето) – ситуация, когда за областью значений с высокой частотой следует асимптотически убывающая область значений с низкой частотой. Такие распределения изучались Бенуа Мандельбротом в 1950-х годах, а затем популяризированы в книге Криса Андерсона «The Long Tail: Why the Future of Business is Selling Less of More»[8].

Примером может служить ранжирование товаров в розничном магазине: несколько товаров пользуется очень большой популярностью, но наряду с ними имеется много уникальных товаров, которые продаются в небольших объемах. Такое распределение ранга по частоте (в основном популярности или «количеству проданных») часто подчиняется степенному закону. Поэтому мы можем рассматривать его как распределение с длинным хвостом.

Распределения с длинным хвостом встречаются в следующих случаях:

- *ущерб от землетрясения* – ущерб тем выше, чем сильнее землетрясение. Ущерб в худшем случае сдвигается в область длинного хвоста;
- *урожайность* – иногда наблюдаются события, выходящие за пределы исторических данных. Но в целом модель сосредоточена вблизи средних значений;
- *прогнозирование смертности после пребывания в палате интенсивной терапии* – на смертность могут оказывать влияние события, слабо связанные с тем, что происходило в палате интенсивной терапии.

Эти примеры имеют отношение к рассматриваемым в книге задачам классификации, потому что большинство статистических моделей опирается на выводы, сделанные на основе очень больших объемов данных. Если наиболее интересные

---

[8] *Крис А.* Длинный хвост. Эффективная модель Интернета в бизнесе. М.: Манн, Иванов и Фербер, 2002.

события случаются в хвосте распределения, а он не представлен в обучающей выборке, то модель может работать непредсказуемо. Этот эффект усиливается в нелинейных моделях, к каковым относятся и нейронные сети. Мы рассматриваем эту ситуацию как частный случай проблемы «входит в выборку/не входит в выборку». Даже профессионал в области машинного обучения может поразиться тому, что модель, прекрасно работающая на асимметричной обучающей выборке, не обобщается на большую совокупность данных.

Распределения с длинным хвостом описывают реальную возможность событий, отстоящих от среднего более чем на пять стандартных отклонений. Мы должны следить за тем, чтобы в обучающих данных были представлены и такие события, чтобы предотвратить переобучение. О том, как этого добиться, мы поговорим подробнее ниже, когда перейдем к переобучению, а также в главе 4, посвященной настройке модели.

Выборка и генеральная совокупность

Под генеральной совокупностью понимается все множество данных, которые мы собираемся изучать или моделировать в эксперименте. Примером может служить «множество всех Java-программистов в штате Теннесси».

Выборкой называется подмножество генеральной совокупности, которое предположительно достаточно точно представляет распределение данных, не внося смещения (обусловленного способом извлечения выборки).

Методы с перевыборкой

Бутстрэппинг и *перекрестная проверка* – два типичных статистических метода с перевыборкой, полезных в машинном обучении. В случае бутстрэппинга мы случайным образом выбираем примеры из другой выборки с целью сгенерировать новую выборку, в которой представители каждого класса были бы сбалансированы. Это полезно при моделировании набора данных с сильно разбалансированными классами.

Перекрестная проверка позволяет оценить, насколько хорошо обобщается обученная модель. Весь набор данных разбивается на *N* порций, а затем каждая порция разделяется на обучающий и тестовый наборы. Сначала модель обучается на обучающих наборах, а затем проверяется на тестовых. Порции меняются местами, пока не будут исчерпаны все варианты. На количество порций не налагается никаких ограничений, но на практике обычно берут 10 порций, это дает хорошие результаты. Нередко часть данных резервируют под контрольный набор, используемый во время обучения.

Смещение выборки

Смещение возникает, когда метод выборки не обеспечивает надлежащую рандомизацию, вследствие чего выборка оказывается нерепрезентативной для моделируемой генеральной совокупности. Мы должны помнить о смещении, производя перевыборку из набора данных, чтобы не внести искажений в модель, что приведет к снижению ее верности на данных из большей генеральной совокупности.

Правдоподобие

Обсуждая возможность события, мы не указываем его числовую вероятность, а употребляем менее формальный термин – *правдоподобие*. Обычно мы имеем в виду, что событие может произойти с довольно высокой вероятностью, но все же не наверняка. На событие могут повлиять пока еще не наблюдавшиеся факторы. Но в целом слово «правдоподобие» используется как неформальный синоним «вероятности».

Как работает машинное обучение?

В предыдущем разделе, говоря о решении систем линейных уравнений, мы ввели обозначение $Ax = b$. По существу, машинное обучение сводится к алгоритмам, которые минимизируют погрешность решения этого уравнения посредством *оптимизации*.

Оптимизация означает, что мы изменяем элементы вектора параметров x, пока не найдем такой набор значений, при котором получается наилучшее приближение к значениям b. Весовая матрица корректируется после вычисления функции потерь (основанной на расхождении с вектором-столбцом b), т. е. ошибки, порожденной сетью. Матрица ошибок, приписывающая некоторую часть потери каждому весу, умножается на сами веса.

Ниже в этой главе мы обсудим метод стохастического градиентного спуска – один из основных методов оптимизации в машинном обучении, а далее в книге рассмотрим и другие алгоритмы оптимизации. Мы также поговорим о гиперпараметрах и, в частности, о регуляризации и скорости обучения.

Регрессия

Термин «регрессия» относится к функциям, пытающимся предсказать вещественное значение. Функции такого типа оценивают зависимую величину, зная независимую. Наиболее распространена *линейная регрессия*, в основе которой лежат идеи, описанные выше в контексте моделирования систем линейных уравнений. Задача линейной регрессии заключается в отыскании функции, описывающей связь между x и y, так чтобы для известных значений x предсказанные ей значения y были верны.

Структура модели

Предсказание, порождаемое линейной регрессионной моделью, является линейной комбинацией коэффициентов (вектора параметров x) и входных величин (входного вектора). Это можно выразить в виде следующего уравнения:

$$y = a + Bx,$$

где a – свободный член, B – входные признаки, x – вектор параметров.

В скалярном виде уравнение записывается так:

$$y = a + b_0 * x_0 + b_1 * x_1 + \ldots + b_n * x_n.$$

Простой пример задачи линейной регрессии – предсказание месячных расходов на бензин на основании длины маршрута. В данном случае сумма, уплаченная на заправке, является функцией расстояния, которое вы проехали. Стоимость

бензина – зависимая величина, а длина маршрута – независимая. Мы можем связать эти величины функциональной зависимостью:

стоимость = *f* (*расстояние*).

Это позволяет дать разумный прогноз расходов на бензин в зависимости от длины поездки.

Вот еще несколько примеров линейных регрессионных моделей:

- предсказание веса как функции роста;
- предсказание продажной цены дома как функции его площади.

Визуализация линейной регрессии

Визуально линейную регрессию можно представить как отыскание прямой линии, которая проходит максимально близко к максимально возможному числу экспериментальных точек, как показано на рис. 1.6.

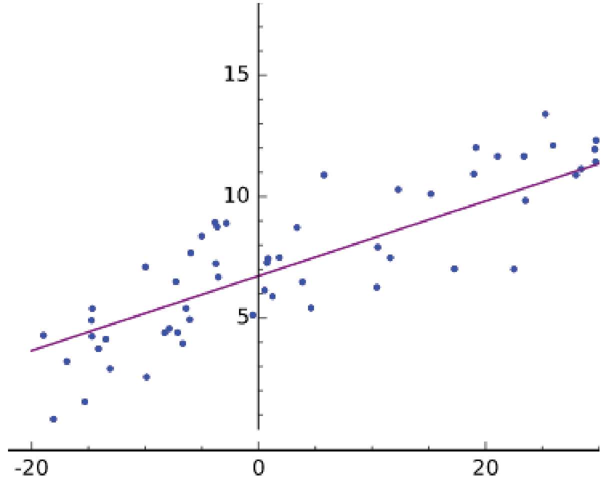


Рис. 1.6 ❖ График линейной регрессии

Подгонкой (fitting), или аппроксимацией, называется нахождение функции $f(x)$, значения которой близки к измеренным значениям y.

Линейная регрессионная модель и система уравнений

Мы можем связать эту функцию с приведенным выше уравнением $Ax = b$, где A – признаки (например, «вес» или «площадь дома») для всех входных примеров модели. Каждая входная запись – строка матрицы A. Вектор-столбец b – известные метки, соответствующие входным записям. Применяя некоторую функцию ошибок и метод оптимизации (например, СГС), мы можем найти такой набор параметров x, который минимизирует отклонение от истинных меток на всем множестве входных данных.

Для применения СГС нам понадобятся три вещи:

1) гипотеза о данных – произведение вектора параметров x и матрицы входных признаков;
2) функция стоимости – квадрат ошибки (предсказание – факт);

3) функция обновления – производная квадратичной функции потерь (функции стоимости).

Если линейная регрессия имеет дело с прямыми линиями, то нелинейная аппроксимация – со всем остальным, чаще всего с полиномами от *x* степени, большей 1. (Потому-то машинное обучение иногда описывают как «подбор аппроксимирующей кривой».) Точная аппроксимация означает, что кривая проходит через все экспериментальные точки. Но точная аппроксимация – обычно очень плохой результат, поскольку это означает, что модель идеально подогнана к обучающему набору и за его пределами не имеет почти никакой предсказательной силы (т. е. не обобщается).

Классификация

Под классификацией понимают моделирование, цель которого – разделить классы данных на основе некоторого множества входных признаков. Если регрессия отвечает на вопрос «сколько», то классификация – на вопрос «какого вида». Зависимая величина *y* не числовая, а категориальная.

Самая простая форма классификации – бинарный классификатор, порождающий на выходе одну из двух меток (всего два класса: 0 и 1). Результатом классификации может быть также число с плавающей точкой от 0.0 до 1.0, означающее степень уверенности. В этом случае требуется задать пороговую величину (обычно 0.5), определяющую границу между двумя классами. В литературе классы обычно называют положительным (например, 1.0) и отрицательным (например, 0.0). Мы еще вернемся к этому вопросу в разделе «Оценивание моделей» ниже.

Вот несколько примеров бинарной классификации:

- имеется у пациента некоторое заболевание или нет;
- является почтовое сообщение спамом или нет;
- является банковская транзакция мошеннической или законной.

Существуют модели классификации не только с двумя, но и с *N* метками, когда мы вычисляем для каждой метки оценку, а затем в качестве результирующей выбираем метку с наивысшей оценкой. Мы вернемся к этой теме, когда будем говорить о нейронных сетях с несколькими выходами. А в этой главе мы еще обсудим классификацию, когда перейдем к логистической регрессии и архитектурам нейронных сетей в целом.

Рекомендование

Рекомендованием называется процесс предложения товаров пользователю системы в зависимости от того, какие товары он просматривал прежде или какие товары покупают похожие на него пользователи. Один из самых известных рекомендательных алгоритмов – *коллаборативная фильтрация* – стал знаменитым благодаря сайту Amazon.com.

Кластеризация

Кластеризация – это метод обучения без учителя, в котором измеряется расстояние между объектами и похожие объекты собираются вместе. В конце процесса объекты, концентрирующиеся вокруг *n* центроидов, считаются принадлежащими одной группе – кластеру. Метод *K* средних – один из самых известных алгоритмов кластеризации в машинном обучении.

Недообучение и переобучение

Как уже отмечалось, алгоритмы оптимизации в первую очередь решают проблему недообучения, т. е., взяв линию, которая плохо аппроксимирует данные, стараются преобразовать ее так, чтобы аппроксимация стала лучше. Прямая линия, пересекающая диаграмму рассеяния, – пример недообучения (см. рис. 1.7).

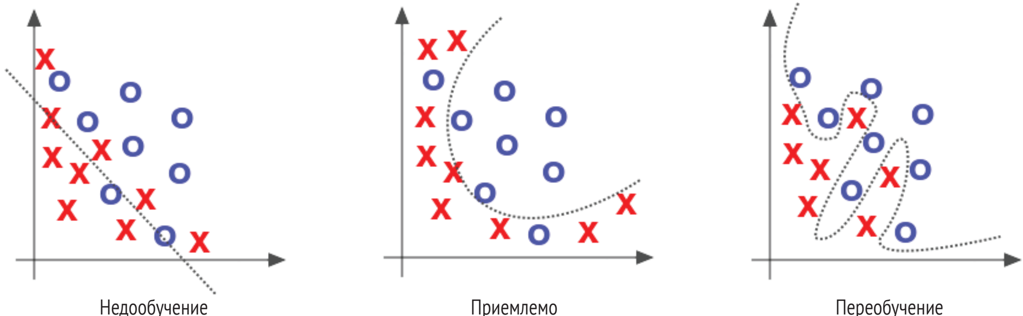


Рис. 1.7 ❖ Недообучение и переобучение

Если линия аппроксимирует данные слишком хорошо, то мы имеем противоположную проблему – переобучение. Устранить недообучение – более приоритетная задача, но в машинном обучении много усилий тратится на то, чтобы воспрепятствовать чрезмерно близкой подгонке к данным. Говоря, что модель переобучена на наборе данных, мы имеем в виду, что частота ошибок на обучающих данных, возможно, и мала, но модель плохо обобщается на всю интересующую нас генеральную совокупность.

По-другому объяснить суть переобучения можно, представив себе вероятные распределения данных. Обучающий набор, который мы пытаемся аппроксимировать линией, – это просто выборка из большого неизвестного множества, и если мы хотим, чтобы модель имела предсказательную силу, то линия, аппроксимирующая эту выборку, должна столь же хорошо аппроксимировать и все множество. Поэтому мы должны предположить, что наша выборка более-менее репрезентативна для всего множества.

Оптимизация

Вышеупомянутый процесс корректировки весов для порождения все более точных гипотез о данных называется *параметрической оптимизацией*. Его можно рассматривать как научный метод. Мы формулируем гипотезу, сравниваем ее с действительностью и либо уточняем, либо заменяем гипотезу, так чтобы добиться лучшего описания реальных явлений. Этот процесс повторяется многократно.

Всякий набор весов представляет собой одну гипотезу о смысле входных данных, т. е. о том, как они соотносятся с информацией, заключенной в метках. Веса – это представление заключений о корреляции между входами сети и выходными метками. Все возможные веса и их комбинации можно рассматривать как пространство гипотез для данной задачи. Попытка сформулировать наилучшую гипотезу сводится к поиску в этом пространстве, а поиск производится с помощью

алгоритмов оптимизации и функций ошибок. Чем больше параметров, тем обширнее пространство поиска. Значительная часть процесса обучения заключается в том, чтобы решить, какие параметры можно игнорировать, а какие играют важную роль.

Решающая граница и гиперплоскости
Под «решающей границей» мы понимаем *n*-мерную гиперплоскость, образованную вектором параметров при линейном моделировании.

Аппроксимация данных линией путем измерения ее стоимости (т. е. расстояния до экспериментальных точек) – это и есть смысл машинного обучения. Чтобы линия более-менее прилично аппроксимировала данные, следует минимизировать суммарное расстояние от нее до всех точек. Мы минимизируем сумму расстояний между точкой x, принадлежащей линии, и соответствующей ей экспериментальной точкой y. В трехмерном пространстве можно представить ошибку как ландшафт, состоящий из гор и долин, а алгоритм оптимизации – как слепого альпиниста, ощупью исследующего склон. Алгоритм оптимизации, например градиентный спуск, информирует альпиниста о направлении понижения склона, чтобы он знал, куда поставить ногу.

Задача состоит в том, чтобы найти веса, минимизирующие разность между тем, что предсказывает сеть ($\hat{b}$ – произведение A и x), и заведомо правильными тестовыми метками b (см. рис. 1.4). Вектор параметров x – это и есть искомые веса. Верность сети является функцией входов и параметров, а скорость приближения к верным значениям параметров – функцией гиперпараметров.

Гиперпараметры
В машинном обучении имеются как параметры модели, так и параметры, которые мы настраиваем, чтобы сеть обучалась быстрее и лучше. Эти настраиваемые параметры называются *гиперпараметрами*, они управляют оптимизацией и выбором модели в ходе обучения с применением алгоритма обучения.

Сходимость
Под *сходимостью* понимается способность алгоритма оптимизации находить вектор параметров, при котором достигается наименьшая возможная ошибка на совокупности обучающих примеров. Говорят, что алгоритм итеративно сходится к решению после опробования нескольких вариантов параметров.

Ниже перечислены три важные функции, применяемые в ходе оптимизации:

- параметры: преобразует входные данные с целью определения найденной сетью классификации;
- функция потерь: измеряет качество классификации на каждом шаге (с точки зрения минимизации ошибки);
- функция оптимизации: направляет процесс в сторону точки с наименьшей ошибкой.

Теперь рассмотрим один класс задач оптимизации – выпуклую оптимизацию.

Выпуклая оптимизация

В случае *выпуклой оптимизации* алгоритм обучения имеет дело с выпуклой функцией стоимости. Если ось x представляет один вес, а ось y – стоимость, то стои-

мость будет равна 0 в какой-то точке оси x и экспоненциально возрастать по обе стороны от нее, когда вес удаляется от идеального значения.

На рис. 1.8 показано, что эту идею функции стоимости можно также поставить с ног на голову.

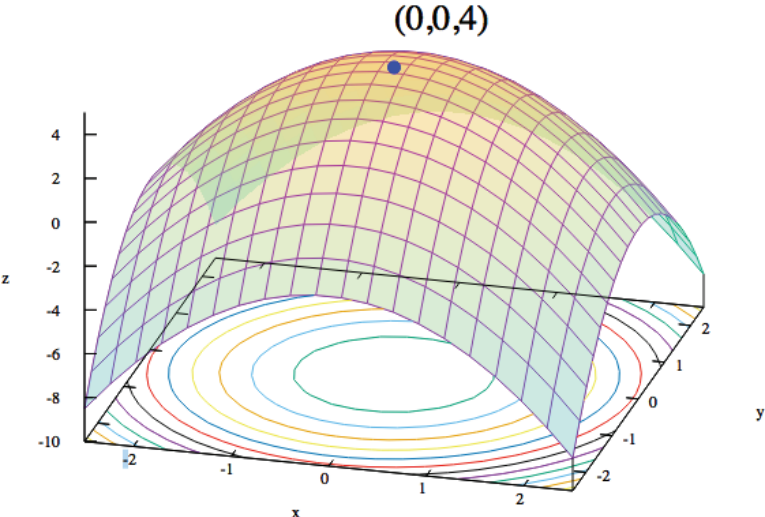


Рис. 1.8 ❖ Наглядное представление выпуклой функции

Еще один способ связать параметры с данными дает метод максимального правдоподобия (ММП). Оценка по методу ММП имеет вид параболы, обращенной «рогами» вниз, при этом правдоподобие откладывается по вертикальной оси, а параметр – по горизонтальной. Каждая точка на параболе измеряет правдоподобие данных при условии некоторого набора параметров. Идея ММП состоит в переборе возможных значений параметров с целью найти такой набор, при котором достигается максимальное правдоподобие данных.

В некотором смысле максимальное правдоподобие и минимальная стоимость – две стороны одной медали. Вычисление функции стоимости от двух весов (при этом мы выходим в трехмерное пространство) дает поверхность, которая получается, если взять лист бумаги за четыре угла, позволив ему свободно провиснуть в середине – что-то вроде миски. По наклону этих выпуклых кривых алгоритм понимает, в каком направлении пространства параметров делать следующий шаг, – мы увидим это на примере алгоритма градиентного спуска, рассматриваемого в следующем разделе.

Градиентный спуск

В алгоритме градиентного спуска мы рассматриваем качество предсказаний сети (функцию от значений весов, или параметров) как ландшафт. Возвышенности соответствуют точкам (наборам параметров) с большой ошибкой предсказания, а долины – точкам с меньшей ошибкой. Мы выбираем какую-то точку в качестве начальных значений весов. Начальные веса можно выбирать, исходя из знаний

о предметной области (если мы обучаем сеть классифицировать виды цветов, то знаем, что длина лепестка существенна, а цвет – нет). Или если мы готовы поручить сети всю работу, то начальные веса можно выбрать случайным образом.

Наша цель состоит в том, чтобы двигаться вниз по склону, к области меньших значений ошибки, настолько быстро, насколько возможно. Алгоритм градиентного спуска может измерить величину наклона поверхности по каждому весу (изменение ошибки, вызванное изменением веса) и тем самым понять, какое направление ведет вниз. Это делается путем вычисления градиента – производной функции потерь. Зная направление, алгоритм изменяет вес, так чтобы стать на один шаг ближе ко дну долины (см. рис. 1.9).

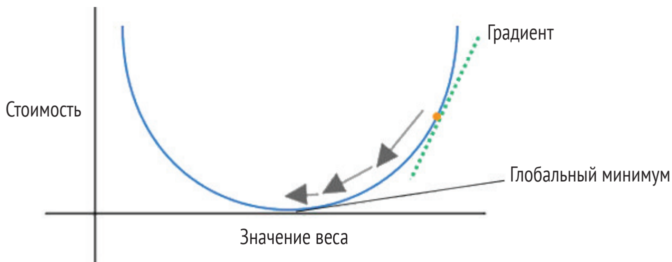


Рис. 1.9 ❖ Изменение веса с целью приближения к глобальному минимуму в алгоритме СГС

Производная измеряет «скорость изменения» функции. В выпуклой оптимизации мы ищем точку, в которой производная функции равна 0. Эта точка называется *стационарной точкой*, или *точкой минимума*. Под оптимизацией понимается минимизация функции (или максимизация, если выпуклость функции направлена вверх, но тогда можно свести задачу к минимизации той же функции с обратным знаком).

Что такое градиент?

Градиент – это обобщение производной функции одной переменной на многомерный случай. Он представляется в виде вектора n частных производных функции f. Понятие градиента полезно в оптимизации, поскольку он указывает направление наибольшей скорости изменения функции, измеряемой как крутизна поверхности графика в этом направлении.

В методе градиентного спуска наклон функции потерь вычисляется путем взятия производной, что должно быть вам знакомо из курса математического анализа. Если функция потерь зависит от одной переменной, то производная в некоторой точке параболы равна тангенсу угла наклона касательной к параболе в этой точке, т. е. отношению изменения y к изменению x. (Из тригонометрии мы знаем, что тангенс угла прямоугольного треугольника равен отношению противолежащего катета к прилежащему.)

Наклон в каждой точке на кривой равен тангенсу угла наклона касательной в этой точке. Как найти эту величину? Мы берем две близко расположенные точки на кривой, вычисляем тангенс угла наклона соединяющей их прямой, а затем уменьшаем расстояние между ними, устремляя его к нулю. В анализе это называется *пределом*.

Процесс измерения потери и изменения веса на каждом шаге в направлении уменьшения ошибки повторяется, пока не будет достигнута точка, в которой дальнейшее уменьшение невозможно. Процесс останавливается в котловине – точке, где верность максимальна. Если функция потерь выпуклая (типичный случай в линейном моделировании), то она имеет единственный глобальный минимум.

Линейное моделирование, цель которого – нахождение вектора параметров *x*, включает четыре компонента:

- гипотеза о данных, например уравнение, используемое моделью для порождения предсказаний;
- функция стоимости, называемая также функцией потерь, например сумма квадратов ошибок;
- функция обновления; в качестве таковой мы вычисляем производную функции потерь.

Наша гипотеза – комбинация обучаемых параметров *x* и входных значений (признаков), ее результатом является дискретный класс или вещественное число (в случае регрессии). Функция стоимости сообщает, насколько далеко мы находимся от глобального минимума функции потерь, а в качестве функции обновления для изменения вектора параметров *x* мы используем производную функции потерь.

Взятие производной функции потерь показывает, насколько нужно подкорректировать каждый параметр, чтобы приблизиться к стационарной точке на кривой потери. Ниже в этой главе мы приведем соответствующие уравнения и покажем, как они работают для линейной и логистической регрессии.

Однако в нелинейных задачах кривая потери не всегда так удобна. Проблема в том, что в гипотетическом ландшафте может быть несколько долин, и механизм градиентного спуска, с помощью которого оптимизируются веса, не знает, достиг ли он самой глубокой долины или просто нижней точки расположенной высоко долины. Нижняя точка самой глубокой долины называется *глобальным минимумом*, а нижние точки остальных долин – *локальными минимумами*. Достигнув локального минимума, градиентный спуск оказывается в тупике, и это один из недостатков алгоритма. В главе 6 мы будем обсуждать гиперпараметры и скорость обучения и тогда поговорим о том, как можно преодолеть эту проблему.

Вторая проблема касается ненормированных признаков, т. е. признаков с сильно различающимися масштабами измерения. Если порядок одного признака – миллионы, а другого – десятые доли, то у метода градиентного спуска возникнут сложности при нахождении самого крутого склона.

Нормировка

В главе 8 мы подробно рассмотрим методы нормировки в контексте векторизации и покажем, как справиться с этой проблемой.

Стохастический градиентный спуск

В методе градиентного спуска вычисляется общая потеря по всем обучающим примерам, и только затем вычисляется градиент и обновляется вектор параметров. В методе СГС градиент и новый вектор параметров вычисляются после

каждого обучающего примера. Показано, что это позволяет ускорить обучение и распараллелить процесс, мы еще вернемся к этому вопросу позже. СГС – аппроксимация «полнопакетного» градиентного спуска.

Обучение на мини-пакетах и СГС

Еще в одном варианте СГС для вычисления градиента используется не один обучающий пример, но и не весь обучающий набор. Этот вариант называется *мини-пакетным*; показано, что его качество выше, чем при использовании обучающих примеров поодиночке. Стохастический градиентный спуск с мини-пакетами сходится быстрее, потому что для вычисления градиента на каждом шаге используется больше примеров.

По мере увеличения размера мини-пакета вычисленный градиент приближается к «истинному» градиенту, вычисленному по всему обучающему набору. Одновременно повышается вычислительная эффективность. Если размер пакета слишком мал (например, всего один обучающий пример), то оборудование используется не так эффективно, как могло бы, особенно если вычисления выполняются на графических процессорах (GPU). С другой стороны, слишком большой мини-пакет тоже неэффективен, потому что тот же градиент можно было бы вычислить с меньшими затратами (в некоторых случаях), воспользовавшись обычным градиентным спуском.

Квазиньютоновские методы оптимизации

Квазиньютоновские методы оптимизации – это итеративные алгоритмы, в которых производится последовательность «линейных поисков». Их отличительная особенность состоит в том, как выбирается направление поиска. Мы будем обсуждать эти методы в последующих главах.

Якобиан и гессиан

Якобианом называется матрица размера $m \times n$, содержащая первые частные производные векторов по векторам.

Гессиан, или матрица Гессе, – это квадратная матрица, содержащая вторые частные производные функции. Она описывает локальную кривизну функции многих переменных. Гессиан встречается в крупномасштабных задачах оптимизации с применением ньютоновских методов, поскольку содержит коэффициенты квадратичного члена в локальном разложении в ряд Тейлора. Практическое вычисление гессиана наталкивается на вычислительные трудности. Обычно в квазиньютоновских алгоритмах применяется аппроксимация гессиана. Примером может служить алгоритм L-BFGS, который мы подробно рассмотрим в главе 2.

Матрицы Якоби и Гессе нечасто будут встречаться на страницах этой книги, но мы хотели, чтобы читатель знал о них и понимал их место в более широком контексте машинного обучения.

Порождающие и дискриминантные модели

На выходе модели можно получить различную информацию в зависимости от того, как она устроена. Есть два основных типа моделей: *порождающие* и *дискриминантные*. Порождающие модели понимают, как были созданы данные, и по-

рождают выход соответствующего типа. Дискриминантным моделям безразлично, как создавались данные, они просто выдают класс или категорию входного сигнала. Для дискриминантной модели важно точно смоделировать границу между классами, и она может дать более детальное представление этой границы, чем порождающая модель. В машинном обучении дискриминантные модели применяются в основном для классификации.

Порождающая модель обучается *совместному* распределению вероятности $p(x, y)$, тогда как дискриминантная – условному распределению вероятности $p(y|x)$. Распределение $p(y|x)$ естественно возникает, когда нужно взять входной сигнал x и породить классифицирующий его выход y, отсюда и название «дискриминантная» (различительная). А поскольку порождающая модель обучается распределению $p(x, y)$, то она порождает вероятный выход по данному входу. Порождающие модели обычно используются в качестве вероятностных графических моделей, улавливающих тонкие связи в данных.

Логистическая регрессия

Логистическая регрессия – хорошо известный вид классификации в линейном моделировании. Она применима как для бинарной классификации, так и для многоклассовой – в форме мультиномиальной логистической регрессии. Технически логистическая регрессия представляет собой регрессионную модель с категориальной зависимой переменной. Бинарная логистическая модель применяется для оценивания вероятности бинарного отклика на одну или несколько входных переменных (независимых переменных, или признаков). На выходе получается статистическая вероятность категории при условии заданных входных предикторов.

Как и в случае линейной регрессии, мы можем записать задачу логистической регрессии в виде $Ax = b$, где A – матрица признаков (например, «вес» или «площадь дома») для всех входных примеров. Каждый входной пример представлен строкой матрицы A, а вектор-столбец b содержит метки, соответствующие входным примерам. Применяя функцию стоимости и метод оптимизации, мы можем найти набор параметров x, минимизирующий ошибку относительно истинных меток по всем предсказаниям.

Как и раньше, будем использовать для решения задачи нахождения вектора параметров x алгоритм СГС. Нам необходимы три компонента:

- гипотеза о данных

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

- функция стоимости – оценка максимального правдоподобия;
- функция обновления – производная функции стоимости.

В этом случае вход состоит из независимых величин (например, входных столбцов, или признаков), а выход – из зависимых (например, «оценок меток»). Интерпретировать это можно так, что функция логистической регрессии объединяет входные значения с весами, чтобы определить, насколько правдоподобен выход. Рассмотрим логистическую функцию более пристально.

Логистическая функция

Логистическая функция (гипотеза) определяется такой формулой:

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

Польза этой функции в том, что она преобразует любой вещественный аргумент в число из интервала от 0.0 до 1.0. Это позволяет интерпретировать результат как вероятность. На рис. 1.10 показан график логистической функции.

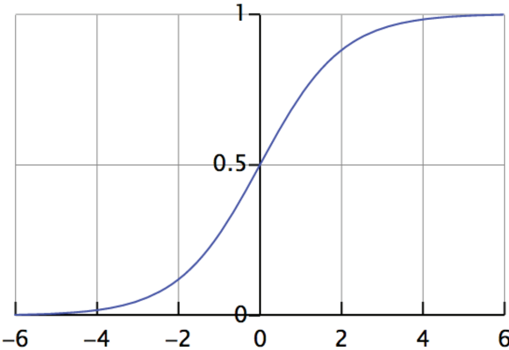


Рис. 1.10 ❖ График логистической функции

Эта функция называется еще *непрерывной логарифмической сигмоидой* с диапазоном от 0.0 до 1.0. Мы еще вернемся к ней в разделе «Функции активации» главы 2.

Интерпретация результата логистической регрессии

Логистическая функция часто обозначается греческой буквой σ (сигма), поскольку ее график напоминает наклоненную и сильно вытянутую букву s. Слева он асимптотически приближается к 0, а справа – к 1.

Если y – сигмоидная функция от x, то при увеличении x функция стремится к 1/1, поскольку $e^{-\theta x}$ стремится к нулю. Напротив, когда x стремится к минус бесконечности, выражение $(1 + e^{-\theta x})$ неограниченно возрастает, а т. к. оно находится в знаменателе, то функция стремится к нулю.

В случае логистической регрессии $f(x)$ представляет вероятность того, что y равно 1 (т. е. истинно) при заданном входе x. Если в задаче оценивания вероятности спама оказывается, что $f(x)$ равно 0.6, то мы можем сказать, что y равно 1 с вероятностью 60%, или что сообщение с вероятностью 60% является спамом. Если определить машинное обучение как метод вывода неизвестных результатов по известным входным данным, то вектор параметров x в модели логистической регрессии описывает силу и степень достоверности наших дедуктивных рассуждений.

Функция logit

Функция logit является обратной к логистической функции.

Оценивание модели

Оценивание модели – это процесс, цель которого состоит в том, чтобы понять, насколько правильно она классифицирует данные. Иногда нас интересует только, как часто предсказания модели правильны, а иногда важно, чтобы предсказания одного вида были правильны гораздо чаще, чем другого. В этом разделе мы обсудим, что такое ложноположительные результаты, безвредные ложноотрицательные результаты, несбалансированные классы и неравная стоимость предсказаний. Начнем с рассмотрения основного инструмента оценивания моделей: *матрицы неточностей* (confusion matrix).

Матрица неточностей

Матрица неточностей (рис. 1.11) – это таблица, в которой представлены предсказанные классификатором и фактические выходные значения (метки). С ее помощью проще понять, насколько хорошо работает модель классификации.

	P′ (Предсказанный)	N′ (Предсказанный)
P (Фактический)	Истинно положительный	Ложноотрицательный
N (Фактический)	Ложноположительный	Истинно отрицательный

Рис. 1.11 ❖ Матрица неточностей

Качество модели измеряется путем подсчета следующих величин:

- истинно положительные результаты:
  - предсказание положительное;
  - метка положительная;
- ложноположительные результаты:
  - предсказание положительное;
  - метка отрицательная;
- истинно отрицательные результаты:
  - предсказание отрицательное;
  - метка отрицательная;
- ложноотрицательные результаты:
  - предсказание отрицательное;
  - метка положительная.

В традиционной статистике ложноположительный результат называют «ошибкой типа I», а ложноотрицательный – «ошибкой типа II». С помощью этих показателей мы можем детально проанализировать качество модели, не ограничиваясь

одним лишь процентом правильных предсказаний. Вычисляя различные комбинации четырех счетчиков в матрице неточностей, мы можем получить такие метрики качества модели:

```
Верность:  0.94
Точность:  0.8662
Полнота:   0.8955
F-мера:    0.8806
```

Мы рассмотрим эти метрики чуть ниже, но начнем с определения чувствительности и специфичности модели.

Чувствительность и специфичность

Чувствительность и *специфичность* – две метрики модели бинарной классификации. Доля истинно положительных результатов показывает, как часто мы правильно классифицируем положительные примеры. Этот показатель называют еще чувствительностью, или полнотой: примером может служить классификация больного пациента как страдающего заболеванием. Чувствительность говорит, насколько хорошо модели удается избегать ложноотрицательных результатов.

Чувствительность = TP / (TP + FN).

Если модель классифицировала здорового пациента как не страдающего заболеванием, то мы имеем истинно отрицательный результат (доля таких результатов называется специфичностью). Специфичность говорит, насколько хорошо модели удается избегать ложноположительных результатов.

Специфичность = TN / (TN + FP).

Зачастую нам необходим компромисс между чувствительностью и специфичностью. Например, модель должна обнаруживать у пациентов серьезное заболевание чаще, поскольку цена неправильной диагностики больного пациента высока. Мы говорим, что такая модель должна иметь низкую специфичность. Серьезное заболевание может угрожать жизни самого пациента и окружающих его людей, поэтому модель должна проявлять высокую чувствительность к этой ситуации. В идеальном мире модель обладала бы чувствительностью 100% (т. е. распознаются все заболевшие) и специфичностью 100% (т. е. ни один здоровый пациент не классифицируется как заболевший).

Верность

Верность – это показатель близости результатов измерений некоторой величины к ее истинному значению.

Верность = (TP + TN) / (TP + FP + FN + TN).

В роли оценки качества модели верность не годится, если велик дисбаланс классов. Если мы просто будем классифицировать все примеры как принадлежащие большему классу, то количество правильных предсказаний будет велико, поэтому мы получим высокую верность, хотя истинного качества модели она не отражает (ведь модель вообще никогда не предсказывает меньший класс или редкое событие).

Точность

В науке и в статистике точностью (precision) называется степень повторяемости результатов измерений при одних и тех же условиях. Другое название – *положительная прогностическая значимость* (positive prediction value). В разговорной речи термины «precision» и «accuracy» часто употребляют как синонимы, но в контексте научного метода их значения различаются.

Точность = TP / (TP + FP).

Измерение может быть верным, но неточным, неверным и тем не менее точным, а также неверным и неточным или верным и точным. Мы считаем измерение достоверным, если оно одновременно верно и точно.

Полнота

То же самое, что чувствительность, употребляется также термин *частота истинно положительных результатов*.

F-мера

В случае бинарной классификации F-мера (или F-оценка, или F1) является мерой верности модели. Это среднее гармоническое точности и полноты:

F-мера = 2TP / (2TP + FP + FN).

Легко видеть, что F-мера находится в диапазоне от 0.0 до 1.0, причем 0.0 соответствует худшей оценке, а 1.0 – лучшей из возможных. Обычно F-мера применяется в информационном поиске и показывает, насколько хорошо модель ищет релевантные результаты. В машинном обучении F-мера используется как обобщенная оценка качества модели.

Контекст и интерпретация оценок

Контекст может играть важную роль в оценивании модели и диктовать применение той или иной из описанных выше метрик. Дисбаланс классов также влияет на выбор метрики, и, как выясняется, существует много наборов данных, в которых число меток, принадлежащих разным классам, сильно различается. Вот несколько типичных примеров такой ситуации:

- предсказание переходов по ссылкам в вебе;
- предсказание смертности в палате интенсивной терапии;
- обнаружение мошеннических операций.

В этих контекстах общий «процент правильных предсказаний» может ничего не говорить о практической ценности модели. Примером может служить набор данных PhysioNet Challenge 2012 года (https://physionet.org/challenge/2012/).

Цель этого конкурса – «предсказание внутрибольничной *смертности* с максимальной верностью при помощи бинарного классификатора». Трудность моделирования этого набора данных состоит в том, что предсказать, что пациент останется жив, легко, поскольку в большинстве примеров исход именно такой. А вот верное предсказание смерти является проблемой: именно ее решение важно для реальной клинической практики. В этом конкурсе принята следующая оценка качества:

Оценка = MIN(Точность, Полнота).

Такая оценка выбрана для того, чтобы участники не стремились в большинстве случаев предсказывать выживание пациента и получать хорошую F-меру, а сосредоточились на правильном предсказании смерти. Это хороший пример того, как контекст может изменять наши представления о качестве модели.

Работа в условиях дисбаланса классов
В главе 6 мы продемонстрируем практические методы компенсации дисбаланса классов. Мы рассмотрим различные аспекты дисбаланса и распределения ошибок в контексте классификации и регрессии.

Итоги

В этой главе мы ввели основные понятия, необходимые для применения машинного обучения на практике. Мы рассмотрели математические основания моделирования как решения уравнения

$$Ax = b.$$

Мы поняли, что матрица A содержит признаки, что x – это вектор параметров, а b – вектор выходных меток. Мы продемонстрировали базовые способы изменения вектора x с целью минимизации потери целевой функции.

Далее в этой книге мы будем возводить здание на фундаменте этих концепций. Мы увидим, что нейронные сети и глубокое обучение основаны на тех же идеях, но включают более сложные способы построения матрицы A, изменения вектора параметров x с помощью методов оптимизации и измерения потери в процессе обучения. А теперь перейдем к главе 2, где будут изложены основы нейронных сетей.

Глава 2

Основы нейронных сетей и глубокого обучения

Когда твои ноги зависли в воздухе,
а голова уткнулась в землю,
Попробуй исхитриться и повернуть её.
Твоя голова взорвется,
Но в ней ничего нет.
И ты спросишь себя:
Где мой разум?

– *The Pixies, «Where is My Mind?»*

Нейронные сети

Нейронные сети – это вычислительная модель, имеющая ряд общих черт с мозгом животного, в котором множество простых блоков параллельно работает в отсутствие центрального управляющего блока. Веса блоков – основное средство долговременного запоминания информации в нейронных сетях, а обновление весов – основной способ обучения новой информации.

В главе 1 мы обсуждали представление модели в виде системы уравнений $Ax = b$. В контексте нейронных сетей матрица A по-прежнему содержит входные данные, а вектор-столбец b – выходные метки для каждой строки матрицы A. В качестве вектора параметров x выступают веса связей между блоками сети. Поведение нейронной сети зависит от ее архитектуры. Архитектура сети определяется (частично) следующими характеристиками:

- количество нейронов;
- количество слоев;
- типы связей между слоями.

Самый известный и простой для понимания тип нейронной сети – многослойная сеть прямого распространения. У нее имеется входной слой, один или несколько скрытых слоев и один выходной слой. Число нейронов в каждом слое может быть различно, и каждый нейрон одного слоя связан со всеми нейронами соседних слоев. Связи между нейронами образуют ациклический граф, показанный на рис. 2.1.

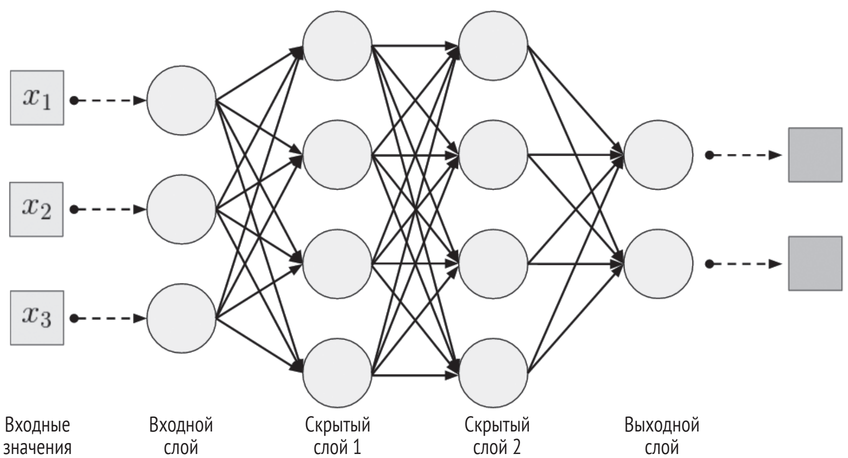


Рис. 2.1 ❖ Топология многослойной нейронной сети

Многослойной сетью прямого распространения можно представить любую функцию при наличии достаточного количества искусственных нейронов. Обычно для ее обучения используется алгоритм *обратного распространения*. В нем для минимизации ошибки на выходе сети применяется градиентный спуск по весам связей.

Локальные минимумы и обратное распространение

Алгоритм обратного распространения может застрять в локальном минимуме, но на практике обычно работает хорошо.

Исторически алгоритм обратного распространения считался медленным, но прогресс в области повышения вычислительной мощности за счет параллелизма и использования графических процессоров (GPU) привел к возрождению интереса к нейронным сетям.

По поводу связи между нейронными сетями и мозгом человека в Интернете и в литературе произнесено немало необдуманных слов. Давайте отделим сигнал от шума и начнем с биологических истоков искусственных нейронных сетей.

Механистический взгляд на разум

Вместо создания жесткой конструкции, требующей, чтобы все входные данные принадлежали какому-то одному типу, мы можем построить модель, отражающую реальный мир, который посылает нам неполную, неоднозначную информацию, так что выведенные из нее умозаключения будут не вполне достоверны.

Этот аспект нейронных сетей знаменует отход от механистического взгляда на разум, который преобладал в начале XX века. Тогда предполагалось, что наш мозг сцеплен с миром детерминированно – как две шестерни, и четкий входной сигнал порождает не менее четкий выходной отклик. Теперь же мы считаем, что на основе неполной и зачастую противоречивой информации человек находит способы устремляться вперед и действовать. Мозг человека делает выводы из вероятностных исходных данных, и так же поступают нейронные сети.

Мы кратко расскажем о биологическом нейроне, а затем бросим взгляд на предтечу нейронных сетей: *перцептрон*. Затем мы увидим, как перцептрон эволюционировал в обобщенный искусственный нейрон, который поддерживает

современные многослойные перцептроны с прямым распространением. По завершении этой главы у вас будут необходимые базовые знания для знакомства с более сложными архитектурами глубоких сетей.

Биологический нейрон

Биологический нейрон (рис. 2.2) – это нервная клетка, являющаяся базовой функциональной единицей нервной системы любого животного. Нейроны взаимодействуют друг с другом, обмениваясь электрохимическими импульсами через синапсы, соединяющие клетки, при условии что импульс достаточно силен, чтобы активировать выпуск химических веществ в синаптическую щель. Сила импульса должна быть выше минимального порога, иначе выпуск химических веществ не произойдет.

На рис. 2.2 показаны основные части нервной клетки:

- тело клетки;
- дендриты;
- аксоны;
- синапсы.

Нейрон состоит из нервной клетки, в которой имеется тело клетки, содержащее много дендритов, но только один аксон. Но этот единственный аксон может многократно ветвиться. Дендриты – это тонкие отростки, исходящие из тела клетки. Аксоны – нервные волокна, основание которых прикреплено к телу клетки.

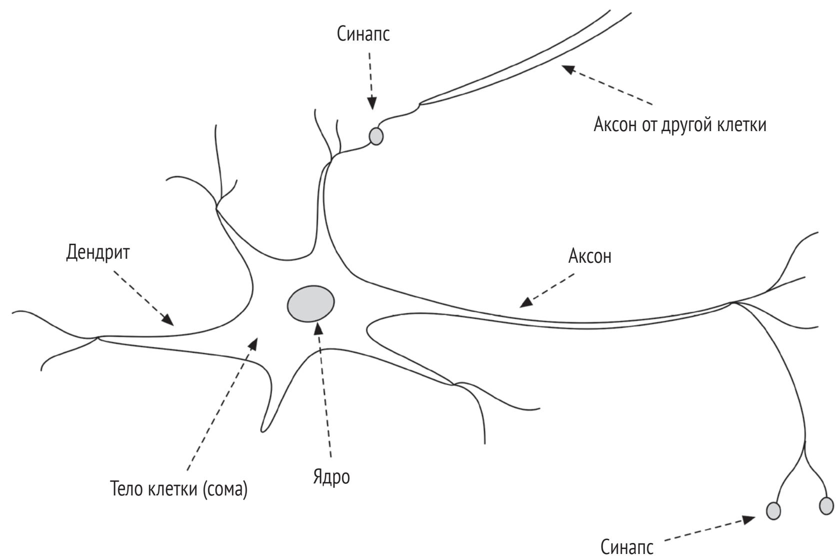


Рис. 2.2 ❖ Биологический нейрон

Синапсы

Синапсы – это соединительные звенья между аксоном и дендритами. Большая часть синапсов передает сигналы от аксона одного нейрона к дендриту другого. Но бывают и исключения, когда у нейрона нет дендритов или аксона или когда синапс соединяет один аксон с другим.

Дендриты

У дендритов есть волокна, образующие сильно разветвленную сеть вокруг нервной клетки. Дендриты дают клетке возможность принимать сигналы от соседних нейронов, и каждый дендрит способен умножать сигнал на свой вес. Здесь под умножением понимается увеличение или уменьшение отношения синаптических нейромедиаторов к количеству химических веществ, поступающих в дендрит.

Аксоны

Аксоны представляют собой длинные отростки, исходящие из тела клетки – сомы. Их длина больше, чем у дендритов, и обычно составляет 1 см (в 100 раз больше диаметра сомы). В конечном итоге аксон разветвляется и соединяется с дендритами других нейронов. Нейроны могут посылать электрохимические импульсы путем изменения трансмембранного потенциала, в результате чего генерируется *потенциал действия*. Сигнал проходит вдоль аксона и активирует синаптические связи с другими нейронами.

Поток информации в биологическом нейроне

Синапсы, повышающие потенциал, называются *возбуждающими*, а уменьшающие потенциал – *тормозящими*. Под *пластичностью* понимаются длительные изменения силы связей в ответ на входной стимул. Доказано, что с течением времени нейроны формируют новые связи и даже мигрируют. Этот комбинированный механизм изменения связей и отвечает за процесс обучения человеческого мозга.

От биологического нейрона к искусственному

Доказано, что мозг животного отвечает за базовые составляющие психической деятельности. Мы можем изучить основные компоненты мозга и понять, как они функционируют. В результате исследований была составлена карта функциональных отделов мозга и продемонстрировано, как проследить прохождение сигнала через нейроны.

Сверточные нейронные сети и зрительная система млекопитающих

Далее в этой книге мы будем рассматривать разновидность глубоких сетей – сверточные нейронные сети (СНС). В СНС изображение представляется в виде различных слоев аналогично тому, как мозг обрабатывает зрительную информацию. Хотя это исследование само по себе интересно, оно не означает, что СНС дает хорошую аппроксимацию мозговой деятельности млекопитающих.

Однако мы все еще не до конца понимаем, каким образом из этой децентрализованной совокупности функциональных блоков возникают мышление и сознание.

Центр сознания

В XVIII веке возникло понимание, что мозг является «центром сознания». К концу XIX века начали составлять карту функциональных отделов мозга животных. Раньше средоточием сознания считалось сердце и, как ни странно, селезенка.

Теперь, когда мы лучше понимаем, как работает биологический нейрон, познакомимся с первыми попытками моделирования нейрона – перцептроном.

Перцептрон

Перцептрон – это линейная модель, применяемая для бинарной классификации. В нейронных сетях перцептрон рассматривается как искусственный нейрон со ступенчатой функцией Хевисайда в качестве функции активации. То и другое мы определим ниже в этой главе. Предшественником перцептрона был блок пороговой логики (БПЛ), разработанный Маккалоком и Питтсом в 1943 году и способный обучаться логическим функциям И и ИЛИ. Алгоритм обучения перцептрона относится к алгоритмам обучения с учителем. В основе идеи как БПЛ, так и перцептрона лежал биологический нейрон.

История перцептрона

Перцептрон был изобретен в 1957 году в Корнелльской авиационной лаборатории Фрэнком Розенблаттом. Проект финансировался Управлением военно-морских исследований США и освещался в газете New York Times:

> Зачаточный электронный компьютер, который, как ожидает ВМС, будет способен ходить, говорить, видеть, писать, воспроизводить себя и осознавать свое существование.

Очевидно, эти прогнозы были несколько преждевременны – как много раз случалось с обещаниями, которые давали адепты машинного обучения и ИИ. Первые версии предполагалось изготовить в виде физической машины, а не компьютерной программы. Первая программная реализация была разработана для IBM 704, а затем воплощена в машине Mark I Perceptron.

Отметим также, что в 1943 году Маккалок и Питтс[1] ввели в обиход важную идею анализа нейронной активности, основанную на пороговых значениях и взвешенных суммах. Эти идеи легли в основу разработки модели для более поздних вариантов, в т. ч. перцептрона.

Mark I Perceptron

Машина Mark I Perceptron проектировалась для распознавания образов в интересах ВМС США. В ней было 400 фотоэлементов, соединенных с искусственными нейронами, а веса были реализованы как потенциометры. Обновление весов производилось физически с помощью электромоторов.

Определение перцептрона

Перцептрон – это линейный бинарный классификатор с простой связью между входом и выходом, изображенный на рис. 2.3. Как видим, он вычисляет взвешенную сумму n входов и передает сумму ступенчатой функции с заданным пороговым значением. Обычно в перцептронах применяется ступенчатая функция Хевисайда с порогом 0.5. Эта функция возвращает число 0 или 1 в зависимости от входного аргумента.

Решающую границу и результат классификации функцией Хевисайда можно описать следующим образом:

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x >= 0 \end{cases}.$$

[1] McCulloch and Pitts, 1943. A logical calculus of the ideas immanent in nervous activity // http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf.

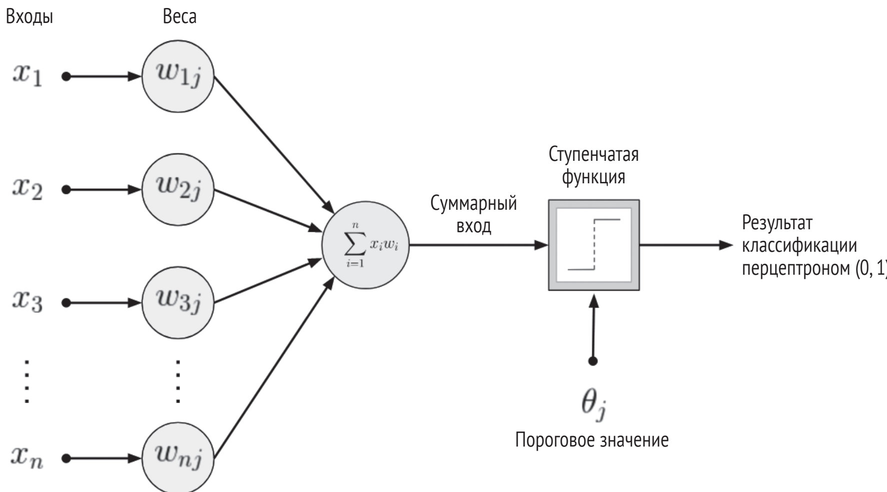


Рис. 2.3 ❖ Однослойный перцептрон

Суммарным входом функции активации (в данном случае функции Хевисайда) является скалярное произведение входов и весов связей. В табл. 2.1 объясняется, как производится это вычисление.

Таблица 2.1. Параметры функции суммирования

Параметр функции	**Описание**
w	Вектор вещественных весов связей
w · x	Скалярное произведение $\sum_{i=1}^{n} w_i x_i$
n	Количество входов перцептрона
b	Смещение (от входных значений не зависит, отодвигает решающую границу от начала координат)

Результат ступенчатой функции (функции активации) является выходом всего перцептрона и определяет классификацию входных значений. Если смещение отрицательно, то обученные веса должны быть значительно больше, чтобы получить на выходе классификатора значение 1. Смещение *b* приводит к сдвигу решающей границы. От входных значений оно не зависит, а определяется в процессе работы алгоритма обучения перцептрона.

Однослойный перцептрон

В работах по нейронным сетям такой перцептрон чаще всего называют однослойным, чтобы отличить его от «многослойного перцептрона», изобретенного позже.

Однослойный перцептрон в роли линейного классификатора является простейшей формой семейства нейронных сетей прямого распространения.

Связь перцептрона с биологическим нейроном

Хотя мы не располагаем полной моделью работы мозга, все же легко видеть, что перцептрон был разработан по образцу биологического нейрона. Входные сигналы поступают в него через взвешенные связи, и это похоже на передачу информации по синапсам.

Алгоритм обучения перцептрона

Алгоритм обучения перцептрона изменяет веса до тех пор, пока не будут правильно классифицированы все входные данные. Алгоритм не завершается, если входные данные не допускают линейного разделения. Набор данных называется линейно разделимым, если существует гиперплоскость, делящая его на два класса.

В начале алгоритма обучения вектор весов инициализируется небольшими случайными значениями или нулями. Затем на вход алгоритма по очереди подаются входные данные, и результат классификации сравнивается с известной меткой. Для выполнения классификации вычисляется взвешенная сумма признаков (столбцов) и весов. Первым входным значением всегда является 1.0, а первым весом – смещение модели. Вычисленное скалярное произведение входов и весов подается на вход рассмотренной выше функции активации.

Если классификация оказалась правильной, веса не изменяются. В противном случае веса корректируются. Веса отдельных связей изменяются в режиме «онлайнового обучения». Цикл продолжается, пока все входные примеры не будут классифицированы правильно. Если набор данных не является линейно разделимым, то алгоритм не завершается. На рис. 2.4 показан линейно неразделимый набор – логическая функция ИСКЛЮЧАЮЩЕЕ ИЛИ.

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Рис. 2.4 ❖ Функция ИСКЛЮЧАЮЩЕЕ ИЛИ

Однослойный перцептрон не способен решить задачу моделирования этой функции, что является доказательством его ограниченности.

Ограничения раннего варианта перцептрона

Когда прошла первая эйфория, выяснилось, что распознаваемое перцептроном множество образов ограничено. Считалось, что неспособность решать нелинейные задачи (например, классифицировать линейно неразделимые наборы данных) ставит крест на всей идее нейронных сетей. В книге Минского и Пейперта «Перцептроны», вышедшей в 1969 году, были проиллюстрированы ограничения однослойного перцептрона. Но тогда индустрия еще не поняла, что многослойный перцептрон может решить и задачу ИСКЛЮЧАЮЩЕГО ИЛИ, и многие другие нелинейные задачи.

Первая зима ИИ: 1974–1980

Непонимание возможностей многослойных перцептронов стало причиной утраты интереса к нейронным сетям и прекращения финансирования их исследований на следующее десятилетие. Возрождение нейронных сетей пришлось на середину 1980-х годов, когда приобрел популярность алгоритм обратного распространения (хотя открыт он был Уэбосом еще в 1974 году).

Многослойные сети прямого распространения

В многослойной сети прямого распространения имеется входной слой, один или несколько скрытых слоев и выходной слой. В каждом слое расположено один или несколько искусственных нейронов. Эти нейроны похожи на однослойный перцептрон, но у каждого имеется своя функция активации, вид которой зависит от назначения слоя в сети. Ниже в этой главе мы рассмотрим типы слоев многослойного перцептрона, а пока поговорим об эволюции искусственного нейрона, обусловленной ограничениями однослойного перцептрона.

Эволюция искусственного нейрона

Искусственный нейрон в многослойном перцептроне напоминает своего предшественника – однослойный перцептрон, но обладает большей гибкостью в плане выбора функции активации. На рис. 2.5 показана модифицированная диаграмма искусственного нейрона, основанного на перцептроне.

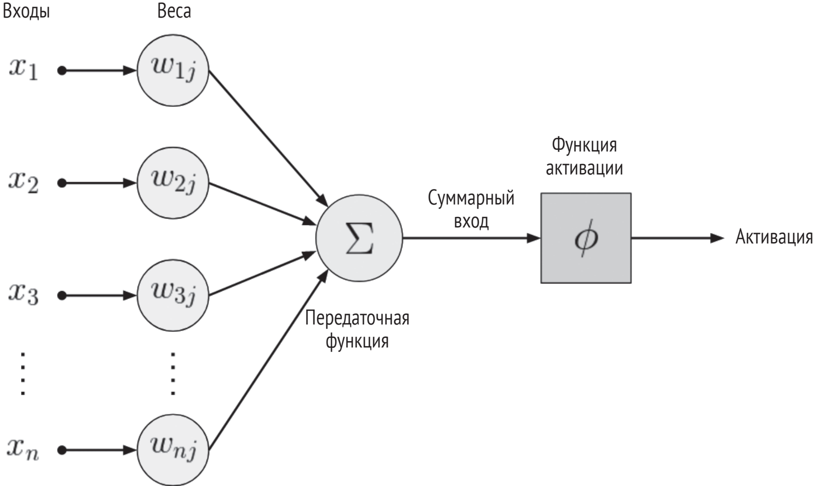


Рис. 2.5 ❖ Искусственный нейрон в многослойном перцептроне

Этот рисунок похож на рис. 2.3, но теперь мы видим обобщенную функцию активации. По мере изучения искусственного нейрона мы детализируем эту диаграмму.

Замечание о термине «нейрон»
Начиная с этого момента, слово «нейрон» будет обозначать искусственный нейрон, изображенный на рис. 2.5.

На вход функции активации по-прежнему подается скалярное произведение весов и входных признаков, но благодаря гибкости функции мы можем по-разному порождать выходное значение. В этом и состоит основное отличие от раннего перцептрона, в котором использовалась кусочно-постоянная функция Хевисайда. Это усовершенствование позволило выразить более сложные результаты активации.

Вход искусственного нейрона. Искусственный нейрон (рис. 2.6) может игнорировать входные сигналы (если вес соответствующей связи равен 0) или передавать их функции активации. Функция активации также имеет возможность отфильтровать данные, если на ее выходе получается нулевое значение.

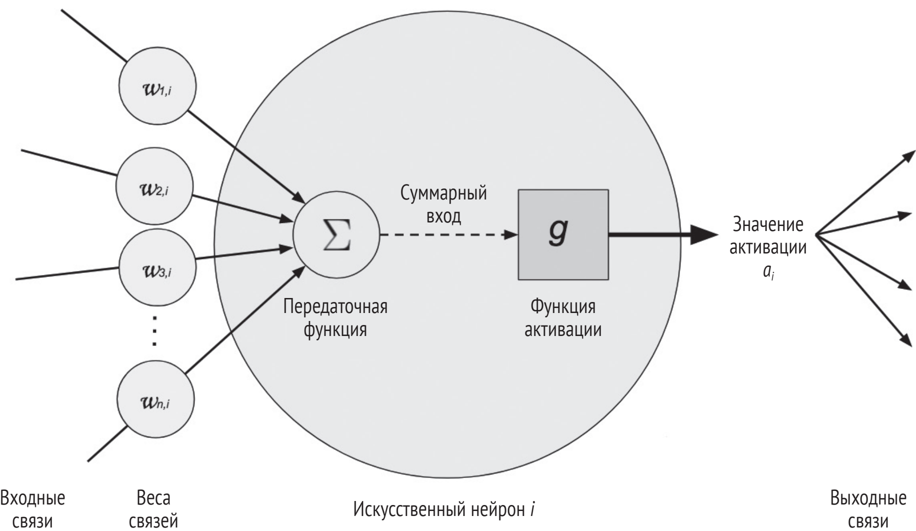


Рис. 2.6 ❖ Детальное изображение искусственного нейрона в нейронной сети в виде многослойного перцептрона

Суммарный вход нейрона равен взвешенной сумме сигналов активации, поступающих по входным связям (рис. 2.6). Во входном слое функция активации линейна (она просто передает значение входного признака дальше). В скрытых слоях входами являются выходы функции активации других нейронов. Математически суммарный вход (полную взвешенную сумму) нейрона можно записать в виде:

$$input_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i,$$

где $\mathbf{W}_i$ – вектор весов связей, входящих в нейрон i, а $\mathbf{A}_i$ – вектор значений активации для входов в нейрон i. Включим еще в это уравнение аддитивный член смещения, свой для каждого слоя:

$$input_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i + \boldsymbol{b}.$$

На выходе нейрона получается результат применения функции активации к этому суммарному входу:

$$a_i = g(input_sum_i).$$

Подставим в это выражение определение $input_sum_i$:

$$a_i = g(\mathbf{W}_i \cdot \mathbf{A}_i + b).$$

Это значение активации нейрона i передается следующему слою по связям, ведущим к другим нейронам.

Другие системы обозначений

В литературе можно встретить и другую запись выхода искусственного нейрона:

$h_{w,b}(x) = g(\mathbf{w} \cdot \mathbf{x} + b)$.

Она несколько отличается от приведенного выше уравнения. Мы приводим ее только для того, чтобы вы знали о возможных вариантах изложения одних и тех же концепций в разных источниках.

Если в роли функции активации выступает сигмоида, то имеем:

$$g(z) = \frac{1}{1+e^{-z}}.$$

Область значений этой функции – тот же интервал (0, 1), что и у функции логистической регрессии.

Сигмоидные функции активации на практике

Мы упомянули сигмоидную функцию активации только ради исторической полноты. Ниже мы увидим, что в наше время сигмоиды вышли из моды.

На вход поступают данные, из которых мы хотим извлечь информацию, а веса связей и смещения – величины, управляющие процессом извлечения, способные активировать или подавлять входные сигналы. Как и в случае перцептрона, существует алгоритм обучения, который изменяет веса и смещения каждого нейрона. Ниже мы рассмотрим его.

Мы знаем, что биологические нейроны не передают каждый полученный электрохимический импульс. Так и искусственные нейроны не являются просто проводами или диодами, передающими сигналы. Они действуют избирательно. Они фильтруют поступающие данные, агрегируют их, преобразуют и передают последующим нейронам только полезную часть информации. В следующем разделе мы проиллюстрируем этот эффект.

Нейроны можно классифицировать по виду получаемых входных данных (бинарные или непрерывные) и по характеру преобразования (функции активации), порождающего выходной сигнал. В DL4J у всех нейронов одного слоя функция активации одна и та же.

Веса связей в нейронной сети – это коэффициенты, которые масштабируют (усиливают или ослабляют) входной сигнал, поступающий к нейрону. В общепринятых представлениях нейронных сетей узлы соединены линиями со стрелками – ребрами математического графа. Обычно веса связей обозначаются буквой w.

Смещения – это скалярные величины, прибавляемые к входному сигналу, чтобы хотя бы несколько узлов в каждом слое активировалось вне зависимости от силы сигнала. Смещения позволяют продолжить обучение, заставляя сеть реагировать даже на слабый сигнал. Благодаря им сеть пробует новые интерпретации, или виды поведения. Обычно смещения обозначаются буквой b и, как и веса, модифицируются в процессе обучения.

Функции активации. Так называются функции, которые управляют поведением нейрона. Передача входного сигнала по сети называется *прямым распространением*. Функции активации преобразуют линейные комбинации входов, сигналов и смещений. Результаты преобразований подаются на вход следующего слоя. Многие (но не все) нелинейные преобразования, применяемые в нейрон-

ных сетях, приводят данные к какому-нибудь удобному диапазону, например от 0 до 1 или от –1 до 1. Если нейрон передает ненулевое значение следующему нейрону, то говорят, что он *активируется*, или *возбуждается*.

Значения активации
Так называются значения, передаваемые от предыдущего слоя следующему. Это результаты, вычисленные функциями активации всех нейронов предыдущего слоя.

В разделе «Функции активации» ниже мы рассмотрим различные типы функций активации и их место в более широком контексте нейронных сетей.

Функции активации и их важность
Функции активации и их использование – тема, проходящая красной нитью почти по всем главам книги. В библиотеке DL4J применена послойная архитектура, основанная на различных типах функций активации.

Сравнение биологического и искусственного нейронов

Мы можем отступить на шаг и задаться вопросом: «Насколько близко искусственный нейрон соответствует биологическому?» Прослеживается аналогия с функцией входных связей, выполняемой дендритами биологического нейрона, и функцией суммирования, присущей соме. Наконец, роль функции активации в биологическом нейроне играет аксон.

Ограниченность сравнения
Отметим (в очередной раз), что биологический нейрон сложнее искусственного аналога. Продолжаются исследования, цель которых – лучше понять функционирование биологического нейрона.

Архитектура нейронных сетей прямого распространения

Теперь, осознав различия между искусственным нейроном и перцептроном, мы можем лучше понять структуру полной многослойной нейронной сети прямого распространения. В таких сетях нейроны организованы в группы, называемые слоями. В многослойной нейронной сети имеются:

- один входной слой;
- один или несколько полносвязных скрытых слоев;
- один выходной слой.

Как показано на рис. 2.7, каждый нейрон одного слоя (представлены кружочками) связан во всеми нейронами следующего слоя.

У всех нейронов одного слоя одна и та же функция активации (как правило). Для входного слоя входным сигналом является исходный вектор. На входы нейронов остальных слоев подаются выходы (значения активации) нейронов предыдущего слоя. На данные, распространяющиеся по сети в прямом направлении, оказывают влияние веса связей и тип функции активации. Рассмотрим теперь специфические особенности слоев различных типов.

Входной слой. Через этот слой в сеть поступают входные данные (векторы). Количество нейронов во входном слое обычно совпадает с количеством входных признаков. За входным слоем расположен один или несколько скрытых слоев. Входной слой в классической сети прямого распространения полносвязан со следующим за ним скрытым слоем, но существуют и другие архитектуры сетей, в которых полносвязность не требуется.

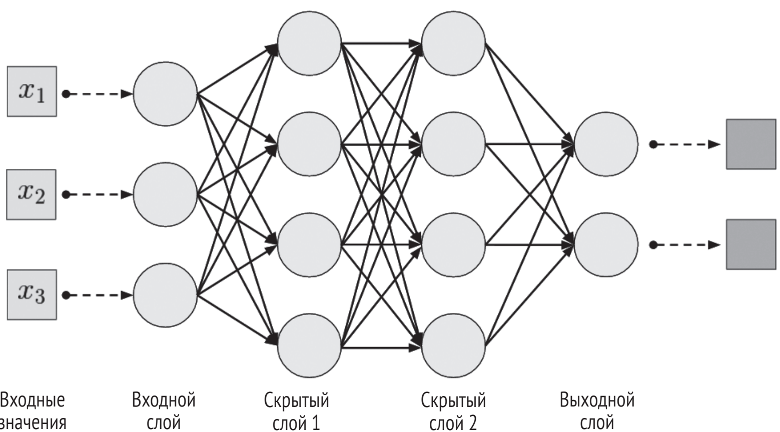


Рис. 2.7 ❖ Топология полносвязной многослойной нейронной сети прямого распространения

Скрытый слой. В нейронной сети прямого распространения имеется один или несколько скрытых слоев. В весах связей между слоями закодирована информация, извлеченная сетью из исходных данных в процессе обучения. Именно благодаря скрытым слоям сеть способна моделировать нелинейные функции, неприступные для однослойных перцептронов.

Выходной слой. Предсказание модели мы получаем от выходного слоя. Поскольку нейронная сеть отображает пространство входов на пространство выходов, то выходной слой дает ответ, зависящий от входных данных. В зависимости от конфигурации сети ответ может быть вещественным числом (регрессия) или набором вероятностей (классификация). Это определяется типом функции активации в выходном слое. Обычно в выходном слое для классификации применяется функция *softmax* или *сигмоида*. Различие между этими функциями активации мы рассмотрим ниже в этой главе.

Связи между слоями. В полносвязной сети прямого распространения каждый нейрон предыдущего слоя связан со всеми нейронами следующего слоя. Веса связей изменяются по мере того, как алгоритм обратного распространения ищет оптимальное решение. Математически веса можно представлять себе как вектор параметров в том смысле, в котором это слово употреблялось ранее в разделе, посвященном линейной алгебре, где процесс машинного обучения описывался как нахождение вектора параметров, минимизирующего ошибку.

Итак, мы рассмотрели базовую структуру нейронных сетей прямого распространения. Далее в этой главе мы поговорим о механизме обучения методом обратного распространения, а также об особенностях функций активации. И завершим главу обзором наиболее употребительных функций потерь и гиперпараметров.

Обучение нейронных сетей

В хорошо обученной нейронной сети веса подобраны так, чтобы сигнал усиливался, а шум ослаблялся. Чем больше вес, тем сильнее корреляция между сигналом

и выходом сети. Входы с большими весами влияют на интерпретацию данных сетью сильнее, чем входы с малыми весами.

Процесс обучения в любом алгоритме на основе весов заключается в пересчете весов и смещений, так что одни становятся больше, а другие меньше. В результате значимость одних частей информации увеличивается, а других – уменьшается. Это позволяет модели узнать, какие предикторы (признаки) с какими выходами связаны, и соответствующим образом подстроить веса и смещения.

В большинстве наборов данных определенные признаки сильно коррелируют с определенными метками (например, площадь дома напрямую связана с его продажной ценой). Нейронные сети обучаются этим связям вслепую, генерируя гипотезы, основанные на входных данных, а затем измеряя верность полученных результатов. Функции потерь в таких алгоритмах оптимизации, как стохастический градиентный спуск (СГС), вознаграждают сеть за правильные гипотезы и штрафуют за неправильные. СГС сдвигает параметры сети в сторону хороших предсказаний, подальше от плохих.

По-другому на процесс обучения можно взглянуть, рассматривая метки как теории, а набор признаков – как факты. Тогда можно сказать, что сеть стремится выявить корреляцию между теорией и фактами. Модель пытается ответить на вопрос «какая теория поддержана фактами?». Памятуя об этих соображениях, рассмотрим алгоритм, который чаще всего упоминается в контексте нейронных сетей: *обучение методом обратного распространения*.

Обучение методом обратного распространения

Алгоритм обратного распространения – важная часть процесса уменьшения ошибки в модели нейронной сети. Чтобы объяснить его, вернемся к обсуждению того, как информация циркулирует в сети прямого распространения. Сначала объясним принцип работы на пальцах, а затем перейдем к математической нотации и псевдокоду.

Истоки алгоритма обратного распространения

Обучение методом обратного распространения было открыто Брайсоном и Хо в 1969 году. В практическом применении нейронных сетей на него не обращали внимания вплоть до возрождения интереса к ним в середине 1980-х.

Интуитивное описание алгоритма

Обучение методом обратного распространения похоже на обучение перцептрона. Мы хотим вычислить отклик на входной сигнал путем прямого распространения по сети. Если отклик совпадает с меткой, то не надо делать ничего. Если же не совпадает, то нужно подправить веса связей в сети.

Для иллюстрации обучения нейронной сети общего вида рассмотрим псевдокод алгоритма, приведенный в примере 2.1.

Пример 2.1 ❖ Псевдокод алгоритма обучения нейронной сети общего вида

```
function neural-network-learning( training-records ) returns network
    network <- initialize weights (randomly)
    start loop
        for each example in training-records do
            network-output = neural-network-output( network, example )
```

```
            actual-output = наблюдаемый выход, ассоциированный с примером
            обновить веса связей в сети на основе
                { example, network-output, actual-output }
        end for
    end loop когда предсказания для всех примеров правильны или выполнены
             условия остановки
    return network
```

Идея в том, чтобы распределить штраф за ошибку между весами, внесшими вклад в выход. В случае алгоритма обучения перцептрона это легко, потому что на выходное значение влияет только один вес для каждого входа. Но в многослойных сетях прямого распространения ситуация сложнее, поскольку на пути от входов к выходам находится много весов. Каждый вес вносит вклад в несколько выходов, поэтому алгоритм обучения должен быть более изощренным.

Обратное распространение – это прагматичный подход к распределению вклада в ошибку между отдельными весами. Здесь есть сходство с алгоритмом обучения перцептрона. Мы пытаемся минимизировать расхождение между меткой (истинным выходным значением), ассоциированной с данным входом, и значением, сгенерированным сетью. В следующем разделе мы рассмотрим математическую нотацию, встречающуюся в литературе по обратному распространению ошибки в нейронных сетях прямого распространения.

Предостережение по поводу алгоритмов обучения в многослойных сетях

Не гарантируется, что алгоритм обучения в многослойных сетях сходится к глобальному оптимуму, равно как не гарантируется его эффективность во всех случаях. Это прямое следствие того, что обучение функции общего вида в худшем случае является нерешаемой задачей. Но на практике алгоритм обучения работает вполне прилично, особенно после правильной настройки гиперпараметров.

Более пристальный взгляд на обратное распространение

Как правило, в этой книге мы не злоупотребляем математикой. Но в вопросе обратного распространения, которое является основой едва ли не всего, о чем пойдет речь, мы чувствуем необходимость привести иллюстрацию хотя бы на уровне обозначений.

Замечание по поводу обозначений

Приведенные ниже обозначения похожи на те, с которыми вы можете столкнуться, читая текст доклада на конференции или известный учебник по машинному обучению. Надеемся, что сумеем объяснить их так, чтобы все было понятно. В идеале это станет отправной точкой для самостоятельного изучения огромной существующей литературы по нейронным сетям и глубокому обучению.

Вернемся к предыдущему рисунку и сосредоточим внимание на входном и первом скрытом слое (рис. 2.8).

На рис. 2.8 показаны некоторые обозначения. Мы будем использовать их в этом разделе для объяснения обратного распространения. См. табл. 2.2.

Чтобы подготовить сцену для объяснения алгоритма, рассмотрим его псевдокод в примере 2.2.

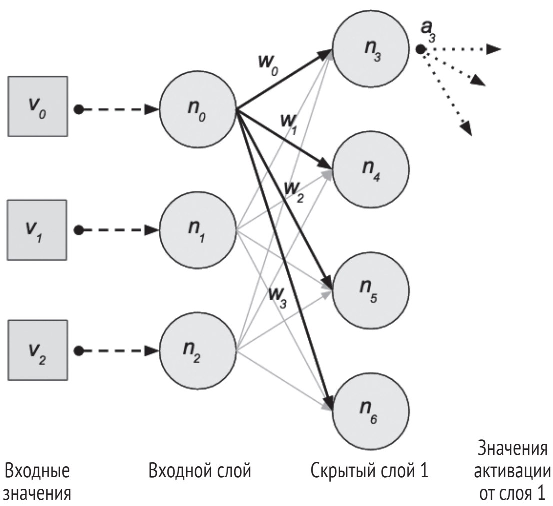


Рис. 2.8 ❖ Часть многослойного перцептрона в увеличенном масштабе

Таблица 2.2. Обозначения, относящиеся к нейронной сети

Обозначение	**Пояснение**
i	Индекс нейрона
n_i	Нейрон с индексом i
j	Индекс нейрона предыдущего слоя, связанного с нейроном i
a_j	Значение активации нейрона i (выход нейрона i)
$\mathbf{A}_i$	Вектор значений активации нейрона i
g	Функция активации
g'	Производная функции активации
Err_i	Разность между выходом сети и фактическим значением для обучающего примера
$\mathbf{W}_i$	Вектор весов связей, ведущих в нейрон i
$W_{j,i}$	Вес связи, идущей от нейрона j в нейрон i
$input_sum_i$	Взвешенная сумма входов в нейрон i
$input_sum_j$	Взвешенная сумма входов в нейрон j предыдущего слоя (в алгоритме обратного распространения)
a	Скорость обучения
Δ_j	Часть ошибки, приходящаяся на нейрон j предыдущего слоя
Δ_i	Часть ошибки, приходящаяся на нейрон i; $Err_i \times g'(input_sum_i)$

Пример 2.2 ❖ Псевдокод алгоритма обратного распространения для обновления весов

```
function backpropagation-algorithm
    ( network, training-records, learning-rate ) returns network
    network <- initialize weights (randomly)
    start loop
        for each example in training-records do

            // вычислить выход этого входного примера
            network-output <- neural-network-output( network, example )

            // вычислить ошибку и дельту для нейронов выходного слоя
```

```
            example_err <- target-output - network-output

            // обновить веса связей, ведущих в выходной слой
            W_{j,i} ← W_{j,i} + α × a_j × Err_i × g'(input_sum_i)

            for each subsequent-layer in network do

                // вычислить ошибку в каждом блоке
                Δ_j ← g'(input_sum_j)Σ_i W_{j,i} Δ_i

                // обновить веса входящих в слой связей
                W_{k,j} ← W_{k,j} + α × a_k × Δ_j
            end for

        end for
    end loop когда сеть сойдется
    return network
```

Замечание о функциях потерь в псевдокоде

В псевдокоде из примера 2.2 функция потерь (описывается ниже в этой главе) явно не вызывается. Мы представили алгоритм обратного распространения в виде псевдокода, поскольку считаем, что так будет понятно специалистам-практикам. Но в приложение C мы включили также описание, более подходящее для читателей с математическим складом ума.

В данном случае член Err_i зависит от производной функции потерь. Мы пользуемся среднеквадратической ошибкой (СКО), поэтому производная оказывается разностью.

Разбор псевдокода алгоритма обратного распространения

На вход алгоритма из примера 2.2 подаются следующие данные:

- сеть: многослойная нейронная сеть прямого распространения;
- обучающие примеры: набор обучающих векторов и соответствующих им выходных меток;
- скорость обучения, обычно обозначаемая греческой буквой альфа.

В начале работы мы инициализируем веса нейронной сети, после чего входим в цикл обработки входных примеров (цикл продолжается, пока не будет выполнено условие остановки или не достигнуто максимальное число периодов). Сначала вычисляется выход сети для текущего примера. Затем этот выход сравнивается с фактической меткой для данного примера и вычисляется ошибка (*example_err*).

Теперь все готово к вычислению обновлений весов связей, ведущих в выходной слой.

Обновление весов выходного слоя. Новые веса выходного слоя вычисляются по формуле:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'\,(input_sum_i).$$

Это правило обновления веса связи между нейронами j и i. Связь, ведущая в выходной слой, выделена на рис. 2.9.

Как видим, мы работаем с весом связи между нейроном j предыдущего скрытого слоя и текущим нейроном i. Скорость обучения α (обсуждается ниже) умножается на значение активации нейрона j, равное результату применения функции активации этого нейрона к суммарному входу в него.

Суммарный вход, передаваемый функции активации нейрона i, равен скалярному произведению вектора входящих весов $\mathbf{W}_j$ на вектор активации $\mathbf{A}_j$ плюс смещение:

$$input_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i + b.$$

Таким образом, значение активации нейрона *j* вычисляется по формуле:

$a_j = g(input_sum_j)$.

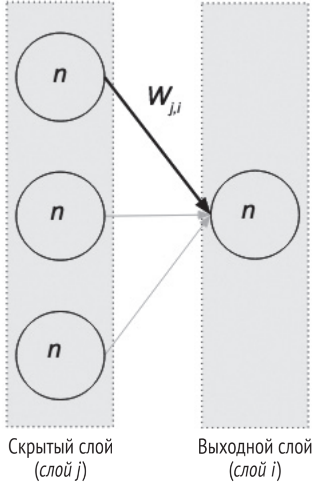


Рис. 2.9 ❖ Обновление весов связей, ведущих в выходной слой

Ошибка для примера *e* в нейроне *i* обозначается Err_i. Вклад производной функции активации $g'(x)$, применяемой к суммарному входу нейрона *i*, описывается членом

$g'(input_sum_i)$.

Эта формула похожа на формулу обновления весов перцептрона, только вместо исходных входных значений используются значения активации предыдущего слоя. Кроме того, в нее входит производная функции активации, выражающая ее градиент.

Выражение для члена ошибки. Выше мы обозначили $g'(z)$ производную функции активации. Член ошибки, обычно обозначаемый Δ_i, вычисляется по формуле:

$\Delta_i = Err_i \times g'(input_sum_i)$.

Это позволяет записать правило обновления в более компактном виде:

$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$.

Это выражение встречается во внутреннем цикле псевдокода, который мы обсудим ниже.

Градиентный спуск в пространстве весов
Обратное распространение рассматривается как градиентный спуск в пространстве весов, когда градиент вычисляется относительно поверхности функции ошибки, которая описывает ошибку на входных признаках в виде функции от значений весов нейронной сети.

Новое правило обновления ошибки. Правило обновления значений дельты теперь принимает вид:

$\Delta_j \leftarrow g'(input_sum_j)\Sigma_i W_{j,i}\Delta_i$.

Тем самым мы получаем новое правило обновления весов связей между входами и скрытым слоем.

Обновление скрытых слоев. Применяя алгоритм обратного распространения, мы движемся назад по скрытым слоям, обновляя веса связей, до тех пор, пока не дойдем до входного слоя. На рис. 2.10 выделена связь между двумя скрытыми слоями.

Чтобы обновить вес этой связи, мы берем дельту, вычисленную на предыдущем шаге, и умножаем ее на значение активации нейрона предыдущего слоя и на скорость обучения.

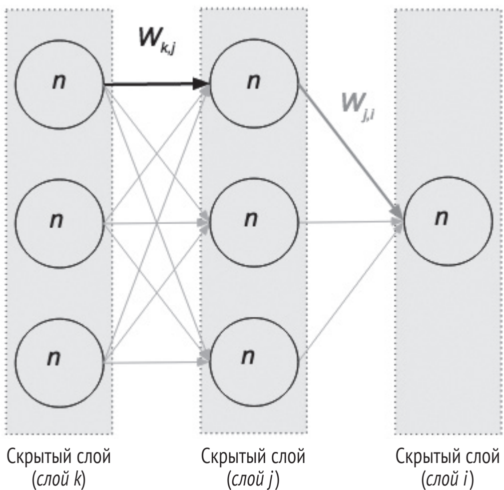


Рис. 2.10 ❖ Связи между скрытыми слоями

Сложив с предыдущим значением веса, мы получаем обновленное значение:

$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$.

Веса и смещения, ответственные за ошибку, уменьшаются, чтобы уменьшить сигнал от них. Веса и смещения для связей, по которым поступали сигналы, поддерживающие правильные ответы, увеличиваются. Нахождение оптимальных значений весов – итеративный процесс, состоящий из нескольких шагов.

Обратное распространение и доля ответственности за ошибку

Смысл обратного распространения – распределение ответственности за ошибку в процессе движения по сети в обратном направлении. Каждый скрытый блок, посылающий сигнал текущему блоку, несет долю ответственности за ошибку в каждом из нейронов, с которыми он связан.

Первый скрытый слой получает данные от вектора исходных признаков, а все последующие пользуются значениями активации нейронов предыдущего слоя. Мы должны правильно распределить ошибку между нейронами скрытых слоев. В алгоритме обратного распространения мы делим значения Δ_i пропорционально весам связей между скрытым и выходным блоками.

Для вычисления Δ_j применяется формула

$\Delta_j \leftarrow g'(\text{input_sum}_j)\Sigma_i W_{j,i}\Delta_i$.

Здесь мы берем каждый блок *i* в текущем слое и умножаем текущее значение ошибки Δ_i на вес входящей связи и на производную функцию активации. В результате получается доля ошибки, приходящаяся на блок предыдущего слоя, которая используется для обновления весов связей, входящих в этот слой. Этот алгоритм применяется слой за слоем, пока не будут обновлены все слои сети.

Величина шагов обучения, т. е. количество весов, изменяемых на каждой итерации, определяется *скоростью обучения*. Это задаваемый нами параметр (а не результат измерения качества сети). Мы еще вернемся к скорости обучения, когда будем обсуждать гиперпараметры в целом.

Обратное распространение и мини-пакетный стохастический градиентный спуск

В главе 1 мы узнали о мини-пакетном варианте СГС, когда модель обучается сразу на нескольких примерах, а не по одному примеру за раз. Мини-пакеты используются и при обратном распространении в нейронных сетях для повышения качества обучения.

За кулисами мы вычисляем усреднение градиента по всем примерам из мини-пакета. Точнее, на прямом проходе с помощью матричных операций обсчитываются все примеры для получения выходных откликов. А на обратном проходе для каждого слоя вычисляется средний градиент по слою. При такой реализации обратного распространения удается получить более точное приближение к градиенту и одновременно более полно задействовать возможности оборудования.

Вторая зима ИИ: начало 1990-х

В конце 1980-х и начале 1990-х годов вовсю рекламировались такие технологии, как экспертные системы и LISP-машины, но ни те, ни другие не оправдали ожиданий. Фонд Стратегической инициативы в области компьютерной техники отказался продолжать финансирование по истечении этого цикла. Программа разработки компьютеров пятого поколения закончилась провалом.

Функции активации

Функции активации используются для передачи выхода блоков одного слоя блокам следующего слоя (до выходного слоя включительно). Это скалярные функции скалярного аргумента, возвращающие значение активации нейрона. С их помощью нейроны скрытых слоев вносят нелинейность в модель нейронной сети. Многие функции активации принадлежат логистическому классу *сигмоидных* функций, график которых напоминает букву S. В семействе сигмоидных функций есть несколько вариаций, одна из которых называется просто сигмоидой. Рассмотрим некоторые полезные функции активации.

Линейная функция

Линейная функция (рис. 2.11) $f(x) = Wx$ описывает пропорциональное соотношение между зависимой и независимой величинами. На практике часто встречается тождественная функция ($W = 1$), означающая, что сигнал вообще не изменяется.

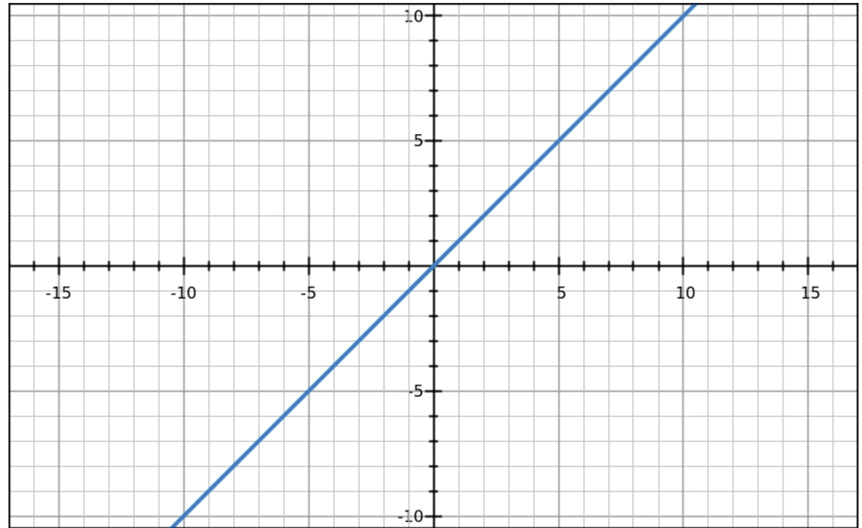


Рис. 2.11 ❖ Линейная функция активации

Такая функция используется во входном слое нейронной сети.

Сигмоида

Как и все логистические преобразования, сигмоида сглаживает выбросы (аномально большие значения), не удаляя их. Вертикальная прямая на рис. 2.12 является решающей границей.

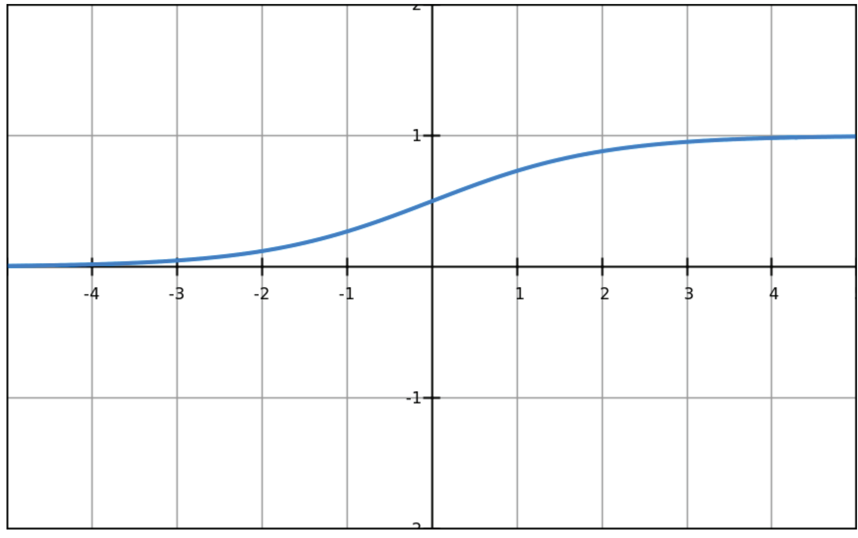


Рис. 2.12 ❖ Сигмоидная функция активации

Сигмоидная функция преобразует независимую величину, принимающую значения от минус до плюс бесконечности, в вероятность из диапазона от 0 до 1 и в большей части своей области определения очень близка к 0 или 1.

Интерпретация входа сигмоиды
Сигмоидная функция активации возвращает вероятность независимо для каждого класса.

Функция tanh

Функция tanh – это гиперболический тангенс (рис. 2.13). Если обычный тангенс равен отношению противолежащего катета прямоугольного треугольника к прилежащему, то tanh равен отношению гиперболического синуса к гиперболическому косинусу: $\tanh(x) = \sinh(x) / \cosh(x)$. В отличие от сигмоиды, tanh приводит аргумент к диапазону от –1 до 1. Достоинство этой функции заключается в том, что она проще при работе с отрицательными числами.

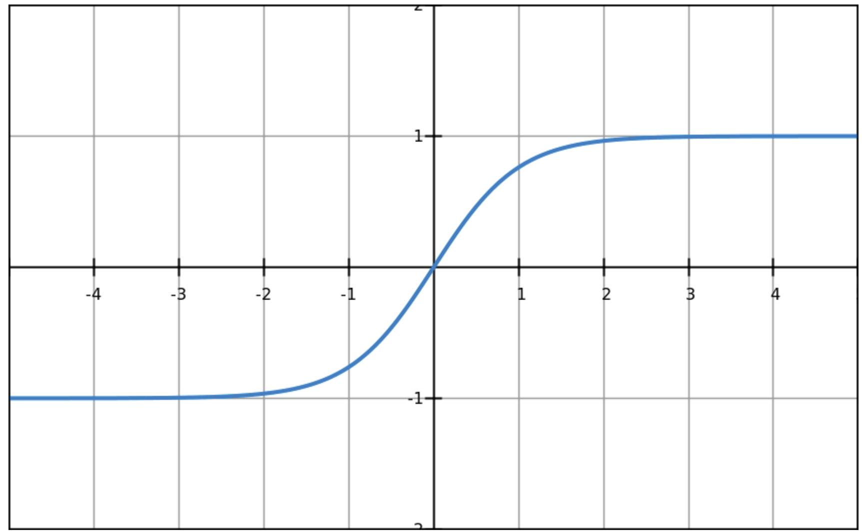


Рис. 2.13 ❖ Функция активации tanh

Функция hardtanh

Функция hardtanh просто обрезает все значения, выходящие за пределы нормированного диапазона. Если значение меньше –1, то оно заменяется на –1, а если больше 1, то на 1. Получается более устойчивая функция активации с ограниченной решающей границей.

Функция softmax

Softmax – обобщение логистической регрессии в том смысле, что она применима к непрерывным данным (а не для бинарной классификации) и может иметь несколько решающих границ. Она используется для мультиномиальной классификации и потому часто применяется в выходном слое классификатора.

Интерпретация выхода softmax

Функция активации softmax возвращает распределение вероятности непересекающихся классов.

Чтобы лучше понять идею выходного слоя с функцией softmax, рассмотрим два сценария. Если имеется многоклассовая задача классификации, но нас интересует только класс с наивысшей оценкой, то имеет смысл включить выходной softmax-слой с функцией `argmax()`, которая возвращает класс с наивысшей оценкой.

Отнесение к нескольким классам

Если мы хотим отнести пример к нескольким классам (например, «человек + автомобиль»), то выходной слой с функцией softmax не подойдет. Вместо этого следует взять слой с сигмоидной функцией активации, которая дает независимые вероятности каждого класса.

В случае, когда меток много (порядка нескольких тысяч), следует использовать *иерархическую функцию активации softmax*. Она располагает метки в узлах древовидной структуры, и в каждом узле обучается softmax-классификатор, определяющий, какую ветвь выбрать на следующем шаге.

Линейная ректификация

Функция линейной ректификации активирует блок, только если входной сигнал больше заданной величины. Функция описывается формулой $f(x) = max(0, x)$, ее график показан на рис. 2.14.

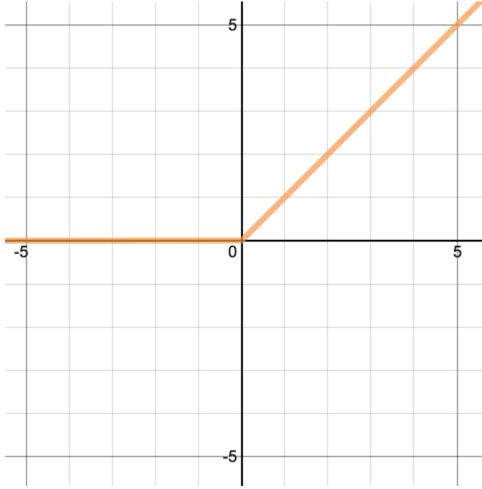


Рис. 2.14 ❖ Функция линейной ректификации

Блоки линейной ректификации (ReLU) находят широкое применение в современных глубоких сетях, поскольку хорошо работают во многих ситуациях. Так как производная ReLU равна нулю или константе, эта функция позволяет справиться с проблемой исчезающего и взрывного градиента. На практике сети с функцией активации ReLU обучаются лучше, чем сети с сигмоидными функциями активации.

Удивительная эффективность функции активации ReLU
В отличие от сигмоиды и tanh, функция активации ReLU не страдает от проблемы исчезающего градиента. Использование максимума в качестве функции активации может внести разреженность в выход слоя. Исследования показывают, что глубокие сети с блоками линейной ректификации хорошо обучаются даже без применения методов предобучения.

ReLU с утечкой

Блоки ReLU с утечкой призваны сгладить проблему «умирающего ReLU»[2]. При $x < 0$ ReLU с утечкой не обращается в нуль, а имеет небольшой отрицательный наклон (например, порядка 0.01). При использовании этого варианта ReLU иногда удавалось добиться успеха, но результаты не всегда стабильны. Функция определена следующим образом:

$$f(x) = \begin{cases} x, \text{ если } x > 0 \\ 0.01x \text{ в противном случае} \end{cases}$$

Функция softplus

Эта функция активации, изображенная на рис. 2.15, считается «гладким вариантом ReLU».

Как видим, функция softplus, определяемая уравнением $f(x) = \ln[1 + \exp(x)]$, по форме похожа на ReLU. Но, в отличие от ReLU, она всюду дифференцируема, и ее производная нигде не обращается в нуль.

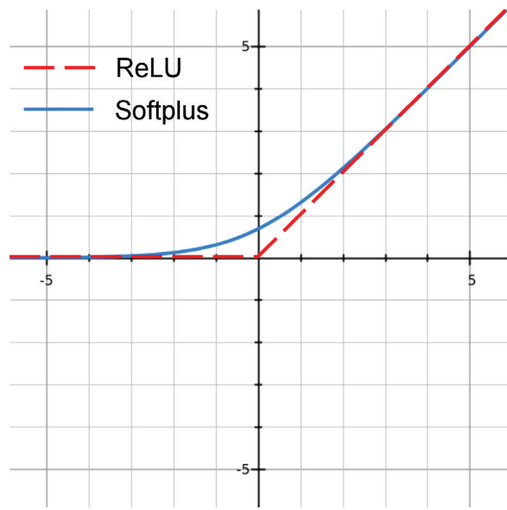


Рис. 2.15 ❖ Сравнение функций активации ReLU и softplus

[2] Karpathy Li. CS231n: Convolutional Neural Networks for Visual Recognition (Course Notes) // *http://cs231n.stanford.edu and http://cs231n.github.io*.

Функции потерь

Функция потерь количественно выражает близость нейронной сети к идеалу, ради которого она обучается. Идея проста. Вычисляется некая метрика, основанная на наблюдаемой ошибке предсказаний. Затем ошибки агрегируются по всему набору данных и усредняются, в результате чего остается единственное число, показывающее, насколько близко мы подошли к идеалу.

Поиск идеального состояния эквивалентен поиску параметров (весов и смещений), которые минимизируют «потерю» из-за ошибок. Тем самым функция потерь позволяет переформулировать задачу обучения нейронной сети как задачу оптимизации. В большинстве случаев найти оптимальные значения параметров аналитически невозможно, но очень часто их можно аппроксимировать с помощью таких алгоритмов оптимизации, как градиентный спуск. В следующем разделе приводится обзор наиболее употребительных функций потерь, иногда с указанием их истоков в машинном обучении.

Применяемые обозначения

В этом разделе используются следующие обозначения:

- рассмотрим набор данных для обучения нейронной сети. Обозначим N количество примеров в нем (входных данных вместе с соответствующими выходными метками);
- каждый пример состоит из уникального набора входных и выходных признаков. Обозначим P количество собранных входных признаков, а M – количество наблюдаемых выходных признаков;
- обозначим (X,Y) собранные входные и выходные данные. Отметим, что существует N пар, в которых вход принадлежит множеству, содержащему P значений, а выход – множеству, содержащему M значений. Будем обозначать i-ю пару в наборе данных (X_i, Y_i);
- будем обозначать выход нейронной сети $\hat{Y}$. Разумеется, $\hat{Y}$ – гипотеза, выдвинутая сетью о значении Y, и потому состоит из M признаков;
- обозначим $h(X_i) = \hat{Y}_i$ нейронную сеть, преобразующую вход X_i в выход $\hat{Y}_i$. Позже мы немного изменим это обозначение, чтобы подчеркнуть зависимость от весов и смещений;
- говоря о признаках, мы всегда имеем в виду матрицу, строки которой представляют примеры, а столбцы – уникальные признаки. Следовательно, $y_{i,j}$ обозначает j-ый признак, наблюдаемый в i-м примере;
- функцию потерь будем обозначать $L(W, b)$.

Обозначение функции потерь подсказывает, что ее значение зависит только от W и b – весов и смещений нейронной сети. Это очень важно. При наличии сети со всеми ее слоями, конфигурационными данными и всем прочим, обученной на некотором наборе данных, значение функции потерь зависит исключительно от состояния сети, определяемого весами и смещениями. Пошевелите их – и потери изменятся. Пошевелите их при заданных входах – и изменятся выходы.

Таким образом, обозначение $h(X) = \hat{Y}$ следует обусловить набором весов и смещений, поэтому будем писать $h_{w,b}(X) = \hat{Y}$. Теперь мы готовы заняться непосредственно функциями потерь.

Функции потерь для регрессии

В этом разделе мы рассмотрим функции потерь для регрессионных моделей.

Среднеквадратическая ошибка

При работе с регрессионной моделью, порождающей вещественное число, мы будем использовать квадратичную функцию потерь, как в случае обычного метода наименьших квадратов в задаче линейной регрессии. Предположим, что требуется предсказать всего один выходной признак ($M = 1$). Ошибка предсказания возводится в квадрат и усредняется по всем примерам:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}(\widehat{Y_i} - Y_i)^2.$$

А что, если M больше 1 и мы хотим предсказать несколько выходных признаков для заданного набора входных? В таком случае фактические и предсказанные значения, Y и $\hat{Y}$ соответственно, – это упорядоченные списки чисел, иначе говоря, векторы.

Замечание о функциях потерь

Функция потерь сводит различие между фактическими и предсказанными значениями к единственному числу, даже если они являются векторами.

Рассмотрим теперь другой вариант среднеквадратической функции потерь:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{M}\sum_{j=1}^{M}(\widehat{y_{ij}} - y_{ij})^2.$$

Если вы знакомы с линейной алгеброй, то легко узнаете во внутренней сумме квадрат евклидова расстояния. Вообще, среднеквадратическая ошибка (СКО) часто определяется в этих терминах. Отметим, что N, размер набора данных, и M, число предсказываемых сетью признаков, – константы. Так что их можно рассматривать просто как масштабные коэффициенты, которые можно учесть и другими способами (например, умножением на скорость обучения). Во многих случаях (в т. ч. в библиотеке DL4J) M опускается и для удобства производится еще деление на 2 (зачем это нужно, мы увидим позже, когда будем рассматривать градиент этой функции в алгоритме обратного распространения). Следующее уравнение содержит определение СКО, используемой в библиотеке DL4J для регрессии:

$$L(W, b) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\widehat{y_{ij}} - y_{ij})^2.$$

Является ли СКО выпуклой функцией потерь?

Технически СКО – выпуклая функция. Но при работе со скрытыми слоями нейронной сети она перестает быть таковой, потому что может существовать много наборов параметров, дающих одинаковую потерю.

Оптимизация СКО

Оптимизация СКО эквивалентна оптимизации среднего.

Другие функции потерь для регрессии

СКО широко используется, но она очень чувствительна к выбросам, и это нельзя упускать из виду при выборе функции потерь. Решая, в какие акции инвестиро-

вать, мы хотим принимать выбросы во внимание, а при покупке дома – вряд ли. В последнем случае нас больше интересует, какую сумму готово заплатить за него большинство людей, т. е. не столько среднее, сколько медиана.

Средняя абсолютная ошибка (САО, англ. MAE). Альтернатива СКО, вычисляемая по следующей формуле:

$$L(W,b) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}\left|\widehat{y_{ij}} - y_{ij}\right|.$$

Это просто усреднение абсолютной величины ошибки по всему набору данных.

Среднеквадратическая логарифмическая ошибка (СКЛО, англ. MSLE). Вычисляется по формуле:

$$L(W,b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\log\widehat{y_{ij}} - \log y_{ij})^2.$$

Средняя абсолютная ошибка в процентах (САОП, англ. MAPE). Вычисляется по формуле:

$$L(W,b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{100\times\left|\widehat{y_{ij}} - y_{ij}\right|}{y_{ij}}.$$

Обсуждение функций потерь для регрессии

Ни один из вышеупомянутых вариантов не имеет абсолютного преимущества во всех возможных ситуациях. СКО применяется очень широко, и в большинстве случаев на нее можно положиться. То же справедливо и в отношении САО. Функции потерь СКЛО и САОП стоит рассматривать, если сеть предсказывает значения, находящиеся в сильно различающихся диапазонах. Предположим, что сеть должна предсказать две выходные величины: одну – в диапазоне [0, 10], другую – в диапазоне [0, 100]. В таком случае САО и СКО будут штрафовать второе предсказание сильнее, чем первое. САОП оперирует относительной ошибкой, поэтому для нее ширина диапазона не играет роли. СКЛО «сплющивает» выходной диапазон, так что 10 и 100 преобразуются соответственно в 1 и 2 (если логарифм берется по основанию 10).

Типичное применение регрессии в нейронных сетях

Хотя СКЛО и САОП пригодны для работы с широкими диапазонами, обычно в нейронных сетях принято нормировать входные данные на подходящий диапазон и использовать СКО или САО для оптимизации среднего или медианы.

Функции потерь для классификации

Можно построить нейронную сеть для распределения данных по нескольким категориям, например мошенническая или честная операция. Но при таком их применении обычно возвращаются вероятности классов (30% за мошенническую, 70% за честную). Для этого необходимы другие функции потерь.

Кусочно-линейная функция потерь

Кусочно-линейная функция (hinge loss) применяется в основном тогда, когда сеть оптимизируется для безусловной классификации. Например, 0 = честная опера-

ция, 1 = мошенническая (бинарный классификатор). Выбор чисел 0 и 1 произволен, с тем же успехом можно было бы взять –1 и 1. Кусочно-линейная функция потерь часто встречается в классе моделей классификации с максимальным зазором (например, в методе опорных векторов, отдаленном родственнике нейронных сетей).

Ниже приведено уравнение кусочно-линейной функции потерь для случая, когда данным нужно присвоить класс –1 или 1:

$$L(W,b)=\frac{1}{N}\sum_{i=1}^{N}\max(0,1-y_{ij}\times\hat{y}_{ij}).$$

Чаще всего кусочно-линейная функция потерь применяется для бинарной классификации. Существуют ее обобщения на случай многоклассовой классификации («один против всех», «один против одного»), но мы их рассматривать не будем.

Кусочно-линейная функция потерь – выпуклая функция
Отметим, что кусочно-линейная функция потерь, как и СКО, – выпуклая функция.

Логистическая функция потерь

Логистическая функция потерь применяется, когда интерес представляет не безусловная классификация, а вероятностная. Примером может служить привлечение внимания человека-оператора к потенциальной мошеннической операции или предсказание «вероятности щелчка по рекламному объявлению», которую затем можно связать с платой за его размещение.

Предсказание вероятности означает, что сеть должна породить число от 0 до 1. Кроме того, сумма вероятностей взаимно исключающих исходов должна быть равна 1. Поэтому важно, чтобы последним слоем нейронной сети, используемой для классификации, был слой softmax. Отметим, что сигмоидная функция активации также возвращает значения от 0 до 1. Но она неприменима в случае, когда выходы взаимно исключающие, поскольку не моделирует зависимости между выходными значениями.

Гарантировав, что наша нейронная сеть будет порождать корректные вероятности классов, мы можем заняться самой функцией потерь и решить, что следует оптимизировать. Мы хотим максимизировать так называемое «максимальное правдоподобие». Иными словами, требуется максимизировать предсказанную вероятность правильного класса И сделать это для каждого имеющегося примера.

Рассмотрим сеть, предсказывающую вероятности двух классов, например мошеннической и честной операций. В приведенных выше обозначениях вероятности 1 и 0 для заданного входа X_i при заданных весах W и смещениях b будут обозначаться соответственно:

$$P(y_i = 1|X_i; \mathbf{W}, \mathbf{b}) = h_{\mathbf{W},\mathbf{b}}(X_i);$$
$$P(y_i = 0|X_i; \mathbf{W}, \mathbf{b}) = 1 - h_{\mathbf{W},\mathbf{b}}(X_i).$$

Эти выражения можно объединить в одно:

$$P(y_i \mid X_i; \mathbf{W}, \mathbf{b}) = (h_{\mathbf{W},\mathbf{b}}(X_i))^{y_i} \times (1 - h_{\mathbf{W},\mathbf{b}}(X_i))^{1-y_i}.$$

Слово «И» в словесном определении максимального правдоподобия в контексте вероятностей должно вызвать рефлекторную реакцию. «И» для всех имеющихся примеров означает, что вероятности перемножаются:

$$L(W,b) = \prod_{i=1}^{N} \hat{y}_i^{y_i} \times (1-\hat{y}_i)^{1-y_i}.$$

Теперь перейдем к отрицательному логарифмическому правдоподобию.

Отрицательное логарифмическое правдоподобие. При работе с произведением вероятностей удобнее взять логарифм, тогда произведение преобразуется в сумму логарифмов вероятностей.

Отрицательное логарифмическое правдоподобие и максимизация вероятности
Логарифм – монотонно возрастающая функция. Поэтому минимизация отрицательного логарифмического правдоподобия эквивалентна максимизации вероятности.

Дополнительно мы добавим знак минус, чтобы формула описывала «потерю». Таким образом, получается следующая функция, которую принято называть отрицательным логарифмическим правдоподобием:

$$L(W,b) = -\sum_{i=1}^{N} y_i \times \log \hat{y}_i + (1-y_i) \times \log(1-\hat{y}_i).$$

Обобщение с двух классов на M дает уравнение логистической функции потерь:

$$L(W,b) = -\sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} \times \log \hat{y}_{i,j}.$$

Математически это эквивалентно перекрестной энтропии двух распределений вероятности – в данном случае предсказанного и наблюдаемого при одном и том же критерии. Мы поговорим об этом несколько подробнее в следующем разделе.

Унифицированный взгляд на функции потерь
Мы хотим минимизировать функцию потерь, для чего максимизируем вероятность, для чего минимизируем отрицательное логарифмическое правдоподобие. Не слишком ли запутанно? Пожалуй, да.
Перекрестная энтропия берет начало в теории информации, а отрицательное логарифмическое правдоподобие – в статистическом моделировании. Математически оба метода эквивалентны, поэтому, какой использовать, не важно, только путаницу создает.

Функции потерь для реконструкции

Идея *реконструкции* проста. Нейронная сеть обучается максимально точно воспроизводить свой вход. Но почему бы просто не запомнить весь набор данных? Вся хитрость в том, чтобы настроить процесс обучения так, чтобы сеть обучилась характерным особенностям, присутствующим в наборе данных.

При одном из подходов количество параметров сети ограничивается таким образом, чтобы сеть была вынуждена сжимать данные, а затем воссоздавать их. Другой популярный подход – исказить входные данные бессмысленным «шумом» и обучить сеть игнорировать шум и определять данные. В качестве примеров упомянем ограниченные машины Больцмана и автокодировщики. Во всех таких сетях используются функции потерь, уходящие корнями в теорию информации.

Следующая формула описывает расхождение Кульбака-Лейблера (KL-расхождение):

$$D_{KL}(Y \,||\, \hat{Y}) = -\sum_{i=1}^{N} Y_i \times \log\left(\frac{Y_i}{\hat{Y}_i}\right).$$

Хотя мы только что кратко упомянули перекрестную энтропию и обосновали переход к логарифмам вероятностей, чтобы преобразовать произведение в сумму, мы не сказали, что эта операция переносит нас прямиком в вотчину теории информации, где царит понятие энтропии.

Различные подходы
Несмотря на математическую эквивалентность отрицательного логарифмического правдоподобия и перекрестной энтропии, они берут начало в совершенно разных математических теориях.

Гиперпараметры

В машинном обучении имеются как параметры модели, так и настраиваемые параметры, позволяющие сделать процесс обучения быстрее и лучше. Последние называются *гиперпараметрами*, их цель – управление функциями оптимизации и выбором модели в ходе обучения согласно выбранному алгоритму. В DL4J алгоритмы оптимизации называются корректорами (updater), потому что действия, выполняемые алгоритмом в пространстве весов для минимизации ошибки, и есть коррекция. Настройка гиперпараметров призвана избежать недообученности и переобученности сети, обеспечив вместе с тем максимально быстрое обучение структуре данных.

Скорость обучения

Скорость обучения говорит, как сильно следует корректировать параметры в процессе оптимизации, чтобы минимизировать ошибку в предсказаниях нейронной сети. Это коэффициент, управляющий величиной шагов коррекции вектора параметров x при перемещении в пространстве функций потерь.

В процессе обратного распространения мы умножаем градиент ошибки на скорость обучения, а затем вычисляем новый вес связи, прибавляя произведение к весу, полученному на предыдущей итерации. Скорость обучения определяет, какую часть градиента мы хотим использовать на очередном шаге алгоритма. Если ошибка велика, а градиент крутой, то при умножении на скорость обучения получится большой шаг. Когда мы приближаемся к области минимальной ошибки и плоского градиента, величина шага уменьшается.

При большой скорости обучения (скажем, 1) параметры меняются скачкообразно, а при малой (например, 0.00001) – в час по чайной ложке. Большие скачки могут сэкономить время на начальном этапе, но приведут к катастрофе, если мы проскочим минимум. Тогда алгоритм начнет буксовать на месте, перепрыгивая с одной стороны минимума на другую, и никогда не остановится.

С другой стороны, при небольшой скорости обучения мы в конце концов достигнем минимума (возможно, локального, а не глобального), но это может потребовать очень много времени и усложнить и без того сложные вычисления. Когда обучение сети на большом наборе данных занимает несколько недель, время имеет значение. Если вы не можете ждать результатов лишнюю неделю, то выберите умеренную скорость обучения (например, 0.1) и поэкспериментируйте, стремясь одновременно добиться наилучшей скорости и верности. Скорость обучения необязательно задавать раз и навсегда, мы рассмотрим также методы, в которых она изменяется динамически, чтобы взять лучшее от обоих миров.

Регуляризация

Регуляризация помогает справляться с чрезмерно большим числом параметров с помощью различных методов уменьшения их количества со временем.

Контроль переобучения в машинном обучении
Основная цель регуляризации – контроль переобучения.

В математических формулах регуляризация представляется коэффициентом лямбда, который определяет компромисс между поиском хорошей аппроксимации и присвоением низкого веса некоторым признакам, когда их общее число возрастает.

L1- и L2-регуляризации предотвращают переобучение за счет уменьшения некоторых весов. Чем меньше веса, тем проще гипотезы, а более простые гипотезы лучше обобщаются. Полиномы высокой степени от признаков с нерегуляризованными весами часто дают переобученную модель.

По мере роста обучающего набора эффект регуляризации снижается, а число параметров возрастает. Но в этом нет ничего страшного, потому что основной причиной переобучения является избыток признаков по сравнению с количеством обучающих примеров. Увеличение набора само по себе является лучшим регуляризатором.

Импульс

Импульс помогает алгоритму обучения выбираться из областей пространства поиска, в которых иначе он застрял бы навечно. Благодаря ему корректор находит в ландшафте ошибок долины, ведущие к минимумам. По отношению к скорости обучения импульс – то же самое, что скорость обучения по отношению к весам, он позволяет находить более качественные модели. В последующих главах мы не раз увидим импульс в действии.

Разреженность

Гиперпараметр «разреженность» позволяет учесть тот факт, что в некоторых входных примерах релевантными являются лишь немногие признаки. Пусть, например, сеть должна классифицировать миллион изображений. Каждое изображение представлено ограниченным количеством признаков. Но чтобы эффективно классифицировать миллионы изображений, сеть должна уметь распознавать гораздо больше признаков, многие из которых в большинстве примеров отсутствуют. В качестве примера приведем фотографии морских ежей, на которых вообще не бывает ни носа, ни кормы, и фотографии подводных лодок, на которых признаки носа и кормы отсутствуют.

Признаки, выделяющие морских ежей, немногочисленны и теряются в обширности слоев нейронной сети. Это проблема, потому что разреженные признаки могут ограничить число активируемых нейронов и снизить способность сети к обучению. Противовесом разреженности служат смещения, которые заставляют нейроны активироваться, так что число активаций колеблется вокруг среднего значения, не давая сети застрять.

Глава 3

Основания глубоких сетей

> Ну, а здесь, знаешь ли, приходится бежать со всех ног, чтобы только остаться на том же месте! Если же хочешь попасть в другое место, тогда нужно бежать по меньшей мере вдвое быстрее!
>
> – *Красная Королева, «Алиса в Зазеркалье»*[1]

Определение глубокого обучения

В главе 2 мы познакомились с основами машинного обучения и нейронных сетей. Теперь мы воспользуемся этими знаниями, чтобы изложить базовые понятия глубоких сетей. Это поможет понять, что происходит, когда в главе 4 мы будем изучать различные архитектуры, а в главе 5 писать практические примеры. Для начала переформулируем определения глубокого обучения и глубоких сетей.

Что такое глубокое обучение?

Возвращаясь к определению глубокого обучения, данному в главе 1, мы можем выделить следующие аспекты, отличающие сети глубокого обучения от «канонических» многослойных сетей прямого распространения:

- больше нейронов, чем в предшествующих сетях;
- более сложные способы соединения слоев;
- «кембрийский взрыв» вычислительных мощностей, доступных для обучения;
- автоматическое выделение признаков.

Говоря «больше нейронов», мы имеем в виду происходившее с годами увеличение количества нейронов с целью выразить более сложные модели. Сами слои также эволюционировали: если раньше это были полносвязные слои в многослойных сетях, то теперь появились локально связные слои с небольшим количеством связей между слоями в сверточных нейронных сетях и рекуррентные связи нейрона с самим собой в рекуррентных нейронных сетях (в дополнение к связям с нейронами предыдущего слоя).

Чем больше связей, тем больше параметров, подлежащих оптимизации, и это стало возможным только благодаря взрывообразному росту вычислительных мощностей, происходившему в течение последних 20 лет. Эти достижения заложили фундамент для построения нейронных сетей следующего поколения,

---

[1] Перевод Н. Демуровой.

способных самостоятельно выделять признаки с большим успехом, чем раньше. А это дало возможность использовать глубокие сети для моделирования более сложных задач (например, для распознавания изображений). Поскольку отраслевые требования постоянно изменяются, возможности нейронных сетей также не стоят месте. В точном соответствии со словами Красной Королевы, приведенными в эпиграфе к этой главе.

Определение глубоких сетей

Чтобы придать конкретики нашему определению глубокого обучения, определим четыре основные архитектуры глубоких сетей:

- сети, предобученные без учителя;
- сверточные нейронные сети;
- рекуррентные нейронные сети;
- рекурсивные нейронные сети.

В области нейронных сетей ведутся активные исследования, но в этой книге мы ограничимся только этими четырьмя архитектурами, которые выкристаллизовались за прошедшие 20 лет. Продолжим начатый в главе 1 краткий экскурс в историю многослойных сетей прямого распространения.

Глубокое обучение с подкреплением

Обучение с подкреплением определено в книге Саттона[2] следующим образом:

> Чтобы определить обучение с подкреплением, нужно охарактеризовать не методы обучения, а задачу обучения.

Далее говорится, что любой метод, пригодный для решения этой задачи, можно рассматривать как метод обучения с подкреплением. В обучении с подкреплением мы не сообщаем обучаемому агенту, какие действия он должен предпринять, а позволяем ему экспериментально выяснить, какие действия приносят наибольшее вознаграждение.

В самом начале обучения у агента нет обученной модели окружающей среды, а синонимом вознаграждения, к которому стремится агент, является функция полезности. Обучающая система предлагает агенту входные данные и вознаграждает его, если результат цикла (или *раунда*) имитационной модели (или *игры*) оказался положительным. Часто бывает, что от действий агента зависит не только непосредственное вознаграждение, но и вознаграждение в будущем. Механизм проб и ошибок и отложенного вознаграждения – ключевые особенности обучения с подкреплением.

Глубокое обучение с подкреплением – это вариант обучения с подкреплением, при котором нейронная сеть используется в качестве универсального аппроксиматора функции. Недостаток этого подхода – в том, что на поведение нейронной сети невозможно наложить никаких ограничений, так что доказательство сходимости больше не проходит. Но, несмотря на это, нейронные сети в роли аппроксиматоров функций дают на удивление хорошие результаты.

В 2013 году группа DeepMind опубликовала текст доклада на семинаре по глубокому обучению NIPS 2013 Deep Learning Workshop, посвященного обучению машины

[2] https://www.ozon.ru/context/detail/id/7107485/.

играм ATARI с помощью глубокого Q-обучения[3]. Авторы использовали стандартный алгоритм (Q-обучение с аппроксимацией функции). В качестве аппроксиматора использовалась сверточная нейронная сеть. Был продемонстрирован агент, способный играть в игры для компьютера Atari 2600, обученный на экранных пикселях.
Агент получает вознаграждение, если его действия привели к положительному результату в игре. В некоторые игры алгоритм научился играть лучше человека.
В процессе работы над этой книгой популярность глубокого обучения с подкреплением возросла, и мы надеемся включить эту тему в следующее издание. А пока адресуем вас к примеру в приложении В.

Эволюция и возрождение

В главе 2 мы остановились на том, что нейронные сети вступили в «период зимы», когда в середине 1980-х годов ИИ не смог выполнить данных обещаний. Как много раз случалось и раньше, когда многообещающая технология скатывалась в яму избавления от иллюзий (рис. 3.1), нашлось немало ученых, продолжавших важную работу в области нейронных сетей.

Рис. 3.1 ❖ Яма избавления от иллюзий
(https://en.wikipedia.org/wiki/Hype_cycle)

Важным достижением стала работа Яна Лекуна из AT&T Bell Labs по оптическому распознаванию символов[4]. Его лаборатория занималась задачей распознавания чеков для сектора финансовых услуг. По ходу дела группа Лекуна разработала концепцию биокомпьютерной модели распознавания изображений, которую мы сегодня знаем под названием «сверточная нейронная сеть» (СНС). Впоследствии это привело к созданию тестового набора рукописных цифр MNIST[5] (мы еще вер-

[3] Mnih et al., 2013. Human-level control through deep reinforcement learning // https://www.nature.com/articles/nature14236.

[4] LeCun et al., 1998. Gradient-based learning applied to document recognition // http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

[5] http://yann.lecun.com/exdb/mnist/.

немся к нему ниже в этой главе) и последовательности вех, отмечающих все возрастающую верность моделей глубокого обучения.

Улучшенная разметка данных
Еще одним фактором эволюции и успеха глубоких сетей стало создание более крупных и лучше размеченных наборов данных, таких как MNIST и ImageNet[6].

В конце 1980-х и начале 1990-х годов усилиями Сеппа Хохрайтера и других ученых был достигнут значительный прогресс в моделировании последовательных данных с помощью рекуррентных нейронных сетей. Во второй половине 1990-х ученое сообщество создало улучшенные варианты искусственного нейрона (например, долгая краткосрочная память [Long Short-Term Memory – LSTM] и память с вентилем забывания). В тиши исследовательских лабораторий по всему миру нейронные сети стали возрождаться к жизни.

На протяжении 2000-х годов достижения исследователей стали воплощаться в промышленные продукты[7], например:

- беспилотные автомобили;
- Google Translate[8];
- Amazon Echo;
- AlphaGo[9].

В беспилотных автомобилях, принимавших участие в конкурсе 2006 Darpa Grand Challenge, использовались многочисленные технологии, помимо глубокого обучения. Победители (команды Стэнфордского университета и университета Карнеги-Меллона) сумели воспользоваться серьезными достижениями в области обработки изображений.

Достижения в компьютерном зрении
В 2012 году Алекс Крижевский, Илья Суцкевер и Джеффри Хинтон разработали «большую глубокую сверточную нейронную сеть», победившую на конкурсе 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge).
Сеть AlexNet[10] подавалась как достижение в области компьютерного зрения, и некоторые считают ее прорывом в применении глубокого обучения в этой сфере. Но на самом деле это масштабированный (более глубокий и широкий) вариант СНС, известных с 1990-х годов. Прогресс последних лет в компьютерном зрении связан не столько с новыми алгоритмами, сколько с улучшенными вычислительными средствами, данными и инфраструктурой.

Благодаря усовершенствованному анализу изображений подсистема планирования в автомобиле может лучше выбирать маршрут движения по неизвестной местности и более надежно избегать препятствий. Другие достижения глубокого обучения позволяют модели правильнее переводить и распознавать звуковые данные, что нашло применение в линейках продуктов Google Translate и Amazon Echo. Совсем недавно мы видели, как машина играет в сложную игру на уровне

---

[6] http://image-net.org/.
[7] https://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html.
[8] https://translate.google.com/.
[9] https://deepmind.com/research/alphago/.
[10] Krizhevsky, Sutskever and Hinton, 2012. ImageNet Classification with Deep Convolutional Neural Networks.

мастера, – когда система AlphaGo победила Ли Седоля – профессионального игрока в го, имеющего девятый дан.

Прогресс машинного обучения не всегда виден. Публичное признание часто является результатом многих направлений работы, демонстрируемым в широко разрекламированных мероприятиях, таких как конкурс Darpa Grand Challenge или победа программы Watson над Кеном Дженнигсом в телевизионной игре Jeopardy[11]. Тем временем за кулисами идет медленная, но неуклонная работа. Ее плоды, как и смену времен года, мы замечаем, только когда они перешагивают порог нашего дома.

В недалеком будущем мы станем свидетелями того, как глубокое обучение находит новые необычные применения. Сочетание скрытого интеллекта (например, рекомендации или распознавание голоса) с прагматичной конструкцией сделает эти приложения полезными в повседневной жизни. Но вот чего мы, скорее всего, не увидим (по крайней мере, в ближайшем будущем), так это вышедших из-под контроля злобных искусственных агентов, которые выбрасывают нас из воздушного шлюза в самое неподходящее время (вспомните о компьютере ЭАЛ 9000 из «Космической одиссеи 2001 года»).

ЭАЛ 9000

ЭАЛ 9000 (в оригинале HAL 9000) – придуманный компьютер, управляющий космическим кораблем «Дискавери» в романе Артура Кларка «Космическая одиссея 2001 года». ЭАЛ означает «эвристически программированный алгоритмический компьютер». В фильме ЭАЛ выглядит как объектив камеры с горящей красной точкой, а для доступа к нему применяется интерфейс на базе системы распознавания речи. В фильме ЭАЛ приходит к выводу, что для успешного завершения задания он должен убить команду «Дискавери».
Дэйв: Открой шлюз, ЭАЛ.
ЭАЛ: Извини, Дэйв. Боюсь, что не могу это сделать.

Глубокое обучение продолжает двигать прогресс во многих областях и при решении многих важнейших задач машинного обучения. Вот лишь несколько крупных успехов, достигнутых в глубоком обучении в последние годы:

- синтез речи (Fan et al., Microsoft, Interspeech 2014);
- распознавание языка (Gonzalez-Dominguez et al., Google, Interspeech 2014);
- распознавание речи со словарем большого размера (Sak et al., Google, Interspeech 2014);
- предсказание интонации (Fernandez et al., IBM, Interspeech 2014);
- распознавание речи со словарем среднего размера (Geiger et al., Interspeech 2014);
- перевод с английского на французский (Sutskever et al., Google, NIPS 2014);
- обнаружение начала звукового сигнала (Marchi et al., ICASSP 2014);
- классификация социальных сигналов (Brueckner & Schulter, ICASSP 2014);
- распознавание арабских рукописных текстов (Bluche et al., DAS 2014);
- распознавание фонем из набора данных TIMIT (Graves et al., ICASSP 2013);

[11] Аналог «Кто хочет стать миллионером». – *Прим. перев.*

- оптическое распознавание символов (Breuel et al., ICDAR 2013);
- генерация подписей к изображениям (Vinyals et al., Google, 2014);
- порождение текстового описания видеоизображения (Donahue et al., 2014);
- синтаксический разбор естественного языка (Vinyals et al., Google, 2014);
- фотореалистичные говорящие головы (Soong and Wang, Microsoft, 2014).

На основе этих достижений мы легко можем спрогнозировать, что в следующем десятилетии глубокое обучение проникнет во многие приложения. Вот несколько наиболее впечатляющих демонстраций прикладного глубокого обучения:

- автоматическое повышение резкости изображения (https://github.com/alexjc/neural-enhance);
- автоматическое увеличение изображения (https://github.com/nagadomi/waifu2x);
- WaveNet: генерация речи, имитирующей голос другого человека (https://deepmind.com/blog/wavenet-generative-model-raw-audio/);
- WaveNet: генерация правдоподобно звучащей классической музыки;
- реконструкция речи по видео без звука (http://www.vision.huji.ac.il/vid-2speech/);
- генерация шрифтов (https://erikbern.com/2016/01/21/analyzing-50k-fonts-using-deep-neural-networks.html);
- автоматическое заполнение отсутствующих в изображении участков (http://bamos.github.io/2016/08/09/deep-completion/);
- автоматическое подписывание изображений (http://cs.stanford.edu/people/karpathy/deepimagesent/, см. также https://github.com/karpathy/neuraltalk2);
- превращение любительских рисунков в картины в стиле известных художников (https://github.com/alexjc/neural-doodle).

Мы, как правило, не осознаем всю прелесть важных коммерческих приложений, пока их не подсунут нам прямо под нос. Быть в курсе прогресса в архитектурах глубоких сетей важно, чтобы понимать, как развиваются прикладные идеи.

Прогресс в области сетевых архитектур

По мере того как с развитием науки фокус исследований переместился с многоуровневых сетей прямого распространения к новым архитектурам типа СНС и рекуррентных нейронных сетей, изменениям подверглись организация слоев, конструирование нейронов и способы соединения слоев. Сетевые архитектуры эволюционировали с учетом конкретных типов входных данных.

Новые типы слоев. С появлением новых типов архитектур увеличилось и разнообразие слоев. Глубокие сети доверия (ГСД, англ. DBN) доказали успешность, когда в качестве слоев для предобучения использовались ограниченные машины Больцмана (ОМБ, англ. RBM). В СНС использовались новые типы функций активации в слоях и изменился способ соединения слоев (вместо полносвязных стали применяться локально связные слои). В рекуррентных нейронных сетях применялись связи, которые позволяют лучше моделировать время в данных, представляющих собой временные ряды.

Новые типы нейронов. Прогресс в создании новых типов нейронов (блоков) особенно заметен в рекуррентных нейронных сетях, построенных на базе LSTM-сетей. Именно здесь появились такие блоки, как ячейка LSTM-памяти и вентильные рекуррентные блоки (Gated Recurrent Units – GRU).

Гибридные архитектуры. Если речь зашла о зависимости архитектуры от типа входных данных, то нельзя не отметить появления гибридных архитектур для данных, в которых одновременно присутствуют временной аспект и изображение. Например, благодаря объединению СНС и рекуррентных нейронных сетей в одну гибридную сеть удалось классифицировать объекты в видеоряде. В некоторых случаях гибридные архитектуры нейронных сетей позволяют взять лучшее из обоих миров.

От конструирования признаков к автоматическому обучению признакам

Хотя глубокие сети и обогатились новыми видами внутренних блоков и слоев, в конечном итоге они увенчиваются дискриминантным классификатором, на вход которого подаются сконструированные признаки. Автоматическое выделение признаков – то общее, что роднит различные архитектуры. В каждой архитектуре признаки конструируются по-своему, в соответствии с ее специализацией на определенных типах данных. Ян Лекун в своем описании глубокого обучения выразил эту мысль, упомянув «машины, которые учатся представлять мир».

Джеффри Хинтон упоминает эту тему в статье о ГСД, где объясняет, что ограниченные машины Больцмана используются для разложения данных на признаки высшего порядка[12].

Классификация ГСД

В этой книге мы относим глубокие сети доверия (и автокодировщики) к классу глубоких сетей, предобученных без учителя.

В рамках классификации изображений мы можем привести в качестве примеров определения лиц в кадре. В исходных изображениях изменяются ориентация лица, освещение и положение ключевых признаков. К числу ключевых признаков, обычно ассоциируемых с лицом, относятся границы лица, границы отдельных его частей, например глаз и носа, а также различные более тонкие признаки, присутствующие не всегда, например ямочки.

Конструирование признаков. Ручной отбор признаков долгое время был отличительной чертой машинного обучения. Победители состязаний по машинному обучению зачастую тщательно изучали набор данных и применяли различные хитроумные трюки, чтобы максимально упростить работу алгоритму обучения. Набор данных обычно представляет собой табличные текстовые данные, и мы можем применять к отдельным столбцам знания о предметной области, чтобы сделать процесс выделения признаков более осмысленным.

Вспомнив, как в главе 1 моделировались входные данные в виде матрицы A в уравнении $Ax = b$, мы поймем, что требуется вручную поместить значения, выделенные из данных, в столбцы A. В результате получаются модели с высокой степенью верности, но для их создания нужно много времени и большой опыт. С точки зрения представления знаний, это можно сравнить с чтением плохо и хорошо написанной книги. Первую мы будем читать дольше второй и потратим больше времени на извлечение той же информации.

[12] Hinton, Osindero and Teh, 2006. A Fast Learning Algorithm for Deep Belief Nets.

Классификация изображений – интересный пример, потому что ручное выделение признаков изображения труднее, чем создание признаков по табличным данным. Содержащаяся в изображении информация не находится в фиксированном столбце, и на нее могут оказывать влияние такие факторы, как освещение, угол съемки и прочее. Для выделения признаков изображения нужен новый подход, и это отчасти стало стимулом для эволюции СНС.

Обучение признакам. Вернемся к примеру с определением лица: нос может находиться в любом множестве пикселей – в отличие от баланса счета, который всегда находится в определенном столбце таблицы. В случае СНС мы обучаем сеть понимать, где проходят границы носа, а затем определять общую форму носа по низкоуровневым признакам «границы носа». Нижние слои сети выделяют эти признаки носа и передают их последующим слоям в виде *карт признаков*.

Эти мелкие карты признаков в конечном итоге объединяются в признак «лицо» на верхних слоях СНС. Это дает СНС возможность подойти к задаче, которую много раз пытались решить и раньше («Является ли это лицом?»), но поставить вопрос в более простой форме, так что для получения верного ответа требуется куда меньше усилий.

Автоматизированное обучение признакам на сложных данных

Автоматическое создание признаков высшего порядка на основе необработанных исходных данных с целью упростить классификацию (или регрессию) – отличительный признак глубокого обучения.

Читая эту книгу, вы сможете лучше понять, как выбрать архитектуру глубокой сети, наиболее точно соответствующую типу входных данных, и как настроить ее для создания оптимальной модели набора данных.

Порождающее моделирование

Идея *порождающего моделирования* не нова, но глубокие сети позволили вывести его на такой уровень, где оно уже соперничает с творческими способностями человека. Мы каждый день встречаемся с его проявлениями: от создания картин и музыкальных произведений до написания обзоров сортов пива. Вот лишь несколько недавних примеров порождающего моделирования в действии:

- инцепционизм;
- моделирование манеры конкретного художника;
- порождающие состязательные сети;
- рекуррентные нейронные сети.

Кратко рассмотрим каждый из этих пунктов.

Инцепционизм. Инцепционизм[13] – это техника, при которой порядок слоев обученной сверточной сети меняется на противоположный, и ей на вход подается изображение с априорным ограничением. Изображения последовательно модифицируются способом, который можно было бы назвать «галлюцинаторным». Если на входе имелось изображение неба, то на выходе мы можем увидеть рыбьи морды на фоне облаков. Это направление исследований, начатое компанией Google, показало, что дискриминантная модель нейронной сети содержит достаточно информации для порождения изображений.

[13] https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.

Моделирование манеры конкретного художника. Было показано, что сверточные сети можно обучить манере конкретных художников, а затем генерировать с их помощью изображения с произвольных фотографий в такой же манере. На рис. 3.2 (он уже встречался в главе 1) приведены потрясающие результаты. Представьте только свою семейную фотографию, написанную Винсентом ван Гогом. (К моменту, когда эта книга выйдет из печати, наверное, в Snapchat уже будет такой фильтр, так что ждать придется не так уж долго.)

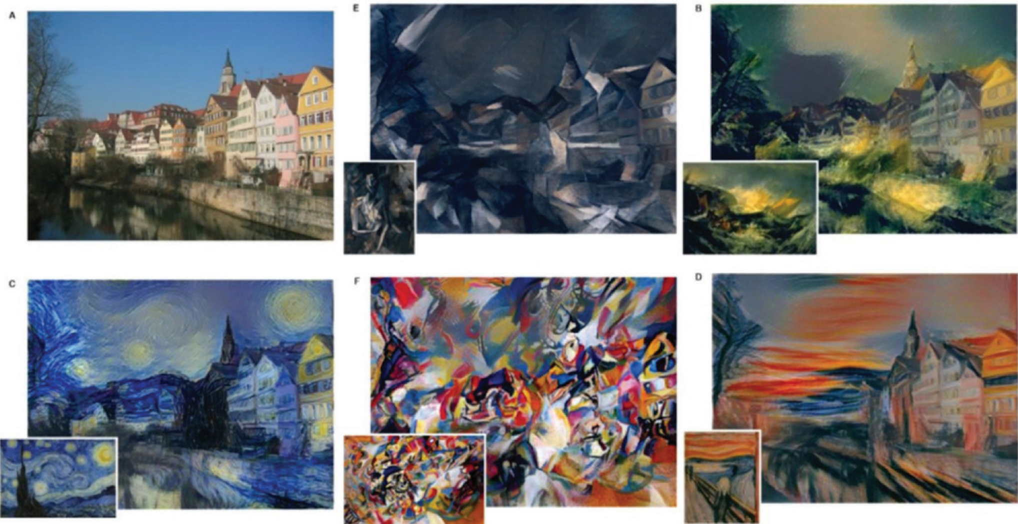


Рис. 3.2 ❖ Стилизованные изображения, взятые из работы Gatys et al., 2015

В 2015 году была опубликована работа Gatys et al. «A Neural Algorithm of Artistic Style»[14], авторы которой разделяют манеру письма и содержание картины. СНС кодирует манеру художника в параметрах сети, которая впоследствии может быть применена к любому изображению, чтобы представить его в той же манере.

Порождающие состязательные сети (ПСС). Результат работы ПСС[15] проще всего описать как синтез новых изображений путем моделирования распределения входных данных, предъявленных сети. Более подробно мы будем обсуждать ПСС в главе 4.

Рекуррентные нейронные сети (РНС). Было продемонстрировано, что РНС могут моделировать последовательности символов и порождать новые последовательности, обладающие видимой логической связностью. В главе 5 мы рассмотрим пример РНС, которая порождает стихотворные строки в стиле Шекспира после обучения на всех его произведениях.

Еще одно интересное применение РНС описано в работе Липтона и Элкана, где сеть моделирует имена собственные типа «Coors Light»[16] и другие образчики

---

[14] Gatys et al., 2015. A Neural Algorithm of Artistic Style // https://arxiv.org/abs/1508.06576.

[15] Goodfellow et al., 2014. Generative Adversarial Networks // https://arxiv.org/abs/1406.2661.

[16] Марка американского пива. – *Прим. перев.*

пивного жаргона. Можно заказать, описание какого пива генерировать, добавив специальные указания (например, «хочу обзор немецкого лагера на 3 звезды»), результаты впечатляют. Вот пример сгенерированного программой описания:

> Разливное в ресторане при пивоварне. Приятный темно-красный цвет с хорошей пенной шапкой, оставляющей заметные следы на стенках бокала. Аромат смородины и шоколада. Полнота вкуса недостаточная, несмотря на присутствие смородины. Привкус бурбона слабо выражен. Честно говоря, не знаю, с каким сортом сравнить это пиво. Я бы предпочел, чтобы карбонизация продолжалась немного дольше. Вполне питкое, я бы не возражал, если бы это пиво было более доступно[17].

Дао глубокого обучения

Сейчас по поводу глубокого обучения много шумихи и ажиотажа, хотя кое-что и оправдано. Но, по существу, глубокое обучение по-прежнему пытается ответить на фундаментальные вопросы машинного обучения типа «Является ли это лицом?». Разница в том, что глубокое обучение переняло методики нейронных сетей предыдущего поколения и добавило к ним автоматизированное конструирование признаков, чтобы было проще отвечать на вычислительно трудные вопросы о данных сложной структуры.

На практике лучший способ воспользоваться этой мощью состоит в том, чтобы подобрать архитектуру сети, отвечающую входным данным. Если это получится, то вы сможете с успехом применить глубокое обучение новыми интересными способами. Если нет, то не получите ничего сверх базовых методов моделирования типа логистической регрессии. Задача этой книги – вооружить вас навыками и знаниями, необходимыми для принятия таких решений и правильного применения глубокого обучения.

Как организована эта глава

Эта глава базируется на материале главы 1, но мы детальнее рассмотрим конкретные архитектуры глубоких сетей. Мы покажем, чем отличаются архитектуры, и опишем, почему конкретная архитектура больше подходит для извлечения признаков из данных определенного вида. И в заключение главы поделимся замечаниями о практическом использовании глубокого обучения и развеем некоторые нынешние заблуждения об этой области знаний.

Общие архитектурные принципы глубоких сетей

Прежде чем переходить к конкретным архитектурам основных глубоких сетей, расширим свои знания о базовых компонентах. Сначала мы более пристально рассмотрим те компоненты, о которых уже знаем:

- параметры;
- слои;
- функции активации;
- функции потерь;

---

[17] Источник: IEEE Spectrum // http://spectrum.ieee.org/computing/software/the-neural-network-that-remembers.

- методы оптимизации;
- гиперпараметры.

Затем дополним их новыми строительными блоками:

- ограниченные машины Больцмана (ОМБ);
- автокодировщики.

Далее на основе этих идей мы рассмотрим следующие архитектуры глубоких сетей:

- сети, предобученные без учителя;
- сверточные нейронные сети (СНС);
- рекуррентные нейронные сети (РНС);
- рекурсивные нейронные сети.

По ходу этой главы мы будем сообщать о том, как в DL4J реализованы некоторые аспекты глубоких сетей. А пока начнем с вопроса об обобщении параметров на глубокие сети.

Параметры

В главе 1 мы узнали, что в машинном обучении параметры – это вектор x в уравнении $Ax = b$. Параметры нейронной сети имеют непосредственное отношение к весам связей в сети. На рис. 1.4 показано, что вектор параметров x представлен вектором-столбцом. Для получения вектора-столбца результатов b мы умножаем матрицу A на вектор параметров x. Чем ближе вектор b к фактическим меткам, присутствующим в обучающих данных, тем лучше наша модель. Для нахождения хороших значений вектора параметров, доставляющих минимум функции потерь на обучающем наборе, применяются методы оптимизации, например градиентный спуск.

В глубоких сетях также имеется вектор параметров, представляющий связи в оптимизируемой модели. Но важнейшее отличие глубоких сетей в части параметров состоит в том, как соединены между собой слои в различных архитектурах. В ГСД мы видим два параллельных набора прямых связей между двумя раздельными сетями. Слои одной сети состоят из ГМБ (вполне самостоятельных подсетей, которые мы рассмотрим ниже в этой главе), которые служат для выделения признаков в интересах другой сети. Другой сетью в ГСД является многослойная нейронная сеть прямого распространения, в которой признаки, выделенные из слоев ГМБ, используются для инициализации весов. Это лишь один из многих примеров того, как параметры (веса) специализируются в разных архитектурах.

Параметры и структуры NDArray

В том, что касается линейной алгебры, DL4J опирается на библиотеку ND4J, содержащую линейно-алгебраические примитивы. Структуры NDArray и линейная алгебра – ключ к работе с нейронными сетями в DL4J.

Слои

В главе 1 мы узнали о входном, скрытых и выходном слоях нейронной сети прямого распространения. В главе 2 мы обогатили свой багаж знаниями о других типах слоев и обсудили, как они связаны с архитектурой глубокой сети. В некоторых архитектурах слои могут быть представлены подсетями. В предыдущем разделе было показано, что глубокая сеть доверия включает слои, составленные из ОМБ.

Слои – это фундаментальная архитектурная единица глубокой сети. В DL4J для настройки слоя изменяется тип функции активации (или тип подсети в случае ОМБ). Мы также увидим, как использовать комбинации слоев для решения поставленной задачи (классификации или регрессии). Наконец, мы узнаем, какие гиперпараметры следует задавать для слоя каждого типа (в зависимости от архитектуры), чтобы эффективнее обучить сеть на начальном этапе. Затем для предотвращения переобучения может понадобиться дополнительная настройка гиперпараметров.

Функции активации

В главе 1 мы познакомились с основными функциями активации, применяемыми в нейронных сетях прямого распространения. В этой главе мы покажем, как функции активации используются в конкретных архитектурах для выделения признаков. Признаки высшего порядка, которым глубокая сеть обучается на данных, являются результатом нелинейного преобразования выхода предыдущего слоя. Это позволяет сети выявлять закономерности в данных, оставаясь в пространстве ограниченной размерности.

Функции активации для архитектуры общего вида

В зависимости от выбранной функции активации может оказаться, что для определенных видов данных (скажем, разреженных или плотных) больше подходит та или иная целевая функция. Такого рода проектные решения мы объединим в две большие группы, свойственные всем архитектурам:

- скрытые слои;
- выходные слои.

Задача скрытых слоев – выделение все более абстрактных признаков из исходных данных. В зависимости от архитектуры используется то или иное подмножество функций активации слоя. Далее в этой главе мы проиллюстрируем, как это происходит в ГСД, СНС и РНС. В главе 4 мы изучим влияние функции активации на архитектуру сети в контексте настройки глубоких сетей.

Замечание о входных слоях

Для входного слоя мы обычно хотим передать исходный вектор признаков без изменения, так что на практике функция активации для входного слоя не задается.

Функции активации в скрытых слоях. Чаще всего используются такие функции:

- сигмоида;
- tanh;
- hardtanh;
- блок линейной ректификации (ReLU) и его варианты.

Непрерывное распределение входных данных обычно лучше моделируется с помощью функции активации ReLU. Если же ReLU дает не слишком хорошие результаты, то мы рекомендуем попробовать функцию tanh (если сеть не слишком глубокая); правда, следует оговориться, что проблемы сети могут быть связаны с другими гиперпараметрами.

Сигмоидные функции активации на практике

В последние годы сигмоиду перестали использовать в качестве функции активации в скрытых слоях – как в практических разработках, так и в теоретических исследованиях.

В этой книге мы не раз увидим, как эти функции активации применяются в различных архитектурах.

Эволюция функций активации на практике

В теоретических работах по глубокому обучению встречается целое семейство блоков ReLU, например «ReLU с утечкой». Более подробно эта тема освещена в главе 6.

Выходной слой для регрессии. Проектное решение зависит от того, какой ответ мы хотим получить от модели. Если мы хотим получить одно вещественное число, то воспользуемся линейной функции активации.

Выходной слой для бинарной классификации. В этом случае мы возьмем сигмоидный выходной слой с одним нейроном, который дает вещественное значение в диапазоне от 0.0 до 1.0 (не включая граничных значений). Это число обычно интерпретируется как распределение вероятности одного класса.

Выходной слой для многоклассовой классификации. Если имеется задача многоклассового моделирования, но нас интересует только наилучшая оценка, то мы возьмем выходной softmax-слой с функцией `argmax()`, которая дает оценку для наиболее вероятного класса. Выходной softmax-слой дает распределение вероятности всех классов.

Получение нескольких классификаций

Если требуется получить несколько классификаций одного выхода (например, человек + автомобиль), то softmax в выходном слое не подойдет. Вместо этого следует взять сигмоидный выходной слой с n нейронами, что даст независимое распределение вероятности каждого класса.

Функции потерь

В главе 2 мы обсуждали функции потерь и их роль в машинном обучении. Функция потерь количественно выражает отклонение предсказанного выхода от фактического. С помощью функции потерь мы определяем штраф за неправильную классификацию входного вектора. До сих пор нам встречались такие функции потерь:

- квадратичная потеря;
- логистическая потеря;
- кусочно-линейная потеря;
- отрицательное логарифмическое правдоподобие.

Выше мы отнесли функции потерь к одной из трех категорий:

- регрессия;
- классификация;
- реконструкция.

Первые две категории были рассмотрены в главе 1. Третья же, реконструкция, связана с автоматическим выделением признаков, благодаря чему сети глубокого обучения достигли побившей все рекорды верности. В некоторых архитектурах глубоких сетей функции потерь из категории реконструкции помогают сети более эффективно выделять признаки, особенно если используются в паре с подходящей функцией активации. Примером может служить классификация с многоклассовой перекрестной энтропией в качестве функции потерь в сочетании с функцией активации softmax. Специализированную функцию потерь мы рассмотрим в следующем разделе.

Использование перекрестной энтропии для реконструкции

Когда энтропия используется для реконструкции, мы сначала накладываем «гауссов шум» (вид статистического белого шума), а затем функция потерь штрафует сеть за отличия результата реконструкции от исходных данных. Эта обратная связь заставляет сеть обучаться различным признакам в попытке более эффективно реконструировать вход и минимизировать ошибку. В глубоком обучении энтропийная функция потерь при реконструкции применяется для конструирования признаков на этапе предобучения сети с использованием ОМБ.

Алгоритмы оптимизации

Обучение модели означает отыскание наилучшего множества значений параметров. Машинное обучение можно представлять себе как задачу оптимизации, в которой требуется минимизировать функцию потерь относительно параметров функции предсказания (основанной на модели).

Определение «наилучшего» в терминах функции потерь

В алгоритмах оптимизации «наилучшее множество значений» параметров определяется как множество значений, доставляющих минимум функции потерь.

В главе 1 были введены понятия оптимизации, градиентного спуска и вектора параметров. В этом разделе мы рассмотрим более сложные методы оптимизации и их применение для обучения глубоких сетей. Мы разобьем алгоритмы оптимизации на две категории:

- первого порядка;
- второго порядка.

В алгоритмах первого порядка вычисляется матрица Якоби, или *якобиан*.

Якобиан

Якобиан – это матрица, составленная из частных производных функции потерь по каждому параметру.

В якобиане присутствует одна частная производная для каждого параметра (при вычислении частных производных по какой-то переменной все остальные переменные временно считаются постоянными). Алгоритм выполняет шаг в направлении, определяемом якобианом.

В алгоритмах второго порядка вычисляется производная якобиана (производная матрицы производных), т. е. производится аппроксимация матрицы Гессе, или *гессиана*. Методы второго порядка учитывают зависимости между параметрами при решении о том, на какую величину изменять каждый параметр.

Методы второго порядка

Методы второго порядка способны выбирать «лучшие» шаги, но на каждом шаге производится больше вычислений.

Практическое применение алгоритмов оптимизации

Мы сообщаем достаточно детальную информацию об алгоритмах оптимизации, чтобы вы знали о том, что за ними стоит. В следующих главах мы будем просто говорить, когда и в каком контексте следует использовать каждый алгоритм.

Другие алгоритмы оптимизации
Существуют и другие алгоритмы оптимизации (например, «метаэвристики»), не рассматриваемые в этой книге. К ним относятся:

- генетические алгоритмы;
- оптимизация методом роя частиц;
- оптимизация методом колонии муравьев;
- имитация отжига.

Методы первого порядка

Упомянутая выше матрица Якоби состоит из частных производных функции потерь по параметрам сети. На практике она вычисляется в точке, определяемой текущими значениями параметров.

Если считать, что для приближения к цели мы делаем по одному шагу за раз, то методы первого порядка на каждом шаге вычисляют градиент (якобиан), чтобы определить направление следующего шага. Это означает, что на каждой итерации мы пытаемся найти наилучшее направление движения, определяемое нашей функцией потерь. Потому-то алгоритмы оптимизации часто называют «поиском». Они ищут путь к точке с минимальной ошибкой.

Градиентный спуск – член семейства алгоритмов поиска пути. Существуют разные варианты градиентного спуска, но в основе своей он сводится к нахождению направления следующего шага, оптимального с точки зрения целевой функции. Эти шаги приближают нас к глобальному минимуму ошибки или к максимальному правдоподобию.

Стохастический градиентный спуск (СГС) – основной алгоритм оптимизации в машинном обучении. Он на несколько порядков быстрее таких методов, как пакетный градиентный спуск, но при этом не приносит в жертву верность модели.

Почему СГС называется «стохастическим»?
Это связано с тем, что градиент вычисляется для одного обучающего примера (или мини-пакета обучающих примеров). Вычисленный градиент – это «зашумленная» аппроксимация истинного градиента, но СГС все же сходится, причем заметно быстрее.

Достоинства СГС – простота реализации и высокая скорость обработки больших наборов данных. Для настройки СГС можно изменять скорость обучения (например, в рассматриваемом далее методе Adagrad) или использовать информацию второго порядка (т. е. гессиан), как будет показано ниже. Популярность СГС для обучения нейронных сетей объясняется также его устойчивостью к зашумленным обновлениям. Иными словами, построенные с его помощью модели хорошо обобщаются.

Другие факторы, влияющие на скорость обучения
Стоит отметить, что на скорость обучения может влиять также применение таких методов, как импульс или RMSProp.

Методы второго порядка

Во всех методах второго порядка вычисляется гессиан или его аппроксимация. Как уже было сказано, мы можем считать гессиан производной якобиана. То есть это матрица вторых производных – «ускорение, а не скорость». Цель гессиана – описать кривизну в каждой точке якобиана. К методам второго порядка относятся:

- BFGS с ограниченной памятью (L-BFGS)[18];
- метод сопряженных градиентов[19];
- безгессианная оптимизация[20].

Рассматривайте эти алгоритмы оптимизации как черный ящик, позволяющий найти наилучший путь к минимальной ошибке при заданной целевой функции и способе определения градиента.

Компромиссы при оптимизации

Главное различие между методами первого и второго порядков состоит в том, что методы второго порядка сходятся за меньшее число шагов, но каждый шаг требует большего объема вычислений.

L-BFGS. Алгоритм L-BFGS принадлежит семейству квазиньютоновских методов. Это вариант алгоритма Бройдена–Флетчера–Гольдфарба–Шанно (BFGS) с ограничением на часть градиента, хранящуюся в памяти. Это значит, что алгоритм не вычисляет полную матрицу Гессе, что было бы дороже с вычислительной точки зрения.

L-BFGS аппроксимирует обратную матрицу Гессе, чтобы направить поиск весов в наиболее многообещающие области пространства параметров. Если в BFGS хранится полная обратная матрица градиента размера $n \times n$, то в L-BFGS от гессиана сохраняется лишь несколько векторов, дающих приближенную информацию второго порядка. На практике L-BFGS и метод сопряженных градиентов могут оказаться быстрее и устойчивее стохастического градиентного спуска.

L-BFGS на практике

Хотя L-BFGS обладает рядом интересных свойств, на практике он редко используется в глубоких сетях.

Метод сопряженных градиентов. В этом методе направление линейного поиска определяется на основе информации о сопряженности. В части выполнения линейного поиска метода очень похож на градиентный спуск. Основное отличие заключается в том, что направление поиска на следующем шаге является сопряженным направлению на предыдущем шаге.

Безгессианная оптимизация. Этот метод родствен методу Ньютона, но лучше минимизирует имеющуюся квадратичную функцию. Этот мощный метод оптимизации был адаптирован для нейронных сетей Джеймсом Мартенсом в 2010 году. Минимум квадратичной функции ищется итеративно с помощью метода сопряженных градиентов.

Гиперпараметры

Под гиперпараметром мы понимаем любой конфигурационный параметр, который задается пользователем и может повлиять на качество или производительность модели.

---

[18] Le et al., 2011. On Optimization Methods for Deep Learning // https://sites.wustl.edu/machine-learning/.

[19] LeCun et al., 1998. Efficient BackProp // http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

[20] Martens, 2010. Deep learning via Hessian-free optimization // https://sites.wustl.edu/machine-learning/.

Можно выделить несколько категорий гиперпараметров:

- размер слоя;
- масштаб (импульс, скорость обучения);
- регуляризация (прореживание, прореживание связей, L1, L2);
- активация (и семейства функций активации);
- стратегия инициализации весов;
- функции потерь;
- настройка периодов обучения (размер мини-пакета);
- схема нормировки входных данных (векторизация).

В этом разделе мы дополним материал главы 1 описанием нескольких новых гиперпараметров, имеющих отношение к глубокому обучению.

Несколько предостережений о гиперпараметрах

Некоторые гиперпараметры применимы не всегда. Соответствующие детали мы обсудим в главах 6 и 7. Кроме того, после изменения одного гиперпараметра может понадобиться изменить и другие. Наконец, некоторые гиперпараметры несовместимы между собой (например, Adagrad и импульс).

Размер слоя

Размер слоя – это количество нейронов в нем. С входным и выходным слоями все ясно, потому что они напрямую соответствуют входу и выходу модели. Размер входного слоя равен числу признаков во входном векторе. А в выходном слое будет либо один нейрон, либо столько, сколько классов мы пытаемся предсказать.

А вот выбор числа нейронов в скрытых слоях – проблема настройки гиперпараметра. Мы можем задать любое число, никаких правил, диктующих размер слоя, не существует. Но сложность моделируемой задачи напрямую связана с количеством нейронов в скрытых слоях сети. Возможно, вы решите с самого начала взять побольше нейронов, но за это решение придется заплатить.

Схема связей между слоями зависит от архитектуры глубокой сети. Но, как мы видели в главе 1, обучиться сеть должна весам связей. Включив в модель больше параметров, мы увеличиваем объем работы, необходимой для обучения. Увеличивается время обучения, и могут возникнуть проблемы со сходимостью.

Большое число параметров и переобучение

Иногда модель с большим числом параметров сходится быстрее просто потому, что «запоминает» обучающие данные. В главе 6 мы обсудим способы борьбы с переобучением.

В главе 6 мы поговорим об эвристиках для определения числа нейронов в одном слое и о том, как итеративно находить хорошие значения гиперпараметров.

Масштабные параметры

К гиперпараметрам из этой категории относятся скорость обучения, величина шага и импульс.

Скорость обучения. Скорость обучения определяет, как быстро следует изменять вектор параметров при перемещении в пространстве поиска. Если скорость обучения слишком велика, то мы, возможно, приблизимся к цели быстрее (реже придется вычислять ошибку), но из-за большой величины шага можем проскочить мимо наилучшего решения.

Высокая скорость обучения и устойчивость

Еще один побочный эффект высокой скорости обучения – опасность, что процесс обучения вообще не сойдется.

Если же скорость обучения слишком мала, то процесс обучения может занять больше времени, чем мы рассчитывали. В таком случае алгоритм обучения становится неэффективным. Проблема заключается в том, что скорость обучения сильно зависит от набора данных и даже от других гиперпараметров. Из-за этого подбор правильных гиперпараметров отнимает много времени и усилий.

Можно также постепенно уменьшать скорость обучения, руководствуясь некоторым правилом. Мы вернемся к этому вопросу в главах 6 и 7.

Важность гиперпараметра «скорость обучения»

Скорость обучения считается одним из важнейших гиперпараметров нейронных сетей.

Импульс Нестерова. В «бесхитростном» варианте СГС градиент используется непосредственно, и это создает проблему, потому что градиенты по всем параметрам могут оказаться близки к нулю. Тогда СГС будет совершать очень малые шаги. И наоборот, если градиент очень большой, то шаги будут слишком велики. Чтобы как-то справиться с этими трудностями, можно воспользоваться несколькими методами:

- метод импульса Нестерова;
- RMSProp;
- Adam;
- AdaDelta.

DL4J и корректоры

Метод Нестерова, RMSProp, Adam и AdaDelta в DL4J называются «корректорами». Многие термины, употребляемые в этой книге, имеют общее значение во всей литературе по глубокому обучению, но этот специфичен для DL4J.

Обучение можно ускорить, увеличив импульс, но тогда модель может проскочить оптимальные значения параметров и не найти минимальную ошибку. Импульс – это коэффициент в диапазоне от 0.0 до 1.0, применяемый к скорости изменения весов со временем. Обычно выбирают значение между 0.9 и 0.99.

AdaGrad. Метод AdaGrad[21] – еще один способ нахождения «правильной» скорости обучения. В названии отражена идея адаптивного использования субградиентных методов для динамического управления скоростью обучения[22] алгоритма оптимизации. AdaGrad демонстрирует монотонное убывание и никогда не приводит к выставлению скорости обучения, превышающей начальное значение.

В AdaGrad вычисляется квадратный корень из суммы квадратов прошлых градиентов. В начале процесса AdaGrad ускоряет обучение, а ближе к точке сходимости замедляет его, обеспечивая более гладкое обучение.

RMSProp. RMSprop – очень эффективный, но до сих пор не опубликованный метод адаптивного изменения скорости обучения. Забавно, но все, кто его используют, ссылаются на слайд 29 из лекции Джеффри Хинтона на сайте Coursera[23].

[21] Duchi, Hazan and Singer, 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization // http://jmlr.org/papers/v12/duchi11a.html.

[22] http://cs231n.github.io/neural-networks-3/.

[23] http://cs231n.github.io/neural-networks-3/.

AdaDelta. Метод AdaDelta[24] – вариант AdaGrad, в котором хранится лишь недавняя история, а не вся, как в AdaGrad.

ADAM. В методе ADAM (последний по времени алгоритм обновления, разработанный в Торонтском университете) скорость обучения вычисляется на основе оценок первого и второго моментов градиентов.

Регуляризация

Обсудим идею регуляризации глубже, чем в главе. Регуляризация – это способ борьбы с переобучением. Переобучение возникает, когда модель хорошо описывает обучающий набор, но не обобщается на новые данные. Переобученные модели не обладают предсказательной способностью для данных, которых не видели раньше. Джеффри Хинтон так описывает наилучший способ построения модели на основе нейронной сети:

> Доведите ее до переобучения, а затем зарегуляризируйте до смерти.

Регуляризация помогает изменять градиент, так чтобы не производить шаги в направлениях, ведущих к переобучению. К методам регуляризации относятся следующие:

- прореживание (Dropout);
- прореживание связей (DropConnect);
- штраф по норме L1;
- штраф по норме L2.

Методы Dropout и DropConnect скрывают части входных данных на каждом слое, так чтобы нейронная сеть обучалась на других частях. Частичное обнуление данных заставляет сеть обучаться более общим представлениям. Принцип работы регуляризации заключается в прибавлении дополнительного члена к вычисленному стандартным способом градиенту.

Прореживание (DropOut). Механизм прореживания[25] применяется, чтобы улучшить процесс обучения нейронной сети за счет устранения скрытых блоков. Заодно повышается и скорость обучения. Алгоритм случайным образом выбирает нейроны, которые не будут участвовать ни в прямом, ни в обратном распространении.

Прореживание и усреднение модели

Мы можем также сопоставить прореживание с идеей усреднения выхода нескольких моделей. Взяв коэффициент прореживания 0.5, мы получим среднее значение модели. Случайное прореживание признаков – это выборка из 2^N возможных архитектур, где N – число параметров.

Прореживание связей (DropConnect). Метод DropConnect[26] аналогичен Dropout, но маскируется не сам нейрон скрытого слоя, а связь между двумя нейронами.

Штраф по норме L1. Методы штрафования по норме L1 или L2 предотвращают чрезмерное разрастание пространства параметров сети в одном направлении. Они уменьшают слишком большие веса.

---

[24] Zeiler, 2012. ADADELTA: An Adaptive Learning Rate Method // https://arxiv.org/abs/1212.5701.

[25] Bengio et al., 2015. Deep Learning (In Preparation).

[26] Bengio et al., 2015. Deep Learning (In Preparation).

L1-регуляризация считается вычислительно неэффективной в неразреженном случае, она дает разреженные выходы и включает встроенный механизм отбора признаков. В этом случае умножаются абсолютные величины весов, а не их квадраты. Метод приводит к обнулению многих весов, позволяя некоторым достигать большой величины; в результате веса проще интерпретировать.

Штраф по норме L2. Напротив, L2-регуляризация вычислительно эффективна, поскольку имеется аналитическое решение, она дает неразреженные выходы, но автоматического отбора признаков не обеспечивает. При задании гиперпараметра L2 к целевой функции прибавляется член, уменьшающий квадраты весов. Полусумма квадратов весов умножается на коэффициент, называемый весовой стоимостью (weight-cost). L2-регуляризация повышает обобщаемость, сглаживает зависимость выхода модели от входов и дает сети возможность игнорировать неиспользуемые веса.

Мини-пакеты

Смысл мини-пакетов[27] – в том, чтобы передать системе обучения не один входной вектор, а целую группу (пакет). Это позволяет более эффективно задействовать оборудование и ресурсы на уровне архитектуры вычислительной системы. Кроме того, мы получаем возможность применить векторные вычисления к некоторым линейно-алгебраическим операциям (точнее, к умножению матриц). И тогда векторные вычисления можно перенести на GPU, если таковые имеются в системе.

Итоги

В главе 2 мы узнали о некоторых базовых средствах регуляризации, применяемых в многослойных сетях прямого распространения. В этой главе мы добавили в свой арсенал ряд новых методов и гиперпараметров для нахождения лучших векторов параметров. Теперь применим эти идеи к конструированию строительных блоков глубоких сетей.

Строительные блоки глубоких сетей

Глубокие сети не ограничиваются простыми многослойными сетями прямого распространения. В некоторых случаях глубокая сеть состоит из нескольких меньших сетей, рассматриваемых как строительные блоки, а иногда используется специализированный набор слоев. Вот несколько строительных блоков, о которых мы хотели бы поговорить подробнее:

- многослойные нейронные сети прямого распространения;
- ОМБ;
- автокодировщики.

Канонические сети прямого распространения были рассмотрены в главе 2. Эти сети, основанные на идее биологического нейрона, являются простейшими примерами искусственных нейронных сетей. Они состоят из входного слоя, одного или нескольких скрытых слоев и выходного слоя. В этом разделе мы познакомим-

[27] Bengio, 2012. Practical recommendations for gradient-based training of deep architectures // https://arxiv.org/abs/1206.5533.

ся с сетями, которые используются в качестве строительных блоков более крупных глубоких сетей:

- ОМБ;
- автокодировщики.

Те и другие характеризуются наличием дополнительного шага послойного обучения. Зачастую они применяются на этапе предобучения в других более крупных сетях.

Послойное предобучение без учителя

Послойное предобучение без учителя[28] помогает при некоторых условиях. Со временем появление более качественных методов оптимизации, функций активации и способов инициализации весов снизило значение глубоких сетей с предобучением. Интересными они становятся, когда имеется много непомеченных данных и лишь сравнительно небольшой набор помеченных обучающих данных. Но предобучение сопряжено с дополнительными расходами на настройку и обучение.

В методе послойного предобучения мы сначала обучаем первый слой (например, ОМБ) без учителя на основе входных данных. Это дает первый слой весов для основной нейронной сети (например, многослойного перцептрона). Этот процесс продолжается для каждого слоя сети, при этом выход предыдущего слоя используется как вход последующего. Процесс предобучения позволяет инициализировать параметры главной нейронной сети хорошими начальными значениями.

ОМБ моделирует вероятность и прекрасно работает в роли инструмента выделения признаков. Это сеть прямого распространения, в которой данные распространяются в одном направлении с двумя смещениями, а не с одним, как в традиционных сетях прямого распространения с обратным распространением ошибки.

Автокодировщики – вариант нейронных сетей прямого распространения, в котором имеется дополнительное смещение для вычисления ошибки реконструкции исходных входных данных. Обученный автокодировщик используется как обычная сеть прямого распространения для активаций. Это форма выделения признаков без учителя, поскольку нейронная сеть использует для обучения весов только сами входные данные, а не обратное распространение ошибки, для которого нужны метки. ОМБ или автокодировщики могут применяться в качестве строительных блоков более крупных сетей (но редко бывает, что применяются сразу оба). В следующих разделах мы рассмотрим их подробнее.

Ограниченные машины Больцмана

ОМБ применяются для следующих целей:

- выделение признаков;
- понижение размерности.

Слово «ограниченные» в названии означает, что связи между блоками одного слоя запрещены (т. е. отсутствуют прямые взаимодействия между любыми видимыми или любыми двумя скрытыми блоками). Джеффри Хинтон, пионер глубокого обучения, популяризировавший ОМБ[29] десять лет назад, описывает более общую машину Больцмана следующим образом:

---

[28] Bengio et al., 2007. Greedy Layer-Wise Training of Deep Networks // https://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf.

[29] https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf.

Сеть, состоящая из симметрично связанных нейроноподобных блоков, которые принимают стохастические решения о своем состоянии: возбужден или нет.

ОМБ является также частным случаем автокодировщика, о котором мы будем говорить в следующем разделе. ОМБ используются для предобучения слоев в более крупных сетях, например в глубоких сетях доверия.

Джеффри Хинтон

Джеффри Хинтон занимает должность заслуженного исследователя в Google (по совместительству) и работает в Торонтском университете заслуженным профессором в отставке[30]. Группой д-ра Хинтона были выполнены основополагающие работы по ОМБ и глубоким сетям доверия.

Результаты, полученные группой д-ра Хинтона, немало способствовали привлечению интереса к глубокому обучению в том виде, в котором мы знаем его сейчас. В 2012 году Алекс Крижевский, Илья Суцкевер и Джеффри Хинтон разработали «большую глубокую сверточную сеть», выигравшую состязание 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). Эта сеть, получившая название «AlexNet»[31], прославилась как крупное достижение в компьютерном зрении и положила начало повальному увлечению глубоким обучением.

Д-р Хинтон всегда ратовал за необходимость фундаментальных исследований и терпеливого ожидания результатов:

> …после того как мы с Терри Сейновским придумали алгоритм обучения машины Больцмана, мне понадобилось 17 лет, чтобы найти такой его вариант, который работал эффективно. Если вы верите в идею, надо продолжать попытки.

Структура сети

Базовая ОМБ состоит из пяти основных частей:

- видимые блоки;
- скрытые блоки;
- веса;
- видимый блок смещения;
- скрытый блок смещения.

В стандартной ОМБ, показанной на рис. 3.3, имеются видимый и скрытый слои, а также граф весов (связей) между видимыми и скрытыми блоками. Можно считать, что это такие же веса, как в классической нейронной сети.

Видимый и скрытый слои. В ОМБ каждый видимый блок связан с каждым скрытым блоком, но никакие два блока одного слоя не связаны. Скрытые блоки являются детекторами признаков, они обучаются признакам, присутствующим во входных данных. Блоки каждого слоя устроены по аналогии с биологическим нейроном, как и в сетях прямого распространения из главы 1. Блоки видимого слоя «наблюдаемые» в том смысле, что принимают на входе обучающие векторы. В каждом слое имеется блок смещения, который всегда активен.

---

[30] https://www.cs.toronto.edu/~hinton/.

[31] Krizhevsky, Sutskever and Hinton, 2012. ImageNet Classification with Deep Convolutional Neural Networks // https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

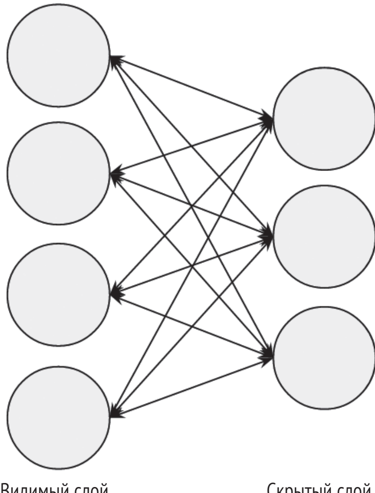


Рис. 3.3 ❖ Сеть ОМБ

Каждый блок выполняет вычисление со своим входом и выводит результат, сообразуясь со стохастическим решением о том, передавать данные с помощью функции активации или нет. Как и в искусственном нейроне из главы 1, значение активации вычисляется на основе весов связей и входных значений. Начальные веса генерируются случайным образом.

Связи и веса. Связи могут существовать только между видимыми и скрытыми блоками. Ребра графа представляют связи, по которым передаются сигналы. Кружочки можно уподобить биологическим нейронам. Это блоки, принимающие решения о том, следует в результате вычисления активироваться или нет. «Да» означает, что сигнал передается дальше по сети, «нет» – что не передается.

Обычно состояние «активен» означает, что данные, проходящие через блок, представляют ценность, т. е. содержат информацию, которая поможет сети принять решение. «Неактивен» означает, что сеть воспринимает данный вход как несущественный шум. Сеть обучается распознавать, какие признаки (сигналы) коррелируют с какими метками (какой код содержится в сообщении). Обученная сеть правильно классифицирует поступающие данные.

Смещение. Блоки смещения на каждом уровне связаны со всеми блоками следующего уровня, но не получают входных сигналов. Благодаря им сеть лучше моделирует случаи, когда входной блок всегда активен или всегда неактивен.

Обучение

Техника *предобучения* с применением ОМБ подразумевает, что она обучается реконструировать исходные данные по ограниченной выборке из них. То есть, видя подбородок, обученная сеть сможет аппроксимировать («реконструировать») все лицо. ОМБ обучается реконструировать входной набор данных. Концепцию реконструкции мы рассмотрим в следующем разделе.

Сопоставительное расхождение

ОМБ вычисляет градиенты, применяя алгоритм сопоставительного расхождения (contrastive divergence), который осуществляет выборку для послойного предобучения. Это алгоритм,

известный также под названием CD-k, минимизирует расхождение Кульбака–Лейблера (между реальным распределением данных и гипотезой), производя выборки на *k* шагах марковской цепи для вычисления гипотезы.

Реконструкция

Глубокие нейронные сети с предобучением без учителя (ОМБ, автокодировщики) конструируют признаки из непомеченных данных с помощью реконструкции. Веса, найденные в результате предобучения без учителя, используются для инициализации весов в других сетях, например в глубоких сетях доверия.

Реконструкция как факторизация матрицы
Реконструкция является задачей факторизации, или разложения, матрицы.

На рис. 3.4 наглядно показана сеть ОМБ, участвующая в реконструкции.

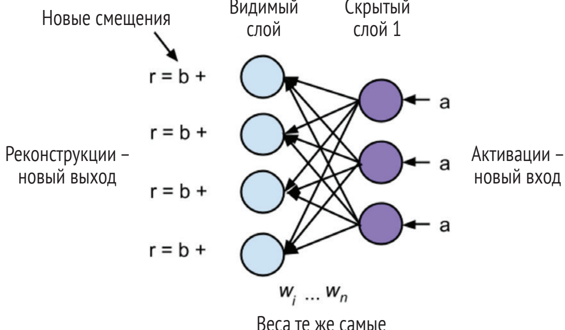


Рис. 3.4 ❖ Реконструкция с помощью ОМБ

Пояснить, как работает реконструкция в ОМБ, можно на примере набора данных MNIST[32] (Mixed National Institute of Standards and Technology), содержащего изображения рукописных цифр. На рис. 3.5 показана выборка из этого набора.

Рис. 3.5 ❖ Выборка из набора данных MNIST

[32] http://yann.lecun.com/exdb/mnist/.

Обучающий набор MNIST содержит 60 000 записей, а тестовый – 10 000 записей. Если обучить ОМБ на наборе MNIST, то можно будет сделать выборку из обученной сети, чтобы посмотреть[33], насколько хорошо она способна реконструировать цифры. На рис. 3.6 показано, как ОМБ постепенно выполняла реконструкцию.

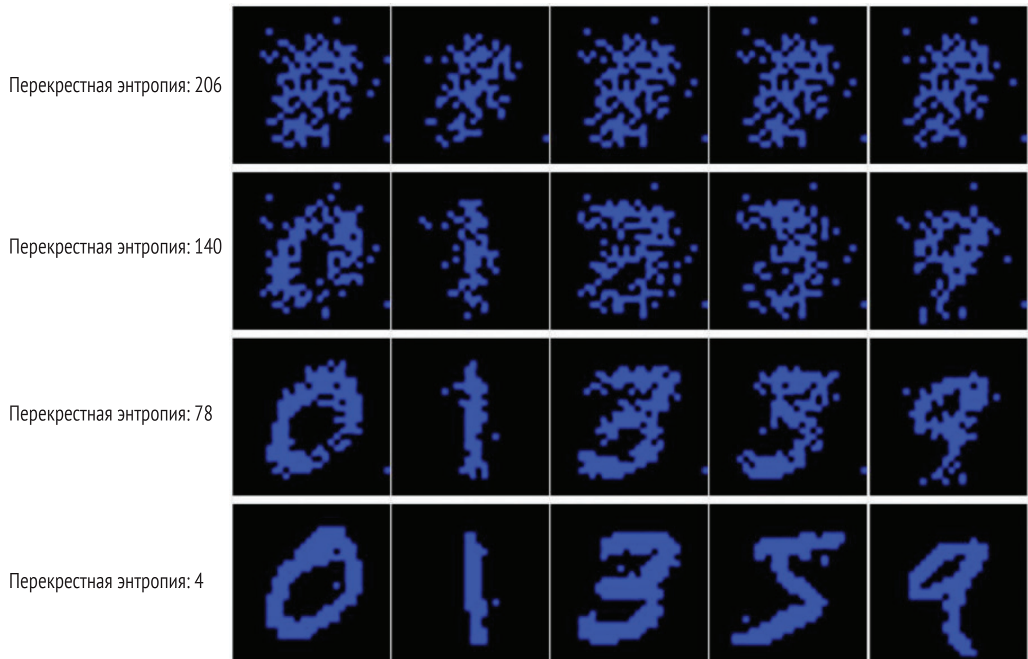


Рис. 3.6 ❖ Реконструкция цифр из набора MNIST с помощью ОМБ

Если обучающие данные имеют нормальное распределение, то большая их часть концентрируется вокруг *среднего*, а чем дальше от среднего, тем реже встречаются данные. Такое распределение выглядит как колоколообразная кривая. Зная среднее и дисперсию (сигму), мы можем восстановить всю кривую. Но предположим, что среднее и дисперсия неизвестны. Тогда нужно высказать о них гипотезу. Подход, при котором мы случайным образом выбираем эти параметры и сопоставляем получившуюся кривую с оригинальной, работает примерно так же, как функция потерь. Мы измеряем расхождение между двумя распределениями вероятности, как измеряем различие между ошибочной и правильной классификацией, – корректируем параметры и пробуем снова.

Перекрестная энтропия реконструкции

В данном случае целевой функцией является перекрестная энтропия реконструкции, или расхождение Кульбака–Лейблера (математики-криптаналитики Соломон Кульбак и Ричард Лейблер опубликовали статью на эту тему в 1951 году). Слово «перекрестная» намекает

[33] Yosinksi and Lipson, 2012. Visually Debugging Restricted Boltzmann Machine Training with a 3D Example // http://yosinski.com/media/papers/Yosinski2012VisuallyDebuggingRestrictedBoltzmannMachine.pdf.

на сравнение двух распределений, а «энтропия» – понятие из теории информации, характеризующее неопределенность. Например, нормальная кривая с широким разбросом, или дисперсией, характеризуется высокой неопределенностью местонахождения точки, т. е высокой энтропией.

Другие применения ОМБ

ОМБ применяются и в других задачах:

- понижение размерности;
- классификация;
- регрессия;
- коллаборативная фильтрация;
- тематическое моделирование.

Автокодировщики

Автокодировщики применяются для обучения сжатых представлений наборов данных. Как правило, это нужно для понижения размерности набора данных. Выходом сети автокодировщика является более компактная реконструкция входных данных.

Сходство с многослойным перцептроном

Автокодировщики напоминают многослойный перцептрон в том смысле, что имеют входной слой, несколько скрытых слоев и выходной слой. Основное отличие заключается в том, что в выходном слое автокодировщика всегда столько же блоков, сколько во входном. На рис. 3.7 показана схема автокодировщика.

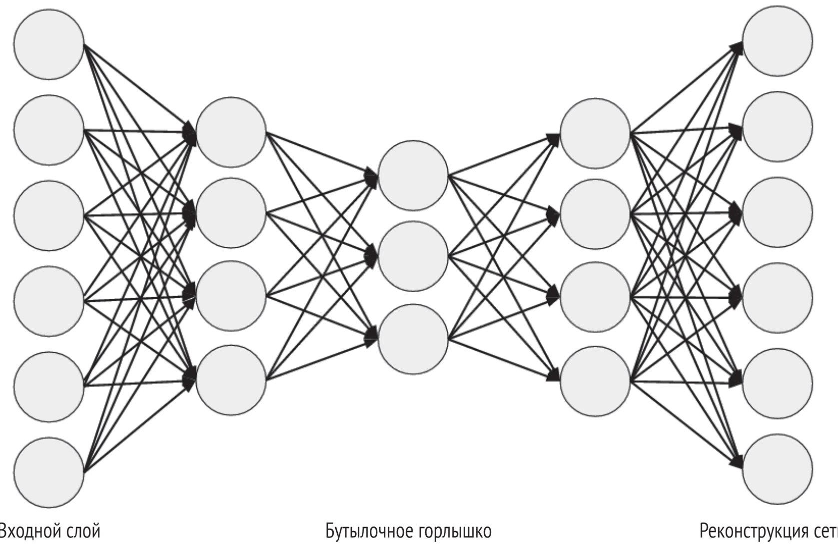


Рис. 3.7 ❖ Архитектура сети автокодировщика

Помимо размера выходного слоя, существуют и другие отличия, которые мы опишем ниже.

Отличительные особенности автокодировщиков

Автокодировщик отличается от многослойного перцептрона в двух отношениях:

- он служит для обучения без учителя на неразмеченных данных;
- он строит сжатое представление входных данных.

Обучение без учителя на неразмеченных данных. Автокодировщик обучается прямо на неразмеченных данных. И это связано со вторым отличием от многослойных перцептронов.

Обучение воспроизведению входных данных. Цель многослойного перцептрона – предсказывать класс (например, мошенническая операция или честная). Автокодировщик обучается воспроизводить собственные входные данные.

Обучение автокодировщика

Для обновления весов в автокодировщике применяется обратное распространение ошибки. Основное отличие между ОМБ и более общим классом автокодировщиков заключается в способе вычисления градиента.

Распространенные варианты автокодировщиков

Две важные разновидности автокодировщиков: *сжимающие* и *шумоподавляющие*.

Сжимающие автокодировщики. Эта архитектура изображена на рис. 3.7. Входные данные должны пройти через бутылочное горлышко и только потом раздвигаются в выходное представление.

Шумоподавляющие автокодировщики. В этом случае[34] автокодировщик получает искаженные входные данные (например, некоторые признаки случайным образом удалены), а сеть обучается выдавать на выходе неискаженный сигнал.

Применение автокодировщиков

На первый взгляд, модель для представления входного набора данных кажется бесполезной. Однако нас интересует не столько ее выход, сколько различие между входным и выходным представлениями. Если мы сможем обучить нейронную сеть распознавать данные, которые она «видит» обычно, то она сможет сообщить нам о том, что наблюдаемые данные необычны, т. е. аномальны.

Автокодировщики как детекторы аномалий

Обычно автокодировщики используются в системах, где мы знаем, как выглядят обычные данные, но формально описать, что такое аномалия, трудно. Автокодировщики прекрасно работают в качестве движка системы распознавания аномалий.

Вариационные автокодировщики

Вариационные автокодировщики (variational autoencoder – VAE) были предложены сравнительно недавно в работе Кингма и Уэллинга[35] (см. рис. 3.8). VAE похожи на сжимающие и шумоподавляющие автокодировщики в том смысле, что обучаются реконструировать свой вход без учителя.

---

[34] Vincent et al., 2010. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion // http://www.jmlr.org/papers/volume11/vincent10a/vincent10a.pdf.

[35] Kingma and Welling, 2013. Auto-Encoding Variational Bayes // https://arxiv.org/abs/1312.6114.

Однако механизм обучения VAE совершенно иной. В сжимающих и шумоподавляющих автокодировщиках значения активации одного слоя отображаются на значения активации следующего слоя, как в стандартной нейронной сети. А в VAE на прямом проходе применяется вероятностный подход.

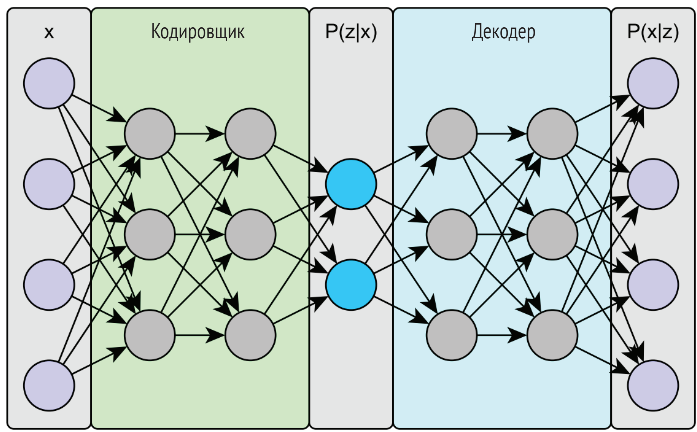


Рис. 3.8 ❖ Архитектура сети VAE

В модели VAE предполагается, что данные x порождаются в два приема: (a) из априорного распределения порождается значение $z^i \sim p(z)$ и (b) порождается пример в соответствии с некоторым условным распределением $x^i \sim p(x|z)$. Конечно, мы не знаем фактических значений z, и точный вывод $p(z|x)$ – вычислительно неразрешимая задача. Чтобы справиться с этой проблемой, мы аппроксимируем оба распределения, $p(z|x)$ и $p(x|z)$, нейронными сетями – соответственно кодировщиком и декодером. Например, если $P(z|x)$ – нормальное распределение, то прямой проход кодировщика даст его параметры μ и σ^2.

Аналогично параметры распределения $P(x|z)$ дает прямой проход декодера[36]. В общем и целом сеть обучается (методом обратного распространения) максимизировать нижнюю границу маргинального правдоподобия обучающих данных, $\log p(x^1, \ldots, x^N)$. Модель VAE также была обобщена на обучение без учителя на временных рядах – это называется вариационным *рекуррентным* автокодировщиком[37]. В главе 5 мы увидим практическое применение VAE для генерации цифр после обучения на наборе MNIST.

[36] Эти параметры распределения не следует путать с параметрами обучаемой сети: на практике это просто значения активации, используемые, например, для задания среднего и дисперсии нормального распределения или среднего распределения Бернулли.

[37] Fabius and van Amersfoort, 2014. Variational Recurrent Auto-Encoders // *https://arxiv.org/abs/1412.6581*.

Глава 4

Основные архитектуры глубоких сетей

Архитектура – мать всех искусств. Без собственной архитектуры у нашей цивилизации не будет души.
– *Фрэнк Ллойд Райт*

Познакомившись с некоторыми компонентами глубоких сетей, рассмотрим четыре основные архитектуры таких сетей и способы их построения из меньших сетей. Ранее мы уже перечислили четыре основные архитектуры:

- сети, предобученные без учителя (Unsupervised Pretrained Networks – UPN);
- сверточные нейронные сети (СНС);
- рекуррентные нейронные сети (РНС);
- рекурсивные нейронные сети.

В этой главе мы изучим эти архитектуры подробнее, чтобы понять, как они применяются на практике.

Наибольшее внимание мы уделим сетям двух типов: СНС для моделирования изображений и сетям с долгой краткосрочной памятью (рекуррентным) для моделирования последовательностей.

Сети, предобученные без учителя

В эту группу входят три архитектуры:

- автокодировщики;
- глубокие сети доверия (ГСД);
- порождающие состязательные сети (ПСС).

Замечание о роли автокодировщиков

Как уже было сказано в главе 3, автокодировщики играют важнейшую роль в глубоких сетях, потому что часто используются как составные части более крупных сетей. Но иногда они играют и самостоятельную роль.

Поскольку об автокодировщиках мы уже говорили много, перейдем к ГСД и ПСС.

Глубокие сети доверия

ГСД состоит из слоев ограниченной машины Больцмана (ОМБ), реализующих этап предобучения, и последующей сети прямого распространения для точной настройки. Архитектура ГСД показана на рис. 4.1.

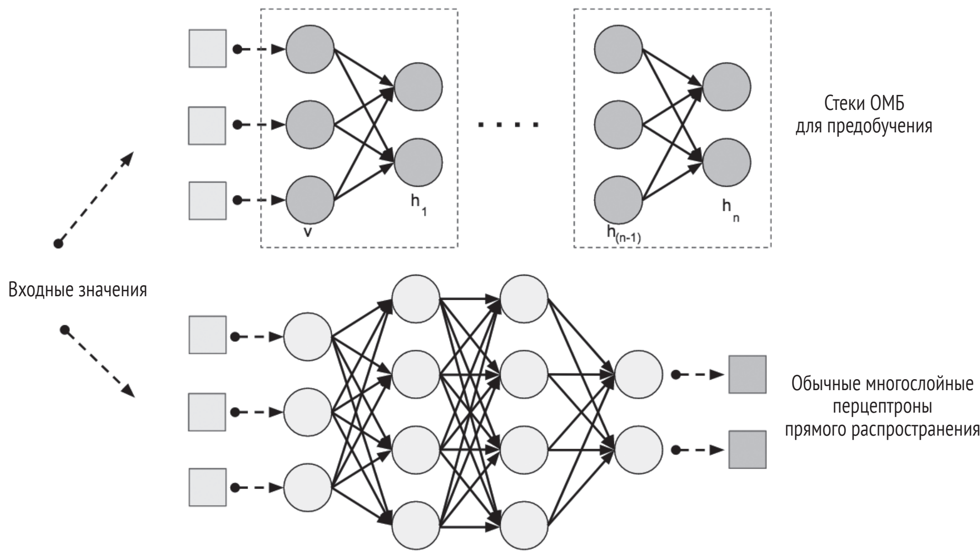


Рис. 4.1 ❖ Архитектура ГСД

В следующих разделах мы объясним, как ГСД пользуются ОМБ для подготовки более качественных обучающих данных.

Выделение признаков с помощью ОМБ-слоев

ОМБ служат для выделения высокоуровневых признаков из исходных входных векторов. Для этого веса связей со скрытыми блоками нужно настроить так, что когда мы предъявляем ОМБ входную запись и просим ее произвести реконструкцию, ОМБ генерирует нечто, весьма напоминающее оригинальный входной вектор. Хинтон называет этот эффект «данными, которые машины видят во сне».

Главное назначение ОМБ в контексте глубокого обучения вообще и ГСД в частности – обучиться этим высокоуровневым признакам набора данных без учителя. Показано, что нейронная сеть обучается лучше, если организовать каскад слоев предобучения ОМБ, в котором последующие ОМБ получают признаки, выделенные предшествующими.

Автоматическое обучение признакам высшего порядка. Обучение признакам без учителя считается стадией предобучения ГСД. Каждый скрытый слой ОМБ на стадии предобучения обучается все более сложным признакам, присутствующим в распределении данных. Эти признаки высшего порядка комбинируются нелинейными способами – и весь этот элегантный процесс можно назвать автоматическим конструированием признаков.

На рис. 4.2, 4.3 и 4.4 показана последовательность активаций, которую порождает ОМБ в процессе обучения на цифрах из набора MNIST.

В главе 6 мы подробнее расскажем о том, как были получены эти рисунки. Как видно, слой ОМБ выделяет фрагменты цифр в процессе обучения. В слоях следующих уровней они комбинируются во все более сложные (нелинейные) признаки.

Порождающий процесс моделирования в каждом слое ОМБ позволяет системе выделять из исходных данных, полученных с помощью векторизации, признаки

все более высокого уровня. Эти признаки распространяются по слоям ОМБ в одном направлении, так что верхний слой выдает самые интересные признаки.

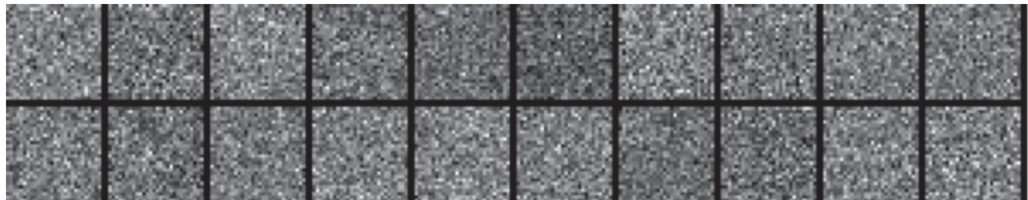


Рис. 4.2 ❖ Визуализация активаций в начале обучения

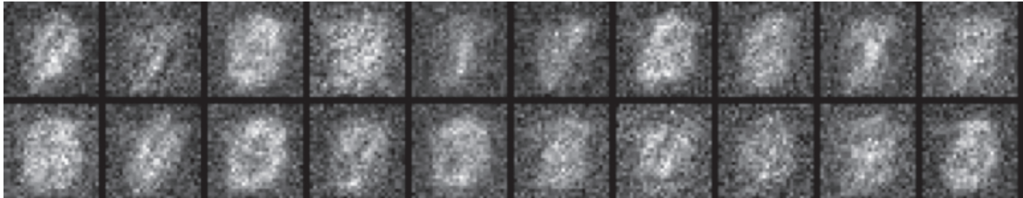


Рис. 4.3 ❖ На более поздних этапах начинают проявляться признаки

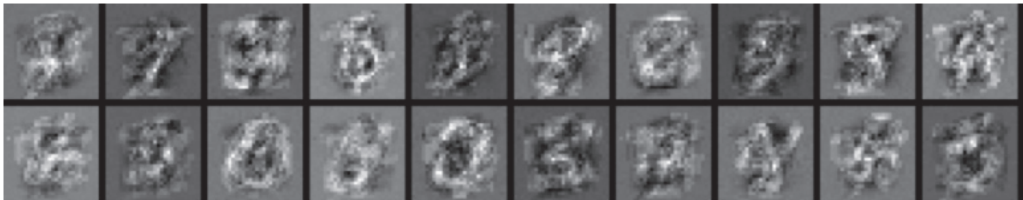


Рис. 4.4 ❖ Ближе к концу обучения мы видим части цифр

Инициализация сети прямого распространения. Затем выделенные признаки используются в качестве начальных весов в традиционной нейронной сети прямого распространения с обратным распространением ошибки. Благодаря этому алгоритм обучения находит лучшие области в пространстве параметров. Этот этап называется *точной настройкой* ГСД.

Точная настройка ГСД с помощью многослойной нейронной сети прямого распространения

На этапе точной настройки ГСД мы используем обычное обратное распространение с пониженной скоростью обучения. Этап предобучения можно рассматривать как неуправляемый поиск в пространстве параметров на основе исходных данных. Напротив, цель точной настройки – специализация сети и ее признаков к конкретной решаемой задаче (например, классификации).

Мягкое обратное распространение. На этапе предобучения с помощью ОМБ из данных были выделены признаки высшего порядка, которые можно использовать в качестве хороших начальных значений весов сети прямого распростра-

нения. Мы хотим еще уточнить эти веса и так получить окончательную модель нейронной сети.

Выходной слой. Обычно цель глубокой сети – обучиться набору признаков. Первый слой сети учится реконструировать оригинальный набор данных, а каждый последующий учится реконструировать распределение вероятности значений активации предыдущего слоя. Выходной слой нейронной сети связан с общей целевой функцией. Обычно это логистическая регрессия, в которой число признаков равно числу входов в последний слой, а число выходов равно числу классов.

Современное состояние ГСД

В этой книге мы не уделяем ГСД столько же внимания, сколько другим сетевым архитектурам. Дело в том, что в настоящее время для моделирования изображений гораздо чаще применяются СНС, поэтому мы решили оставить больше места для этой архитектуры.

Роль ГСД в возрождении глубокого обучения

Хотя в этой книге ГСД остались на заднем плане, в возрождении глубокого обучения они сыграли важнейшую роль. Группа Джеффри Хинтона в Торонтском университете долгое время совершенствовала методы моделирования изображений и добилась выдающихся успехов. Мы считаем необходимым отметить роль ГСД в развитии нейронных сетей.

Порождающие состязательные сети

Не сказать о ПСС невозможно[1]. Было показано, что ПСС прекрасно справляются с синтезом новых изображений после обучения на существующих. Эту идею можно обобщить и на другие предметные области:

- звук[2];
- видео[3];
- порождение изображений по текстовым описаниям[4].

ПСС – это сеть, в которой параллельно обучаются без учителя две модели. Главная отличительная особенность ПСС (и порождающих моделей вообще) состоит в том, что количество параметров относительно объема обучающих данных значительно меньше, чем обычно. Сеть вынуждена представлять данные эффективно, поэтому она лучше справляется с порождением данных, похожих на обучающие.

Обучение порождающих моделей, обучение без учителя и ПСС

Если бы у нас был большой набор обучающих изображений (как, например, набор данных ImageNet[5]), то мы могли бы построить порождающую нейронную сеть, которая выдает изображения (а не занимается классификацией). Сгенерированные изображения рассматривались бы как выборки из модели. Порождающая модель в ПСС как раз и генерирует такие изображения, в то время как вспомогательная дискриминантная сеть пытается сгенерированные изображения классифицировать как настоящие или синтетические.

---

1 Goodfellow et al., 2014. Generative Adversarial Networks // https://arxiv.org/abs/1406.2661.

2 https://github.com/usernaamee/audio-GAN.

3 Vondrick, Pirsiavash and Torralba, 2016. Generating Videos with Scene Dynamics // http://carlvondrick.com/tinyvideo/paper.pdf.

4 Zhang et al., 2016. StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks // https://arxiv.org/abs/1612.03242.

5 *http://image-net.org/*.

В ходе обучения ПСС мы стремимся обновлять параметры, так чтобы сеть генерировала более правдоподобные изображения. Наша цель – сделать изображения настолько реалистичными, чтобы дискриминантная сеть не могла отличить настоящие изображения от синтетических.

Так, при моделировании набора данных ImageNet с помощью ПСС количество примеров составляет примерно 100 миллионов. В процессе обучения набор объемом 200 ГБ сводится ко множеству параметров объемом порядка 100 МБ. Цель обучения – найти наиболее эффективное представление таких признаков, как группы похожих пикселей, границы и другие паттерны.

Дискриминантная сеть. В процессе моделирования изображений в роли дискриминантной сети обычно выступает стандартная СНС. Наличие вспомогательной дискриминантной сети позволяет ПСС обучать обе сети параллельно без учителя. Дискриминантная сеть принимает на входе изображение, а на выходе порождает его классификацию.

Градиент выхода дискриминантной сети по синтетическим входным данным показывает, какие небольшие изменения следует внести в синтетические данные, чтобы сделать их более реалистичными.

Порождающая сеть. Порождающая сеть в ПСС генерирует данные (или изображения) с помощью специального *антисверточного* (deconvolutional) *слоя* (подробнее об антисверточных сетях и слоях читайте во врезке ниже).

В процессе обучения в обеих сетях используется обратное распространение ошибки, чтобы обновить параметры порождающей сети и сделать генерируемые ей изображения более реалистичными – настолько, чтобы «обмануть» дискриминантную сеть.

Антисверточные сети

Сети этого типа были предложены Мэттью Зейлером (Matthew Zeiler) и Робом Фергусом (Rob Fergus) из Нью-Йоркского университета в ходе работы над сетью ZF Net и описаны в статье «Visualizing and Understanding Convolutional Neural Networks»[6] (2013). Антисверточная сеть позволяет исследовать активацию различных признаков и их связь с пространством входов (рис. 4.5).

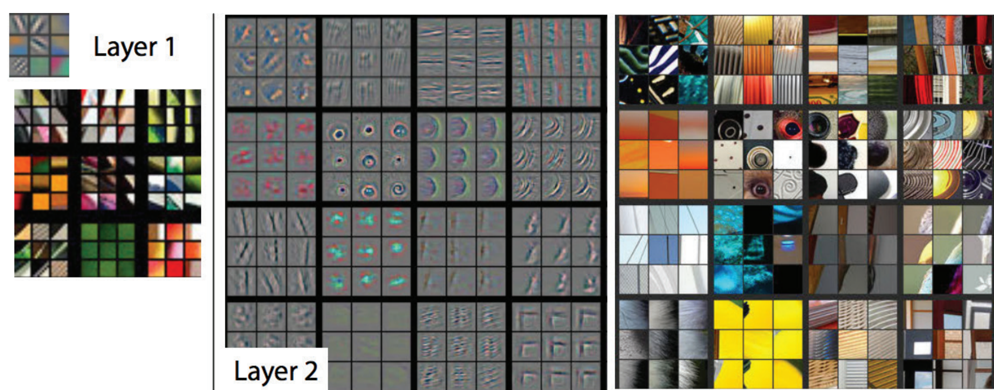


Рис. 4.5 ❖ Наглядное представление антисверточных слоев

[6] *https://arxiv.org/abs/1311.2901v3.*

Как показано на рис. 4.5, антисверточные слои в антисверточной сети (для краткости «deconvnet-слои») отображают признаки на пиксели изображения, т. е. выполняют операцию, противоположную той, что делает обычный сверточный слой. Именно этот аспект антисверточной сети позволяет генерировать изображения на выходе нейронной сети. Антисверточные сети обучаются без учителя послойно, как ГСД. В сети имеется несколько антисверточных слоев, и каждый последующий обучается на выходе предыдущего. Идея заключается в том, что выход слоя является разреженным представлением его входа.

Построение порождающих моделей и глубоких сверточных порождающих состязательных сетей

Один из вариантов ПСС – глубокая сверточная порождающая состязательная сеть[7] (ГСПСС, англ. DCGAN). На рис. 4.6 приведены изображения спален, порожденные ГСПСС.

Рис. 4.6 ❖ Изображения спален, порожденные сетью ГСПСС[8]

Эта сеть принимает случайные числа (с равномерным распределением) и порождает на выходе изображение из модели сети. В ответ на разные числа порождаются разные изображения спален.

Условные ПСС

Условные ПСС[9] также могут использовать метки класса и условно генерировать данные конкретного класса.

---

[7] *https://github.com/Newmu/dcgan_code.*

[8] Изображение взято из репозитория авторов ГСПСС на GitHub.

[9] Mirza and Osindero, 2014. Conditional Generative Adversarial Nets // https://arxiv.org/abs/1411.1784.

Сравнение ПСС с вариационными автокодировщиками

ПСС пытается классифицировать обучающие примеры как выбранные из реального или модельного распределения. Если предсказание дискриминантной сети свидетельствует о различии между распределениями, то порождающая сеть корректирует свои параметры. В конечном итоге генератор сходится к параметрам, которые воспроизводят реальное распределение, а дискриминатор не способен отличить подлинник от подделки.

В случае вариационных автокодировщиков (VAE) мы подходим к той же задаче с помощью вероятностных графических моделей, которые обучаются реконструировать вход без учителя, как было описано в главе 3. VAE пытается максимизировать нижнюю границу логарифмического правдоподобия данных, так чтобы сгенерированные изображения как можно больше походили на реальные.

Еще одно интересное отличие между ПСС и VAE – как именно порождаются изображения. В базовой ПСС порождается какое-то изображение, мы не можем заказать картинку с определенными признаками. Напротив, в VAE имеется конкретная схема кодирования-декодирования, позволяющая сравнить сгенерированное изображение с оригинальным. Побочный эффект – возможность затребовать порождение изображений определенного типа.

Проблемы порождающих моделей

Иногда изображения, созданные порождающей моделью, содержат шум. В случае VAE получаются слегка размытые изображения – это следствие способа их порождения. В случае ПСС порожденное изображение улавливает стиль, характерный для входных данных, но картинка в целом не вполне достоверна (т. е. изображение собаки мы получаем, но какое-то не такое).

Сверточные нейронные сети (СНС)

Цель СНС – отыскать в данных признаки высшего порядка посредством операции свертки. Они хорошо приспособлены к распознаванию объектов на изображениях и регулярно выигрывают соревнования по классификации изображений. С их помощью можно идентифицировать лица отдельных людей, дорожные знаки, утконосов и другие визуальные образы. СНС пересекаются с анализом текста благодаря оптическому распознаванию символов, но полезны также для анализа слов[10] как дискретных текстовых единиц. Хороши они и для анализа звуковых данных.

Эффективность СНС как средства распознавания изображений – одна из основных причин, объясняющих, почему мир признал мощь глубокого обучения. Как показывает рис. 4.7, СНС отлично справляются с задачей конструирования признаков, не зависящих от положения и (в какой-то мере) от угла поворота исходных данных.

СНС стоят за основными достижениями в машинном зрении, которое имеет очевидные приложения в беспилотных автомобилях, робототехнике, дронах и приспособлениях для слабовидящих.

[10] Kalchbrenner et al., 2016. Neural Machine Translation in Linear Time // https://arxiv.org/abs/1610.10099.

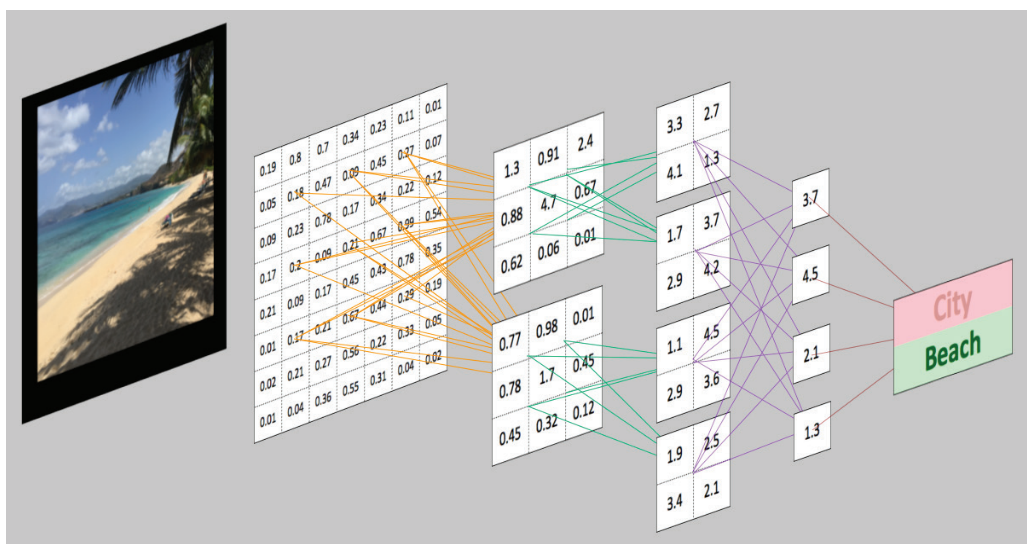


Рис. 4.7 ❖ СНС и компьютерное зрение

СНС и структурные закономерности в данных
СНС особенно полезны, когда во входных данных имеется некоторая структура. Например, изображения и звуковые данные с повторяющимися паттернами, когда между близкими входными значениями имеется пространственная связь. С другой стороны, табличные данные, экспортированные из реляционной СУБД, обычно не имеют пространственных структурных связей. Соседние столбцы просто хранятся таким образом в базе данных.

СНС нашли применение и в других задачах, например в переводе с одного естественного языка на другой и генерации текстов на естественном языке[11], а также в анализе тональности высказываний[12]. Свертка – весьма эффективное средство построения более устойчивого пространства признаков по сигналу.

Биологические корни

Биологические корни СНС – зрительная кора головного мозга животных[13]. Каждая клетка зрительной коры чувствительна к небольшому участку входного сигнала, который называется *рецептивным полем*. В совокупности эти небольшие участки покрывают все поле зрения. Клетки хорошо приспособлены для выявления сильной пространственной корреляции, присутствующей в изображениях, обрабатываемых мозгом, и играют роль фильтров, применяемых к пространству входов. В этой области мозга есть клетки двух видов. Простые клетки возбуждаются, когда обнаруживают паттерны, похожие на границы, а более сложные имеют большее рецептивное поле и инвариантны относительно положения паттерна.

[11] Gehring et al., 2016. A Convolutional Encoder Model for Neural Machine Translation // https://arxiv.org/abs/1611.02344.

[12] Nogueira dos Santos and Gatti, 2014. Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts // http://www.aclweb.org/anthology/C14-1008.

[13] Eickenberg et al., 2017. Seeing it all: Convolutional network layers map the function of the human visual system // http://www.sciencedirect.com/science/article/pii/S1053811916305481.

Интуитивное описание

Многослойные нейронные сети прямого распространения принимают на входе один одномерный вектор и преобразуют данные с помощью одного или нескольких скрытых слоев (полносвязных). Затем результат снимается с выходного слоя. Проблема такой сети в применении к изображениям состоит в том, что сеть плохо масштабируется. Рассмотрим, к примеру, моделирование набора данных CIFAR-10 (см. врезку ниже). Он состоит из изображений 32 × 32 пикселя с тремя цветовыми каналами. В результате получается 3072 веса на один нейрон первого скрытого слоя, а нам, наверное, хотелось бы иметь в этом слое побольше одного нейрона. А зачастую и слоев должно быть несколько, и тогда количество весов перемножается.

Что такое набор данных CIFAR-10?

CIFAR-10 – хорошо известный эталонный набор данных для классификации изображений[14], созданный Алексом Крижевским, Винодом Наиром и Джефри Хинтоном. Набор состоит из 60 000 цветных изображений, разбитых на 10 классов по 6000 изображений в каждом. Размер каждого изображения – 32 × 32 пикселя. В наборе 50 000 обучающих и 10 000 тестовых изображений. На рис. 4.8 показаны классы изображений.

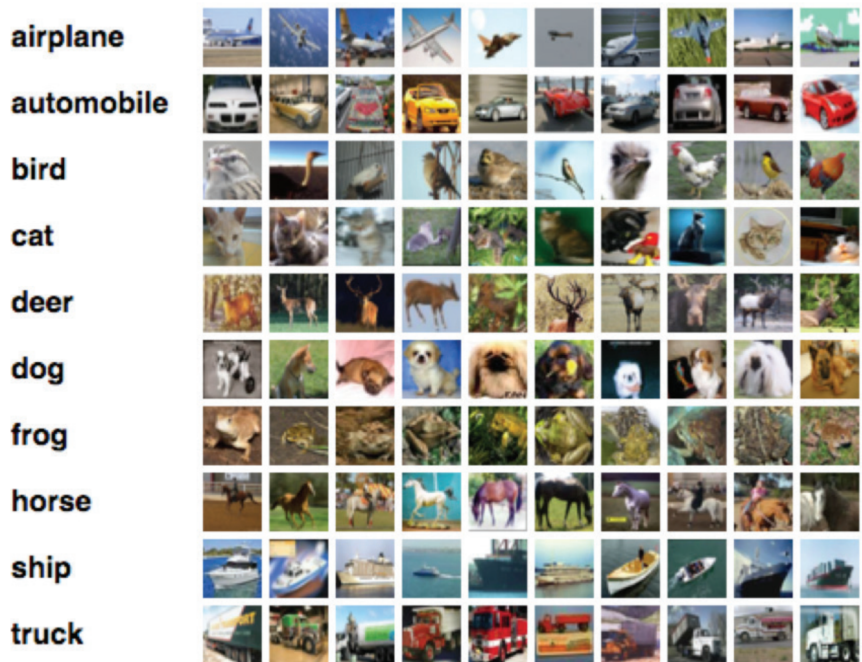


Рис. 4.8 ❖ Набор данных CIFAR-10

Классы не пересекаются, т. е. на изображении грузовика будет только грузовик и ничего более. Размер набора составляет примерно 170 МБ.

[14] http://www.cs.toronto.edu/~kriz/cifar.html.

Типичное изображение вряд ли будет меньше, чем 300 × 300 пикселей с 3 цветовыми каналами. Это получается 270 000 связей на один скрытый нейрон. Так что в полносвязной многослойной сети количество связей при моделировании данных изображений очень быстро возрастает. Но мы можем изменить архитектуру нейронной сети, так чтобы воспользоваться присущей изображению структурой. В СНС мы можем организовать нейроны в виде трехмерной структуры, имеющей три измерения: ширину, высоту и глубину.

Им соответствуют:

- ширина изображения в пикселях;
- высота изображения в пикселях;
- количество RGB-каналов.

Эту структуру можно рассматривать как трехмерный массив нейронов. Важно, как благодаря новым типам слоев СНС добивается вычислительной эффективности. Мы рассмотрим этот вопрос чуть ниже, а пока обратимся к высокоуровневому представлению архитектуры СНС.

Общий взгляд на архитектуру СНС

СНС получает входные данные, преобразует их с помощью ряда взаимосвязанных слоев и на выходе выдает набор вероятностей классов. Существует много вариантов архитектуры СНС, но все они основаны на чередовании слоев, показанных на рис. 4.9.

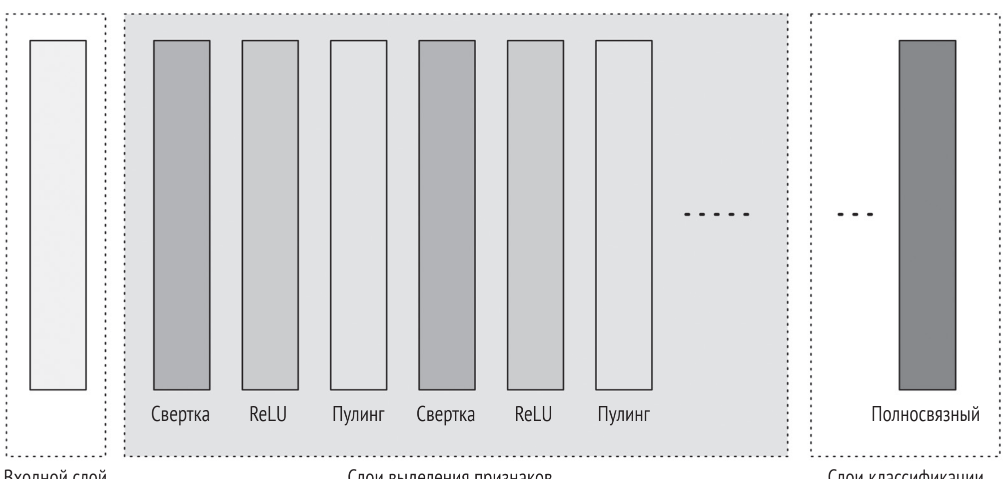


Рис. 4.9 ❖ Высокоуровневое представление архитектуры СНС

На рис. 4.9 показаны три основные группы:

1) входной слой;
2) слои выделения признаков (обучающиеся);
3) слои классификации.

Входной слой принимает трехмерный сигнал, обычно в виде прямоугольного изображения с глубиной, представляющей цветовые каналы (как правило, три канала в формате RGB).

Мини-пакет как четвертое измерение
Когда мы собираем несколько обучающих примеров в мини-пакет, появляется четвертое измерение – индекс примера в мини-пакете. Поэтому в DL4J массив обучающих данных, содержащий изображения, четырехмерный, а не трехмерный.

Слои выделения признаков имеют повторяющуюся структуру:

1. Сверточный слой. Блок линейной ректификации (ReLU), который на самом деле является функцией активации, показан здесь в виде слоя, поскольку так принято в литературе.
2. Пулинговый слой.

Эти слои находят признаки в изображениях и последовательно строят признаки высших порядков. Это вполне соответствует общему принципу глубокого обучения, согласно которому признаки ищутся автоматически, в результате обучения, а не конструируются вручную, как в традиционном машинном обучении.

Наконец, мы имеем один или несколько полносвязных слоев классификации, которые принимают признаки высшего порядка и порождают вероятности (оценки). Каждый нейрон любого их этих слоев связан со всеми нейронами предыдущего слоя. На выходе обычно получается двумерный массив размера $[b \times N]$, где b – количество примеров в мини-пакете, в N – число интересующих нас классов.

Пространственная организация нейронов

Напомним, что в традиционных многослойных нейронных сетях слои полносвязные, т. е. каждый нейрон предыдущего слоя связан со всеми нейронами следующего. Нейроны в слоях СНС располагаются в трех измерениях, в соответствии со структурой входных данных. Под глубиной здесь понимается третье пространственное измерение, а не количество слоев нейронной сети.

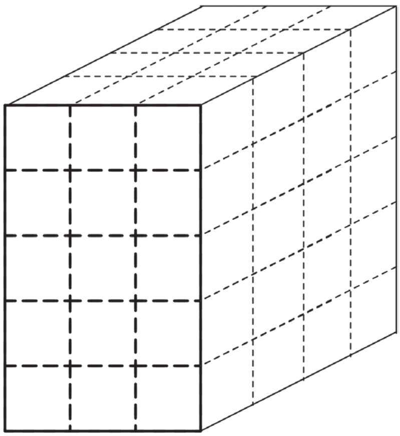


Рис. 4.10 ❖ Входной слой трехмерной области

Эволюция связей между слоями

Еще одно отличие заключается в способе соединения слоев в сверточной архитектуре. Нейроны каждого слоя связаны только с небольшим числом нейронов предшествующего слоя. В СНС сохранена слоистая архитектура, как в традиционных многослойных сетях, но типы слоев другие. Каждый слой преобразует трехмерную входную область предыдущего слоя в трехмерный массив значений активации, применяя некоторую дифференцируемую функцию, с параметрами или без, как показано на рис. 4.10.

Входной слой

Входной слой – это то место, куда загружаются и где хранятся исходные данные обрабатываемого сетью изображения. Эти данные характеризуются шириной, высотой и числом каналов. Обычно каналов три, по числу RGB-компонент каждого пикселя.

Сверточные слои

Сверточные слои – основные строительные блоки в архитектуре СНС. Как показано на рис. 4.11, сверточные слои преобразуют входные данные, воздействуя на небольшой участок (патч) локально связанных нейронов предыдущего уровня. Слой вычисляет свертку области нейронов входного слоя и локальных весов.

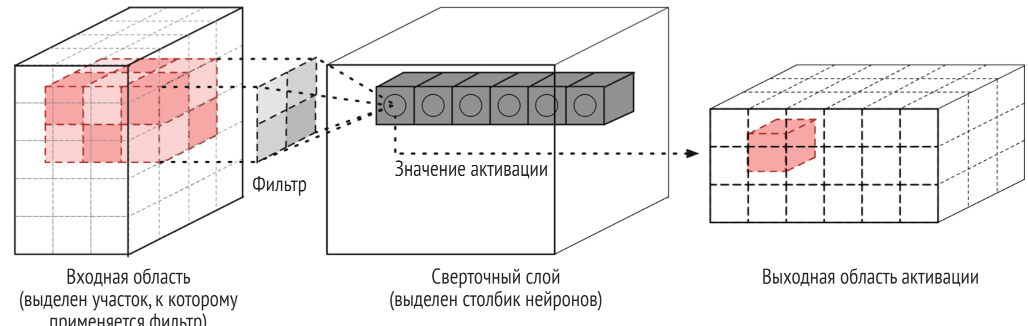


Рис. 4.11 ❖ Сверточный слой с входной и выходной областями

Результат на выходе обычно имеет такие же (или меньшие) пространственные измерения, но размерность по глубине иногда больше. Рассмотрим основную операцию, выполняемую этими слоями, – *свертку*.

Свертка

Математически операция свертки описывает правило объединения двух информационных массивов. Она важна в физике и в математике и посредством преобразования Фурье перекидывает мост между пространственно-временной и частотной областями. Мы получаем вход, применяем к нему ядро свертки и получаем на выходе карту признаков.

Показанная на рис. 4.12 операция свертки является *детектором признаков* в СНС. Входом свертки могут быть исходные данные или карта признаков, вычисленная другой сверткой. Часто она интерпретируется как фильтр, с помощью которого ядро отфильтровывает из данных определенные виды информации; например, фильтр границ пропускает только информацию о присутствующих в изображении границах.

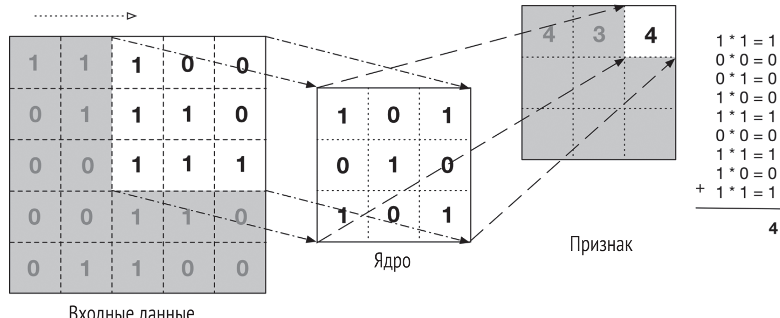


Рис. 4.12 ❖ Операция свертки

На рисунке показано, что ядро скользит по входным данным, порождая результаты свертки – признаки. На каждом шаге ядро сворачивается с входными данными в области под ним, порождая одно значение в выходной карте признаков. На практике величина выходного сигнала велика, если искомый признак присутствует во входных данных.

Набор весов в сверточном слое принято называть фильтром (или ядром). Этот фильтр сворачивается с входом, и в результате получается карта признаков (или карта активации). Сверточные слои производят преобразования областей входных данных, которые являются функциями активаций в этой области и параметров (весов и смещений нейронов). Карты активации каждого фильтра складываются в стопку вдоль третьего измерения (глубины), так что в итоге получается трехмерная выходная область.

В сверточных слоях имеются параметры слоя и дополнительные гиперпараметры. Для обучения параметров каждого слоя применяется градиентный спуск, так чтобы оценки классов были согласованы с метками в обучающем наборе. Перечислим основные компоненты сверточных слоев:

- фильтры;
- карты активации;
- разделение параметров;
- гиперпараметры каждого слоя.

Фильтры

Параметры сверточного слоя определяют набор его фильтров. Фильтр – это матрица, ширина и высота которой меньше ширины и высоты входных данных.

Размеры фильтров в приложениях для обработки естественного языка

Размер фильтра может совпадать с размером входных данных, но обычно только в одном направлении, а не в обоих. Об этом следует помнить, когда СНС применяются к обработке естественного языка (ОЕЯ, англ. NLP).

В процессе применения фильтры сдвигаются по ширине и по высоте входной области, как показано на рис. 4.12. Кроме того, они применяются к каждому вертикальному сечению входной области. Результатом применения фильтра является сумма произведений элементов фильтра на соответственные элементы участка входной области под ним.

Число фильтров и карты активации

Результат применения фильтра к входной области называется *картой активации*, или *картой признаков*. На диаграммах СНС мы часто видим множество небольших карт активации, а как они получаются, не всегда понятно.

Число фильтров – это гиперпараметр каждого сверточного слоя. Он управляет тем, сколько карт активации порождает слой и, значит, каким будет вход следующего слоя. Его можно рассматривать как третье измерение (число карт активации) трехмерной выходной области. Число фильтров можно задавать произвольно, но одни значения работают лучше, другие хуже.

Архитектура СНС устроена так, что обученные фильтры дают максимальную активацию в ответ на пространственно-локальные входные паттерны. Это значит, что каждый фильтр обучается таким образом, чтобы активироваться только в том случае, когда паттерны (или признаки) встречаются в обучающих данных

в его рецептивном поле. В последующих слоях СНС мы встретим фильтры, способные распознавать нелинейные комбинации признаков и для которых масштаб области, где распознаются паттерны, постепенно увеличивается. Показано, что для высокоэффективных сверточных архитектур (которые мы увидим в следующем разделе) глубина сети является важным фактором.

Карты активации

Напомним (см. главу 1), что активация положительна, если нейрон решил пропустить информацию. Она зависит от весов связей, поданных функции активации, и самой этой функции. Говоря, что фильтр «активируется», мы имеем в виду, что он пропускает информацию с входа на выход.

На прямом проходе по СНС каждый фильтр сдвигается вдоль пространственных измерений входной области (по ширине и высоте). В результате получается двумерный выход, который называется картой активации конкретного фильтра. На рис. 14.13 показано, как эта карта активации соотносится с введенным ранее понятием признака, являющимся результатом свертки.

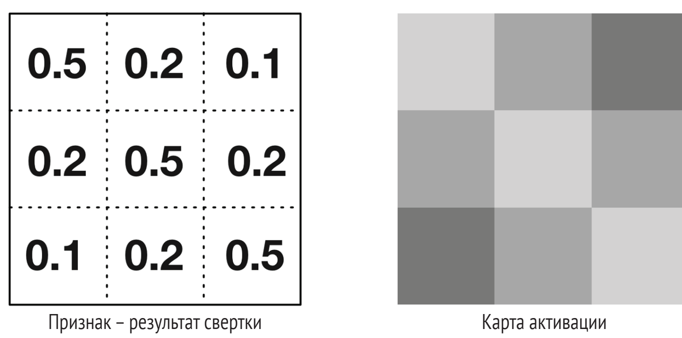


Рис. 4.13 ❖ Свертки и карты активации

Карта активации справа нарисована по-другому – в соответствии с принятой в литературе традицией.

Карты активации

Иногда в литературе карты активации называются *картами признаков*, но мы будем придерживаться первого термина.

Для вычисления карты активации фильтр сдвигается вдоль измерения глубины. Вычисляется скалярное произведение элементов фильтра и элементов находящейся под ним части входной области. Фильтр представляет собой веса, умножаемые на скользящее окно входных активаций. Сеть обучается находить фильтры, которые активируются, когда видят определенные признаки в некоторой пространственной позиции входных данных.

Трехмерная выходная область для сверточного слоя создается путем создания стопки этих карт активации, как показано на рис. 4.14. Элементы выходной области рассматриваются как выходной сигнал нейрона, который видит только небольшое окно входной области.

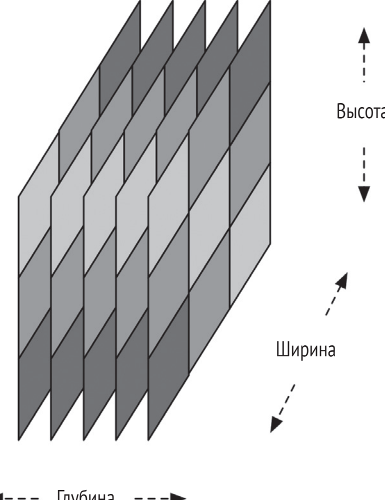


Рис. 4.14 ❖ Выходная область активации сверточного слоя

В некоторых случаях этот выход является результатом применения параметров, общих для всех нейронов в одной и той же карте активации. Каждый нейрон, принимающий участие в генерации выходной области, связан только с небольшим участком входной области, как показано на рис. 4.15.

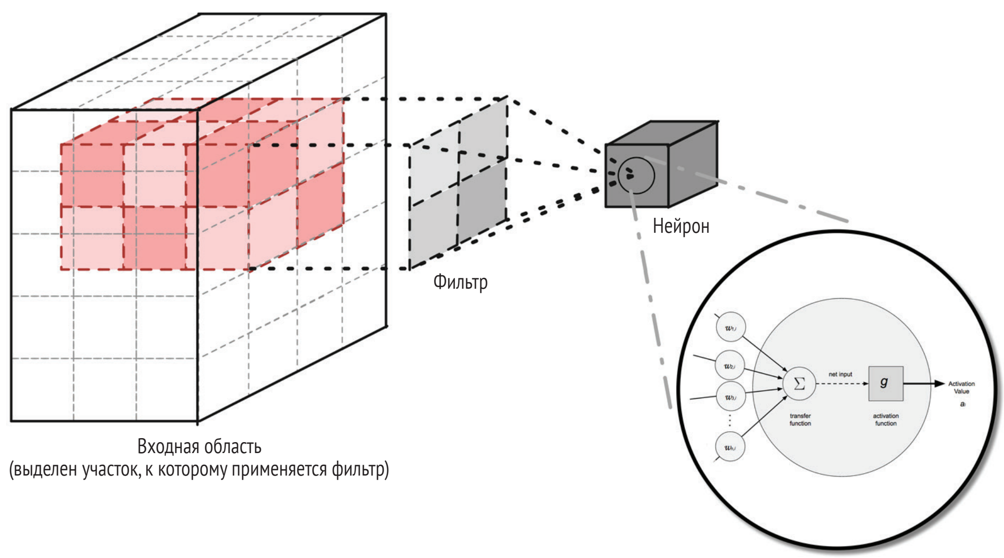


Рис. 4.15 ❖ Генерация выходной области активации

Локальная связность этого процесса управляется гиперпараметром *рецептивное поле*, который задает ширину и высоту области, к которой применяется фильтр.

Управление локальной связностью с помощью рецептивного поля

Нейроны слоя связаны с небольшим пространственным участком входной области, но глубина этого участка всегда совпадает с глубиной области. Следовательно, по глубине мы всегда имеем полную связность. Рассмотрим это на примере набора изображений CIFAR-10, о котором упоминалось выше.

В этом случае размер входной области 32 × 32 × 3, а рецептивное поле задано как 5 × 5. Каждый нейрон сверточного слоя имеет 5 × 5 × 3 связей с элементами входной области, что дает 5 * 5 * 3 = 75 весов.

Отметим, что глубина входной области равна 3. Участок, видимый нейрону, по ширине и высоте меньше изображения, но глубина всегда неизменна. Несмотря на локальную связность нейронов сверточного слоя, сами нейроны остаются такими же, как и раньше, – мы по-прежнему вычисляем скалярное произведение весов с входом, применяя нелинейную функцию.

Единственное отличие состоит в том, что нейрон теперь связан только с подмножеством входных данных, а не с каждым нейроном предыдущего слоя, как в традиционных многослойных нейронных сетях.

Фильтр определяет небольшую ограниченную область, по которой генерируются карты активации. Это позволяет обеспечить качественное выделение признаков, сократив количество обучаемых параметров. При этом сверточные слои еще уменьшают это количество благодаря технике *разделения параметров*.

Разделение параметров

В СНС для управления общим числом параметров применяется техника *разделения параметров*, что позволяет сократить время обучения. Назовем двумерную срезку входной области вдоль измерения глубины «сечением» и потребуем, чтобы у нейронов в каждом сечении были одинаковые веса и смещения. В результате число параметров для данного сверточного слоя существенно уменьшается.

Мы не можем воспользоваться преимуществами разделения параметров, когда обучающие входные изображения имеют специфическую структуру с центром. Этот эффект особенно заметен для лиц, поскольку мы ожидаем найти определенный признак в определенном месте (если лицо центрировано). В таком случае применять разделение параметров не стоит. Кстати говоря, именно благодаря разделению параметров СНС инвариантны к параллельному переносу.

Визуализация обученных фильтров

На рис. 4.16 показаны 96 обученных фильтров размера 11 × 11 × 3. Благодаря разделению параметров мы можем обучить сеть обнаружению горизонтальных границ только в одном месте и, воспользовавшись позиционно-инвариантной природой изображения, не обучать тот же признак во всех точках.

Возьмем двумерное изображение. Если разбить его на четыре части, то нейронная сеть обучится инвариантным относительно положения признакам. Инвариантность объясняется тем, что сеть разбивает данные на квадранты. Затем она обучается на частях изображения и агрегирует результаты в пулинговом слое. Тем самым сеть обучается всему представлению, не локальному относительно конкретного набора признаков. Мы вернемся к этому вопросу в главе 6.

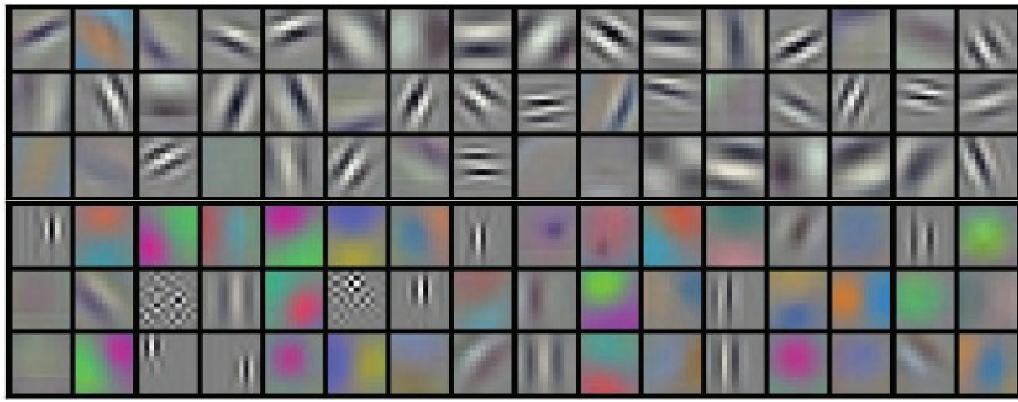


Рис. 4.16 ❖ Примеры фильтров, обученных в работе Krizhevsky et al.[15]
(96 фильтров размера 11 × 11 × 3)

Вращательная инвариантность признаков в СНС
Обученные признаки по построению являются позиционно-инвариантными, но в общем случае они не обладают инвариантностью относительно вращения. Впрочем, добиться некоторой вращательной инвариантности можно с помощью подходящего пополнения данных.

Функции активации ReLU как слои

В СНС часто используются ReLU-слои. ReLU-слой применяет к входным данным поэлементную функцию активации с отсечением в нуле, например max(0,x), порождая выход такой же размерности, как вход.

Типы слоев и функции активации в DL4J
В DL4J слой определяется типом функции активации (хотя это не всегда отражено в имени класса слоя). Функции активации в DL4J встроены в сами слои. В других библиотеках, например Caffe, используются отдельные слои активации.

Применение этой функции к входной области изменяет значения пикселей, но не пространственные размеры входных данных. У ReLU-слоев нет ни параметров, ни дополнительных гиперпараметров.

Гиперпараметры сверточных слоев

Следующие гиперпараметры[16] определяют пространственную организацию и размер выхода сверточного слоя:

- размер фильтра (ядра), т. е. рецептивное поле;
- глубина выхода;
- шаг;
- заполнение нулями.

---

[15] Krizhevsky et al., 2012. ImageNet Classification with Deep Convolutional Neural Networks // https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[16] В СНС часто встречается также параметр «наращивание» (dilation), но в версии DL4J 0.7 он не поддерживается.

В этом разделе объясняется, как работают эти гиперпараметры. А в главе 7 мы расскажем о настройке слоев СНС.

Размер фильтра. Фильтр невелик по сравнению с входным изображением. Например, в первом сверточном слое может применяться фильтр размера 5 × 5 × 3, имеющий ширину и высоту по 5 пикселей, а глубину 3 цветовых канала (предполагается, что входное изображение представлено в 3-канальном формате RGB).

Глубина выхода. Мы можем вручную выбрать глубину выходной области. Гиперпараметр «глубина» задает количество нейронов в сверточном слое, связанных с одной и той же входной областью.

Границы и активация

Различные нейроны вдоль третьего измерения (глубины) обучаются активироваться в ответ на созданные входные данные (например, цвет или границы).

Множество нейронов, связанных с одним и тем же участком входной области, называется столбиком (depth column).

Шаг. Этот гиперпараметр задает расстояние, на которое сдвигается окно фильтра на одном шаге. При каждом применении фильтра к входной области создается новый столбик в выходной области. Чем меньше величина шага (например, 1 означает, что фильтр сдвигается на один пиксель), тем больше будет выходных столбиков. При этом рецептивные поля столбиков сильнее перекрываются. Наоборот, чем больше величина шага, тем меньше перекрытие рецептивных полей и пространственный размер выходной области.

Дополнение нулями. Этот гиперпараметр управляет поведением фильтра на границах входной области и влияет на размер выходной области.

Пакетная нормировка и слои

Для ускорения обучения СНС можно нормировать активации предыдущего слоя в каждом пакете[17], т. е. применить преобразование, в результате которого средняя активация будет близка к 0.0, а стандартное отклонение – к 1.0.

Было показано, что пакетная нормировка ускоряет обучение (https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md). Применяя нормировку к каждому мини-пакету входных примеров, мы можем задавать гораздо более высокую скорость обучения. Пакетная нормировка[18] также уменьшает чувствительность обучения к начальным весам и выступает в роли регуляризатора (устраняя необходимость в других видах регуляризации). Пакетная нормировка применялась и в глубоких LSTM-сетях[19], которые мы обсудим ниже в этой главе.

Пулинговые слои

Пулинговые слои часто вставляются между соседними сверточными слоями, поскольку они уменьшают пространственный размер (ширину и высоту) представления данных и тем самым предотвращают переобучение. Пулинговый слой независимо воздействует на каждое сечение входных данных.

---

[17] Ioffe and Szegedy, 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift // https://arxiv.org/abs/1502.03167.

[18] https://www.quora.com/Why-does-batch-normalization-help.

[19] Cooijmans et al., 2016. Recurrent Batch Normalization // https://arxiv.org/abs/1603.09025.

Типичные операции понижающей передискретизации
Чаще всего встречается операция взятия максимума, а следующая по частоте – операция усреднения.

Пулинг с операцией `max()` называется *max-пулингом*. Если размер фильтра равен 2 × 2, то `max()` вычисляет максимум из четырех чисел в области фильтра. На глубину эта операция не влияет.

Пулинговые слои применяют пространственную понижающую передискретизацию к входным данным. Это означает, что если входное изображение имело размер 32 × 32 пикселя, то выходное будет меньше по ширине и по высоте (например, 16 × 16). Чаще всего в пулинговых слоях применяются фильтры 2 × 2 с шагом 2, в результате чего размер входной области уменьшается вдвое по каждому измерению. Это значит, что пулинг отбрасывает 75% активаций.

У пулинговых слоев нет собственных параметров, но есть дополнительные гиперпараметры. Отсутствие параметров связано с тем, что вычисляется фиксированная функция входных данных. К пулинговым слоям обычно не применяется заполнение нулями.

Полносвязные слои

Такой слой вычисляет вероятности классов, выдаваемые сетью на выходе (например, это может быть последний слой сети). Размер выхода равен $[1 \times 1 \times N]$, где N – число вычисляемых выходных классов. В случае набора данных CIFAR N равно 10, т. к. в этот набор входят изображения, разбитые на 10 классов. Каждый нейрон полносвязного слоя связан со всеми нейронами предыдущего слоя.

У полносвязных слоев имеются обычные параметры и гиперпараметры. К входным данным применяется преобразование, зависящее от параметров (весов связей и смещений нейронов).

Несколько полносвязных слоев
В некоторых архитектурах СНС применяется несколько полносвязных слоев, находящихся в конце сети. Примером может служить сеть AlexNet: в ней есть два полносвязных слоя, за которыми следует слой softmax.

Другие применения СНС

Помимо двумерных изображений, СНС применялись и к трехмерным наборам данных, например:

- данные МРТ[20];
- трехмерные объекты[21];
- графы[22];
- обработка естественных языков[23].

[20] Milletari, Navab and Ahmadi, 2016. V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation // https://arxiv.org/abs/1606.04797.

[21] Maturana and Scherer, 2015. VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition // https://www.ri.cmu.edu/pub_files/2015/9/voxnet_maturana_scherer_iros15.pdf.

[22] Henaff, Bruna and LeCun, 2015. Deep Convolutional Networks on Graph-Structured Data // https://arxiv.org/abs/1506.05163.

[23] Conneau et al., 2016. Very Deep Convolutional Networks for Text Classification // https://arxiv.org/abs/1606.01781.

Свойство позиционной инвариантности СНС оказалось полезно во всех этих задачах, поскольку необязательно вручную кодировать появление признаков в определенных местах входного вектора.

Самые известные СНС

Ниже приведен перечень самых популярных архитектур СНС.

- LeNet[24]:
  - одна из самых первых успешных архитектур СНС;
  - разработана Яном Лекуном;
  - первоначально применялась для распознавания цифр в изображениях.
- AlexNet[25]:
  - способствовала популяризации СНС в компьютерном зрении;
  - разработана Алексом Крижевским, Ильей Суцкевером и Джеффри Хинтоном;
  - победила в соревновании ILSVRC 2012.
- ZF Net[26]:
  - победила в соревновании ILSVRC 2013;
  - разработана Мэттью Зейлером и Робом Фергусом;
  - была предложена антисверточная сеть для визуализации.
- GoogLeNet[27]:
  - победила в соревновании ILSVRC 2014;
  - разработана Кристианом Сегеди и его группой в компании Google;
  - кодовое название «Inception», в одном из вариантов имеются 22 слоя.
- VGGNet[28]:
  - заняла второе место в соревновании ILSVRC 2014;
  - разработана Кареном Симоняном и Эндрю Циссерманом;
  - продемонстрировала, что глубина сети имеет важнейшее значение для качества ее работы.
- ResNet[29]:
  - очень глубокая сеть (до 1200 слоев);
  - победила в соревновании ILSVRC 2015 по классификации изображений.

Итоги

СНС появились из-за необходимости специализированного метода выделения признаков из изображений. Мы видели, что ее слои хорошо находят признаки,

[24] LeCun et al., 1998. Gradient-based learning applied to document recognition // http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

[25] Krizhevsky, Sutskever and Hinton, 2012. ImageNet Classification with Deep Convolutional Neural Networks // https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[26] Zeiler and Fergus, 2013. Visualizing and Understanding Convolutional Networks // https://arxiv.org/pdf/1311.2901v3.pdf.

[27] Szegedy et al., 2015. Going Deeper with Convolutions // https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf.

[28] Simonyan and Zisserman, 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition // https://arxiv.org/pdf/1409.1556v6.pdf.

[29] He et al., 2015. Deep Residual Learning for Image Recognition // https://arxiv.org/abs/1512.03385.

в каких бы столбцах они ни находились. Мы видели, как сверточные слои, пулинговые слои и обычные полносвязные слои совместно выполняют классификацию изображений. Теперь перейдем к архитектуре, предназначенной для моделирования временных рядов: рекуррентным нейронным сетям.

Рекуррентные нейронные сети

Рекуррентные нейронные сети (РНС) – семейство сетей прямого распространения. Их отличие от других подобных сетей состоит в том, что они могут передавать информацию между временными шагами. Вот любопытное объяснение рекуррентных нейронных сетей, предложенное Юргеном Шмидхубером:

> [Рекуррентные нейронные сети] позволяют производить как параллельные, так и последовательные вычисления и в принципе способны вычислить все, что может вычислить традиционный компьютер. Но, в отличие от традиционных компьютеров, рекуррентные нейронные сети напоминают человеческий мозг, представляющий собой большую сеть взаимосвязанных нейронов с обратной связью, которая каким-то образом обучается транслировать входной поток чувственных ощущений, полученных на протяжении всей жизни, в последовательность полезных моторных реакций. Мозг – потрясающий образец для подражания, поскольку он умеет решать задачи, недоступные современным машинам.

Поначалу обучать эти сети было трудно, но недавние достижения (оптимизация, сетевые архитектуры, распараллеливание и графические процессоры) сделали их доступными для практического применения.

Рекуррентная нейронная сеть выбирает по одному вектору из входной последовательности и моделирует его. Это позволяет сети сохранять состояние на протяжении некоторого окна входных векторов. Моделирование временных рядов – отличительная особенность РНС.

Моделирование времени

Рекуррентные нейронные сети являются полными по Тьюрингу, т. е. могут имитировать любую программу (при наличии достаточно большого количества весов). Если рассматривать нейронные сети как механизм оптимизации функций, то РНС можно назвать «оптимизацией программ». РНС подходят для моделирования функций, вход и (или) выход которых состоят из векторов, между значениями которых имеются временные зависимости. РНС моделируют временной аспект данных посредством создания циклов в сети (отсюда и название «рекуррентные»).

Заблудившийся во времени

Многие средства классификации (метод опорных векторов, логистическая регрессия и стандартные сети прямого распространения) особенно успешно применяются, когда между входными данными нет временных зависимостей, т. е. они считаются независимыми. Существуют варианты этих инструментов, которые улавливают временную динамику путем моделирования входного окна (например, предыдущее, текущее и следующее значения рассматриваются как один входной вектор).

Недостатком этих методов является тот факт, что предположение о независимости входов не позволяет модели уловить долгосрочные временные зависи-

мости. У скользящего окна ограниченная ширина, с его помощью невозможно уловить эффекты, длительность которых превышает размер окна. В качестве примера приведем то, как машина понимает речь и с течением времени может дать связный ответ. Хорошо обученная рекуррентная нейронная сеть смогла пройти знаменитый тест Тьюринга, смысл которого состоит в том, чтобы убедить человека, что он разговаривает с живым собеседником, а не с машиной.

Обратная связь по времени и циклические связи

В рекуррентных нейронных сетях возможны циклические связи. Благодаря этому они могут моделировать поведение во времени, что является преимуществом в таких предметных областях, как временные ряды, обработка естественного языка, звуковых данных и текста.

Замечание о связях, направленных от выходного слоя к скрытому
На практике такая схема связей встречается достаточно редко, и в DL4J она тоже не используется. Но мы считаем необходимым упомянуть ее для полноты. Чаще всего можно встретить связи между нейронами рекуррентных слоев, относящихся к соседним временным шагам.

В этих задачах данные внутренне упорядочены, и более поздние значения зависят от более ранних. РНС позволяет организовать обратную связь и тем самым уловить эти временные эффекты. Основной областью применения архитектуры РНС являются временные ряды.

В РНС имеется петля обратной связи, которая позволяет обучаться на последовательностях, в т. ч. переменной длины. РНС содержит дополнительную матрицу связей между временными шагами, в которой улавливаются временные связи, присутствующие в данных.

РНС обучается порождать последовательности, в которых выход на каждом временном шаге зависит как от текущего входа, так и от входов на всех предыдущих временных шагах. Обычно РНС вычисляет градиент с помощью алгоритма *обратного распространения во времени* (backpropagation through time – BPTT). Детали этого алгоритма мы обсудим далее в этой главе.

Последовательности и временные ряды

Существует много задач, в которых модель должна выводить последовательность векторов:

- подписывание изображений[30];
- синтез речи[31];
- порождение музыки[32];
- видеоигры;
- моделирование языка[33];
- модели порождения текста на уровне символов[34].

---

[30] http://cs.stanford.edu/people/karpathy/sfmltalk.pdf.

[31] Graves and Jaitly, 2014. Towards End-to-End Speech Recognition with Recurrent Neural Network // http://proceedings.mlr.press/v32/graves14.pdf.

[32] Nayebi and Vitelli, 2015. GRUV: Algorithmic Music Generation using Recurrent Neural Networks // https://cs224d.stanford.edu/reports/NayebiAran.pdf.

[33] http://www.fit.vutbr.cz/~imikolov/rnnlm/thesis.pdf.

[34] https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

В других задачах присутствует последовательность входных векторов:

- предсказание временных рядов;
- анализ видеоряда;
- извлечение музыкальной информации.

Наконец, есть задачи, в которых последовательности векторов присутствуют как на входе, так и на выходе:

- перевод с естественного языка[35];
- поддержание диалога;
- управление роботом.

РНС отличается от других глубоких сетей типом входных данных, которые они способны моделировать (нефиксированный вход):

- нефиксированные шаги вычислений;
- нефиксированный размер выхода;
- возможность оперировать последовательностями векторов, например видеокадрами.

Особенности входа и выхода модели

В традиционном машинном обучении мы имеем один входной вектор, фиксированного размера и размер выхода также фиксирован. Именно так обычно строятся классификаторы изображений и табличных данных.

В рекуррентных нейронных сетях на входе может быть несколько векторов, по одному для каждого временного шага, и у каждого вектора может быть несколько столбцов. Ниже приведены примеры последовательностей входных и выходных векторов в различных рекуррентных сетях.

- Один ко многим: последовательность на выходе. Например, в задаче подписывания изображений на вход подается изображение, а на выходе получается последовательность слов.
- Многие к одному: последовательность на входе. Например, в задаче анализа тональности на вход подается предложение – последовательность слов.
- Многие ко многим. Например, в задаче классификации видеоряда следует снабдить меткой каждый кадр.

Посмотрим теперь, как представляются входные данные.

Трехмерный вход

Входные данные рекуррентной нейронной сети имеют больше измерений, чем стандартная модель в машинном обучении. В этом они концептуально похожи на СНС. Вход имеет три измерения:

1) размер мини-пакета;
2) количество столбцов вектора на каждом временном шаге;
3) количество временных шагов.

Размер мини-пакета – это количество входных примеров (коллекций временных рядов из одного источника), моделируемых за раз. Количество столбцов соответствует количеству столбцов-признаков в обычном входном векторе. С помощью количества временных шагов мы представляем изменения входного

[35] Sutskever, Vinyals and Le, 2014. Sequence to Sequence Learning with Neural Networks // https://arxiv.org/abs/1409.3215.

вектора во времени. Это временной аспект входных данных. В терминах предыдущего раздела количество временных шагов, большее 1, соответствует разновидности архитектуры «многие к».

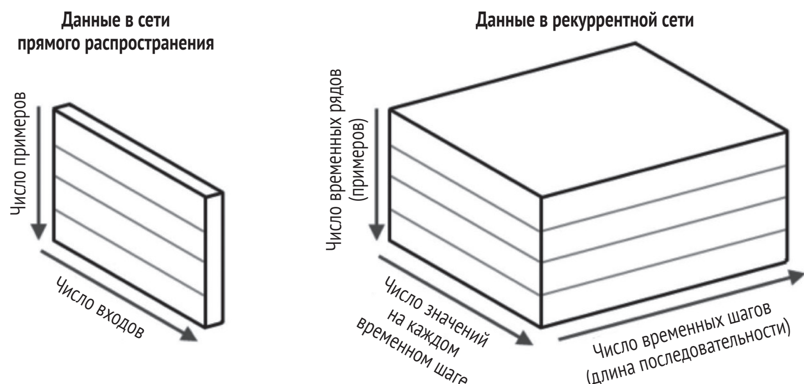


Рис. 4.17 ❖ Сравнение входов обычной и рекуррентной нейронных сетей

Неполные временные ряды и маскирование

Как уже было сказано, входные данные РНС характеризуются временными шагами, помимо признаков во входном векторе. На рис. 4.18 это показано наглядно.

Векторы-столбцы

	Values
albumin	0.0
alp	1.0
alt	0.5
ast	0.0
...	

Временные шаги

Векторы-столбцы

	0	1	2	3	4	...
albumin	0.0	0.0	0.5	0.0	0.0	
alp	0.0	0.1	0.0	0.0	0.2	
alt	0.0	0.0	0.0	0.9	0.0	
ast	0.0	0.0	0.0	0.0	0.4	
...						

Рис. 4.18 ❖ Временной аспект входных данных РНС

Значения в определенном столбце присутствуют не на каждом временном шаге, особенно в тех случаях, когда описательные данные (например, столбцы, взятые из статической базы данных) перемешаны с данными временного ряда (например, с ежеминутными измерениями пульса пациента в палате интенсивной терапии). Для тех случаев, когда значения могут отсутствовать, необходимо *маскирование*, чтобы DL4J знала, где в векторе находятся реальные данные. Для этого мы заводим дополнительную матрицу-маску, где отмечаются временные шаги, на которых имеются входные данные хотя бы в одном столбце (см. рис. 4.19).

В главе 7 будут приведены примеры задания таких масок.

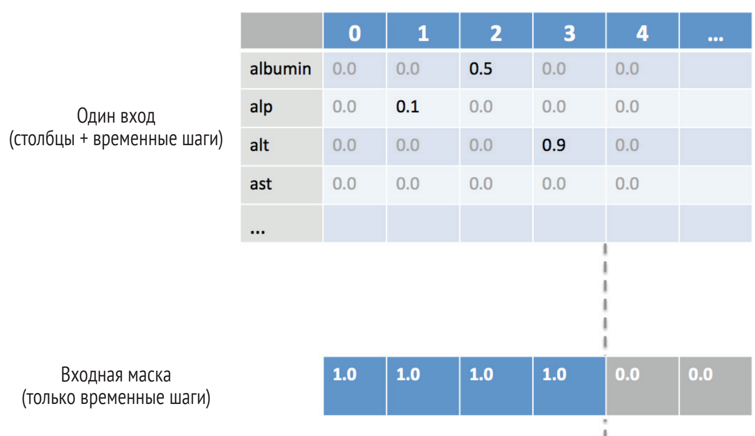


Рис. 4.19 ❖ Маскирование временных шагов

Почему не марковские модели?

Для учета времени было бы естественно подумать об использовании марковских моделей[36]. Это еще один класс моделей машинного обучения, широко применяемый для моделирования последовательностей. Но их применение ограничивает тот факт, что с ростом контекстного окна объем вычислений становится слишком велик, так что моделировать с их помощью долгосрочные зависимости невозможно.

Рекуррентные нейронные сети (коннекционистские модели) лучше марковских моделей (и других моделей с временным окном), потому что могут улавливать долгосрочные временные зависимости в данных. Связано это с тем, что в скрытом состоянии РНС запоминается информация из сколь угодно длинного контекстного окна, так что ограничения, присущие другим методам, отсутствуют. Более того, количество состояний, моделируемых с помощью такой сети, представлено скрытым слоем, и это количество экспоненциально зависит от числа блоков в слое. Поэтому РНС прекрасно справляется с улавливанием большого объема зависящей от времени информации во многих входных векторах.

Рекуррентные нейронные сети, скрытые слои и число состояний

Если бы на вход подавались только бинарные значения (0,1), то сеть смогла бы представить 2^N состояний, где N – число узлов в скрытом слое. Если бы на выходе были вещественные 64-разрядные числа, то один скрытый слой мог бы представить 2^{64N} различных состояний.

Время обучения такой сети растет всего лишь квадратично с ростом числа скрытых блоков, а ее выразительная мощность при этом растет экспоненциально.

---

36 https://en.wikipedia.org/wiki/Hidden_Markov_model.

Общая архитектура рекуррентной нейронной сети

Рекуррентные нейронные сети – надмножество сетей прямого распространения, характеризующееся добавлением рекуррентных связей (или рекуррентных ребер). Они связывают соседние временные шаги, внося в модель концепцию времени. Традиционные связи в РНС не содержат циклов, но рекуррентные связи могут образовывать циклы, в т. ч. связи от нейрона на будущем временном шаге к нему же в прошлом.

Архитектура рекуррентных нейронных сетей и временные шаги

На каждом временном шаге распространения информации по рекуррентной сети блоки, в которые входные данные поступают через рекуррентные связи, получают активации от текущего входного вектора и от скрытых блоков в предыдущем состоянии сети.

Выход вычисляется по скрытому состоянию на данном временном шаге. Предыдущий входной вектор на предыдущем временном шаге может оказывать влияние на текущий выход на текущем временном шаге посредством рекуррентных связей.

Мы можем сцеплять эти специализированные рекуррентные нейроны для построения лучших моделей. Выход предыдущего слоя соединяется с входом следующего слоя так же, как в многослойных сетях прямого распространения.

Проблема исчезающего градиента

Известно, что РНС страдают от «проблемы исчезающего градиента». Эта проблема возникает, когда градиент становится слишком мал, и затрудняет моделирование долгосрочных зависимостей (10 и более шагов) во входном наборе данных. Самый эффективный способ борьбы с ней – использовать LSTM-вариант архитектуры РНС, поддерживаемый библиотекой DL4J.

LSTM-сети

LSTM-сети – наиболее распространенный вариант рекуррентных нейронных сетей. Они были изобретены в 1997 году Хохрайтером и Шмидбауэром[37].

Важнейшие элементы LSTM[38] – ячейка памяти и вентили (включая вентиль забывания[39], а также входной вентиль). Содержимое ячейки памяти модулируется входным вентилем и вентилем забывания[40]. Если оба вентиля закрыты, то содержимое ячейки остается неизменным при переходе от предыдущего временного шага к следующему. Вентильная структура позволяет сохранять информацию на протяжении многих временных шагов, а следовательно, обеспечивает течение градиентов во времени. Это позволяет LSTM-модели преодолевать проблему исчезающего градиента, терзающую другие виды рекуррентных нейронных сетей.

---

[37] Hochreiter and Schmidhuber, 1997. Long short-term memory // http://www.bioinf.jku.at/publications/older/2604.pdf.

[38] Graves, 2012. Supervised Sequence Labelling with Recurrent Neural Networks // http://www.cs.toronto.edu/~graves/preprint.pdf.

[39] Gers, Schmidhuber and Cummins, 1999. Learning to Forget: Continual Prediction with LSTM // http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.5709&rep=rep1&type=pdf.

[40] Graves, 2012. Supervised Sequence Labelling with Recurrent Neural Networks // http://www.cs.toronto.edu/~graves/preprint.pdf.

Свойства LSTM-сетей

LSTM-сети известны следующими свойствами:

- улучшенные уравнения обновления;
- улучшенное обратное распространение.

Приведем несколько примеров использования LSTM-сетей:

- генерация последовательностей (например, в языковых моделях языка на уровне символов);
- классификация временных рядов;
- распознавание речи;
- распознавание рукописного текста;
- полиморфное моделирование музыки.

LSTM и двунаправленные рекуррентные нейронные сети (Bidirectional Recurrent Neural Networks – BRNN) в последние годы стали основным средством решения следующих задач:

- подписывание изображений;
- машинный перевод;
- распознавание рукописного текста.

LSTM-сеть, состоящая из большого числа связанных ячеек памяти, эффективно обучается.

Замечание о сложности обучения LSTM

Вычислительная сложность прямого и обратного проходов линейно зависит от числа временных шагов во входной последовательности.

Далее мы кратко опишем архитектуру и компоненты LSTM-сети.

Архитектура LSTM-сети

Ранее в этой книге была представлена концепция многослойной нейронной сети прямого распространения (рис. 4.20).

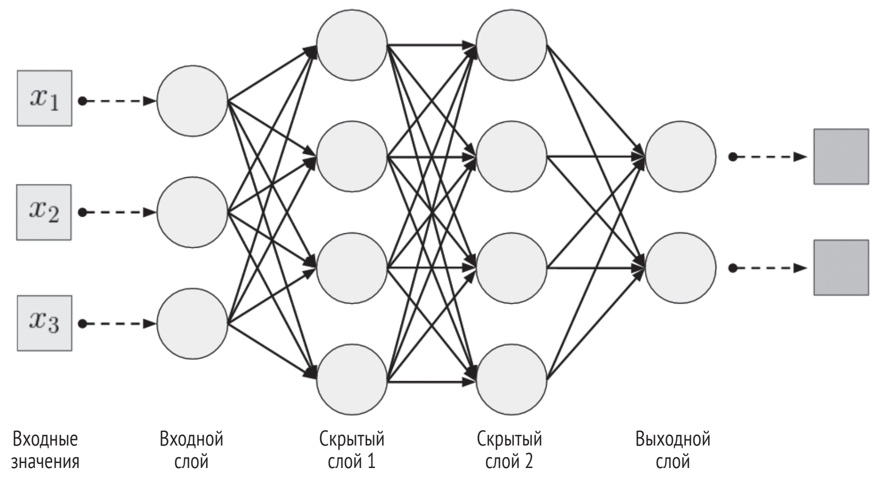


Рис. 4.20 ❖ Архитектура многослойной нейронной сети прямого распространения

Мы можем «сплющить» это представление, показав каждый слой сети в виде одного узла, как на рис. 4.21.

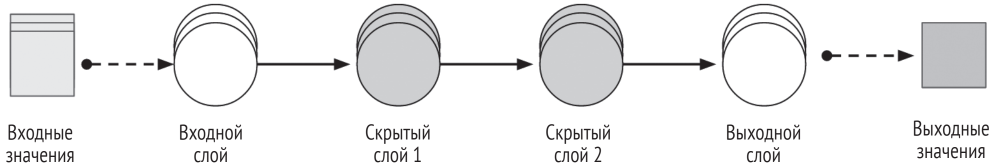


Рис. 4.21 ❖ Упрощенное представление многослойной сети прямого распространения

В рекуррентных нейронных сетях вводится идея связи между выходом нейрона скрытого слоя и входом другого (или того же самого) нейрона того же скрытого слоя. Это позволяет подавать на вход нейрона информацию с предыдущего временного шага.

На рис. 4.22 рекуррентные связи показаны на «сплющенном» представлении сети.

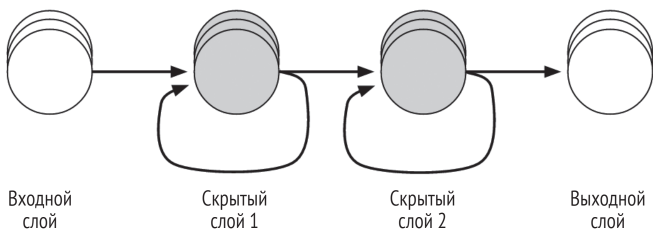


Рис. 4.22 ❖ Рекуррентные связи в скрытых слоях

На рис. 4.23 показано «развернутое» представление той же сети, что на рис. 4.22. Мы видим, что информация распространяется по сети в прямом направлении и «сквозь время».

Как мы увидим в следующем разделе, в LSTM-сетях по рекуррентной связи передается больше информации, чем в традиционных РНС.

Элемент LSTM-сети

Элемент слоя РНС – вариант классического искусственного нейрона. У каждого элемента LSTM-сети имеются связи двух типов:

- связи с предыдущим временным шагом (выходы блоков на этом шаге);
- связи с предыдущим слоем.

Главной идеей, благодаря которой LSTM-сеть может запоминать состояние, является ячейка памяти. Тело элемента LSTM-сети называется *блоком LSTM* (рис. 4.24).

Перечислим компоненты блока LSTM:

- три вентиля:
  - входной вентиль (вентиль модуляции входа);
  - вентиль забывания;
  - выходной вентиль;
- вход блока;

- ячейка памяти (карусель константной ошибки);
- выходная функция активации;
- глазковые связи.

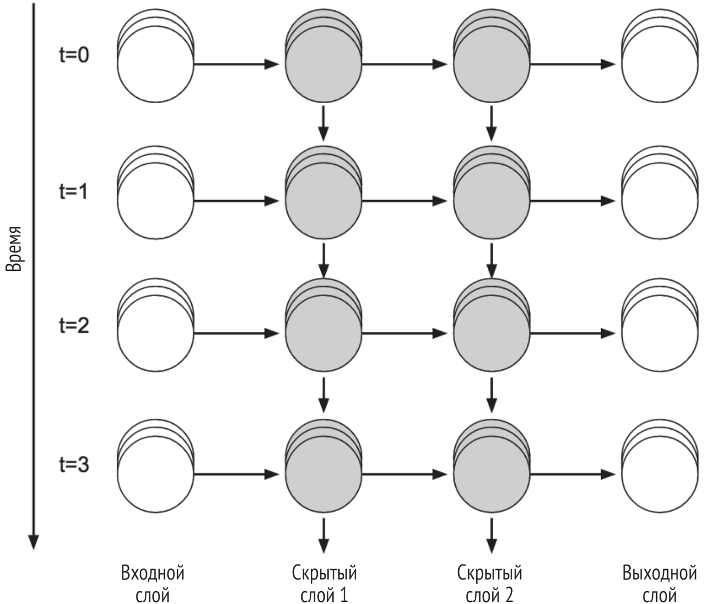


Рис. 4.23 ❖ Рекуррентная нейронная сеть, развернутая вдоль временной оси

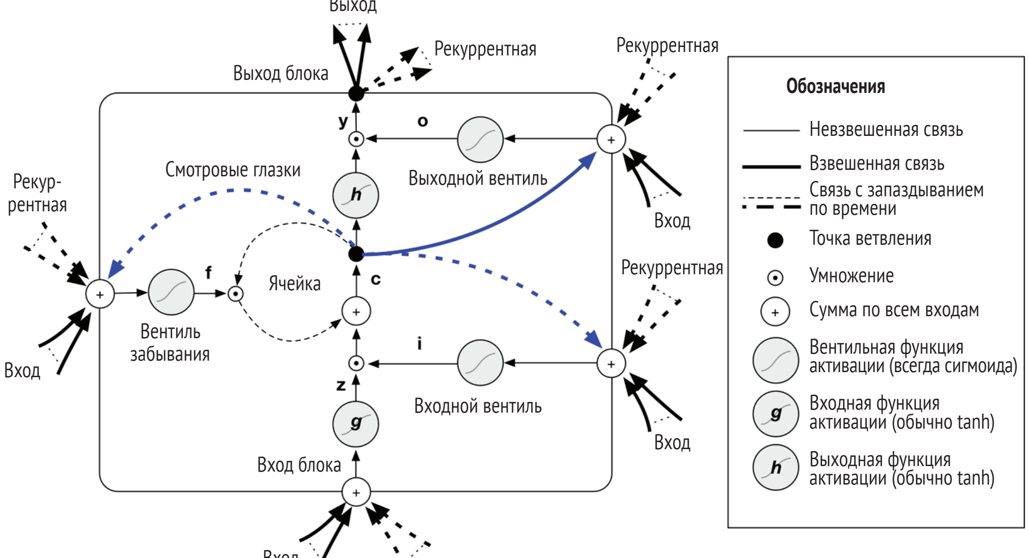


Рис. 4.24 ❖ Схема блока LSTM

Три вентиля обучаются защищать линейный блок от ложных сигналов:

- входной вентиль защищает блок от несущественных входных событий;
- вентиль забывания позволяет блоку забывать предыдущее содержимое памяти;
- выходной вентиль раскрывает (или не раскрывает) содержимое памяти на выходе блока LSTM.

Выход блока LSTM рекуррентно соединяется с его входом и всеми вентилями блока. Во всех трех вентилях блока LSTM применяются сигмоидные функции активации (для приведения аргумента к диапазону [0, 1]). В качестве функции активации на входе и выходе блока обычно используется tanh.

Замечание о вентиле забывания

Значение активации 1.0 означает «помнить все», а значение 0.0 – «забыть все». Так что этот вентиль было бы правильнее назвать «вентилем запоминания»!

Памятуя об этом, мы обычно инициализируем смещение вентиля забывания большим значением, чтобы он обучался долгосрочным зависимостям (в DL4J «большое» значение по умолчанию равно 1.0).

На рис. 4.25 приведены векторные формулы прямого прохода для слоя LSTM, взятые из работы Greff et al.[41]

$$\mathbf{z}^t = g(\mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z) \quad \text{Вход блока}$$
$$\mathbf{i}^t = \sigma(\mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i) \quad \text{Входной вентиль}$$
$$\mathbf{f}^t = \sigma(\mathbf{W}_f\mathbf{x}^t + \mathbf{R}_f\mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f) \quad \text{Вентиль забывания}$$
$$\mathbf{c}^t = \mathbf{i}^t \odot \mathbf{z}^t + \mathbf{f}^t \odot \mathbf{c}^{t-1} \quad \text{Состояние ячейки}$$
$$\mathbf{o}^t = \sigma(\mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o) \quad \text{Выходной вентиль}$$
$$\mathbf{y}^t = \mathbf{o}^t \odot h(\mathbf{c}^t) \quad \text{Выход блока}$$

Рис. 4.25 ❖ Векторные формулы прямого прохода для слоя LSTM

В табл. 4.1 описаны переменные, встречающиеся на рис. 4.25.

Таблица 4.1. Описание переменных в векторных формулах LSTM

Имя переменной	**Описание**
x^t	Входной вектор в момент *t*
W	Прямоугольные матрицы входных весов
R	Квадратные матрицы рекуррентных весов
p	Векторы весов глазковых связей
b	Векторы смещений

Саморекуррентная связь имеет фиксированный вес 1.0 (за исключением случая, когда она модулируется), чтобы преодолеть проблему исчезающего градиента. В такой базовой форме блоки LSTM могут обнаруживать далеко отстоящие события в последовательности – находящиеся на удалении до 1000 временных

[41] Greff et al., 2015. LSTM: A Search Space Odyssey // http://arxiv.org/pdf/1503.04069v1.pdf.

шагов. Сравните с предшествующими архитектурами, которые могли моделировать только события в пределах 10 временных шагов или около того.

Другие варианты LSTM
Обратите внимание на статью «LSTM: A Search Space Odyssey» (https://arxiv.org/pdf/1503.04069v1.pdf).

Вентильные рекуррентные блоки

С LSTM имеет некоторое сходство вентильный рекуррентный блок (Gated Recurrent Unit – GRU)[42]. В GRU имеются вентиль забывания и вентиль обновления, которые похожи на вентиль забывания и входной вентиль в блоке LSTM. Основное отличие заключается в том, что GRU полностью раскрывает содержимое своей памяти, применяя интеграцию с утечкой (но с адаптивной временной константой, управляемой вентилем обновления). На дизайн GRU оказал влияние блок LSTM, но считается, что GRU проще для вычислений и реализации.

LSTM-слои

LSTM-слой принимает входной вектор *x* (нефиксированный) и порождает выход *y*. На значение *y* оказывают влияние вход *x* и история всех входов. На слой влияет история входов через рекуррентные связи. В РНС имеется внутреннее состояние, которое обновляется при подаче на вход слоя каждого нового вектора. Состоянием является один скрытый вектор.

Обучение

В LSTM-сетях для обновления весов применяется обучение с учителем. На вход алгоритма обучения подается по одному вектору из последовательности. Векторы вещественные и становятся последовательностями активаций входных блоков. Все остальные блоки вычисляют свое текущее значение активации на каждом временном шаге. Это значение является нелинейной функцией от взвешенной суммы активаций всех блоков, связанных с данным.

Для каждого вектора во входной последовательности ошибка равна сумме отклонений вычисленных активаций от известных меток. Рассмотрим вариант алгоритма обратного распространения во времени (BPTT), применяемый в рекуррентных нейронных сетях, включая LSTM.

BPTT и усеченный BPTT

Обучение РНС может быть сопряжено с высокими вычислительными расходами. Традиционно для этой цели используется алгоритм BPTT.

По существу, BPTT не отличается от стандартного алгоритма обратного распространения: мы применяем правило вычисления производной сложной функции (градиентов) с учетом структуры связей в сети. Слова «*во времени*» означают, что некоторые градиенты (ошибки) распространяются в направлении от будущих временных шагов к прошлым, а не от следующих слоев к предыдущим, как в стандартном алгоритме.

Если сеть имеет дело с длинными последовательностями, насчитывающими много временных шагов, то мы рекомендуем подумать об использовании усечен-

[42] Cho et al., 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation // https://arxiv.org/abs/1406.1078.

ного алгоритма BPTT. В нем уменьшена вычислительная сложность обновления каждого параметра РНС.

Рекуррентные нейронные сети и обратное распространение

Вычисление градиента для РНС на последовательности длиной 1000 имеет такую же вычислительную сложность, как выполнение прямого и обратного проходов в перцептроне с 1000 слоев.

Более частое обновление параметров ускоряет обучение рекуррентной нейронной сети. Мы рекомендуем применять усеченный BPTT, когда число временных шагов превышает несколько сотен.

Чтобы лучше понять идею усеченного BPTT, посмотрим, что происходит при обучении сети с 12 временными шагами. В этом случае мы должны выполнить прямой проход на 12 шагов, вычислить ошибку сети и произвести обратный проход на 12 шагов (см. рис. 4.26).

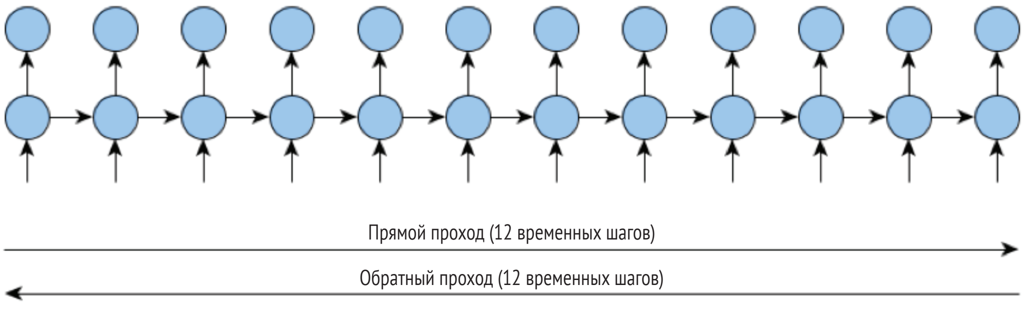


Рис. 4.26 ❖ Стандартный BPTT

Выполнить 12 временных шагов в процессе обучения не так уж трудно. Но когда временных шагов сотни, обучение модели усложняется. А 1000 шагов в обоих направлениях – это уже вычислительно дорогостоящая задача, и мы начинаем искать альтернативы.

В усеченном алгоритме BPTT прямой и обратный проходы разбиваются на несколько меньших операций. На рис. 4.27 видно, что производятся небольшой прямой проход и соответствующий ему обратный проход, на котором обновляются параметры. Размер этого прохода – гиперпараметр, задаваемый пользователем. На рисунке показан проход длиной 4 временных шага.

Усеченный BPTT считается наиболее практичным из современных методов обучения РНС. Он позволяет улавливать долгосрочные зависимости с меньшими вычислительными затратами, чем стандартный BPTT.

Порядок сложности стандартного и усеченного алгоритмов BPTT один и тот же, и количество временных шагов в процессе обучения одинаково. Но при одинаковых затратах мы получаем больше обновлений параметров (хотя с каждым обновлением сопряжены некоторые издержки). Как всегда при аппроксимации, использование усеченного BPTT не свободно от недостатков: протяженность зависимостей, обучаемых в усеченном BPTT, может оказаться меньше, чем в пол-

ном. На практике в этом обычно нет ничего страшного, нужно только правильно задать гиперпараметры усеченного BPTT.

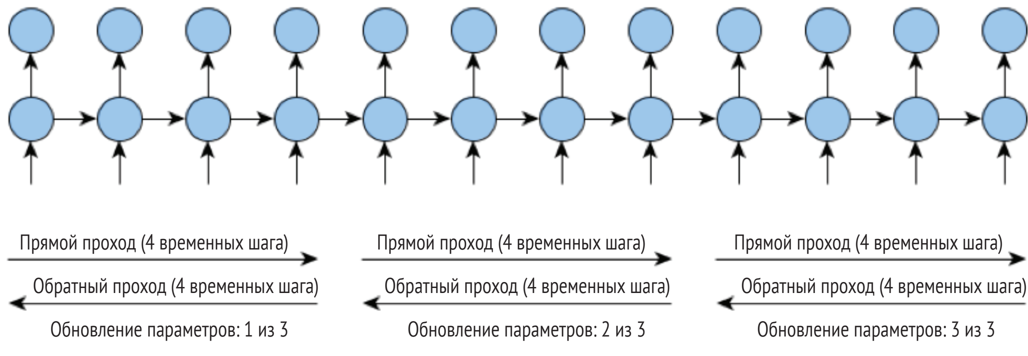


Рис. 4.27 ❖ Усеченный BPTT

Предметно-ориентированные приложения и гибридные сети

Как уже отмечалось, рекуррентные нейронные сети применяются в различных предметных областях, например: транскрипция речи в текст, машинный перевод, порождение рукописного текста. РНС оказались также весьма востребованы в компьютерном зрении и позволяют решать следующие задачи:

- анализ видеоряда на уровне кадров[43];
- подписывание изображений;
- подписывание видео[44];
- ответы на визуальные вопросы[45].

Еще одна быстро развивающаяся область исследований в компьютерном зрении – применение РНС для извлечения информации из изображения в результате обработки лишь небольших его участков, это называется рекуррентной моделью зрительного внимания[46]. Эти модели эффективны при работе с изображениями, загроможденными большим количеством объектов, которые с трудом поддаются классификации с помощью СНС. В таких приложениях СНС применяется для чистого восприятия, а РНС – для моделирования временного аспекта.

Стоит отметить еще одну гибридную сеть СНС+РНС из работы Андрея Карпатого и Ли Фей-Фея[47], в которой сеть порождает описания изображений и их участков на естественном языке. Эта модель способна также порождать подписи под изображениями, будучи обучена на наборах данных, состоящих из изображений и соответствующих фраз (рис. 4.28).

---

[43] Srivastava, Mansimov and Salakhutdinov, 2015. Unsupervised Learning of Video Representations using LSTMs // https://arxiv.org/abs/1502.04681.

[44] Venugopalan et al., 2014. Translating Videos to Natural Language Using Deep Recurrent Neural Networks // https://arxiv.org/abs/1412.4729.

[45] Wu et al., 2016. Image Captioning and Visual Question Answering Based on Attributes and External Knowledge // https://arxiv.org/abs/1603.02814.

[46] Mnih et al., 2014. Recurrent Models of Visual Attention // https://arxiv.org/abs/1406.6247.

[47] Karpathy and Fei-Fei, 2014. Deep Visual-Semantic Alignments for Generating Image Descriptions // https://arxiv.org/abs/1412.2306v2.

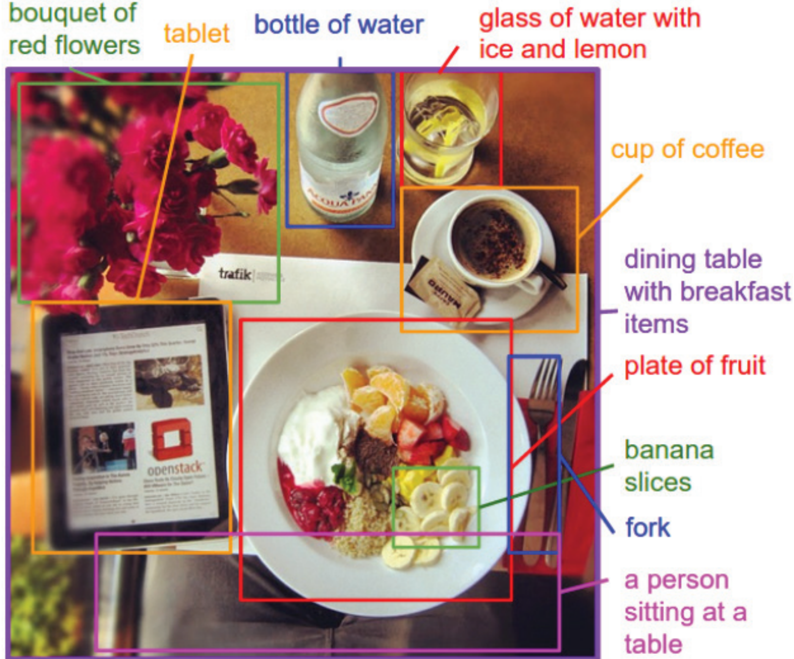


Рис. 4.28 ❖ Пометка изображений с помощью гибридной сети СНС+РНС

Это комбинация СНС и двунаправленной РНС.

Рекурсивные нейронные сети

Рекурсивные нейронные сети, как и рекуррентные, могут работать с входными данными переменной длины. Основное отличие заключается в том, что рекурсивная сеть способна моделировать иерархические структуры в обучающем наборе данных. Изображение обычно содержит сцену, на которой присутствует много объектов. Деконструкция сцены часто представляет собой интересную, но нетривиальную проблему. Требуется не только идентифицировать объекты на сцене, но и понять, как они соотносятся друг с другом, – в этом проявляется рекурсивный характер деконструкции.

Архитектура сети

Архитектурно рекурсивная нейронная сеть состоит из матрицы разделяемых весов и двоичного дерева, наделяющего сеть способностью обучаться последовательностям слов или частей изображения, имеющим переменную длину. Такие сети полезны для анализа изображений и предложений. Для обучения используется алгоритм *обратного распространения сквозь структуру* (backpropagation through structure – BPTS). Прямой проход выполняется снизу вверх, обратный – сверху вниз. Можно считать, что цель находится на вершине дереве, а входные данные расположены внизу.

Разновидности рекурсивных нейронных сетей

Есть несколько видов рекурсивных нейронных сетей. Один из них – рекурсивный автокодировщик. Как и его тезка прямого распространения, рекурсивный автокодировщик обучается воспроизводить вход. Конкретно в случае ОЕЯ он учится воспроизводить контексты. Результатом обучения рекурсивного автокодировщика с частичным привлечением учителя являются правдоподобия определенных меток в каждом контексте.

Еще один вид – рекурсивная нейронная тензорная сеть, обучаемая с учителем, которая вычисляет цель в каждом узле дерева. Слово «тензорная» в названии означает, что градиент вычисляется несколько иначе, включая больше информации в каждом узле, для чего нужен тензор (матрица с тремя и более измерениями).

Применение рекурсивных нейронных сетей

У рекурсивных и рекуррентных нейронных сетей много общих применений. Рекурсивные нейронные сети традиционно используются в ОЕЯ в силу присущей им связи с двоичными деревьями, контекстами и анализаторами естественных языков. Например, анализаторы на основе грамматики составляющих могут разложить предложение в двоичное дерево, ориентируясь на его лингвистические свойства. Рекурсивная нейронная сеть налагает ограничение – анализатор должен строить древовидную структуру (обычно подразумевается грамматика составляющих).

Рекурсивная нейронная сеть может восстановить как низкоуровневую, так и высокоуровневую иерархическую структуру в наборах изображений или предложений. Вот несколько типичных приложений:

- декомпозиция изображенной сцены;
- ОЕЯ;
- транскрипция речи в текст.

На практике чаще всего встречаются рекурсивные автокодировщики и рекурсивные нейронные тензорные сети (RNTN). Первые используются для разложения предложений на сегменты в ОЕЯ. Вторые – для разложения изображения на объекты и сопоставления каждому объекту семантической метки.

Рекуррентные нейронные сети обучаются быстрее, поэтому они чаще используются в приложениях для работы с временными рядами, но продемонстрирована также успешная работа в таких связанных с ОЕЯ задачах, как анализ тональности высказываний.

Итоги и обсуждение

В этой главе мы рассмотрели основные современные архитектуры, применяемые в глубоком обучении. Сгруппируем их по применениям.

- Для порождения данных (например, изображений, звука и текста) используются:
  - порождающие состязательные сети;
  - вариационные автокодировщики;
  - рекуррентные нейронные сети.

- Для моделирования изображений используются:
  - сверточные нейронные сети;
  - глубокие сети доверия.
- Для моделирования последовательных данных используются:
  - рекуррентные нейронные сети, LSTM.

В последующих главах мы приведем примеры реального кода для большинства сетей и обсудим особенности их обучения и настройки. В главе 5 мы увидим, как эти концепции претворятся в API библиотеки DL4J. Но прежде поговорим о нескольких вопросах, часто возникающих в контексте глубокого обучения.

Приведет ли глубокое обучение к отмиранию всех прочих алгоритмов?

Тема ненужности всех остальных алгоритмов моделирования в связи с пришествием глубокого обучения раз за разом возникает на интернет-форумах. Ответ на этот вопрос отрицательный, потому что для многих сравнительно простых приложений машинного обучения гораздо более простые алгоритмы прекрасно работают и обеспечивают требуемую верность модели. С моделями типа логистической регрессии проще работать, поэтому, принимая решение, всегда нужно сопоставлять трудоемкость с требованиями к верности. Алгоритмы же глубокого обучения особенно хорошо работают в случаях, когда мы мало знаем о предметной области, и конструирование качественных признаков вручную обходится дорого.

Оптимальный метод зависит от задачи

Правильное применение машинного обучения подразумевает поиск подхода, отвечающего поставленной задаче. Мы пока не можем назвать единый метод, пригодный для всего на свете, поэтому должны всякий раз оценивать задачу и данные в поисках наилучшей модели. В этом смысл «теоремы об отсутствии бесплатных завтраков».

Теорема об отсутствии бесплатных завтраков

Эта теорема утверждает, что не существует модели, оптимальной для всех задач. Предположения, при которых лучше всего работает некоторая модель, могут не выполняться в другой задаче. В машинном обучении нередко пробуют разные модели, стремясь найти такую, которая лучше других подходит в конкретном случае.

У любого метода машинного обучения есть смещение и дисперсия. Чем ближе модель к истинному распределению данных, тем лучше результаты алгоритма обучения в среднем.

Попробуем взглянуть на проблему с точки зрения практического примера. Если визуализация показывает, что данные очевидно линейны, то станете ли вы аппроксимировать их нелинейной моделью (например, многослойным перцептроном)? Наверное, нет, а возьмете что-нибудь попроще, скажем, логистическую регрессию. В конкурсах на сайте Kaggle оптимальный метод все время меняется. Но в тех случаях, когда победителем оказывается не глубокое обучение, первое место занимают обычно случайные леса и ансамблевые методы.

Размер набора данных также следует учитывать, принимая решение об использовании глубокого обучения. Полученные в последние годы эмпирические

результаты свидетельствуют, что предсказательная сила глубокого обучения высока, когда набор данных достаточно велик, т. е. результаты тем лучше, чем больше набор данных. Нейронные сети обладают большей репрезентативной емкостью, чем линейные модели, и лучше приспособлены для исследования данных. Эвристическое правило таково: обучить нейронную сеть можно, если имеется по меньшей мере 5000 помеченных примеров.

Когда мне может понадобиться глубокое обучение?

В завершение этой главы сформулируем ряд простых правил для ответа на вопрос: нужно ли в данном проекте глубокое обучение?

Когда стоит использовать глубокое обучение

Глубокое обучение стоит использовать, когда:

- более простые модели (логистическая регрессия) не дают требуемой верности;
- требуется распознавать сложные паттерны в изображениях, ОЕЯ или звуковых данных;
- размерность данных велика;
- входные данные обладают временным измерением (последовательности).

Когда стоит ограничиться традиционным машинным обучением

Традиционное машинное обучение стоит использовать, когда:

- имеются высококачественные данные низкой размерности, например экспортированные из базы данных в виде таблицы;
- вы не пытаетесь найти в данных изображения сложные паттерны.

Оба метода дадут неудовлетворительные результаты, если данные неполны или плохого качества.

Глава 5

Построение глубоких сетей

Теперь не время думать о том, чего у тебя нет. Подумай о том, как бы обойтись с тем, что есть.

– *Эрнест Хемингуэй. «Старик и море»*

В этой главе мы познакомимся с инструментарием библиотеки DL4J и некоторыми реальными примерами, которыми вы сможете воспользоваться в своих проектах. Мы начнем с вопроса о выборе конкретной глубокой сети для решения поставленной задачи. А закончим главу подробным обзором примеров, входящих в комплект поставки библиотеки.

Об установке и поддержке DL4J см. приложение G.

Выбор глубокой сети для решения задачи

В главе 4 мы уже говорили о том, что смысл глубокого обучения состоит в том, чтобы спроектировать архитектуру сети, соответствующую задаче, а не пытаться вручную сконструировать признаки по входным данным. В этой главе мы будем рассматривать приложения для решения следующих задач:

- моделирование табличных данных;
- моделирование изображений;
- моделирование последовательностей и временных рядов;
- обработка естественного языка.

Представленные в этой главе приложения иллюстрируют концепции глубоких сетей, которые мы изучали начиная с главы 1. И хотя мы не включили примеры для каждой архитектуры, упомянутой в главе 4, предложенных примеров достаточно для освещения базовых идей, и при желании вы легко сможете развить их в интересах собственных проектов. Начнем с обзора типов данных в контексте подбора подходящей архитектуры.

Примеры в сети

Мы создали репозиторий с примерами на сайте GitHub (https://github.com/deeplearning4j/oreilly-book-dl4j-examples/). Проект DL4J постоянно развивается, но мы хотели, чтобы у читателей был мгновенный снимок кода, который гарантированно будет работать с версией, используемой в этой книге.

Табличные данные и многослойные перцептроны

Табличные данные, как правило, имеют статическую структуру, и в DL4J для их моделирования наиболее подходит многослойный перцептрон. В таких за-

дачах иногда полезно приложить руку к конструированию признаков, но зачастую сеть находит оптимальные веса самостоятельно. Одна из основных проблем при работе с моделью на основе многослойного перцептрона – настройка гиперпараметров. В главе 6 мы рассмотрим различные техники, помогающие в этом деле.

Изображения и сверточные нейронные сети

Сверточные нейронные сети (СНС) демонстрируют удивительные успехи, когда требуется найти структуру в данных изображения. Исторически в моделировании изображений превалировали разнообразные методы предварительной обработки, имеющие целью выровнять входные изображения и преобразовать их к виду, удобному для моделирования. Небольшие вариации угла поворота или масштаба сильно затрудняли обработку изображений. СНС позволили работать с изображениями без предварительной обработки, а на долю человека осталась задача настройки сетевой архитектуры. Ниже в этой главе мы рассмотрим классификацию рукописных цифр с помощью СНС.

Нейронные сети и исходные данные изображения

В табличных данных признаки собраны в столбцах, для чего требуется приложить немалые усилия. Исторически модели машинного обучения вели себя ненадежно, если признаки оказывались не там, где модель ожидала их найти.

На изображениях объекты редко находятся там, где мы хотели бы их видеть, поэтому задача выделения признаков для классических методов машинного обучения отнюдь не тривиальна. Сила СНС в том и состоит, что ей можно подать на вход исходное изображение, в котором объекты расположены в произвольных местах и положениях, и тем не менее сеть успешно распознает признаки.

Эволюция приложений СНС

Всё новые приложения СНС появляются с головокружительной скоростью. Мы еще только рекомендуем практикам начинать с применения СНС к моделированию изображений, а тут уже подоспели приложения для моделирования текста, например:

- машинный перевод[1];
- классификация предложений[2];
- анализ тональности высказываний[3].

Точно так же, как окно фильтра СНС сдвигается по изображению, оно может сдвигаться и по последовательности символов. Эта инвариантность относительно положения признаков и делает СНС эффективным инструментом в арсенале специалиста по глубоким сетям. В этой главе мы будем рассматривать только применение СНС к данным изображений. Но сразу хотим оговориться, что это не ограничение сетевой архитектуры, а лишь отправная точка для дальнейших исследований.

---

[1] Kalchbrenner et al., 2016. Neural Machine Translation in Linear Time // https://arxiv.org/abs/1610.10099.

[2] Kim, 2014. Convolutional Neural Networks for Sentence Classification // https://arxiv.org/abs/1408.5882.

[3] Nogueira dos Santos and Gatti, 2014. Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts // http://www.aclweb.org/anthology/C14-1008.

Последовательности, временные ряды и рекуррентные нейронные сети

При работе с последовательными данными, например журналами веб-сервера, рекуррентные нейронные сети – как раз то, что надо. Если в последовательности данных каждый пример содержит временную метку, то данные можно рассматривать как временной ряд.

На графике последовательные данные и временные ряды выглядят как волнообразная форма. Зачастую нам интересно найти на графике какие-то паттерны, повторяющиеся со временем. Отличие временных рядов от изображений состоит в том, что для последних не существует временного измерения, а данные образуют двумерную сетку. Но в любом случае мы ищем в данных паттерны. Искомые объекты могут различаться по размеру и обычно находятся в разных местах, что представляет проблему.

Рекуррентные нейронные сети лучше приспособлены для моделирования временных рядов, чем многослойные перцептроны, т. к. последовательность входных векторов является для них одним логическим входом (см. главу 4).

Датчики, журналы и другие измерения

Рекуррентная нейронная сеть лучше моделирует изменение во времени данных, полученных от источника измерений.

Рекуррентные нейронные сети можно использовать для классификации, регрессии и порождения новых данных. Ниже в этой главе мы приведем примеры классификации и порождения данных.

Выступление в роли адвоката дьявола при выборе модели

Иногда можно услышать о достижении большого успеха при использовании других методов, например случайных лесов. Но в мире последовательных данных этот путь хорош разве что в очень специальных случаях.

Некоторые практики говорят, что не могут применить случайные леса к последовательным данным или временным рядам так же, как рекуррентные нейронные сети, потому что временной ряд нужно сначала преобразовать в какое-то плоское векторное представление[4] (например, выделить признаки) и только потом приступать к моделированию.

Обычно мы считаем все споры по поводу сравнительных достоинств модели глубокого обучения и другой мощной модели в сочетании с признаками, сконструированными вручную, беспредметными, поскольку специалист, предпочитающий ручное конструирование, сможет найти хорошие признаки, учитывающие все особенности набора данных (при наличии достаточного времени).

Истинные преимущества глубокого обучения в применении к временным рядам (да и к любой другой задаче) сводятся к следующим.

Пойдя по пути вручную подобранных признаков и алгоритмов, мы окажемся в ситуации, когда должны писать новый код для десятков совершенно различных алго-

---

[4] Пример такого рода приведен в работе: Shieh and Keogh, 2008. iSAX: Indexing and Mining Terabyte Sized Time Series // http://www.cs.ucr.edu/~eamonn/iSAX.pdf.

ритмов (моделей) в зависимости от задачи или набора данных. В программной инженерии эту ситуацию называют *техническим долгом*. Раньше в машинном обучении все обстояло именно так.

Если в начале пути подходящие признаки неизвестны, то современное состояние глубокого обучения допускает полуавтоматическую настройку гиперпараметров нейронной сети, т. е. полуавтоматическое конструирование признаков. Хорошо продуманную систему глубокого обучения (включая настройку гиперпараметров) можно эффективно использовать многократно, и она будет работать для самых разных задач.

Когда приходится иметь дело с временными рядами переменной длины с неизвестными, но, возможно, долгосрочными зависимостями, на передний план выходят рекуррентные нейронные сети. Если же длина последовательности фиксирована, то вполне может подойти многослойный перцептрон или СНС.

Но если допустимы входные последовательности произвольной длины и не исключено, что нужно принимать во внимание всю историю, то единственная возможность – рекуррентная нейронная сеть (или тщательно спроектированная марковская модель с большим пространством скрытых состояний).

Применение гибридных сетей

Если данные представляют собой комбинацию изображений и времени (например, видеоряд), то применяется специальная гибридная сеть с долгой краткосрочной памятью (LSTM) и сверточными слоями. Мы рекомендуем такой подход, потому что LSTM-сети способны улавливать временной аспект изменения изображений в видео, а сверточные слои – структуру каждого кадра.

Инструментарий DL4J

DL4J представляет собой группу инструментов глубокого обучения, предназначенных для решения следующих задач:

- интеграции;
- векторизации;
- моделирования;
- оценивания.

Эти инструменты работают на разных платформах и могут выполняться последовательно или параллельно. С самого начала DL4J проектировалась с учетом современных аппаратных платформ и не испытывает проблем с распараллеливанием, как некоторые другие библиотеки машинного обучения, разработанные в прошлом десятилетии. Сообщество DL4J уделяло также внимание качественной интеграции с такими платформами, как Spark и Hadoop Distributed File System (HDFS). Проект DL4J обрел надежные средства векторизации с включением в него библиотеки DataVec в качестве полноправного компонента.

DL4J рассчитана на задачи масштаба предприятия и ориентирована на практиков, которые хотят использовать в глубоком обучении виртуальную машину (JVM), но в то же время нуждаются в быстродействии C++ и возможностях Spark для параллельных вычислений. Сделаем краткий обзор этих аспектов DL4J, прежде чем приступать к установке библиотеки на ноутбук.

Векторизация и DataVec

Поскольку нейронные сети обучаются только на векторных данных, векторизация является необходимым этапом предварительной обработки. В состав DL4J входит библиотека DataVec, позволяющая быстро создавать векторизованные данные, пригодные для моделирования средствами DL4J. Как мы увидим ниже, DataVec может работать как в локальном режиме, так и в режиме параллельных вычислений на платформе Apache Spark.

Среды выполнения и ND4J

ND4J – библиотека научных расчетов для JVM. Синтаксически она имитирует Numpy и MATLAB. В ней реализованы *n*-мерные массивы для Java, которые можно использовать для решения задач линейной алгебры и операций с очень большими матрицами. ND4J предлагает простой фасад, который позволяет менять скрытые за ним аппаратные средства – центральные или графические процессоры, – не трогая написанного кода. Мы можем один раз написать реализацию какой-то линейно-алгебраической операции, а затем с помощью Maven указать предпочтительную аппаратную платформу. ND4J поддерживает среди прочих платформы x86 и jcublas. Разработка проекта с использованием DL4J рано или поздно потребует включения OpenCL и Power8. ND4J поддерживает API для Java и Scala, чтобы программисты на платформе JVM могли работать в знакомой среде. API библиотеки ND4J спроектированы так, чтобы по возможности имитировать лаконичность и простоту Numpy.

DL4J Java API позволяет программисту конфигурировать свои собственные нейронные сети из отдельных компонентов и настраивать гиперпараметры. Предлагаются средства для построения нейронной сети, ее настройки, оценивания и загрузки в нее данных. ND4J API содержит функции для выполнения операций линейной алгебры, математического анализа и обработки сигналов в окружении, напоминающем Numpy (которое можно развернуть в распределенных средах выполнения на нескольких CPU или GPU). Совместно Java API и ND4J дают программный доступ к самым быстрым средствам, доступным на сегодняшний день для приложений глубокого обучения в масштабе предприятия.

ND4J и жажда скорости

В мире глубокого обучения всегда существует потребность в более быстрых вычислениях. Библиотека ND4J достигает высокой скорости работы тремя способами:

- использованием JavaCPP;
- реализацией базовых алгоритмов на CPU;
- реализацией базовых алгоритмов на GPU.

Несколько замечаний о том, как это помогает ускорить обучение глубоких моделей.

JavaCPP

- Автоматическая генерация привязок JNI для C++.
- Простота сопровождения и развертывания двоичного кода на C++ в среде Java.

Реализация базовых алгоритмов на CPU

- OpenMP (многопоточность с встроенными примитивами).
- OpenBLAS или MKL (линейная алгебра).
- Расширения системы команд SIMD.

Реализация базовых алгоритмов на GPU

- DL4J уже поддерживает Cuda 7.5 (+cuBLAS) и будет поддерживать версию 8.0 с момента ее выхода.
- Используется также cuDNN.

Конфигурирование поддержки GPU
Дополнительную информацию о том, как сконфигурировать DL4J для работы на GPU, см. в приложении I.

Эталонные тесты производительности ND4J и DL4J

Результаты сравнения DL4J с другими популярными библиотеками глубокого обучения имеются на странице https://github.com/deeplearning4j/dl4j-benchmark. На сегодняшний день DL4J вполне конкурентоспособна, как видно из табл. 5.1.

***Таблица 5.1.** Реализация сети LeNet с использованием cuDNN*

Пакет	CPU	GPU	Multi	Верность
DL4J	20м08с	3с13с	1м18с	~99.0%
Caffe	19м52с	53с	1м14с	~99.0%
TensorFlow	5м15с	1м44с	2м44с	~98.6%
Torch	18с03с	6м25с	3м50с	~98.3%

Сообщество DL4J стремится еще больше повысить производительность ND4J и DL4J. Время от времени посматривайте на публикуемые результаты тестов производительности.

Основные концепции DL4J API

В этом разделе мы кратко перечислим основные средства, используемые в большинстве примеров. Более полные сведения об использовании API см. в приложении E или в онлайновой документации по DL4J.

Загрузка и сохранение моделей

Запись обученной модели на диск

Для записи архитектуры и параметров модели на диск служит класс `ModelSerializer`, например:

```
BufferedOutputStream stream = new BufferedOutputStream(...);
ModelSerializer.writeModel( trainedNetwork, stream, true );
```

Этот код работает как для записи на локальный диск, так и в другую файловую систему, например HDFS.

Запись в HDFS. Для записи в HDFS нужно только использовать правильный класс `Path` (`org.apache.hadoop.fs.Path`) и правильно отформатировать путь к файлу HDFS:

```
Path modelPath = new Path( hdfsPathToSaveModel );
BufferedOutputStream stream = new BufferedOutputStream( os );
ModelSerializer.writeModel( trainedNetwork, stream, true );
```

Пути в HDFS имеют такой формат:

```
hdfs:///path/to/my/file.txt
```

Чтение сохраненной модели с диска

Чтобы загрузить ранее сохраненную модель DL4J с диска, мы снова используем класс `ModelSerializer`:

```
InputStream stream = ...
MultiLayerNetwork network = ModelSerializer.restoreMultiLayerNetwork( stream );
```

Чтение из HDFS. Чтобы загрузить модель непосредственно из HDFS, используется тот же самый вызов API, но при создании экземпляра `InputStream` указывается класс `FileSystem` для Hadoop:

```
org.apache.hadoop.conf.Configuration hadoopConfig = new org.apache.hadoop.conf
  .Configuration();
FileSystem hdfs = FileSystem.get(hadoopConfig);
InputStream stream = hdfs.open( new Path( hdfsPathToSavedModelFile ) );
MultiLayerNetwork network = ModelSerializer.restoreMultiLayerNetwork( stream );
```

Получение входных данных для модели

При обучении и оценивании моделей в DL4J используется структура данных NDArray. Для задания входа и выхода массивы NDArray часто применяются в паре с объектом `DataSet`. Дополнительные сведения о классе `DataSet` см. в приложении E, посвященном ND4J.

Векторизация данных
О том, как преобразовать исходные данные в векторы типа NDArray для использования в моделях DL4J, см. приложения F и E.

Загрузка данных на этапе обучения

Для загрузки обучающих и тестовых данных в модели DL4J применяются различные объекты `RecordReader`, которые разбирают данные в различных файловых форматах и собирают из них мини-пакеты.

- Класс `RecordReader` – этот класс читает входной вектор из файла в любом из поддерживаемых форматов и преобразует данные в стандартный массив NDArray.
- Класс `DataSetIterator` – этот класс работает совместно с `RecordReader`, он создает обучающие мини-пакеты из полученных объектов NDArray.

Загрузка данных в Spark
Методы загрузки данных в системе Spark аналогичны, но используются специальные классы, предназначенные для Spark и HDFS. Мы рассмотрим эту тему в главе 9.

Задание архитектуры модели

Для любой модели в DL4J требуется задать архитектуру нейронной сети, выбрав ее из тех, что обсуждались в этой книге выше.

Построение послойной архитектуры

NeuralNetConfiguration – основной объект для конструирования слоев сети в DL4J. Глубокая нейронная сеть состоит из большого числа слоев, как показано в следующем примере:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(1)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .learningRate(learningRate)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
                .weightInit(WeightInit.XAVIER)
                .activation(Activation.RELU)
                .build())
        .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
                .weightInit(WeightInit.XAVIER)
                .activation(Activation.SOFTMAX)
                .nIn(numHiddenNodes).nOut(numOutputs).build())
        .pretrain(false).backprop(true).build();
```

Чтобы построить нужную нам архитектуру сети, мы добавляем слои по одному и конфигурируем каждый слой отдельно.

Гиперпараметры

Объект NeuralNetConfiguration позволяет задать различные гиперпараметры, относящиеся к обучению, например:

```
.learningRate(learningRate)
.updater(Updater.NESTEROVS).momentum(0.9)
```

Здесь задаются скорость обучения, корректор и значение импульса для созданного ранее объекта MultiLayerConfiguration. Стратегии задания гиперпараметров для других архитектур сети будут показаны ниже.

Обучение и оценивание

Сконфигурировав сеть и загрузив данные с помощью объектов RecordReader и Iterator, мы должны обучить модель. Мы хотим задать несколько периодов и вызвать метод .fit(), передав ему обучающие данные в качестве аргумента:

```
model.fit( trainingDataIter );
```

Этот метод с помощью итератора перебирает в цикле все обучающие примеры и возвращает управление, когда все примеры будут исчерпаны.

Периодом обучения называется полный проход по всему набору данных. В следующем фрагменте метод модели fit() вызывается с одним и тем же итератором столько раз, сколько задано периодов:

```
for ( int n = 0; n < nEpochs; n++) {
    model.fit( trainingDataIter );
}
```

В примерах из этой главы мы еще не раз встретим этот цикл по периодам.

Предсказание

Чтобы лучше понять процесс порождения предсказаний, обратитесь к приложению E.

Обучающие, контрольные и тестовые данные

Считается правильным не только разбивать набор данных на обучающие и тестовые, но еще и выделять контрольные данные. С их помощью мы решаем, не пора ли прекратить обучение досрочно.

Моделирование CSV-данных с помощью многослойных перцептронов

Освоить работу с DL4J человеку, только приступающему к практическому глубокому обучению, нелегко, поэтому начнем с построения модели многослойного перцептрона. Это позволит познакомиться с простыми приемами использования DL4J API в контексте довольно старой и, наверное, знакомой архитектуры нейронных сетей. В репозитории на GitHub имеется Java-программа для моделирования синтетического нелинейного набора данных[5]. В примере 5.1 приведен ее код (https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/feedforward/classification/MLPClassifierSaturn.java).

Пример 5.1 ❖ Пример многослойного перцептрона

```
public class MLPClassifierSaturn {

    public static void main(String[] args) throws Exception {
        Nd4j.ENFORCE_NUMERICAL_STABILITY = true;
        int batchSize = 50;
        int seed = 123;
        double learningRate = 0.005;
        // Число периодов (полных проходов по данным)
        int nEpochs = 30;

        int numInputs = 2;
        int numOutputs = 2;
        int numHiddenNodes = 20;
        final String filenameTrain =
            new ClassPathResource("/classification/saturn_data_train.csv")
                .getFile().getPath();
        final String filenameTest =
            new ClassPathResource("/classification/saturn_data_eval.csv")
                .getFile().getPath();

        // Загрузить обучающие данные
        RecordReader rr = new CSVRecordReader();
        rr.initialize(new FileSplit(new File(filenameTrain)));
        DataSetIterator trainIter =
```

[5] Этот набор данных создан д-ром Джейсоном Бэлбриджем для тестирования базовой функциональности библиотек нейронных сетей.

```
        new RecordReaderDataSetIterator(rr,batchSize,0,2);

    // Загрузить тестовые данные для оценивания
    RecordReader rrTest = new CSVRecordReader();
    rrTest.initialize(new FileSplit(new File(filenameTest)));
    DataSetIterator testIter =
        new RecordReaderDataSetIterator(rrTest,batchSize,0,2);

    //log.info("Построение модели....");
    MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(seed)
            .iterations(1)
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .learningRate(learningRate)
            .updater(Updater.NESTEROVS).momentum(0.9)
            .list()
            .layer(0, new DenseLayer.Builder().nIn(numInputs)
                    .nOut(numHiddenNodes)
                    .weightInit(WeightInit.XAVIER)
                    .activation(Activation.RELU)
                    .build())
            .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
                    .weightInit(WeightInit.XAVIER)
                    .activation(Activation.SOFTMAX)
                    .nIn(numHiddenNodes).nOut(numOutputs).build())
            .pretrain(false).backprop(true).build();

    MultiLayerNetwork model = new MultiLayerNetwork(conf);
    model.init();
model.setListeners(new ScoreIterationListener(10));  // Печатать оценку через каждые
                                                     // 10 обновлений параметров

    for ( int n = 0; n < nEpochs; n++) {
        model.fit( trainIter );
    }

    System.out.println("Оценивание модели....");
    Evaluation eval = new Evaluation(numOutputs);
    while(testIter.hasNext()){
        DataSet t = testIter.next();
        INDArray features = t.getFeatureMatrix();
        INDArray lables = t.getLabels();
        INDArray predicted = model.output(features,false);

        eval.eval(lables, predicted);
    }

    System.out.println(eval.stats());
    //------------------------------------------------------------------------
    // Обучение завершено. Следующий далее код служит для построения графиков
    // данных и предсказаний.

    double xMin = -15;
    double xMax = 15;
    double yMin = -15;
```

```
        double yMax = 15;

        // Оцениваем предсказания в каждой точке пространства (x, y) и наносим
        // точки на график
        int nPointsPerAxis = 100;
        double[][] evalPoints = new double[nPointsPerAxis*nPointsPerAxis][2];
        int count = 0;
        for( int i=0; i<nPointsPerAxis; i++ ){
            for( int j=0; j<nPointsPerAxis; j++ ){
                double x = i * (xMax-xMin)/(nPointsPerAxis-1) + xMin;
                double y = j * (yMax-yMin)/(nPointsPerAxis-1) + yMin;

                evalPoints[count][0] = x;
                evalPoints[count][1] = y;

                count++;
            }
        }

        INDArray allXYPoints = Nd4j.create(evalPoints);
        INDArray predictionsAtXYPoints = model.output(allXYPoints);

        // Поместить все обучающие данные в один массив и нанести их на график
        rr.initialize(new FileSplit(new File(filenameTrain)));
        rr.reset();
        int nTrainPoints = 500;
        trainIter = new RecordReaderDataSetIterator(rr,nTrainPoints,0,2);
        DataSet ds = trainIter.next();
        PlotUtil.plotTrainingData(ds.getFeatures(), ds.getLabels(), allXYPoints,
            predictionsAtXYPoints, nPointsPerAxis);


        // Получить тестовые данные, прогнать их через сеть, чтобы сгенерировать
        // предсказания, и нанести предсказания на график.
        rrTest.initialize(new FileSplit(new File(filenameTest)));
        rrTest.reset();
        int nTestPoints = 100;
        testIter = new RecordReaderDataSetIterator(rrTest,nTestPoints,0,2);
        ds = testIter.next();
        INDArray testPredicted = model.output(ds.getFeatures());
        PlotUtil.plotTestData(ds.getFeatures(), ds.getLabels(), testPredicted,
            allXYPoints, predictionsAtXYPoints, nPointsPerAxis);

        System.out.println("***************Конец примера******************");
    }
}
```

В следующих разделах мы разберем этот код по частям.

Подготовка входных данных

На вход подается нелинейный набор данных в таком виде:

```
1,-7.1239700674365,-5.05175898010314
0,1.80771566423302,0.770505522143023
1,8.43184823707231,-4.2287794074931
0,0.451276074541732,0.669574142606103
0,1.52519959303934,-0.953055551414968
```

Первый столбец содержит метку строки, а во втором и третьем представлены две независимые величины. Для чтения этого массива нам понадобится читатель данных, в качестве которого мы возьмем объект `CSVRecordReader`:

```
// Загрузить обучающие данные
RecordReader rr = new CSVRecordReader();
rr.initialize(new FileSplit(new File(filenameTrain)));
DataSetIterator trainIter = new RecordReaderDataSetIterator(rr,batchSize,0,2);
```

В итераторе мы сообщаем читателю, сколько имеется столбцов и в каком столбце находится метка.

Задание архитектуры сети

Нам нужен многослойный перцептрон, и для задания такой архитектуры сети мы воспользуемся объектом `MultiLayerConfiguration`:

```
// log.info("Построение модели....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(1)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .learningRate(learningRate)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
                .weightInit(WeightInit.XAVIER)
                .activation(Activation.RELU)
                .build())
        .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
                .weightInit(WeightInit.XAVIER)
                .activation(Activation.SOFTMAX)
                .nIn(numHiddenNodes).nOut(numOutputs).build())
        .pretrain(false).backprop(true).build();

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(10)); // Печатать оценку через каждые
                                                    // 10 обновлений параметров
```

В этом примере два слоя:

- `DenseLayer`;
- `OutputLayer`.

Рассмотрим особенности именно этой сетевой архитектуры.

Общие гиперпараметры

Алгоритм оптимизации задается с помощью следующего параметра:

```
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
```

Тем самым мы сообщаем DL4J, что собираемся использовать алгоритм стохастического градиентного спуска (СГС). Скорость обучения задается так:

```
.learningRate(learningRate)
```

Мы также сообщаем DL4J, что хотим использовать метод Нестерова при обновлении параметров, и задаем импульс равным 0.9:

```
.updater(Updater.NESTEROVS).momentum(0.9)
```

Первый скрытый слой

Первый скрытый слой принимает от входного слоя исходные данные, порожденные конвейером векторизации. Обычно это значения в диапазоне [–1.0,1.0] или [0.0,1.0], полученные после различных видов нормировки.

```
.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.RELU)
        .build())
```

Количество входных нейронов для этого слоя равно числу независимых величин во входных векторах:

```
.nIn(numInputs)
```

Количество выходов слоя равно числу нейронов в следующем слое нейронной сети. В данном случае это число представлено переменной `numHiddenNodes`. Для этого слоя веса инициализируются с помощью стратегии `WeightInit.XAVIER`, а в качестве функции активации взят блок линейной ректификации:

```
.weightInit(WeightInit.XAVIER)
.activation(Activation.RELU)
```

Теперь перейдем к рассмотрению следующего, выходного слоя.

Выходной слой для классификации

В выходном слое применяется функция активации softmax. Напомним, что эта функция хороша в случае, когда выходной слой осуществляет многоклассовую классификацию. Правда, сейчас мы строим бинарный классификатор и могли бы обойтись сигмоидной функцией активации.

```
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
```

Количество входов совпадает с количеством выходом предыдущего слоя (входного). Выходных нейронов два, потому что это бинарный классификатор. Функция softmax дает вероятности каждого из двух классов.

Сигмоида или softmax?

С точки зрения математики, сигмоидный слой с одним выходом – то же самое, что softmax-слой с двумя нейронами (при использовании многоклассовой перекрестной энтропии/отрицательного логарифмического правдоподобия и унитарного представления: [1,0] или [0,1] вместо 0 или 1). Представление в виде унитарных векторов обсуждается на врезке ниже.

Мы также указываем, что в роли функции потерь в выходном слое используется отрицательное логарифмическое правдоподобие, поскольку эта функция обычно применяется в связке с softmax.

Унитарные векторы

Унитарным называется вектор, в котором лишь один элемент равен 1.0, а все остальные – 0.0. Такой способ часто используется для представления категориальных целочисленных признаков по схеме «1 из K». В табл. 5.2 приведен пример стандартного двоичного представления чисел от 0 до 4 и их представления в виде унитарных векторов.

Таблица 5.2. ***Визуальные векторы***

Значение	Двоичное представление	Унитарное представление
0	000	00000001
1	001	00000010
2	010	00000100
3	011	00001000
4	100	00010000

Обучение модели

Обучение модели производится в цикле `for` по входному набору данных на протяжении заданного числа периодов:

```
for ( int n = 0; n < nEpochs; n++) {
    model.fit( trainIter );
}
```

Для обучения на всем наборе данных служит метод `.fit()` объекта класса `MultiLayerNetwork`. Этот класс применяет заданные гиперпараметры, в т. ч. размер мини-пакета, который был ранее задан при конструировании итератора по набору данных:

```
DataSetIterator testIter = new RecordReaderDataSetIterator(rrTest,batchSize,0,2);
```

Здесь переменная `batchSize` определяет, сколько примеров прочитать с диска и передать модели для обучения на одном пакете. В процессе обучения на консоли будут появляться сообщения вида:

```
o.d.o.l.ScoreIterationListener - Score at iteration 0 is 0.6313823699951172
o.d.o.l.ScoreIterationListener - Score at iteration 10 is 0.4763660430908203
o.d.o.l.ScoreIterationListener - Score at iteration 20 is 0.42963680267333987
o.d.o.l.ScoreIterationListener - Score at iteration 30 is 0.39850467681884766
o.d.o.l.ScoreIterationListener - Score at iteration 40 is 0.3672478103637695
```

Со временем ошибка уменьшается, и когда она станет близка к 0.0, мы можем сделать вывод, что обучение близко к завершению.

Оценивание модели

В следующем фрагменте производится оценивание многослойного перцептрона. Мы используем тестовый набор данных с помощью объекта `testIter`, который загружает фактические и предсказанные метки в объект класса `Evaluation`:

```
System.out.println("Evaluate model....");
Evaluation eval = new Evaluation(numOutputs);
```

```
while(testIter.hasNext()){
    DataSet t = testIter.next();
    INDArray features = t.getFeatureMatrix();
    INDArray lables = t.getLabels();
    INDArray predicted = model.output(features,false);

    eval.eval(lables, predicted);

}

System.out.println(eval.stats());
```

В главе 1 мы обсуждали F-меру и другие метрики оценивания модели. Метод `eval.stats()` выводит на консоль такие сообщения:

```
Оценивание модели....

Examples labeled as 0 classified by model as 0: 48 times
Examples labeled as 1 classified by model as 1: 52 times


==========================Scores========================================
 Accuracy:  1
 Precision: 1
 Recall:    1
 F1 Score:  1
========================================================================
```

Это довольно простой набор данных, и уже после 30 периодов DL4J смогла получить идеальную оценку (1.0) с точки зрения всех метрик. Перейдем теперь к более сложным примерам.

Моделирование рукописных цифр с помощью СНС

В начале этой главы мы сказали, что СНС хороши для классификации изображений. В следующем примере мы покажем, как загрузить и обучить модель, содержащую изображения рукописных цифр. Получившаяся СНС сможет классифицировать ранее не предъявлявшиеся изображения.

В этом примере конструируются итераторы для обучающего и тестового наборов данных. Мы обучаем модель на обучающем наборе, а затем оцениваем ее верность на зарезервированном тестовом наборе. В качестве обучающего используется набор данных MNIST (http://yann.lecun.com/exdb/mnist/).

Архитектура модели отличается от предыдущей другим составом слоев и гиперпараметрами. Эта конкретная СНС известна под названием «LeNet».

LeNet

Сеть LeNet состоит из ряда сверточных слоев, за которыми следуют слои max-пулинга. В примере 5.2 показано, как эта архитектура реализована в DL4J[6].

[6] LeCun et al., 1998. Gradient-based learning applied to document recognition // http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

Реализация СНС LeNet на Java

Пример 5.2 ❖ Построение сети LeNet для набора данных MNIST в DL4J

```
public class LenetMnistExample {
    private static final Logger log = LoggerFactory
        .getLogger(LenetMnistExample.class);
    public static void main(String[] args) throws Exception {
        int nChannels = 1; // Число входных каналов
        int outputNum = 10; // Число возможных исходов
        int batchSize = 64; // Размер тестового пакета
        int nEpochs = 1; // Число периодов обучения
        int iterations = 1; // Номер итерации
        int seed = 123;

        /*
            Создать итератор, указав размер пакета для одной итерации
        */
        log.info("Загружаются данные....");
        DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize,true,12345);
        DataSetIterator mnistTest = new MnistDataSetIterator(batchSize,false,12345);

        /*
            Построить нейронную сеть
        */
        log.info("Построение модели....");
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
                .seed(seed)
                .iterations(iterations) // Загружается число итераций
                .regularization(true).l2(0.0005)
                /*
                    Раскомментировать следующие строки, чтобы задать затухание и смещение
                    скорости обучения
                */
                .learningRate(.01) // .biasLearningRate(0.02)
                // .learningRateDecayPolicy(LearningRatePolicy.Inverse)
                // .lrPolicyDecayRate(0.001).lrPolicyPower(0.75)
                .weightInit(WeightInit.XAVIER)
                .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
                .updater(Updater.NESTEROVS).momentum(0.9)
                .list()
                .layer(0, new ConvolutionLayer.Builder(5, 5)
                        // nIn и nOut задают глубину. nIn равно числу каналов nChannels,
                        // а nOut – числу применяемых фильтров
                        .nIn(nChannels)
                        .stride(1, 1)
                        .nOut(20)
                        .activation(Activation.IDENTITY)
                        .build())
                .layer(1, new SubsamplingLayer
                    .Builder(SubsamplingLayer.PoolingType.MAX)
                        .kernelSize(2,2)
                        .stride(2,2)
                        .build())
```

```
            .layer(2, new ConvolutionLayer.Builder(5, 5)
                    // Заметьте, что в последующих слоях задавать nIn не нужно
                    .stride(1, 1)
                    .nOut(50)
                    .activation(Activation.IDENTITY)
                    .build())
            .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer
                .PoolingType.MAX)
                    .kernelSize(2,2)
                    .stride(2,2)
                    .build())
            .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
                    .nOut(500).build())
            .layer(5, new OutputLayer
                .Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
                    .nOut(outputNum)
                    .activation(Activation.SOFTMAX)
                    .build())
            .setInputType(InputType.convolutionalFlat(28,28,1)) // См. замечание ниже
            .backprop(true).pretrain(false).build();
    /*
    О строке .setInputType(InputType.convolutionalFlat(28,28,1)). Здесь делается
несколько вещей:
     (a) Добавляются препроцессоры, отвечающие за переход между слоями
свертки/пулинга и плотным слоем.
     (b) Производится дополнительная проверка конфигурации.
     (c) Там, где необходимо, задается значение nIn (число входных нейронов,
или глубина входа в случае СНС) для каждого слоя на основе размера предыдущего
слоя (но значения, заданные пользователем вручную, имеют приоритет). Класс
InputType применим и к другим слоям (РНС, МСП), а не только к СНС.
    Для обычных изображений (при использовании ImageRecordReader) вызывается метод
InputType.convolutional(height,width,depth).
    Читатель примеров MNIST – особый случай, он выводит полутоновые (nChannels=1)
изображения размера 28x28 пикселей в "разглаженном" формате вектора-строки
(т. е. векторы 1x784), поэтому мы вызываем метод convolutionalFlat.
    */

    MultiLayerNetwork model = new MultiLayerNetwork(conf);
    model.init();

    log.info("Обучение модели....");
    model.setListeners(new ScoreIterationListener(1));
    for( int i=0; i<nEpochs; i++ ) {
        model.fit(mnistTrain);
        log.info("*** Завершен период {} ***", i);

        log.info("Оценивание модели....");
        Evaluation eval = new Evaluation(outputNum);
        while(mnistTest.hasNext()){
            DataSet ds = mnistTest.next();
            INDArray output = model.output(ds.getFeatureMatrix(), false);
            eval.eval(ds.getLabels(), output);
```

```
            }
            log.info(eval.stats());
            mnistTest.reset();
        }
        log.info("***************Конец примера*******************");
    }
}
```

Далее мы разберем эту программу по частям.

Загрузка и векторизация входных изображений

В этом примере мы используем специальный итератор по набору данных `MnistDataSetIterator`. Связано это с тем, что набор MNIST хранится в специальном двоичном формате – это не множество отдельных файлов типа JPG или PNG, как можно было бы ожидать. Чтобы не усложнять изложения, просим вас просто поверить, что за кулисами исходные данные изображений правильно преобразуются в формат NDArray для обучения средствами DL4J. Вот как конструируются объекты `DataSetIterator`:

```
DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize,true,12345);
DataSetIterator mnistTest = new MnistDataSetIterator(batchSize,false,12345);
```

Здесь для загрузки обучающего и тестового наборов используются два разных итератора. Программа автоматически скачает набор данных MNIST из Интернета и распакует его на локальный диск.

Архитектура сети LeNet в DL4J

Как и в предыдущем примере, для описания архитектуры сети используется объект `MultiLayerConfiguration`. Но у этой сети гораздо больше слоев, и типы их отличаются от тех, что были в перцептроне.

```
/*
    Построить нейронную сеть
 */
log.info("Построение модели....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(iterations) // Задается число итераций
        .regularization(true).l2(0.0005)
        /*
            Раскомментировать следующие строки, чтобы задать затухание
            и смещение скорости обучения
         */
        //.biasLearningRate(0.02)
        //.learningRateDecayPolicy(LearningRatePolicy.Inverse)
        //.lrPolicyDecayRate(0.001).lrPolicyPower(0.75)
        .weightInit(WeightInit.XAVIER)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .list()
        .layer(0, new ConvolutionLayer.Builder(5, 5)
```

```
                // nIn и nOut задают глубину. nIn равно числу каналов nChannels,
                // а nOut – числу применяемых фильтров
                .nIn(nChannels)
                .stride(1, 1)
                .nOut(20)
                .activation(Activation.IDENTITY)
                .build())
        .layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
                .kernelSize(2,2)
                .stride(2,2)
                .build())
        .layer(2, new ConvolutionLayer.Builder(5, 5)
                // Заметьте, что в последующих слоях задавать nIn не нужно
                .stride(1, 1)
                .nOut(50)
                .activation(Activation.IDENTITY)
                .build())
        .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
                .kernelSize(2,2)
                .stride(2,2)
                .build())
        .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
                .nOut(500).build())
        .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
            .NEGATIVELOGLIKELIHOOD)
                .nOut(outputNum)
                .activation(Activation.SOFTMAX)
                .build())
        .setInputType(InputType.convolutionalFlat(28,28,1))
        .backprop(true).pretrain(false).build();
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

Общие гиперпараметры

Некоторые важнейшие гиперпараметры задаются в следующих строчках:

```
.seed(seed)
.iterations(iterations) // Задается число итераций
.regularization(true).l2(0.0005)
/*
    Раскомментировать следующие строки, чтобы задать затухание и смещение скорости обучения
*/
//.biasLearningRate(0.02)
//.learningRateDecayPolicy(LearningRatePolicy.Inverse).lrPolicyDecayRate(0.001)
//.lrPolicyPower(0.75)
.weightInit(WeightInit.XAVIER)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.NESTEROVS).momentum(0.9)
```

Ниже объясняется, что они означают.

- Регуляризация. В данном случае регуляризация включена (установлена в `true`) и выбрана L2-регуляризация с параметром 0.0005.
- Инициализация весов. Мы экспериментально выяснили, что в этом примере для инициализации весов хорошо работает метод Ксавье.

- Алгоритм оптимизации. В качестве алгоритма оптимизации задан СГС. Он часто встречается в задачах глубокого обучения, потому что дает хорошие результаты во многих случаях. Другие варианты алгоритмов оптимизации мы обсудим в главах 6 и 7.
- Корректор. В этом и во многих других примерах мы выбрали метод Нестерова, поскольку он хорошо работает на практике. Идея в том, что масштаб обновлений увеличивается, если обновления устойчиво ведут в одном направлении. Можно интерпретировать это как движение по гладкому и очень пологому холму. Поскольку направление движения ясно, можно выбирать шаги побольше.

Сверточные слои

Здесь мы наблюдаем типичную последовательность чередующихся сверточных слоев и слоев max-пулинга, описанную в главе 4. Вот определение первого сверточного слоя сети LeNet:

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
        // nIn и nOut задают глубину. nIn равно числу каналов nChannels,
        // а nOut - числу применяемых фильтров
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
```

Здесь мы видим пример применения паттерна Построитель для задания свойств сверточного слоя. Рассмотрим эти свойства подробнее.

- Размер фильтра. Создается фильтр размера 5 × 5.
- Число каналов во входных данных. В данном случае число каналов равно 1, потому что специальный итератор по этому набору данных преобразует исходное изображение в черно-белое. В других примерах число каналов может быть равно 3, что соответствует цветному изображению в формате RGB.
- Шаг. Шаг равен (1, 1), т. е. на каждой итерации фильтр сдвигается на один пиксель вправо, а по достижении конца строки – на один пиксель вниз.
- Функция активации. В качестве выхода этого сверточного слоя используется тождественная функция.

Замечание о тождественной функции активации
Сетевая архитектура LeNet (1998)[7] появилась на несколько лет раньше ReLU (2012)[8]. Замена тождественной функции на ReLU ускоряет сходимость СГС. Но мы решили придерживаться оригинальной архитектуры LeNet.

Сверточные слои и функции активации
В СНС, появившихся позднее, в сверточном слое стали использовать блоки линейной ректификации.

---

7 LeCun et al., 1998. Gradient-based learning applied to document recognition // http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

8 Krizhevsky, Sutskever and Hinton, 2012. ImageNet Classification with Deep Convolutional Neural Networks // http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf.

Слои max-пулинга

Далее показано, как создается пулинговый слой после первого сверточного слоя:

```
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
```

Перечислим его свойства.

- Размер фильтра. Размер фильтра равен (2,2), т. е. производится понижающая передискретизация – с фильтра 5 × 5 в предыдущем слое на сетку 2 × 2 в текущем.
- Задание шага. Величина шага равна (2,2), т. е. фильтр сдвигается на два пикселя вправо, а при переходе на следующую строку – на два пикселя вниз.

Выходной слой

Поскольку в нашей модели классификации число меток больше двух (цифры от 0 до 9), то нужен выходной слой с функцией активации softmax. Ниже показано, как это делается.

```
.layer(5, new OutputLayer
        .Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
        .nOut(outputNum)
        .activation(Activation.SOFTMAX)
        .build())
```

В выходном слое в качестве функции потерь используется отрицательное логарифмическое правдоподобие (как обычно в выходном softmax-слое), а число выходных нейронов равно числу классов (меток) в наборе данных.

Обучение СНС

Закончив задание архитектуры сети LeNet, мы можем инициализировать объект `MultiLayerNetwork` для обучения на входном наборе данных. В следующем фрагменте конструктор модели принимает созданную на предыдущей стадии конфигурацию, после чего модель инициализируется.

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

Инициализированную модель можно обучить на наборе данных MNIST, задав желаемое количество периодов:

```
log.info("Обучение модели....");
model.setListeners(new ScoreIterationListener(1));
for( int i=0; i<nEpochs; i++ ) {
    model.fit(mnistTrain);
    log.info("*** Завершен период {} ***", i);

    log.info("Оценивание модели....");
    Evaluation eval = new Evaluation(outputNum);
    while(mnistTest.hasNext()){
        DataSet ds = mnistTest.next();
        INDArray output = model.output(ds.getFeatureMatrix(), false);
```

```
        eval.eval(ds.getLabels(), output);
    }
    log.info(eval.stats());
    mnistTest.reset();
}
log.info("***************Конец примера*******************");
```

Цикл обучения устроен так же, как в предыдущем примере: итератор отвечает за передачу мини-пакета изображений методу модели `.fit()`. После завершения очередного периода мы проверяем, насколько хорошо обучилась модель, прогоняя ее на тестовом наборе. На консоль выводится значение F-меры после каждого периода. С течением времени оценка верности увеличивается, а ошибка уменьшается.

Моделирование последовательных данных с помощью рекуррентной нейронной сети

В этом разделе мы продемонстрируем порождающие и дискриминантные возможности рекуррентных нейронных сетей на двух примерах: порождение текста в стиле Шекспира и классификация временных рядов.

Порождение текста в стиле Шекспира с помощью LSTM-сети

Начнем с забавного примера моделирования произведений Шекспира. Мы обучим рекуррентную LSTM-сеть порождать новые строки в стиле Шекспира. Отметим, что текст рассматривается как последовательность символов, а модель предсказывает следующий наиболее вероятный символ на основе информации о ранее встречавшихся символах. Эту модель можно адаптировать для других типичных последовательностей данных, например журналов или показаний датчиков.

Фантастическая эффективность рекуррентных нейронных сетей
Этот пример отчасти вдохновлен статьей «The Unreasonable Effectiveness of Recurrent Neural Networks» (http://karpathy.github.io/2015/05/21/rnn-effectiveness/) в блоге Андрея Карпатого.

Исходный текст для обучения LSTM-сети
Этот пример обучен на полном собрании сочинений Вильяма Шекспира, загруженном с сайта проекта Гутенберг[9]. Обучение модели на других текстах не должно вызвать серьезных затруднений.

Общее описание процесса моделирования

В этом примере демонстрируется конфигурирование архитектуры рекуррентной нейронной сети в DL4J. Используются уже знакомые нам шаги:

- загрузка входных наборов данных для обучения;
- задание конфигурации сетевой архитектуры.

Но на этот раз мы познакомимся с новыми частями DL4J API, относящимися к рекуррентным нейронным сетям. Модель обучается на последовательных сим-

---

[9] *Complete Works of William Shakespeare* (https://www.gutenberg.org/ebooks/100): файл размера 5.3 МБ в кодировке UTF-8, содержит ~5.4 миллиона символов.

волах, составляющих строки шекспировских текстов. А затем мы просим ее породить новые строки на основе результатов обучения.

По мере продвижения обучения мы видим, как абракадабра вида

```
----- Sample 0 -----
lnee!
  Lhir tape shepyang? Nocw; mame. Budt hlant'nthely ler ild
   Py theu sfochill'ad my and ocs im nereepapd werer;
  Motadid. Mert hatterhirl. Iit nesdoesd'nlowhednanieivetranns deugheuind
   Bred yetide rathane fojlond thivh uweet.
   Thy lametom theuegfast lart souclalitoloe ilntangylrt or

----- Sample 1 -----
l,, ne agly
   Lot Bolncanbom bavantenfircasle womlidibl.
  NTERIOO. IrdmisfUoItolleeeddortiss hot buye.
   The hetenle of ile,
   'merlliydingiponI, bomgule? Shurtstarer of ate,
   Onbibly ot ire pomxatgillant, dakl.
   Oxt Mtanlonfye wiudsimotime raugadent deu'y ondtstes.
   If vonee.
  Whol touEde
```

постепенно сменяется более осмысленным текстом:

```
----- Sample 0 -----
ous reward me, Master Warce! I-will stay
    shall; for I one as mine lord.
  CLOTEN. Come, I will thigh, i'; and what wam! Hath dravelly
    The albowed out, Aside dismernicges could be a
    druck than there's thoughts, here is we with me and rag.
    Thou shalt love it doth my child.
  PERDITA. Ti

----- Sample 1 -----
on,
    Incie Paties, go, thurst with thy flounds by the bands. Exit COURSTIO
  FLORIZEL. Uncle, an if you,
    Abassom the man,
    Stars, you spite-hath loved.
  QUEEN MANGER On stay is! Who is mer?
  CLOTEN. Hang't, what I'll remain,
    Cap nothes same so here;
    My tens
```

Теперь посмотрим на код этой Java-программы.

Java-программа для моделирования Шекспира

Полный код примера приведен в примере 5.3.

Пример 5.3 ❖ Моделирование и порождение текстов в стиле Шекспира с помощью LSTM-сети в DL4J

```
public class GravesLSTMCharModellingExample {
    public static void main( String[] args ) throws Exception {
```

```
int lstmLayerSize = 200;  // Число блоков в каждом слое сети GravesLSTM
int miniBatchSize = 32;   // Размер мини-пакета на этапе обучения
int exampleLength = 1000; // Длина одной обучающей последовательности. Может быть
                          // увеличена
int tbpttLength = 50;     // Длина для усеченного обратного распространения во времени,
                          // т.е. параметры обновляются через каждые 50 символов
int numEpochs = 1;        // Число периодов обучения
int generateSamplesEveryNMinibatches = 10; // С какой частотой порождать примеры?
                                           // 1000 символов / 50 tbpttLength:
                                           // 20 обновлений параметров
                                           // на один мини-пакет
int nSamplesToGenerate = 4; // Сколько примеров порождать после каждого периода обучения
int nCharactersToSample = 300; // Длина одного порождаемого примера
String generationInitialization = null; // Факультативная инициализация символов;
                                        // если null, берется случайный символ

// Эта строка используется для инициализации LSTM последовательностью символов,
// которую следует продолжить
// Начальные символы должны принадлежать множеству CharacterIterator
// .getMinimalCharacterSet()
Random rng = new Random(12345);

// Получить DataSetIterator, отвечающий за преобразование текста в форму,
// пригодную для обучения сети GravesLSTM
CharacterIterator iter = getShakespeareIterator(miniBatchSize,exampleLength);
int nOut = iter.totalOutcomes();

// Задать конфигурацию сети:
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.RMSPROP)
    .list()
    .layer(0, new GravesLSTM.Builder().nIn(iter.inputColumns())
        .nOut(lstmLayerSize)
            .activation(Activation.TANH).build())
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX) // MCXENT + softmax для классификации
            .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
        .tBPTTBackwardLength(tbpttLength)
    .pretrain(false).backprop(true)
    .build();

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

```
        net.setListeners(new ScoreIterationListener(1));

        // Напечатать число параметров сети (и для каждого слоя)
        Layer[] layers = net.getLayers();
        int totalNumParams = 0;
        for( int i=0; i<layers.length; i++ ){
            int nParams = layers[i].numParams();
            System.out
                .println("Число параметров в слое " + i + ": " + nParams);
            totalNumParams += nParams;
        }
        System.out.println("Общее число параметров сети: " + totalNumParams);

        // Обучить сеть, затем породить и напечатать примеры
        int miniBatchNumber = 0;
        for( int i=0; i<numEpochs; i++ ){
            while(iter.hasNext()){
                DataSet ds = iter.next();
                net.fit(ds);
                if(++miniBatchNumber % generateSamplesEveryNMinibatches == 0){
                    System.out.println("--------------------");
                    System.out.println("Завершено " + miniBatchNumber +
                        " мини-пакетов размера " + miniBatchSize + "x" + exampleLength
                            + " символов" );
                    System.out.println("Выбираются символы из сети при заданной
                        инициализации \"" +
                  (generationInitialization == null ? "" :
                  generationInitialization) + "\"");
                    String[] samples = sampleCharactersFromNetwork(
                        generationInitialization,net,iter,rng,nCharactersToSample,
                            nSamplesToGenerate);
                    for( int j=0; j<samples.length; j++ ){
                        System.out.println("----- Пример " + j + " -----");
                        System.out.println(samples[j]);
                        System.out.println();
                    }
                }
            }

            iter.reset(); // Переустановить итератор для следующего периода
        }

        System.out.println("\n\nКонец примера");
    }

    /** Скачивает обучающие тексты Шекспира и сохраняет их во временном каталоге.
     * Затем конфигурирует простой DataSetIterator, который будет выполнять
     * векторизацию текста.
     * @param miniBatchSize число текстовых сегментов в каждом мини-пакете
     * @param sequenceLength число символов в одном текстовом сегменте.
     */
    public static CharacterIterator getShakespeareIterator(int miniBatchSize,
        int sequenceLength) throws Exception{
        // Полное собрание сочинений Вильяма Шекспира
```

```
    // файл 5.3 МБ в кодировке UTF-8, ~5.4 миллиона символов
    // https://www.gutenberg.org/ebooks/100
    String url = "https://s3.amazonaws.com/dl4j-distribution/pg100.txt";
    String tempDir = System.getProperty("java.io.tmpdir");
    String fileLocation = tempDir + "/Shakespeare.txt"; // Где сохранять загруженный файл
    File f = new File(fileLocation);
    if( !f.exists() ){
        FileUtils.copyURLToFile(new URL(url), f);
        System.out.println("Файл загружен в" + f.getAbsolutePath());
    } else {
        System.out.println("Используется существующий файл " + f.getAbsolutePath());
    }

    if(!f.exists()) throw new IOException("Файл не существует: " +
        fileLocation); // Проблемы при скачивании?

    // Какие символы разрешены? Все остальные удаляются
    char[] validCharacters = CharacterIterator.getMinimalCharacterSet();
    return new CharacterIterator(fileLocation, Charset.forName("UTF-8"),
        miniBatchSize, sequenceLength, validCharacters, new Random(12345));
}

/** Порождает пример из сети при заданной (факультативной, возможно, нулевой)
 * начальной строке, которую мы хотим продолжить. Отметим, что эта начальная строка
 * используется для всех примеров.
 * @param initialization String, возможно, null. Если null, выбрать случайный
 * символ для инициализации всех примеров.
 * @param charactersToSample Сколько символов выбирать из сети (начальные
 * символы не считаются).
 * @param net MultiLayerNetwork Сеть с одним или более рекуррентными слоями
 * GravesLSTM и выходным softmax-слоем.
 * @param iter CharacterIterator. Используется для перехода от индексов к символам.
 */
private static String[] sampleCharactersFromNetwork(String initialization,
                                                    MultiLayerNetwork net,
                                                    CharacterIterator iter,
                                                    Random rng,
                                                    int charactersToSample,
                                                    int numSamples ){
    // Настроить начальную строку. Если она не задана, взять случайный символ
    if( initialization == null ){
        initialization = String.valueOf(iter.getRandomCharacter());
    }

    // Преобразовать начальную строку во входные данные
    INDArray initializationInput = Nd4j.zeros(numSamples, iter.inputColumns(),
        initialization.length());
    char[] init = initialization.toCharArray();
    for( int i=0; i<init.length; i++ ){
        int idx = iter.convertCharacterToIndex(init[i]);
        for( int j=0; j<numSamples; j++ ){
            initializationInput.putScalar(new int[]{j,idx,i}, 1.0f);
        }
    }
```

```
        StringBuilder[] sb = new StringBuilder[numSamples];
        for( int i=0; i<numSamples; i++ ) sb[i] = new StringBuilder(initialization);

        // Выбрать пример из сети (подать примеры обратно на вход) по одному символу
        // за раз (для всех примеров)
        // Здесь выборка производится параллельно
        net.rnnClearPreviousState();
        INDArray output = net.rnnTimeStep(initializationInput);
        output = output.tensorAlongDimension(output.size(2)-1,1,0); // Получить выход
                                                                    // для последнего
                                                                    // временного шага

        for( int i=0; i<charactersToSample; i++ ){
            // Подготовить следующий вход (на одном временном шаге) путем выборки
            // из предыдущего выхода
            INDArray nextInput = Nd4j.zeros(numSamples,iter.inputColumns());
            // Выход - это распределение вероятностей. Производим из него выборку
            // для каждого генерируемого примера и добавляем к новому входу
            for( int s=0; s<numSamples; s++ ){
                double[] outputProbDistribution = new double[iter.totalOutcomes()];
                for( int j=0; j<outputProbDistribution.length; j++ )
                    outputProbDistribution[j] = output.getDouble(s,j);
                int sampledCharacterIdx =
                    sampleFromDistribution(outputProbDistribution,rng);

                // Подготовить вход для следующего временного шага
                nextInput.putScalar(new int[]{s,sampledCharacterIdx}, 1.0f);
                // Добавить выбранный символ в StringBuilder (выход для восприятия человеком)
                sb[s].append(iter.convertIndexToCharacter(sampledCharacterIdx));
            }
            output = net.rnnTimeStep(nextInput); // Один временной шаг прямого прохода
        }
        String[] out = new String[numSamples];
        for( int i=0; i<numSamples; i++ ) out[i] = sb[i].toString();
        return out;
    }

    /** Произвести выборку из заданного распределения вероятностей дискретных классов
     * и вернуть сгенерированный индекс класса.
     * @param distribution Распределение вероятностей классов. Сумма должна быть 1.0
     */
    public static int sampleFromDistribution( double[] distribution, Random rng ){
        double d = rng.nextDouble();
        double sum = 0.0;
        for( int i=0; i<distribution.length; i++ ){
            sum += distribution[i];
            if( d <= sum ) return i;
        }
        // Этого не должно быть, если распределение вероятностей корректно
        throw new IllegalArgumentException("Distribution is invalid? d="+d+",
            sum="+sum);
    }
}
```

Далее мы рассмотрим этот код по частям.

Подготовка входных данных и векторизация

Входные данные автоматически скачиваются и преобразуются в NDArray с помощью вспомогательных классов, входящих в состав примера:

```
CharacterIterator iter = getShakespeareIterator(miniBatchSize,exampleLength);
```

Оставляем изучение внутренних деталей любопытствующим.

Архитектура LSTM-сети

Как и в предыдущих примерах, для конфигурирования слоев сети применяется паттерн Построитель:

```
// Задать конфигурацию сети:
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.RMSPROP)
    .list()
    .layer(0, new GravesLSTM.Builder().nIn(iter.inputColumns())
        .nOut(lstmLayerSize)
            .activation(Activation.TANH).build())
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX) // MCXENT + softmax для классификации
            .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
        .tBPTTBackwardLength(tbpttLength)
    .pretrain(false).backprop(true)
    .build();
```

Общие замечания о гиперпараметрах. В этом примере мы снова видим СГС в качестве алгоритма оптимизации и скорость обучения 0.1. Включена регуляризация методом L2 с параметром 0.001. Указан корректор `RMSPROP`, что отличает этот пример от предыдущих.

- Скрытые слои. Используются специальные слои GravesLSTM с функцией активации tanh.
- Выходной слой. Выходной слой отличается от всего, что мы видели раньше. Это специальный слой `RnnOutputLayer`, умеющий обрабатывать различные типы выходов рекуррентной сети:

```
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
    .activation(Activation.SOFTMAX) // MCXENT + softmax для классификации
        .nIn(lstmLayerSize).nOut(nOut).build())
```

Мы снова использовали в качестве функции активации softmax, а в качестве функции потерь `LossFunction.MCXENT`.

RMSProp на практике

По построению RMSProp оказывает нормирующий эффект на масштаб обновлений. Это означает, что после применения этого корректора параметры с разными масштабами (в разных слоях или разные параметры в одном слое) приводятся примерно к одному масштабу.

Обучение LSTM-сети

Ниже показано, как производится обучение этой сети.

```
// Обучить сеть, затем породить и напечатать примеры
int miniBatchNumber = 0;
for( int i=0; i<numEpochs; i++ ){
    while(iter.hasNext()){
        DataSet ds = iter.next();
        net.fit(ds);
        if(++miniBatchNumber % generateSamplesEveryNMinibatches == 0){
            System.out.println("--------------------");
            System.out.println("Завершено " + miniBatchNumber +
                " мини-пакетов размера " + miniBatchSize + "x" + exampleLength
                    + " символов" );
            System.out.println("Выбираются символы из сети при заданной
                инициализации \"" +
          (generationInitialization == null ? "" :
          generationInitialization) + "\"");
            String[] samples = sampleCharactersFromNetwork(
                generationInitialization,net,iter,rng,nCharactersToSample,
                    nSamplesToGenerate);
            for( int j=0; j<samples.length; j++ ){
                System.out.println("----- Пример " + j + " -----");
                System.out.println(samples[j]);
                System.out.println();
            }
        }
    }

    iter.reset(); // Переустановить итератор для следующего периода
}

System.out.println("\n\nКонец примера");
```

В этом примере DL4J API используется несколько иначе. Вместо того чтобы передавать методу .fit() итератор, мы явно передаем ему мини-пакеты. Это позволяет лучше контролировать, что происходит между вызовами .fit() для каждого мини-пакета. В данном случае мы производим выборку из сети в процессе прохода по набору данных.

По ходу обучения на консоли печатаются сообщения, из которых видно, что функция потерь монотонно уменьшается:

```
o.d.o.l.ScoreIterationListener - Score at iteration 0 is 217.28348109866505
o.d.o.l.ScoreIterationListener - Score at iteration 1 is 213.24020789706773
o.d.o.l.ScoreIterationListener - Score at iteration 2 is 212.96001041971766
```

```
o.d.o.l.ScoreIterationListener - Score at iteration 3 is 175.06079409241767
o.d.o.l.ScoreIterationListener - Score at iteration 4 is 165.25272077487378
```

В этом примере сообщения о ходе обучения перемежаются выводом порожденных примеров предложений.

Порождение примеров в стиле Шекспира

Чтобы произвести выборку из сети, мы вызываем вспомогательный метод, который генерирует предложение:

```
String[] samples = sampleCharactersFromNetwork(generationInitialization,net,iter,
     rng,nCharactersToSample, nSamplesToGenerate);
for( int j=0; j<samples.length; j++ ){
    System.out.println("----- Пример " + j + " -----");
    System.out.println(samples[j]);
    System.out.println();
}
```

Этот код периодически производит вывод на консоль в процессе обучения.

Классификация временных рядов, содержащих показания датчика, с помощью LSTM-сети

В этом примере мы будем классифицировать последовательности примеров с помощью рекуррентной LSTM-сети. Используется набор данных Synthetic Control Chart Time Series Data Set (https://archive.ics.uci.edu/ml/datasets/Synthetic+Control+Chart+Time+Series), созданный в Калифорнийском университете в Ирвайне (UCI). Мы построим модель, которая будет относить одномерные временные ряды к одной из шести категорий:

- нормальный (C);
- циклический (B);
- с возрастающим трендом (E);
- с убывающим трендом (A);
- со смещением вверх (D);
- со смещением вниз (F).

На рис. 5.1 приведены примеры[10] из каждой категории.

Эти шесть классов последовательностей – отличный пример данных, которые могли бы генерироваться реальными датчиками, а с нашей точки зрения это практически полезный набор данных для работы.

Java-программа рекуррентной классификации

В примере 5.4 приведен код Java-программы, в которой мы строим модель для классификации последовательностей из набора данных Synthetic Control Chart Time Series Data Set, созданного в UCI.

[10] Рисунок взят со страницы https://archive.ics.uci.edu/ml/datasets/Synthetic+Control+Chart+Time+Series.

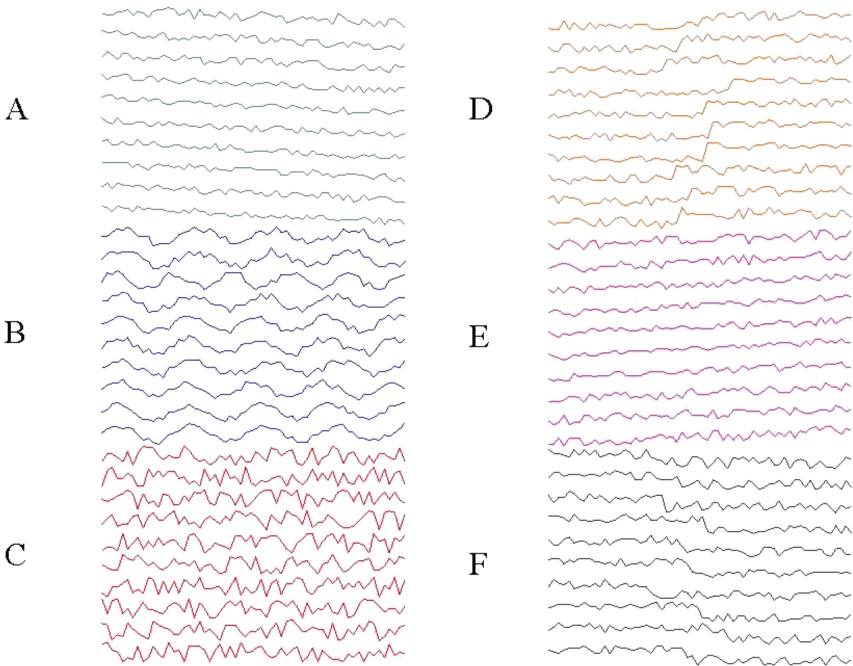


Рис. 5.1 ❖ Синтетические последовательности временных рядов из репозитория UCI

Пример 5.4 ❖ Java-программа для классификации последовательностей из набора данных UCI

```
public class UCISequenceClassificationExample {
    private static final Logger log = LoggerFactory
        .getLogger(UCISequenceClassificationExample.class);

    // 'baseDir': базовый каталог для данных. Измените его, если хотите поместить
    // данные в другое место
    private static File baseDir = new File("src/main/resources/uci/");
    private static File baseTrainDir = new File(baseDir, "train");
    private static File featuresDirTrain = new File(baseTrainDir, "features");
    private static File labelsDirTrain = new File(baseTrainDir, "labels");
    private static File baseTestDir = new File(baseDir, "test");
    private static File featuresDirTest = new File(baseTestDir, "features");
    private static File labelsDirTest = new File(baseTestDir, "labels");

    public static void main(String[] args) throws Exception {
        downloadUCIData();

        // ----- Загрузить обучающие данные -----
        // Отметим, что имеется 450 обучающих файлов для признаков:
        // от train/features/0.csv до train/features/449.csv
        SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
        trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
            .getAbsolutePath() + "/%d.csv", 0, 449));
        SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
        trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
```

```
    .getAbsolutePath() + "/%d.csv", 0, 449));

int miniBatchSize = 10;
int numLabelClasses = 6;
DataSetIterator trainData = new SequenceRecordReaderDataSetIterator(
    trainFeatures, trainLabels, miniBatchSize, numLabelClasses,
    false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

// Нормировать обучающие данные
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainData);              // Собрать статистику обучающих данных
trainData.reset();

// Использовать собранную статистику для нормировки на лету. Каждый объект
// DataSet, возвращенный итератором 'trainData', будет нормирован
trainData.setPreProcessor(normalizer);


// ----- Загрузить тестовые данные -----
// Та же процедура, что для обучающих данных.
SequenceRecordReader testFeatures = new CSVSequenceRecordReader();
testFeatures.initialize(new NumberedFileInputSplit(featuresDirTest
    .getAbsolutePath() + "/%d.csv", 0, 149));
SequenceRecordReader testLabels = new CSVSequenceRecordReader();
testLabels.initialize(new NumberedFileInputSplit(
    labelsDirTest.getAbsolutePath() + "/%d.csv", 0, 149));

DataSetIterator testData = new SequenceRecordReaderDataSetIterator(
    testFeatures, testLabels, miniBatchSize, numLabelClasses,
    false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

testData.setPreProcessor(normalizer);   // Отметим, что используется такой же
                                        // процесс нормировки, как для обучающих
                                        // данных


// ----- Сконфигурировать сеть -----
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(123)      // Начальное значение генератора случайных чисел
                        // для воспроизводимости результатов. Не обязательно
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .learningRate(0.005)
        .gradientNormalization(GradientNormalization
            .ClipElementWiseAbsoluteValue)  // Требуется не всегда, но для этого
                                            // набора данных полезно
        .gradientNormalizationThreshold(0.5)
        .list()
        .layer(0, new GravesLSTM.Builder().activation(Activation.TANH).nIn(1)
            .nOut(10).build())
        .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction
            .MCXENT)
                .activation(Activation.SOFTMAX).nIn(10).nOut(numLabelClasses)
                    .build())
```

```
            .pretrain(false).backprop(true).build();

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

    net.setListeners(new ScoreIterationListener(20));   // Печатать оценку (значение
                                                    // функции потерь) через каждые 20 итераций

    // -- Обучить сеть, вычисляя верность на тестовом наборе в конце каждого периода --
    int nEpochs = 40;
    String str = "Test set evaluation at epoch %d: Accuracy = %.2f, F1 = %.2f";
    for (int i = 0; i < nEpochs; i++) {
        net.fit(trainData);

        // Оценить на тестовом наборе:
        Evaluation evaluation = net.evaluate(testData);
        log.info(String.format(str, i, evaluation.accuracy(), evaluation.f1()));

        testData.reset();
        trainData.reset();
    }

    log.info("----- Конец примера -----");
}

// Этот метод скачивает данные и преобразует их из формата "один временной ряд
// на строку" в последовательность в формате CSV, которую может прочитать
// библиотека DataVec (CsvSequenceRecordReader) и DL4J
private static void downloadUCIData() throws Exception {
    if (baseDir.exists()) return;  // Данные уже есть, не надо скачивать снова
    String url =
        "https:// archive.ics.uci.edu/ml/machine-learning-databases/
        synthetic_control-mld/synthetic_control.data";
    String data = IOUtils.toString(new URL(url));

    String[] lines = data.split("\n");

    // Создать каталоги
    baseDir.mkdir();
    baseTrainDir.mkdir();
    featuresDirTrain.mkdir();
    labelsDirTrain.mkdir();
    baseTestDir.mkdir();
    featuresDirTest.mkdir();
    labelsDirTest.mkdir();

    int lineCount = 0;
    List<Pair<String, Integer>> contentAndLabels = new ArrayList<>();
    for (String line : lines) {
        String transposed = line.replaceAll(" +", "\n");

        // Метки: первые 100 примеров (строк) имеют метку 0, следующие 100 примеров –
        // метку 1 и т. д.
        contentAndLabels.add(new Pair<>(transposed, lineCount++ / 100));
    }

    // Перемешать и разделить на обучающий и тестовый наборы:
```

```
        Collections.shuffle(contentAndLabels, new Random(12345));

        int nTrain = 450;// 75% train, 25% test
        int trainCount = 0;
        int testCount = 0;
        for (Pair<String, Integer> p : contentAndLabels) {
            // Записать выход в формате, который мы сможем прочитать, поместив
            // в нужный каталог
            File outPathFeatures;
            File outPathLabels;
            if (trainCount < nTrain) {
                outPathFeatures = new File(featuresDirTrain, trainCount + ".csv");
                outPathLabels = new File(labelsDirTrain, trainCount + ".csv");
                trainCount++;
            } else {
                outPathFeatures = new File(featuresDirTest, testCount + ".csv");
                outPathLabels = new File(labelsDirTest, testCount + ".csv");
                testCount++;
            }

            FileUtils.writeStringToFile(outPathFeatures, p.getFirst());
            FileUtils.writeStringToFile(outPathLabels, p.getSecond().toString());
        }
    }
}
```

Далее мы разберем этот код по частям.

Подготовка входных данных и векторизация

В примере 5.4 нам не нужно вручную извлекать данные, потому что это делает за нас метод `downloadUCIData()`. Заодно он разбивает 600 примеров временных рядов на обучающий (450 примеров) и тестовый (150 примеров) наборы. Затем он записывает данные в формате, пригодном для загрузки классом `CSVSequenceRecordReader`. Этот формат подразумевает по одному временному ряду на файл и отдельный файл для меток.

Например, файл *train/features/0.csv* содержит признаки для примера 0, а файл *train/labels/0.csv* – метку примера 0. Поскольку временные ряды одномерные, в каждом CSV-файле будет только один столбец. Обычно каждый столбец содержит несколько значений (например, по одному временному шагу на строку). Поскольку для каждого временного ряда есть только одна метка, в каждом файле меток будет всего одно значение.

Из приведенного ниже фрагмента кода загрузки обучающих данных видно, как используется класс `SequenceRecordReader` для чтения CSV-файла[11], а затем создается объект `DataSetIterator` для работы с этим типом последовательных данных:

```
// ----- Загрузить обучающие данные -----
// Отметим, что имеется 450 обучающих файлов для признаков:
// от train/features/0.csv до train/features/449.csv
SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
```

[11] Дополнительные сведения см. по адресу http://deeplearning4j.org/usingrnns#data.

```
    .getAbsolutePath() + "/%d.csv", 0, 449));
SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
    .getAbsolutePath() + "/%d.csv", 0, 449));

int miniBatchSize = 10;
int numLabelClasses = 6;
DataSetIterator trainData = new SequenceRecordReaderDataSetIterator(
    trainFeatures, trainLabels, miniBatchSize, numLabelClasses,
    false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

// Нормировать обучающие данные
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainData);          // Собрать статистику обучающих данных
trainData.reset();

// Использовать собранную статистику для нормировки на лету. Каждый объект
// DataSet, возвращенный итератором 'trainData', будет нормирован
trainData.setPreProcessor(normalizer);
```

В конце фрагмента мы видим, как с помощью объекта `DataNormalization` собирается статистика по набору данных. Собранная статистика используется для нормировки обучающего набора, что повышает качество результатов обучения.

Архитектура и обучение сети

Мы используем рекуррентную LSTM-сеть, которая конфигурируется с помощью объекта `MultiLayerConfiguration` следующим образом:

```
// ----- Сконфигурировать сеть -----
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(123)    // Начальное значение генератора случайных чисел
                      // для воспроизводимости результатов. Не обязательно
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
           .iterations(1)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .learningRate(0.005)
        .gradientNormalization(GradientNormalization
           .ClipElementWiseAbsoluteValue)  // Требуется не всегда, но для этого
                                           // набора данных полезно
        .gradientNormalizationThreshold(0.5)
        .list()
        .layer(0, new GravesLSTM.Builder().activation(Activation.TANH).nIn(1)
           .nOut(10).build())
        .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
               .activation(Activation.SOFTMAX).nIn(10).nOut(numLabelClasses)
                  .build())
        .pretrain(false).backprop(true).build();
```

Здесь нам нужен только один LSTM-слой, соединенный с выходным softmax-слоем. Используются метод инициализации весов `XAVIE`, стратегия обновления `NESTEROVS` и скорость обучения `0.005`.

Как и в остальных примерах, мы производим обучение в течение нескольких периодов, пока частота ошибок не упадет до приемлемого уровня:

```
// ----- Обучить сеть, вычисляя верность на тестовом наборе в конце каждого периода ---
int nEpochs = 40;
String str = "Test set evaluation at epoch %d: Accuracy = %.2f, F1 = %.2f";
for (int i = 0; i < nEpochs; i++) {
    net.fit(trainData);

    // Оценить на тестовом наборе:
    Evaluation evaluation = net.evaluate(testData);
    log.info(String.format(str, i, evaluation.accuracy(), evaluation.f1()));

    testData.reset();
    trainData.reset();
}
log.info("----- Конец примера -----");
```

Здесь мы снова вызываем метод `.fit()` для обучения на векторизованном и нормированном обучающих наборах данных, а затем используем объект `Evaluation`, чтобы измерить, насколько хорошо модель обобщается на зарезервированные тестовые данные.

Применение автокодировщиков для обнаружения аномалий

Чтобы продемонстрировать использование автокодировщиков на практике, мы рассмотрим обнаружение аномалий в наборе данных MNIST с помощью простого автокодировщика без предобучения.

Нашей целью является выявление аномальных цифр, не похожих на обычные. Для этого мы воспользуемся ошибкой реконструкции: у стереотипных примеров ошибка реконструкции должна быть мала, а у аномальных велика.

Порядок конфигурирования модели такой же, как в других примерах, но ее архитектура другая. В автокодировщике большинство слоев полносвязные – типа `DenseLayer`, но особый интерес для нас представляет последовательность слоев, которые сначала «сужают», а затем – в выходном слое – «расширяют» данные.

Java-программа автокодировщика

В примере 5.5 приведен код программы (https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/feedforward/anomalydetection/MNISTAnomalyExample.java) для обнаружения аномалий.

Пример 5.5 ❖ Java-программа обнаружения аномалий с помощью автокодировщика

```
public class MNISTAnomalyExample {

    public static void main(String[] args) throws Exception {

        // Сконфигурировать сеть. Входов и выходов 784 (поскольку в наборе MNIST
        // изображения имеют размер 28x28). 784 -> 250 -> 10 -> 250 -> 784
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
                .seed(12345)
                .iterations(1)
                .weightInit(WeightInit.XAVIER)
                .updater(Updater.ADAGRAD)
                .activation(Activation.RELU)
```

```
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .learningRate(0.05)
        .regularization(true).l2(0.0001)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
                .build())
        .layer(1, new DenseLayer.Builder().nIn(250).nOut(10)
                .build())
        .layer(2, new DenseLayer.Builder().nIn(10).nOut(250)
                .build())
        .layer(3, new OutputLayer.Builder().nIn(250).nOut(784)
                .lossFunction(LossFunctions.LossFunction.MSE)
                .build())
        .pretrain(false).backprop(true)
                .build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.setListeners(Collections.singletonList((IterationListener) new
    ScoreIterationListener(1)));

// Загрузить данные и разделить их на обучающий и тестовый наборы
// В обучающем 40000 примеров, в тестовов 10000
DataSetIterator iter = new MnistDataSetIterator(100,50000,false);

List<INDArray> featuresTrain = new ArrayList<>();
List<INDArray> featuresTest = new ArrayList<>();
List<INDArray> labelsTest = new ArrayList<>();

Random r = new Random(12345);
while(iter.hasNext()){
    DataSet ds = iter.next();
    SplitTestAndTrain split = ds.splitTestAndTrain(80, r);  // Разделение 80/20
                                                            // (miniBatch = 100)
    featuresTrain.add(split.getTrain().getFeatureMatrix());
    DataSet dsTest = split.getTest();
    featuresTest.add(dsTest.getFeatureMatrix());
    INDArray indexes = Nd4j.argMax(dsTest.getLabels(),1);   // Преобразование
                                                            // из унитарного
                                                            // представления в индекс
    labelsTest.add(indexes);
}

// Обучить модель:
int nEpochs = 30;
for( int epoch=0; epoch<nEpochs; epoch++ ){
    for(INDArray data : featuresTrain){
        net.fit(data,data);
    }
    System.out.println("Epoch " + epoch + " complete");
}

// Оценить модель на тестовых данных
// Каждый пример (цифра) из тестового набора оценивается отдельно
// Затем помещаем тройки (оценка, цифра, данные INDArray) в списки и сортируем
// по оценке
```

```
    // Это позволяет получить N лучших и N худших цифр каждого типа
    Map<Integer,List<Triple<Double,Integer,INDArray>>> listsByDigit =
        new HashMap<>();
    for( int i=0; i<10; i++ ) listsByDigit.put(i,new ArrayList<Triple<Double,
        Integer,INDArray>>());

    int count = 0;
    for(int i=0; i<featuresTest.size(); i++ ){
        INDArray testData = featuresTest.get(i);
        INDArray labels = labelsTest.get(i);
        int nRows = testData.rows();
        for( int j=0; j<nRows; j++){
            INDArray example = testData.getRow(j);
            int label = (int)labels.getDouble(j);
            double score = net.score(new DataSet(example,example));
            listsByDigit.get(label).add(new ImmutableTriple<>(score, count++,
                example));
        }
    }

    // Отсортировать данные по оценке, отдельно для каждой цифры
    Comparator<Triple<Double, Integer, INDArray>> c
        = new Comparator<Triple<Double, Integer, INDArray>>() {
        @Override
        public int compare(Triple<Double, Integer, INDArray> o1, Triple<Double,
            Integer, INDArray> o2) {
            return Double.compare(o1.getLeft(),o2.getLeft());
        }
    };

    for(List<Triple<Double, Integer, INDArray>> list : listsByDigit.values()){
        Collections.sort(list, c);
    }

    // Выбрать 5 лучших и 5 худших примеров (с точки зрения ошибки реконструкции)
    // для каждой цифры
    List<INDArray> best = new ArrayList<>(50);
    List<INDArray> worst = new ArrayList<>(50);
    for( int i=0; i<10; i++ ){
        List<Triple<Double,Integer,INDArray>> list = listsByDigit.get(i);
        for(int j=0; j<5; j++ ){
            best.add(list.get(j).getRight());
            worst.add(list.get(list.size()-j-1).getRight());
        }
    }

    // Визуализировать лучшие и худшие цифры
    MNISTVisualizer bestVisualizer = new MNISTVisualizer(2.0,best,"Best
        (Low Rec. Error)");
    bestVisualizer.visualize();

    MNISTVisualizer worstVisualizer = new MNISTVisualizer(2.0,worst,"Worst
        (High Rec. Error)");
    worstVisualizer.visualize();
}
```

```
    public static class MNISTVisualizer {
        private double imageScale;
        private List<INDArray> digits;      // Цифры (в виде векторов-строк),
                                            // каждая в своем INDArray
        private String title;
        private int gridWidth;

        public MNISTVisualizer(double imageScale, List<INDArray> digits,
            String title ) {
            this(imageScale, digits, title, 5);
        }

        public MNISTVisualizer(double imageScale, List<INDArray> digits,
            String title, int gridWidth ) {
            this.imageScale = imageScale;
            this.digits = digits;
            this.title = title;
            this.gridWidth = gridWidth;
        }

        public void visualize(){
            JFrame frame = new JFrame();
            frame.setTitle(title);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            JPanel panel = new JPanel();
            panel.setLayout(new GridLayout(0,gridWidth));

            List<JLabel> list = getComponents();
            for(JLabel image : list){
                panel.add(image);
            }

            frame.add(panel);
            frame.setVisible(true);
            frame.pack();
        }

        private List<JLabel> getComponents(){
            List<JLabel> images = new ArrayList<>();
            for( INDArray arr : digits ){
                BufferedImage bi = new BufferedImage(28,28,BufferedImage
                    .TYPE_BYTE_GRAY);
                for( int i=0; i<784; i++ ){
                    bi.getRaster().setSample(i % 28, i / 28, 0, (int)(255*arr
                        .getDouble(i)));
                }
                ImageIcon orig = new ImageIcon(bi);
                Image imageScaled = orig.getImage().getScaledInstance((int)
                    (imageScale*28),(int)(imageScale*28),Image.SCALE_REPLICATE);
                ImageIcon scaled = new ImageIcon(imageScaled);
                images.add(new JLabel(scaled));
            }
            return images;
        }
    }
}
```

Далее мы разберем этот код по частям.

Подготовка входных данных

В следующем фрагменте набор данных MNIST загружается с помощью специального итератора.

```
// Загрузить данные и разделить их на обучающий и тестовый наборы
// В обучающем 40000 примеров, в тестовов 10000
DataSetIterator iter = new MnistDataSetIterator(100,50000,false);

List<INDArray> featuresTrain = new ArrayList<>();
List<INDArray> featuresTest = new ArrayList<>();
List<INDArray> labelsTest = new ArrayList<>();

Random r = new Random(12345);
while(iter.hasNext()){
    DataSet ds = iter.next();
    SplitTestAndTrain split = ds.splitTestAndTrain(80, r);  // Разделение 80/20
                                                            // (miniBatch = 100)
    featuresTrain.add(split.getTrain().getFeatureMatrix());
    DataSet dsTest = split.getTest();
    featuresTest.add(dsTest.getFeatureMatrix());
    INDArray indexes = Nd4j.argMax(dsTest.getLabels(),1);   // Преобразование из унитарного
                                                            // представления в индекс
    labelsTest.add(indexes);
}
```

Обучающий и тестовый наборы данных обрабатываются иначе, чем в предыдущих примерах, так что мы можем продемонстрировать дополнительные API DL4J и ND4J. В цикле `while` мы вручную разделяем данные на обучающие и тестовые.

Архитектура и обучение сети автокодировщика

Как уже отмечалось, сеть автокодировщика обычно имеет вид сужающейся воронки, которая затем расширяется в выходном слое до размера, равного размеру входного слоя. Перед автокодировщиком стоит задача найти наиболее эффективное представление входных данных. Ниже приведен код конфигурирования автокодировщика.

```
// Сконфигурировать сеть. Входов и выходов 784 (поскольку в наборе MNIST
// изображения имеют размер 28x28). 784 -> 250 -> 10 -> 250 -> 784
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(12345)
        .iterations(1)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.ADAGRAD)
        .activation(Activation.RELU)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .learningRate(0.05)
        .regularization(true).l2(0.0001)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
                .build())
```

```
        .layer(1, new DenseLayer.Builder().nIn(250).nOut(10)
                .build())
        .layer(2, new DenseLayer.Builder().nIn(10).nOut(250)
                .build())
        .layer(3, new OutputLayer.Builder().nIn(250).nOut(784)
                .lossFunction(LossFunctions.LossFunction.MSE)
                .build())
        .pretrain(false).backprop(true)
                .build();
```

В этой архитектуре четыре слоя, причем в последнем 784 блока – столько же, сколько во входном. В качестве функции активации всюду используется ReLU, поскольку, как мы обнаружили, она работает для этого набора данных лучше всех остальных.

Обучение автокодировщика организовано так же, как в других примерах:

```
// Обучить модель:
int nEpochs = 30;
for( int epoch=0; epoch<nEpochs; epoch++ ){
    for(INDArray data : featuresTrain){
        net.fit(data,data);
    }
    System.out.println("Период " + epoch + " завершен");
}
```

Обучение производится в цикле по эпохам. В следующей строке мы видим интересный вариант API:

```
net.fit(data,data);
```

Здесь `data` используется также в качестве выхода сети. Дело в том, что автокодировщик обучается реконструировать сами данные, поэтому они присутствуют как на входе, так и на выходе, из-за чего метод `fit()` вызывается несколько иначе, чем в других случаях.

Оценивание модели

Программа генерирует два изображения, содержащих цифры с лучшей и худшей ошибками реконструкции. На рис. 5.2 показаны рукописные цифры, для которых ошибка реконструкции минимальна.

А на рис. 5.3 показаны цифры с максимальной ошибкой реконструкции.

Невооруженным взглядом видно, что некоторые изображения действительно выглядят аномально на фоне других. Мы не всегда понимаем, что считать аномальными данными, но с автокодировщиком заранее знать и не нужно.

Рис. 5.2 ❖ Цифры, которым автокодировщик обучился лучше всего

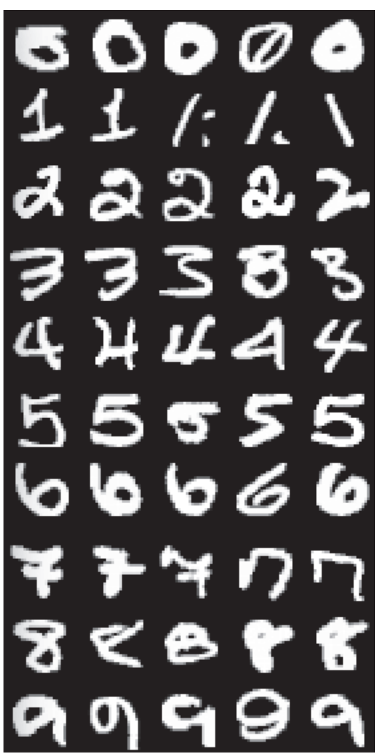


Рис. 5.3 ❖ Рукописные цифры, для которых обучение вызвало трудности

Использование вариационных автокодировщиков для реконструкции цифр из набора MNIST

В главе 3 было введено понятие вариационного автокодировщика (VAE) как способа реконструкции данных без учителя.

У VAE много применений, из них наиболее типичны следующие:

- обучение признакам без учителя или с частичным привлечением учителя. В последнем случае имеется много непомеченных данных и немного помеченных (это позволяет достичь гораздо лучших результатов, чем использование одних лишь помеченных данных, которых мало);
- обнаружение аномалий (без учителя);
- в качестве порождающей модели (например, для порождения примеров), которые могут генерировать изображения (как в этом примере), но также и предложения[12].

[12] Bowman et al., 2015. Generating Sentences from a Continuous Space // https://arxiv.org/abs/1511.06349.

В следующей программе демонстрируются порождающие возможности VAE на примере порождения новых цифр, похожих на представленные в наборе MNIST.

Программа реконструкции цифр для набора MNIST

В примере 5.6 приведен код простой программы обучения VAE на наборе данных MNIST с целью демонстрации порождающих возможностей VAE. В нем сознательно взято небольшое скрытое состояние Z (два значения) для визуализации на двумерной сетке.

По завершении обучения программа строит два изображения:

- реконструкции цифр MNIST путем выборки из латентного пространства состояний;
- значения скрытых состояний в процессе обучения (через каждые *N* минипакетов).

В обоих случаях в верхней части рисунка отображается ползунок. Двигая его, можно наблюдать за изменением реконструкции и латентного пространства во времени.

Пример 5.6 ❖ Моделирование набора данных MNIST с помощью вариационного кодировщика на Java

```
public class VariationalAutoEncoderExample {
    private static final Logger log =
        LoggerFactory.getLogger(VariationalAutoEncoderExample.class);

    public static void main(String[] args) throws IOException {
        int minibatchSize = 128;
        int rngSeed = 12345;
        int nEpochs = 150;      // Число периодов обучения

        // Параметры построения графиков
        int plotEveryNMinibatches = 100;    // Частота сбора данных для последующего
                                            // нанесения на графики
        double plotMin = -4;                // Минимальные значения (x и y)
        double plotMax = 4;                 // Максимальные значения (x и y)
        int plotNumSteps = 16;              // Число шагов для вывода реконструкций
                                            // от plotMin до plotMax

        // Набор данных MNIST
        DataSetIterator trainIter = new MnistDataSetIterator(minibatchSize, true,
            rngSeed);

        // Конфигурация нейронной сети
        Nd4j.getRandom().setSeed(rngSeed);
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(rngSeed)
            .iterations(1).optimizationAlgo(OptimizationAlgorithm
                .STOCHASTIC_GRADIENT_DESCENT)
            .learningRate(1e-3)
            // .updater(Updater.RMSPROP).rmsDecay(0.95)
            .updater(Updater.ADAM)
            .weightInit(WeightInit.RELU)
            .regularization(true).l2(1e-5)
            .list()
```

```
    .layer(0, new VariationalAutoencoder.Builder()
        .activation(Activation.LEAKYRELU)
        .encoderLayerSizes(512, 512)        // 2 слоя кодирования, каждый размером 256
        .decoderLayerSizes(512, 512)        // 2 слоя декодирования,
                                            // каждый размером 256
        .pzxActivationFunction("identity")  // p(z|data) activation function
        .reconstructionDistribution(new BernoulliReconstructionDistribution(
            // распределение Бернулли для p(data|z)
            Activation.SIGMOID.getActivationFunction()))
        .nIn(28 * 28)                       // Размер входа: 28x28
        .nOut(2)                            // Размер пространства латентных
                                            // переменной: p(z|x). Здесь 2 измерения
                                            // для построения графиков.
                                            // В общем случае может быть больше
        .build())
    .pretrain(true).backprop(false).build();

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

// Получить слой вариационного автокодировщика
org.deeplearning4j.nn.layers.variational.VariationalAutoencoder vae
    = (org.deeplearning4j.nn.layers.variational.VariationalAutoencoder)
        net.getLayer(0);

// Тестовые данные для построения графиков
DataSet testdata = new MnistDataSetIterator(10000, false, rngSeed).next();
INDArray testFeatures = testdata.getFeatures();
INDArray testLabels = testdata.getLabels();

// Сетка X/Y, от plotMin до plotMax
INDArray latentSpaceGrid =
    getLatentSpaceGrid(plotMin, plotMax, plotNumSteps);

// Списки, в которых будут хранится данные для последующего построения графиков
List<INDArray> latentSpaceVsEpoch = new ArrayList<>(nEpochs + 1);

// Собрать и сохранить значения латентной переменной до начала обучения
INDArray latentSpaceValues = vae.activate(testFeatures, false);
latentSpaceVsEpoch.add(latentSpaceValues);
List<INDArray> digitsGrid = new ArrayList<>();

// Обучить
int iterationCount = 0;
for (int i = 0; i < nEpochs; i++) {
    log.info("Starting epoch {} of {}",(i+1),nEpochs);
    while (trainIter.hasNext()) {
        DataSet ds = trainIter.next();
        net.fit(ds);

        // Каждые N=100 мини-пакетов:
        // (a) собрать значения латентных переменных на тестовом наборе
        // (b) собрать реконструкции в каждой точке сетки
        if (iterationCount++ % plotEveryNMinibatches == 0) {
            latentSpaceValues = vae.activate(testFeatures, false);
```

```
                latentSpaceVsEpoch.add(latentSpaceValues);

                INDArray out = vae.generateAtMeanGivenZ(latentSpaceGrid);
                digitsGrid.add(out);
            }
        }

        trainIter.reset();
    }

    // Нанести на график тестовый набор MNIST - по одной оси номер итерации,
    // по другой значения латентных переменных.
    // (По умолчанию через каждые 100 мини-пакетов)
    PlotUtil.plotData(latentSpaceVsEpoch, testLabels, plotMin, plotMax,
        plotEveryNMinibatches);

    // Нанести на график реконструкции
    double imageScale = 2.0;        // Увеличьте или уменьшите эту величину,
                                    // чтобы изменить масштаб цифр
    PlotUtil.MNISTLatentSpaceVisualizer v =
        new PlotUtil.MNISTLatentSpaceVisualizer(imageScale, digitsGrid,
            plotEveryNMinibatches);
    v.visualize();
  }

  // Этот метод просто возвращает двумерную сетку: (x,y), где x и y изменяются
  // от plotMin до plotMax
  private static INDArray getLatentSpaceGrid(double plotMin, double plotMax,
      int plotSteps) {
      INDArray data = Nd4j.create(plotSteps * plotSteps, 2);
      INDArray linspaceRow = Nd4j.linspace(plotMin, plotMax, plotSteps);
      for (int i = 0; i < plotSteps; i++) {
          data.get(NDArrayIndex.interval(i * plotSteps, (i + 1) * plotSteps),
              NDArrayIndex.point(0)).assign(linspaceRow);
          int yStart = plotSteps - i - 1;
          data.get(NDArrayIndex.interval(yStart * plotSteps,
              (yStart + 1) * plotSteps), NDArrayIndex.point(1)).assign(linspaceRow
                  .getDouble(i));
      }
      return data;
  }
}
```

Далее мы объясним, как генерируется сетка изображений из латентного пространства.

Изучение модели VAE

На рис. 5.4 показано изображение, сгенерированное программой. Это реконструкции, полученные из латентного пространства на одной итерации процесса обучения сети.

Латентные переменные

В статистике латентными – в отличие от наблюдаемых – называются величины, выведенные с помощью математической модели.

В зависимости от длительности и параметров обучения получаются различные изображения.

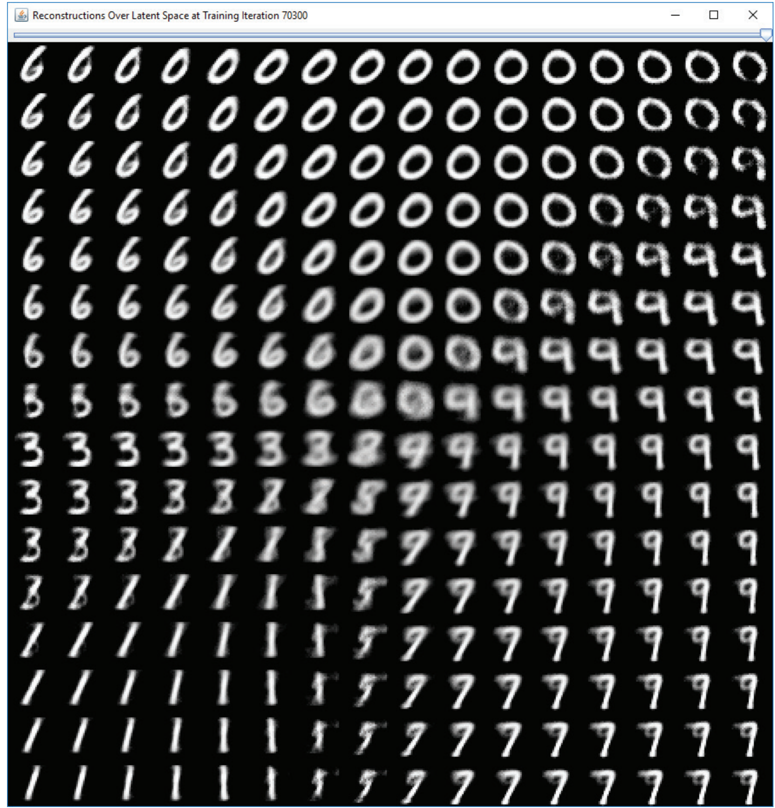


Рис. 5.4 ❖ Цифры MNIST, сгенерированные VAE из этого примера

Чтобы лучше понять, как получены эти изображения, вернемся к архитектуре VAE, описанной в главе 3. Она показана на рис. 5.5.

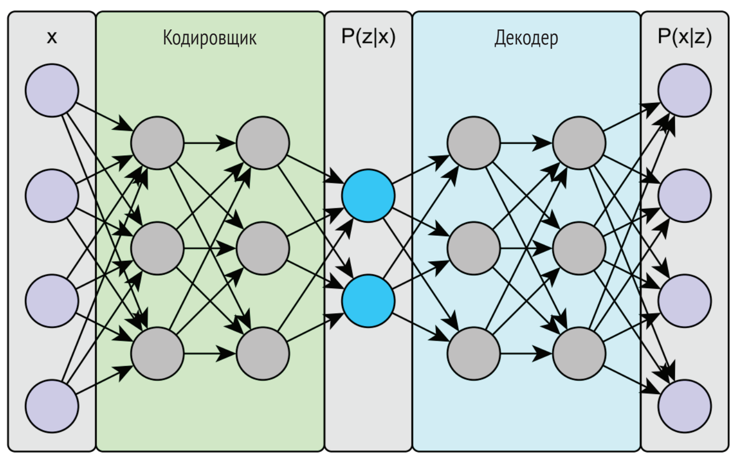


Рис. 5.5 ❖ Архитектура VAE

На рис. 5.6 показана диаграмма рассеяния латентного пространства VAE на тестовых данных MNIST, соответствующая кодирующей части сети на рис. 5.5.

Рис. 5.6 ❖ Диаграмма рассеяния латентного пространства VAE на тестовом наборе данных MNIST в зависимости от периода

Что изображено на диаграмме рассеяния

Диаграмма рассеяния на рис. 5.6 – это снимок пространства латентных переменных после завершения периода. Мы проецируем тестовый набор данных MNIST (например, x) на значения $p(z|x)$, где $p(z|x)$ – нормальное распределение с двумя значениями, а z – латентная переменная.

Переменную z можно интерпретировать как вероятностную версию бутылочного горлышка в сжимающем автокодировщике. В процессе обучения мы производим случайную выборку из z, которая дает некоторые числа. Понятно, что мы не можем распространить вперед распределение вероятности, но можем распространить выбранные из него числа (вариант метода Монте-Карло в статистическом оценивании).

Принятое в модели предположение заключается в том, что сначала данные выбираются из какого-то распределения $p(z)$ в латентном пространстве и что существует некоторый процесс, который порождает данные x с распределением $p(x|z)$.

Мы берем среднее $p(z|x)$ и наносим его на диаграмму рассеяния. В общем случае количество значений больше двух, но мы остановились на 2, чтобы продемонстрировать красивые картинки.

Интерпретация сгенерированных изображений цифр

Изображения, показанные на рис. 5.4, – не что иное, как правая часть сети на рис. 5.6. Точнее, мы генерируем сетку 16 × 16 в диапазоне от –4 до +4 и распространяем эти значения вперед, как если бы они были средними распределения $p(z|x)$.

Можно подойти к этому рисунку и иначе: мы просто фиксируем значения z и выполняем прямой проход через декодер, получая в результате $p(x|z)$ (реконструкции данных MNIST).

В этом случае $p(x|z)$ – распределение Бернулли, принимающее значения из диапазона от 0 до 1. Это значит, что мы можем просто нанести на график эти вероятности (которые, кстати, являются средними значениями распределения Бернулли), так что числа от 0 до 1 преобразуются в яркости пикселей в диапазоне от 0 до 255.

В данном случае размеры x и $p(x|z)$ совпадают: 784 входа и выхода (поскольку в наборе MNIST изображения имеют размер 28 × 28).

Связь между диаграммой рассеяния и сгенерированными изображениями

В определенном смысле сгенерированные изображения и диаграмма рассеяния – одно и то же, только в разных направлениях.

Применение глубокого обучения в обработке естественного языка

Глубокое обучение оказалось эффективным в области обработки естественного языка (ОЕЯ). Типичными его применениями являются частеречная разметка[13], порождение символов[14] и обучение погружениям слов. В этой главе мы рассмотрим следующие применения ОЕЯ:

---

[13] Nogueira dos Santos and Zadrozny, 2014. Learning Character-level Representations for Part-of-Speech Tagging // http://jmlr.org/proceedings/papers/v32/santos14.pdf.

[14] https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/recurrent/character/GravesLSTMCharModelling-Example.java.

- обучение погружениям слов с помощью метода Word2Vec[15];
- распределенные представления предложений посредством векторов абзацев[16];
- классификация документов с помощью векторов абзацев.

Обучение погружениям слов с помощью Word2Vec

Метод Word2Vec математически выявляет сходство между словами путем анализа окружающего контекста. Создаются векторы, являющиеся распределенными числовыми представлениями признаков слов, в данном случае контекста, состоящего из окружающих слов. Word2Vec обучается на корпусе текстов и на выходе порождает список векторов слов (погружений) в качестве модели. Семантика слов и их связи кодируются в порожденных погружениях пространственно. Как мы увидим ниже, погружения обладают полезными свойствами, в т. ч. возможностью производить арифметические операции над векторами.

Модель и алгоритм Word2Vec

Работа алгоритма начинается с построения словаря из входных обучающих данных, после чего строятся представления отдельных слов. Документ не представляется одним вектором, как в других способах векторизации. В методе Word2Vec выходной набор данных, полученный в результате обучения на входном корпусе, – это множество встречающихся в корпусе уникальных слов, к каждому из которых присоединен вектор. Этот вектор содержит контекст слова. В алгоритме Word2Vec используется нейронная модель естественного языка с прямым распространением (Neural Network Language Model – NNLM)[17] для построения контекстных векторов слов без учителя. На практике это двухслойная нейронная сеть, обучаемая алгоритмом SGD.

Векторы слов представляют последовательности слов таким способом, который позволяет моделировать их контекст. Они показывают, как слово соотносится с окружающими его словами. Это особенно удобно, когда имеется достаточно много контекстной информации для классификации слова, как в случаях разрешения именованных сущностей (Named-Entity Resolution – NER), частеречной разметки и присвоения семантических ролей (Semantic Role Labeling – SRL).

Число признаков в сгенерированных Word2Vec векторах

Каждый вектор содержит от 50 до 300 признаков, являющихся результатом распределенного представления слов, которому обучилась нейронная сеть.

Еще один метод из той же области, латентное семантическое индексирование (Latent Semantic Indexing – LSI), определяет измерения, которые встречаются вместе, и объединяет их в одно, что позволяет ускорить различные вычисления, например кластеризацию. Проектировщики Word2Vec рассматривали много под-

---

[15] Mikolov et al., 2013. Efficient Estimation of Word Representations in Vector Space // https://arxiv.org/abs/1301.3781.

[16] Le and Mikolov, 2014. Distributed Representations of Sentences and Documents // https://cs.stanford.edu/~quocle/paragraph_vector.pdf.

[17] Mikolov et al., 2013. Efficient Estimation of Word Representations in Vector Space // https://arxiv.org/pdf/1301.3781v3.pdf.

ходов к оцениванию непрерывных представлений слов, в том числе латентный семантический анализ (Latent Semantic Analysis – LSA) и латентное размещение Дирихле (Latent Dirichlet Allocation – LDA). Word2Vec продемонстрировал значительно лучшее качество, по сравнению с LSA, в том, что касается сохранения линейных регулярных связей между словами. При этом вычислительно он дешевле, чем LDA.

Моделирование контекста

Векторы слов сохраняют контекст исходного текста вокруг слов в форме многословных окон. С точки зрения машинного обучения, семантика слова определяется как множество слов, устойчиво встречающихся вместе с ним. При наличии достаточного объема данных мы можем построить представления слов в корпусе, которые будут с высокой верностью моделировать их семантику. Word2Vec создает признаки на основе окон слов, причем некоторые признаки включают контекст слов.

При построении модели Word2Vec используется скользящее окно, обычно состоящее из пяти слов. Идея скользящего окна встречается и во многих других подобных методах, в т. ч. частеречной разметке, разрешении именованных сущностей и присвоении семантических ролей. В Word2Vec для построения модели применяется *алгоритм Витерби*, который вычисляет наиболее вероятную последовательность событий (меток) при условии вероятности перехода из одного состояния в другое, заданной в виде «матрицы переходов».

Обучение сходной семантике и семантическим связям

Для поиска похожих слов мы можем воспользоваться расстоянием между векторами, например евклидовой метрикой (см. главу 1) или косинусным расстоянием[18]. Это даст слова, находящиеся на небольшом расстоянии друг от друга, т. е. имеющие «близкие представления». В табл. 5.3 перечислены примеры слов, наиболее близких к «France», полученные от обученной методом Word2Vec модели.

***Таблица 5.3.* Сходство между словами в модели Word2Vec**

Слово	Косинусное расстояние
Spain	0.678515
Belgium	0.665923
Netherlands	0.652428
Italy	0.633130
Switzerland	0.622323
Luxembourg	0.610033
Portugal	0.577154
Russia	0.571507
Germany	0.563291
Catalonia	0.534176

Интерпретация косинусного расстояния

Если косинусное расстояние между двумя словами равно 1.0, то слова в точности совпадают. Чем ближе косинусное расстояние к 1.0, тем более похожа семантика двух векторов слов.

[18] https://ru.wikipedia.org/wiki/Коэффициент_Отиаи.

Чтобы в векторном пространстве слов проявилась регулярная структура, модель Word2Vec необходимо обучить на большом наборе слов.

Векторная арифметика и погружение слов

Над векторами в модели Word2Vec можно производить некоторые базовые операции, например:

```
vector('Rome') = vector('Paris') - vector('France') + vector('Italy')
```

Результатом этой операции будет вектор, очень близкий к `vector('Rome')`. Интересно также, что слово «big» (большой) имеет такое же семантическое сходство со словом «bigger» (больше), как «small» (маленький) со «smaller» (меньше). Интересное упражнение – вычислить слово, которое похоже на «small» в том же смысле, в каком «biggest» похоже на «big». Для этого можно применить простые арифметические операции к векторным представлениям слов:

```
vector('smallest') = vector("biggest") – vector("big") + vector("small")
```

Далее мы ищем в векторном пространстве слово, самое близкое к указанному в смысле заданной метрики (в данном случае косинусного расстояния), применяя алгоритм поиска ближайшего соседа[19] для нахождения вектора слов. Если модель хорошо обучена, то мы должны найти правильное слово (smallest). Можно также выявить более тонкие связи, например между столицами стран (например, France относится к Paris так же, как Germany к Berlin).

Эти семантические связи могут заметно улучшить существующие приложения ОЕЯ, например машинный перевод[20], информационный поиск[21] и вопросно-ответные системы[22]. Рассмотрим простой пример использования Word2Vec.

Java-программа, демонстрирующая Word2Vec в действии

Программа[23] в примере 5.7 обучается векторному представлению простых текстовых предложений. Этот скромный пример демонстрирует большой потенциал, заложенный в обучении семантике слов на основе корпуса текстов.

Пример 5.7 ❖ Пример Word2Vec

```
public class Word2VecRawTextExample {

    private static Logger log =
        LoggerFactory.getLogger(Word2VecRawTextExample.class);

    public static void main(String[] args) throws Exception {

        // Получить путь к текстовому файлу
        String filePath = new ClassPathResource("raw_sentences.txt").getFile()
```

---

[19] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

[20] Mikolov, Le, and Sutskever, 2013. Exploiting Similarities among Languages for Machine Translation // https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44931.pdf.

[21] http://web.stanford.edu/class/cs276/handouts/lecture20-distributed-representations.pdf.

[22] Feng et al., 2015. Applying Deep Learning to Answer Selection: A Study and An Open Task // https://arxiv.org/abs/1508.01585v2.

[23] https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/ deeplearning4j/examples/nlp/word2vec/Word2VecRawTextExample.java.

```
            .getAbsolutePath();

        log.info("Загрузка и векторизация предложений....");
        // Убрать пробелы в начале и в конце строк
        SentenceIterator iter = new BasicLineIterator(filePath);
        // Разбить строку на слова по пробелам
        TokenizerFactory t = new DefaultTokenizerFactory();

        /*
            CommonPreprocessor применяет к каждой лексеме такое регулярное выражение:
            [\d\.:,"'\(\)\[\]|/?!;]+
            Поэтому все цифры, знаки препинания и некоторые специальные символы удаляются.
            Кроме того, все лексемы переводятся в нижний регистр.
         */
        t.setTokenPreProcessor(new CommonPreprocessor());

        log.info("Построение модели....");
        Word2Vec vec = new Word2Vec.Builder()
                .minWordFrequency(5)
                .iterations(1)
                .layerSize(100)
                .seed(42)
                .windowSize(5)
                .iterate(iter)
                .tokenizerFactory(t)
                .build();

        log.info("Обучение модели Word2Vec....");
        vec.fit();

        log.info("Запись векторов слов в текстовый файл....");

        // Запись векторов слов в файл
        WordVectorSerializer.writeWordVectors(vec, "pathToWriteto.txt");

        // Печать 10 слов, ближайших к "day". Пример того, что можно сделать
        // с векторами слов.
        log.info("Ближайшие слова:");
        Collection<String> lst = vec.wordsNearest("day", 10);
        System.out.println("10 слов, ближайших к 'day': " + lst);

    }
}
```

Далее мы обсудим некоторые части этой программы.

Обсуждение программы

В следующем фрагменте с помощью объекта `SentenceIterator` загружается файл предложений, включенный в репозиторий код вместе с этим примером.

```
// Получить путь к текстовому файлу
String filePath = new ClassPathResource("raw_sentences.txt").getFile()
    .getAbsolutePath();

log.info("Загрузка и векторизация предложений....");
// Убрать пробелы в начале и в конце строк
SentenceIterator iter = new BasicLineIterator(filePath);
```

```
// Разбить строку на слова по пробелам
TokenizerFactory t = new DefaultTokenizerFactory();
t.setTokenPreProcessor(new CommonPreprocessor());
```

Здесь предварительную обработку выполняет объект класса `CommonPreprocessor`. Этот класс берет на себя большую часть работы, чтобы код примера был проще.

Архитектура сети Word2Vec отличается от рассмотренных выше, поскольку класс `Word2Vec` в DL4J в значительной степени автономен:

```
log.info("Построение модели....");
Word2Vec vec = new Word2Vec.Builder()
        .minWordFrequency(5)
        .iterations(1)
        .layerSize(100)
        .seed(42)
        .windowSize(5)
        .iterate(iter)
        .tokenizerFactory(t)
        .build();
```

Возможно, вы также обратили внимание, что итератор предложений встраивается непосредственно в архитектуру сети. С точки зрения функциональности, это отличие несущественно, но оно является наследием прежнего дизайна API, который сохранился для данной конкретной сети.

Поскольку итератор предложений для сети уже задан, для ее обучения мы просто вызываем метод `.fit()` без параметров:

```
log.info("Обучение модели Word2Vec....");
vec.fit();
```

Когда `.fit()` возвращает управление, программа сохраняет сеть и печатает 10 слов, ближайших к слову «day».

Другие применения Word2Vec

Выходом нейронной сети Word2Vec является словарь, в котором с каждым словом связан вектор. Этот словарь можно подать на вход сети глубокого обучения или просто опрашивать для определения связей между словами. Векторы Word2Vec можно также использовать для классификации документов, как мы увидим далее в этой главе.

Каноническая классификация документов производится так. Сначала по корпусу текстов генерируются векторы слов. Затем мы берем каждое слово из подлежащего классификации документа и вычисляем сумму их векторов слов. Это дает нам вектор, представляющий документ в целом, с учетом всей контекстной информации. После этого полученный вектор документа используется так же, как мы поступили бы с вектором, сгенерированным методом TF-IDF.

Обучение глобальным векторам для представления слов с помощью метода GloVe

GloVe[24] – альтернатива методу Word2Vec. Это алгоритм обучения без учителя для построения погружений слов. Он строит модель на основе агрегированной статистики

[24] https://nlp.stanford.edu/projects/glove/.

совместной встречаемости пар слов, собранной по входным данным (например, корпусу текстовых документов). Основное отличие заключается в том, что Word2Vec – «прогностическая» модель, а GloVe – модель, «основанная на подсчете»[25]. Стоит отметить, что GloVe считается более трудной для обучения и настройки, чем Word2Vec. В состав DL4J входит пример применения GloVe[26], аналогичный рассмотренному выше для Word2Vec.

Распределенные представления предложений с помощью векторов абзацев

Продолжая тему, начатую методом Word2Vec, мы можем обобщить эту модель на произвольно длинные последовательности (например, предложения или целые документы). В этом разделе мы создадим векторы абзацев.

Исторически для алгоритмов машинного обучения требовалось, чтобы представления моделируемых документов имели фиксированную длину. Один из самых распространенных способов достижения этого – метод векторизации с помощью *мешка слов*, когда строится вектор уникальных слов в документе, с каждым из которых связан счетчик его вхождений. Но тут есть две серьезные проблемы:

- теряется информация о порядке слов;
- в модели не отражена семантика слов.

В случае векторов абзацев мы обучаем вектор фиксированной длины, представляющий текстовые последовательности переменной длины (предложения, абзацы или документы). Каждая логическая запись представлена плотным вектором. Исследования[27] показывают, что качество метода векторов абзацев выше, чем у мешка слов и других подходов к представлению текстов. Также показано, что векторы абзацев повышают верность в некоторых задачах классификации текстов и анализа тональности высказываний.

В этом разделе мы работаем с векторами абзацев на примере реализации Doc2Vec в DL4J. Doc2Vec – это обобщение алгоритма Word2Vec, которое обучается корреляции между метками и словами, а не между словами и другими словами. Doc2Vec обучается на корпусе текстов без учителя, так что явно присваивать текстам метки не нужно. Следующая программа моделирует векторы абзацев.

Построение векторов абзацев

В примере 5.8 мы строим распределенное представление (векторы абзацев) всех предложений, встречающихся в корпусе текстов. Обучающие данные имеются в репозитории кода к книге, так что вам остается только клонировать репозиторий и выполнить класс.

---

[25] Baroni, Dinu and Kruszewski, 2014. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors // http://clic.cimec.unitn.it/marco/publications/acl2014/baroni-etal-countpredict-acl2014.pdf.

[26] https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/ deeplearning4j/examples/nlp/glove/GloVeExample.java.

[27] Le and Mikolov, 2014. Distributed Representations of Sentences and Documents // https://cs.stanford.edu/~quocle/paragraph_vector.pdf.

Пример 5.8 ❖ Построение вектора абзацев средствами DL4J на Java

```
public class ParagraphVectorsTextExample {

    private static final Logger log =
        LoggerFactory.getLogger(ParagraphVectorsTextExample.class);

    public static void main(String[] args) throws Exception {
        ClassPathResource resource = new ClassPathResource("/raw_sentences.txt");
        File file = resource.getFile();
        SentenceIterator iter = new BasicLineIterator(file);

        AbstractCache<VocabWord> cache = new AbstractCache<>();

        TokenizerFactory t = new DefaultTokenizerFactory();
        t.setTokenPreProcessor(new CommonPreprocessor());

        /*
            Если под рукой нет класса LabelAwareIterator, то можно воспользоваться
            синхронизированным генератором меток для пометки
            каждого документа/предложения/строки собственной меткой.
            Если же имеется класс LabelAwareIterator, знающий о применяемых
            в организации метках, то можете взять его.
         */
        LabelsSource source = new LabelsSource("DOC_");

        ParagraphVectors vec = new ParagraphVectors.Builder()
                .minWordFrequency(1)
                .iterations(5)
                .epochs(1)
                .layerSize(100)
                .learningRate(0.025)
                .labelsSource(source)
                .windowSize(5)
                .iterate(iter)
                .trainWordVectors(false)
                .vocabCache(cache)
                .tokenizerFactory(t)
                .sampling(0)
                .build();
                vec.fit();

        /*
            В обучающем корпусе есть несколько строк, содержащих довольно близкие слова.
            Следующие предложения должны быть близки друг к другу в векторном пространстве.
            line 3721: This is my way .
            line 6348: This is my case .
            line 9836: This is my house .
            line 12493: This is my world .
            line 16393: This is my work .
            Следующее предложение не имеет ничего общего с предыдущими:
            line 9853: We now have one .
            Отметим, что индексирование документов начинается с 0.
         */

        double similarity1 = vec.similarity("DOC_9835", "DOC_12492");
```

```
        log.info("9836/12493 ('This is my house .'/'This is my world .') similarity:
            " + similarity1);

        double similarity2 = vec.similarity("DOC_3720", "DOC_16392");
        log.info("3721/16393 ('This is my way .'/'This is my work .') similarity:
            " + similarity2);

        double similarity3 = vec.similarity("DOC_6347", "DOC_3720");
        log.info("6348/3721 ('This is my case .'/'This is my way .') similarity: "
            + similarity3);

        // В этом случае вероятность должна быть значительно ниже
        double similarityX = vec.similarity("DOC_3720", "DOC_9852");
        log.info("3721/9853 ('This is my way .'/'We now have one .') similarity: "
            + similarityX + "(should be significantly lower)");
    }
}
```

Из результатов видно, что в первых трех запросах сходство заметно выше, чем в последнем.

Анализ примера с векторами абзацев

Структурно эта программа похожа на пример Word2Vec: мы загружаем входные обучающие данные, конфигурируем сеть и обучаем ее на данных. Работу с файлами и каталогами берут на себя вспомогательные классы.

```
ClassPathResource resource = new ClassPathResource("/raw_sentences.txt");
File file = resource.getFile();
SentenceIterator iter = new BasicLineIterator(file);

AbstractCache<VocabWord> cache = new AbstractCache<>();

TokenizerFactory t = new DefaultTokenizerFactory();
t.setTokenPreProcessor(new CommonPreprocessor());

LabelsSource source = new LabelsSource("DOC_");
```

В реализации вектора абзацев используется вспомогательный класс `ParagraphVectors`:

```
ParagraphVectors vec = new ParagraphVectors.Builder()
        .minWordFrequency(1)
        .iterations(5)
        .epochs(1)
        .layerSize(100)
        .learningRate(0.025)
        .labelsSource(source)
        .windowSize(5)
        .iterate(iter)
        .trainWordVectors(false)
        .vocabCache(cache)
        .tokenizerFactory(t)
        .sampling(0)
        .build();

vec.fit();
```

Основные гиперпараметры задаются прямо с помощью методов этого класса, т. к. у класса `ParagraphVectors` более специализированный API (похожий на Word2Vec), чем у класса `MultiLayerConfiguration`, предназначенного для широкого круга сетей.

Как и в предыдущем примере, мы вызываем метод `.fit()` и в конце программы сравниваем между собой несколько документов, применяя косинусное расстояние. Этот пример показывает, как строить векторы абзацев в DL4J. А в следующем примере мы посмотрим, как использовать их в объемлющем приложении для классификации.

Обобщение векторов последовательностей

Векторы абзацев – это пример реализации векторов последовательностей. В библиотеке DL4J имеется универсальная реализация skip-грамм, лежащая в основе моделей Word2Vec, векторов абзацев и DeepWalk[28].

Хинтон о потенциальных последствиях векторов абзацев

В выступлении 2015 года перед Королевским обществом в Лондоне Джеффри Хинтон сказал:

> Это имеет важнейшие последствия для обработки документов. Если преобразовать предложение в вектор, улавливающий его смысл, то Google сможет искать гораздо лучше, основываясь на содержании документа.
>
> Кроме того, если нам удастся преобразовать в вектор каждое предложение документа, то мы сможем взять эту последовательность векторов и попытаться смоделировать естественное рассуждение. А это то, чего старомодный ИИ никогда не смог бы добиться.
>
> Если мы сможем прочитать каждый англоязычный документ в сети и преобразовать каждое предложение в вектор мысли, то получим достаточно данных для обучения системы, рассуждающей, как человек.
>
> Возможно, мы не захотим, чтобы системы рассуждали, как люди, но, по крайней мере, мы сможем увидеть, что они могли бы подумать.
>
> Я полагаю, что уже через несколько лет эта возможность преобразовывать предложения в векторы мысли поднимет способность систем к пониманию документов на новый уровень. Чтобы понимать документы на уровне человека, система, вероятно, должна иметь ресурсы, как у человека, а ведь в нашем мозгу триллионы связей, тогда как в самых больших на сегодняшний день сетях число связей исчисляется лишь миллиардами. Так что до цели еще несколько порядков, но я уверен, что специалисты по оборудованию с этим справятся.

Применение векторов абзацев для классификации документов

Как уже отмечалось выше, векторы абзацев можно использовать в контексте ОЕЯ и классификации документов. В этом примере мы будем использовать такую же модель, как в предыдущем, для построения классификатора документов, который выдает на выходе оценки трех меток:

```
Документ 'health' попадает в следующие категории:
    health: 0.29721372296220205
    science: 0.011684473733853906
    finance: -0.14755302887323793
```

[28] Perozzi, Al-Rfou and Skiena, 2014. DeepWalk: Online Learning of Social Representations // https://arxiv.org/abs/1403.6652.

Метка с наибольшей оценкой и является итогом классификации. В примере 5.9 приведен исходный код программы классификации[29].

Пример 5.9 ❖ Использование векторов абзацев для классификации документов

```
public class ParagraphVectorsClassifierExample {

    ParagraphVectors paragraphVectors;
    LabelAwareIterator iterator;
    TokenizerFactory tokenizerFactory;

    private static final Logger log =
        LoggerFactory.getLogger(ParagraphVectorsClassifierExample.class);

    public static void main(String[] args) throws Exception {

      ParagraphVectorsClassifierExample app =
         new ParagraphVectorsClassifierExample();
      app.makeParagraphVectors();
      app.checkUnlabeledData();
        /*
                Результат должен выглядеть так:
                Документ 'health' попадает в следующие категории:
                    health: 0.29721372296220205
                    science: 0.011684473733853906
                    finance: -0.14755302887323793
                Документ 'finance' попадает в следующие категории:
                    health: -0.17290237675941766
                    science: -0.09579267574606627
                    finance: 0.4460859189453788
         */
    }

    void makeParagraphVectors()  throws Exception {
      ClassPathResource resource = new ClassPathResource("paravec/labeled");

      // Построить итератор по набору данных
      iterator = new FileLabelAwareIterator.Builder()
              .addSourceFolder(resource.getFile())
              .build();

      tokenizerFactory = new DefaultTokenizerFactory();
      tokenizerFactory.setTokenPreProcessor(new CommonPreprocessor());

      // Сконфигурировать сеть для обучения ParagraphVectors
      paragraphVectors = new ParagraphVectors.Builder()
              .learningRate(0.025)
              .minLearningRate(0.001)
              .batchSize(1000)
              .epochs(20)
              .iterate(iterator)
              .trainWordVectors(true)
```

[29] https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/paragraphvectors/ParagraphVectorsClassifierExample.java.

```
              .tokenizerFactory(tokenizerFactory)
              .build();

      // Обучить модель
      paragraphVectors.fit();
    }

    void checkUnlabeledData() throws FileNotFoundException {
      /*
      Сейчас предполагается, что модель построена и можно проверить, в какие категории
      попадает непомеченный документ. Загрузим непомеченные документы и проверим их.
     */
     ClassPathResource unClassifiedResource =
       new ClassPathResource("paravec/unlabeled");
     FileLabelAwareIterator unClassifiedIterator = new FileLabelAwareIterator
       .Builder()
             .addSourceFolder(unClassifiedResource.getFile())
             .build();

     /*
      Обходим непомеченные данные и смотрим, какие метки им присвоить.
      Примечание: во многих задачах нормально, когда документ попадает сразу
      в несколько категорий, хотя и с различными "весами".
     */
     MeansBuilder meansBuilder = new MeansBuilder(
         (InMemoryLookupTable<VocabWord>)paragraphVectors.getLookupTable(),
           tokenizerFactory);
     LabelSeeker seeker = new LabelSeeker(iterator.getLabelsSource().getLabels(),
         (InMemoryLookupTable<VocabWord>) paragraphVectors.getLookupTable());

     while (unClassifiedIterator.hasNextDocument()) {
         LabelledDocument document = unClassifiedIterator.nextDocument();
         INDArray documentAsCentroid = meansBuilder.documentAsVector(document);
         List<Pair<String, Double>> scores = seeker.getScores(documentAsCentroid);

         /*
          Отметим, что document.getLabel() используется только для того, чтобы
          показать, какой документ мы сейчас анализируем – вместо печати всего
          названия документа. Так что метки этих двух документов играют роль
          заголовка, чтобы мы могли удостовериться в правильности классификации.
         */
         log.info("Документ '" + document.getLabel()
             + "' попадает в следующие категории: ");
         for (Pair<String, Double> score: scores) {
             log.info("        " + score.getFirst() + ": " + score.getSecond());
         }
     }
  }
}
```

Идея примера 5.9 состоит в том, чтобы использовать класс `ParagraphVectors` так же, как мы используем латентное размещение Дирихле[30] (например, для моделирования тематического пространства).

[30] https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation.

В этом примере предполагается, что имеется небольшое число помеченных документов, которые можно использовать для обучения, и ряд непомеченных. Наша цель – определить, в какие категории попадают непомеченные документы, воспользовавшись информацией, содержащейся в векторе абзацев.

Анализ программы классификации с помощью вектора абзацев

Этот пример организован так же, как предыдущий, но обобщен на классификацию непомеченных документов. Вот код, управляющий всей программой:

```
public static void main(String[] args) throws Exception {

  ParagraphVectorsClassifierExample app =
    new ParagraphVectorsClassifierExample();
  app.makeParagraphVectors();
  app.checkUnlabeledData();
```

Как видите, предыдущий пример обернут методом `.makeParagraphVectors()`. В следующей строке эта модель вектора абзацев используется для классификации непомеченных документов. Внутри метода `.checkUnlabledData()` имеется список документов, подлежащих классификации:

```
LabelledDocument document = unClassifiedIterator.nextDocument();
INDArray documentAsCentroid = meansBuilder.documentAsVector(document);
List<Pair<String, Double>> scores = seeker.getScores(documentAsCentroid);
```

В следующей строке вычисленные оценки выводятся на экран:

```
log.info("Документ '" + document.getLabel() + "'
    попадает в следующие категории: ");
for (Pair<String, Double> score: scores) {
    log.info(" " + score.getFirst() + ": " + score.getSecond());
}
```

Вот что будет напечатано:

```
Документ 'health' попадает в следующие категории:
    health: 0.29721372296220205
    science: 0.011684473733853906
    finance: -0.14755302887323793
```

Максимальная оценка

```
    health: 0.29721372296220205
```

и является итогом классификации. Вывод о том, что документ принадлежит к категории «health», правилен, т. к. согласуется с его меткой.

Другие применения метода Word2Vec

Обсуждая СНС, мы уже отмечали, что упоминаемые в книге приложения – лишь отправная точка. Вообще говоря, погружения – это отображения связей между данными, выявленные в ходе обучения нейронной сети. К другим интересным приложениям метода Word2Vec относятся:

- обобщение на другие предметные области;
- анализ графов;
- рекомендательные системы;
- идентификация изображений.

Обобщение на другие предметные области: Gov2Vec. Интересное применение Word2Vec находит в области анализа юридических документов в правительстве. Это приложение называется Gov2Vec. Вот его описание, взятое из статьи автора[31]:

> Мы сравниваем различия в стратегии учреждений, погружая представления всего корпуса юридических документов каждого учреждения и общий словарь всех корпусов в непрерывное векторное пространство. Мы применяем наш метод, Gov2Vec, к заключениям Верховного суда, документам президента и официальным рефератам законопроектов конгресса.

Эта нейронная сеть способна понимать, наложат ли конгресс или президент вето на некоторый законопроект и как будут развиваться события с течением времени. Она умеет отвечать вопросы вида:

> Как Обама и конгресс 113-го созыва различаются в подходах к изменению климата с точки зрения окружающей среды и экономики?

Подходы на основе арифметики векторов, скорее всего, будут привлекать все большее внимание по мере развития глубокого обучения.

Графы и Node2Vec. В области анализа графов существует алгоритмическая инфраструктура Node2Vec[32] для обучения непрерывным представлениям признаков в графах. Интересно, что Node2Vec масштабируется на очень большие графы (миллионы и более вершин и ребер).

Рекомендательные системы и Item2Vec. Еще одно интересное применение идея вектора слов находит в методе Item2Vec для построения рекомендательных систем[33]. В этой работе авторы демонстрируют применение погружений в качестве техники коллаборативной фильтрации для выработки рекомендаций.

Компьютерное зрение и FaceNet. В области идентификации изображений имеется система FaceNet[34], в которой нейронная сеть на базе погружений используется для генерации представлений признаков лица[35].

В общем и целом Word2Vec – интересный метод использования погружений для отображения связей, присутствующих во входных данных. Его можно использовать для кластеризации, классификации и сравнения.

[31] Nay, 2016. Gov2Vec: Learning Distributed Representations of Institutions and Their Legal Text // http://aclweb.org/anthology/W16-5607.

[32] Grover and Leskovec, 2016. node2vec: Scalable Feature Learning for Networks // https://arxiv.org/abs/1607.00653.

[33] Barkan and Koenigstein, 2016. Item2Vec: Neural Item Embedding for Collaborative Filtering // https://arxiv.org/abs/1603.04259.

[34] Schroff, Kalenichenko and Philbin, 2015. FaceNet: A Unified Embedding for Face Recognition and Clustering // https://arxiv.org/abs/1503.03832.

[35] http://crockpotveggies.com/2016/11/05/triplet-embedding-deeplearning4j-facenet.html.

Глава 6

Настройка глубоких сетей

Всё есть яд, и ничто не лишено ядовитости; одна лишь доза делает яд незаметным.

– Парацельс, врач, ботаник, алхимик, астролог и оккультист, живший в XV веке

Основные концепции настройки глубоких сетей

В этой главе мы рассмотрим методы и стратегии обучения нейронных сетей, а именно:

- выбор сетевой архитектуры, соответствующей решаемой задаче;
- элементы настройки гиперпараметров;
- детали процесса обучения.

Понятно, что эта глава не может охватить весь круг опубликованных работ по настройке глубоких сетей. Мы хотим отобрать лишь наиболее важные материалы и построить рассказ так, чтобы дать представление об основных концепциях. Далее в главе 7 будут описаны методы настройки наиболее популярных архитектур глубоких сетей:

- глубокие сети доверия (ГСД);
- сверточные нейронные сети (СНС);
- рекуррентные нейронные сети (РНС).

Начнем с интуитивного описания подхода к построению нейронных сетей для различных целей.

Настройка ограниченных машин Больцмана

В этой главе настройка ограниченных машин Больцмана (ОМБ) рассматривается в контексте настройки ГСД.

Интуитивное описание построения глубоких сетей

В самом начале проекта мы должны задать себе два вопроса:

- какие входные данные я собираюсь моделировать?
- что я хочу получить на выходе построенной модели?

Правильное представление о характере моделируемых данных в значительной мере определяет архитектуру глубокой сети и вид входного слоя. Понимая, что мы хотим узнать от сети, мы сможем решить, нужны ли нам оценки классов (вероятностная классификация) или какое-то одно вещественное значение (регрессия). Заодно определится вид выходного слоя. Подвохи и вариации имеются в самых разных уголках глубокого обучения, но ответы на эти два вопроса помогут зало-

жить надежный фундамент под здание глубокой сети. Определившись с ними, мы сможем перейти к следующему пласту проектных решений, а именно настройке параметров:

- количество слоев;
- количество параметров в одном слое.

Следует также принимать во внимание требования к памяти, необходимой для выбранной сетевой архитектуры при данном количестве параметров. Количество слоев и параметров в одном слое определяет емкость сети, т. е. ее возможности по представлению присутствующей в данных структуры. Для некоторых задач можно создать на удивление сложную модель при сравнительно небольшом числе параметров (общем числе нейронов).

Выбрав архитектуру и количество слоев, мы должны рассмотреть другие вопросы:

- стратегию инициализации весов;
- функцию активации;
- функцию потерь;
- алгоритм оптимизации;
- мини-пакеты;
- регуляризацию.

Стратегия инициализации весов обычно связана с архитектурой и типом входных данных. Удачная инициализация может ускорить процесс обучения, а неудачная – замедлить его. Для каждого слоя необходимо будет выбрать функцию активации, моделирующую нелинейные связи между входными и выходными данными. Функции активации помогают обучаться некоторым типам признаков, а заодно конфигурируют выходные слои для классификации или регрессии. Иногда типы слоев (функции активации) должны соответствовать функциям потерь.

Зависимость обучения от функции потерь

Функция потерь определяет, чему именно должна обучиться сеть (например, классификации или регрессии). Она должна соответствовать функции активации и типу имеющихся данных и меток.

От характера задачи зависит также выбор метода оптимизации.

Методы регуляризации не дают модели отвлекаться на присутствующий в данных шум и удерживают веса от чрезмерного роста, чтобы модель лучше обобщалась на генеральную совокупность данных, т. е. на примеры, не предъявлявшиеся во время обучения. Как мы увидим, многие вышеупомянутые проектные решения взаимосвязаны или оказывают влияние на все архитектурные особенности сети. Объяснение этих взаимозависимостей и составляет большую часть материала этой главы. Теперь формализуем эти идеи и опишем пошаговую процедуру, которая может служить руководством по конструированию архитектуры сети.

Преобразование интуитивных представлений в пошаговую процедуру

В предыдущем разделе мы обсудили, что нужно сделать для построения архитектуры глубокой сети. Теперь формально опишем процедуру, которой следует придерживаться в большинстве задач моделирования с помощью нейронных сетей.

1. Определить, что представляют собой входные данные:
   a. От характера входных данных зависит выбор архитектуры.
2. Определить, каким должен быть результат:
   a. От этого зависит конфигурирование архитектуры.
   b. Тем самым определяется тип выходного слоя.
3. Выбрать архитектуру сети, соответствующую задаче:
   a. Важны выбор модели, архитектуры и функции стоимости.
   b. В зависимости от архитектуры выбирается число скрытых слоев.
   c. Выбрать количество активаций в каждом слое, исходя из архитектуры сети в целом и назначения этого конкретного слоя.
4. Обработать обучающие данные:
   a. Очистить данные.
   b. Подготовить визуализации.
   c. Произвести векторизацию и нормировку.
   d. Сбалансировать классы (если необходимо).
   e. Разделить все множество данных на обучающие, тестовые и контрольные.
5. Разработать стратегию настройки гиперпараметров на сбалансированном подмножестве данных:
   a. По мере необходимости увеличить размер подмножества данных и скорректировать гиперпараметры.
6. Если окончательный обучающий набор данных велик, использовать Spark для ускорения обучения (если есть такая возможность).

Это все еще высокоуровневое описание шагов, но оно задает направление движения при создании любой глубокой сети. Надеемся, что к концу этой главы вы будете в достаточной степени вооружены знанием базовых принципов настройки, чтобы заняться продуктивной работой в появляющихся ныне областях науки о данных. Перейдем теперь к специфике настройки глубоких сетей, начав с первого шага: подбора сетевой архитектуры, соответствующей входным данным.

Подбор сетевой архитектуры, соответствующей входным данным

Как уже отмечалось выше, проектирование глубокой сети должно начинаться с осмысления входного набора данных. Приведем ряд примеров:

- табличные данные;
- изображения;
- звуковые данные;
- видеоданные;
- временные ряды.

В случае табличных данных (в формате CSV) мы обычно имеем дело с экспортом из одной или нескольких соединенных таблиц реляционной базы данных (РСУБД):

```
M,0.455,0.365,0.095,0.514,0.2245,0.101,0.15,15
M,0.35,0.265,0.09,0.2255,0.0995,0.0485,0.07,7
F,0.53,0.42,0.135,0.677,0.2565,0.1415,0.21,9
M,0.44,0.365,0.125,0.516,0.2155,0.114,0.155,10
I,0.33,0.255,0.08,0.205,0.0895,0.0395,0.055,7
I,0.425,0.3,0.095,0.3515,0.141,0.0775,0.12,8
```

Здесь нет ни пикселей (как в изображении), из которых следовало бы выделять признаки, ни временных зависимостей (как во временных рядах), которые нужно обнаружить, так что архитектура может быть относительно простой. Мы рекомендуем начать с многослойного перцептрона.

Для задач классификации изображений лучше использовать СНС. В последнее время именно СНС демонстрируют лучшие результаты в задачах обработки изображений (см. главу 4).

Под последовательными данными понимается любая ситуация, в которой на вход модели подается ряд данных. На практике так чаще всего бывает при обработке журналов, создаваемых серверами или порождаемых датчиками. К этой же категории относятся временные ряды. Временной ряд определяется как ряд значений, с каждым из которых связана временная метка. Эти значения, упорядоченные хронологически, описывают некоторую деятельность, развивающуюся во времени. В большинстве моделей машинного обучения необходим некоторый механизм выделения признаков, строящий из набора временных рядов один входной вектор, который можно было бы передать алгоритму обучения. Для обработки последовательных данных или временных рядов рекомендуем использовать рекуррентную нейронную сеть, обладающую способностью моделировать N входных векторов и не связанную ограничением единственного входного вектора (см. главу 4). Это позволяет моделировать действия во времени.

Обычно рекуррентные нейронные сети дают хорошие результаты при обработке звуковых данных, в которых имеются естественные временные ряды отсчетов сигнала.

При работе с видеоданными требуются более сложные вычисления, чтобы получить какую-то информацию из ряда изображений. Один из возможных подходов – объединить сверточные, max-пулинговые, плотные (прямого распространения) и рекуррентные (LSTM) слои для классификации каждого видеокадра. Другой подход – извлечь отдельные видеокадры в файлы изображений и проанализировать их с помощью СНС.

Обработка видеоданных

На практике очень помогает предобработка с вычислением оптического потока. Это позволяет уловить краткосрочные временные зависимости, присутствующие в данных (например, движение в соседних кадрах), которые невозможно выявить в одном кадре.

Итоги

Мы рассказали, с чего начинать при выборе архитектуры, соответствующей типу входных данных. Наши рекомендации сведены в табл. 6.1.

Таблица 6.1. Соответствие между типом входных данных и архитектурой сети

Тип входных данных	Рекомендуемая архитектура
Табличные (CSV) данные	Многослойный перцептрон
Изображение	СНС
Последовательные	Рекуррентная нейронная сеть, конкретно LSTM
Звуковые	Рекуррентная нейронная сеть, конкретно LSTM
Видео	Гибридная сеть: СНС + РНС

Мир глубоких архитектур не стоит на месте

Хотя рекомендуемая для некоторого типа данных архитектура может работать неплохо, следует помнить о ведущихся исследованиях. В наши дни новые варианты архитектур публикуются с завидной скоростью, поэтому наши рекомендации следует рассматривать как отправные точки и следить за новыми результатами, улучшающими эти архитектуры.

В следующем разделе мы нарастим плоть на скелет архитектурных эвристик, рассмотрев вопрос о задании количества слоев и нейронов.

Соотнесение назначения модели с выходным слоем

В предыдущем разделе обсуждался выбор архитектуры сети в зависимости от типа входных данных. Теперь поговорим о выходных слоях. Этот вопрос сложнее, потому что выбор зависит от того, какой ответ мы ожидаем получить от сети.

С каждым слоем ассоциирована функция активации, служащая для передачи информации следующему слою. Если мы вообще собираемся получить от сети какой-то результат, то должен существовать последний, выходной слой, в котором функция активации соответствует характеру ответа (например, классификация или регрессия).

Выходной слой регрессионной модели

В регрессионных моделях мы порождаем вещественное значение, например стоимость дома в зависимости от площади. Два основных фактора при выборе выходного слоя регрессионной модели – функция потерь и функция активации.

- Функция потерь. Существует несколько вариантов функции потерь в выходном слое регрессионной модели. Самые распространенные – среднеквадратическая ошибка (СКО) и сумма квадратов (норма L2).
- Функция активации. В этом случае используется тождественная (линейная) функция.

Выходной слой регрессионной модели и другие граничные случаи

Иногда в выходном регрессионном слое используется функция активации tanh (только если гарантируется, что все данные будут в диапазоне [−1, 1]) либо softplus, или вариант блока линейной ректификации (ReLU с утечкой, рандомизированный ReLU с утечкой), если гарантируется, что все данные будут в диапазоне [0, ∞).

А что сказать о блоке линейной ректификации (ReLU) для регрессии в случае, когда метки находятся в диапазоне [0, ∞)? Хотя сама функция активации возвращает значения в правильном диапазоне – [0, ∞), существует так называемая проблема «умирающего ReLU».

Суть ее в том, что ReLU может застрять на нулевом участке функции активации. Если это произойдет, то выход ReLU (предсказание сети) всегда будет равен 0, независимо от входов. В других вариантах ReLU (с утечкой и рандомизированный с утечкой), а также в softplus такой проблемы нет.

Выходной слой модели классификации

В моделях классификации мы имеем в выходном слое N блоков, каждый из которых порождает оценку одного класса. Если $N = 1$, то налицо модель с одной меткой,

когда требуется определить, выполняется некоторое условие или нет (например, сообщение спамное или нормальное). Если $N > 1$, то мы оцениваем вероятность принадлежности входа к каждому классу и используем другую конфигурацию выходного слоя. Например, речь может идти о принадлежности документа к некоторой категории: спорт, бизнес, политика. Возможно также, что документ принадлежит сразу к нескольким категориям, скажем, спорт и бизнес.

Модели классификации с одной меткой

В простейшем примере бинарного классификатора с одной меткой в выходном слое используется сигмоидная функция. Она возвращает значение от 0.0 до 1.0, показывающее, например, вероятность, что сообщение является спамом.

Бинарный классификатор: один или два выхода?

Существует вариант бинарного классификатора, в котором выходной слой содержит два нейрона и используется функция активации softmax. Тогда на выходе получаются два вещественных значения, в сумме дающих 1.0, и большее значение определяет индекс метки. В этом случае в качестве функции потерь используется многоклассовая перекрестная энтропия (MCXENT).

Ведутся споры о том, какой выходной слой лучше применять в моделях бинарной классификации. Математически сигмоида с одним выходом эквивалентна softmax-слою с двумя выходными нейронами (с MCXENT/отрицательным логарифмическим правдоподобием и унитарным представлением [1,0], [0,1] вместо 0, 1).

Модели с тремя и более метками

Если меток больше двух, то нужно рассмотреть два случая:

1) модель должна выдать одну наиболее вероятную метку. Это называется многоклассовой классификацией[1];
2) модель должна выдать несколько меток (например, человек + автомобиль). Это называется многометочной классификацией[2].

Многоклассовые модели классификации. Напомним (см. главу 2), что если имеется задача многоклассового моделирования, но нас интересует только класс с наилучшей оценкой, то в выходном слое следует использовать функцию активации softmax. В этом случае число выходных нейронов равно числу классов.

Сумма выходов всех нейронов выходного слоя должна быть равна 1.0. Поскольку выходами являются вероятности классов, то, скорее всего, мы увидим на выходе не ровно 1.0 или 0.0, а некоторое промежуточное значение. Результатом классификации считается класс с наибольшей вероятностью. Чтобы получить индекс этого класса, мы используем функцию `argmax()`.

В следующем примере показана часть нейронной сети с функцией активации softmax в выходном слое:

```
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
```

---

1 https://en.wikipedia.org/wiki/Multiclass_classification.

2 https://en.wikipedia.org/wiki/Multi-label_classification.

В данном случае тип выходного слоя задается методом `.activation()`.

Как правило, в слое с функцией активации softmax в качестве функции потерь используется отрицательное логарифмическое правдоподобие, как в примере выше. Иногда можно встретить вместо него многоклассовую перекрестную энтропию, но эти две функции эквивалентны.

Для выходных данных применяется унитарное представление. Это важно при построении входных или выходных обучающих векторов, поскольку нужно не забыть преобразовать выходной вектор в унитарное представление. В DL4J класс `RecordReaderDataSetIterator` автоматически преобразует индекс класса (от 0 до числа классов – 1) в унитарный вектор.

Случай большого числа меток

Когда число меток достигает десятков тысяч, рекомендуется использовать выходной слой с иерархической функцией softmax. Она дает быструю аппроксимацию softmax[3] при большом числе классов[4]. Этот подход следует предпочесть только из соображений вычислительной эффективности, а не ради повышения верности.

Многометочные модели классификации. Если требуется назначить одному входу несколько классов (например, «человек + автомобиль»), то использовать в качестве функции активации softmax не следует. Вместо нее нужно взять сигмоидную функцию и выходной слой с несколькими блоками, каждый из которых представляет отдельный класс. Тогда каждый блок даст независимую вероятность одного класса.

В этом случае в качестве функции потерь следует использовать бинарную перекрестную энтропию (`LossFunction.XENT`).

Выходной вектор в обучающих данных должен содержать в каждом элементе 0 или 1. Этим он отличается от унитарного представления, поскольку при наличии нескольких меток значение 1.0 могут принимать несколько элементов, а не один, как в унитарном представлении.

Несколько выходных слоев

Предположим, что нейронная сеть должна предсказывать марку автомобиля ((Ford, Toyota, GM и т. д.) и его тип (внедорожник, спорткар, грузовик и т. д.). Тогда нам понадобятся граф вычислений и несколько выходных слоев. Дополнительные сведения см. на странице https://deeplearning4j.org/compgraph#multitask.

Справка: сравнение softmax с сигмоидными функциями активации

Функция softmax гарантирует, что сумма выходов будет равна 1.0 (т. е. дает распределение вероятности), тогда как сигмоидная функция налагает ограничения на каждый выход в отдельности. Например, в сигмоидном выходном слое каждый блок может выдавать значение 0.9. У сигмоидных блоков нет «дополнительного» ограничения.

---

[3] См.: Morin and Bengio, 2005. Hierarchical Probabilistic Neural Network Language Model // http://www.iro.umontreal.ca/~lisa/pointeurs/hierarchical-nnlm-aistats05.pdf; Mikolov et al., 2013. Distributed Representations of Words and Phrases and their Compositionality // https://arxiv.org/abs/1310.4546.

[4] Иерархическая функция softmax – не единственное решение. Альтернативный подход к той же задаче описан в статье: Vishwanathan et al., 2015. BlackOut: Speeding up Recurrent Neural Network Language Models With Very Large Vocabularies // https://arxiv.org/abs/1511.06909.

Количество слоев, количество параметров и объем памяти

В нейронных сетях количество слоев и нейронов в одном слое определяет общее число параметров модели, которое зависит от типов слоев и числа связей (весов) между нейронами в каждом слое (плюс веса смещений).

Тип слоя и количество нейронов

Тип слоя также может влиять на количество параметров сети. При одном и том же количестве входов и выходов LSTM-слой имеет гораздо больше нейронов, чем обычный рекуррентный слой, а тот в DL4J имеет больше нейронов, чем `DenseLayer`. С другой стороны, плотный слой `DenseLayer` имеет больше параметров, чем СНС-слой такого же размера.

Чем больше параметров, тем более сложные функции мы можем моделировать. Но настает такой момент, когда число параметров становится слишком большим, и мы получаем переобученную модель, которая улавливает слишком много несущественных нюансов набора данных и плохо обобщается.

Специфика задания количества слоев, нейронов и весов связей зависит от архитектуры сети. Некоторые общие идеи нейронных сетей применимы в основном к многослойным перцептронам прямого распространения. Конкретные архитектуры будут рассмотрены ниже в этой главе.

Многослойные нейронные сети прямого распространения

В многослойных перцептронах прямого распространения входной слой должен иметь столько блоков, каков размер входного вектора. Число блоков в выходном слое равно числу меток в случае классификации или 1 в случае регрессии. Ниже мы обсудим стратегии задания количества скрытых слоев и нейронов.

Задание числа скрытых слоев

Число скрытых слоев и размер набора данных взаимосвязаны. С ростом набора данных должно увеличиваться и число слоев. Так, в сети для обработки набора MNIST достаточно трех или четырех скрытых слоев (если их число увеличить, то верность будет падать), а в сети DeepFace, построенной компанией Facebook, скрытых слоев девять, так что набор данных, по-видимому, очень велик.

Эвристическое определение числа скрытых слоев

Считается, что чем больше набор данных, тем больше можно использовать скрытых слоев и нейронов без риска переобучения. Когда обучающих данных много, мы можем построить более крупную сеть и, возможно, улучшить верность. Если при большом входном наборе данных сеть слишком мала, то возникает риск недообучения, и верность может не достичь оптимальной для такого набора величины.

Также на выбор числа слоев и параметров влияет – и даже в большей степени – вариативность данных.

Задание числа нейронов в одном слое

Если нейронов в скрытых слоях слишком мало, то будет трудно смоделировать данные в процессе обучения. Если параметров слишком много, то мы либо столкнемся с переобучением, либо потратим больше времени на получение хорошей аппроксимации.

Рекомендуется уменьшать количество нейронов по мере удаления от входного слоя. Кроме того, нужно следить за тем, чтобы ни в одном скрытом слое число

нейронов не опускалось ниже четверти от числа нейронов во входном слое. Если же число нейронов в скрытых слоях слишком велико, то мы снова рискуем переобучить сеть.

Эвристическое определение числа нейронов в слое

Мы стремимся к правильному сочетанию размера слоя, регуляризации и объема данных. Большие слои в сочетании с недостаточной регуляризацией приводят к плохой обобщаемости. Чем больше слой (и их количество, т. е. число параметров), тем более агрессивной должна быть регуляризация для предотвращения переобучения (или же нужно увеличить объем обучающих данных).

Иногда получаются лучшие результаты, если число параметров во всех скрытых слоях одинаково, а не уменьшается. Однако это справедливо только для наборов данных определенного типа.

Управление количеством слоев и параметров

Естественно, в голову приходит мысль исключить из сети параметры (или нейроны), чтобы разрешить проблему переобучения. На практике, однако, число скрытых нейронов часто оставляют на уровне выше оптимального.

Эмпирически установлено, что оптимальное число скрытых нейронов гораздо больше при использовании предобучения без учителя (ОМБ в ГСД). В таком случае их число может увеличиться с нескольких сотен до нескольких тысяч[5]. А для борьбы с переобучением рекомендуется использовать методы регуляризации, например:

- L1;
- L2;
- прореживание;
- шум на входе;
- прореживание связей.

В качестве аргумента выдвигают соображение о том, что чем меньше сеть, тем меньше в ней локальных минимумов (но, возможно, модель хуже), а в большой сети и локальных минимумов много[6]. Не хотелось бы уменьшать размер сети только из боязни переобучения. Наоборот, лучше увеличить число параметров настолько, насколько позволяет вычислительная платформа, и применить вышеупомянутые методы регуляризации.

Не стоит перебарщивать с параметрами

Количество параметров, конечно, стоит увеличить, но в разумных пределах. Один слой с 1 000 000 нейронов – не лучшая идея, даже если оборудование способно его потянуть.

По мере увеличения вариативности и размера обучающего набора данных мы рекомендуем медленно увеличивать количество скрытых слоев и нейронов в одном слое.

---

[5] Bengio, 2012. Practical Recommendations for Gradient-Based Training of Deep Architectures // https://arxiv.org/abs/1206.5533; Muller et al., 2012. Neural Networks: Tricks of the Trade, Second Edition.

[6] Li, Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition (Course Notes) // http://cs231n.stanford.edu/.

Добавление данных для борьбы с переобучением
Один из лучших способов борьбы с переобучением – увеличение объема обучающих данных (там, где это возможно).

Вычисление количества параметров сети

На практике есть два способа получить количество параметров сети:

- вручную вычислить его для выбранной архитектуры;
- воспользоваться DL4J API.

Чтобы вычислить общее количество параметров сети, нужно просто просуммировать параметры в каждом слое. В табл. 6.2 приведено количество параметров для наиболее распространенных слоев.

***Таблица 6.2.** Количество параметров в одном слое в зависимости от его типа*

Тип слоя	Число параметров
Полносвязный (например, Dense, MLP, OutputLayer в DL4J)	$n^{L-1}n^L + n^L$
Сверточный	$d^{L-1}d^Lk^Hk^W + d^L$
LSTM (например, GravesLSTM)	$4n^L(n^{L-1} + 1) + n^L(4n^L + 3)$
Стандартная РНС	$n^{L-1}n^L + (n^L)^2 + n^L$

Ниже описаны применяемые обозначения:

- n^{L-1} – число входов (размер предыдущего слоя, $L - 1$);
- n^L – размер текущего слоя L;
- k^H и k^W – высота и ширина ядер в сверточном слое;
- d^{L-1} и d^L – глубина (число каналов) для входного и выходного сверточных слоев.

Отметим, что для некоторых типов слоев параметров нет вовсе, например в слоях субдискретизации и активации, а также в слое `LossLayer` в DL4J. Некая трудность возникает в сетях, где встречаются сверточные и плотные (или выходные) слои. Для плотного (или выходного) слоя после сверточного или субдискретизирующего слоя мы должны определить размер массива активаций последнего сверточного слоя. Его длина (в расчете на один пример) используется в качестве значения n^{L-1} для плотного (выходного) слоя. Например, если последний сверточный слой порождает активацию 5×5 со 100 каналами, то плотный (выходной) слой будет иметь $n^{L-1} = 5 \times 5 \times 100 = 2500$ входов[7].

Пусть имеется конфигурация DL4J. Как узнать число параметров? Вот простой способ:

```
MultiLayerConfiguration configuration = ...
MultiLayerNetwork network = new MultiLayerNetwork(configuration);
network.init();

System.out.println("Общее число параметров: " + network.numParams());
for(int i=0; i<network.getnLayers(); i++ ){
    System.out.println("Число параметров в слое " + i + ": " +
        network.getLayer(i).numParams());
}
```

[7] Дополнительные сведения о вычислении размера входа см. в примечаниях к курсу CS231n Стэнфордского университета и в Java-документации по перечислению `ConvolutionMode`. (Можно вместо этого воспользоваться функциональностью класса `InputType`, чтобы вычислить значение `numInputs`, а затем использовать DL4J для получения конфигурации сети.)

Оценка требований к объему памяти

Если сеть слишком тяжела для имеющегося оборудования, то обучение остановится после исчерпания памяти. Хотя прямого отношения к настройке это не имеет, вопрос весьма важен для больших сетей – ведь если мы не сможем обучить сеть, то о настройке и говорить нечего.

При обучении нейронной сети память выделяется под многомерные массивы (в DL4J это структуры `INDArray`). Такие массивы используются в шести случаях:

- параметры сети;
- градиенты параметров (того же размера, что параметры сети);
- значения активации сети;
- градиенты значений активации (того же размера, что сами активации);
- состояние корректора (история для `Momentum` или `RMSProp` – размер кратен количеству параметров);
- обучающие данные.

Кроме того, следует учесть накладные расходы на рабочую память и временные массивы, асинхронно предзагруженные данные и все необходимое для работы виртуальной машины Java (JVM).

Минимальные требования к памяти для обучения нейронной сети (в байтах) можно оценить по формуле

$$N_{bytes,\ train} = 4[n_{params}(2 + u) + m(2a + d)].$$

В то же время для тестирования сети (после обучения) нужно только

$$N_{bytes,\ test} = 4[n_{params} + m(a + d)],$$

где u – размер корректора ($u = 0$ для СГС, $u = 1$ для Momentum, RMSProp и Adagrad; $u = 2$ для Adam и Adadelta); m – размер мини-пакета; a – размер значений активации сети для одного примера (все слои); d – размер одного примера.

Параллельная обработка и расчет потребления памяти

Если обучение производится на нескольких машинах (с применением Spark или ParallelWrapper), то $N_{bytes,\ train}$ нужно умножить на число реплик модели.

Замечание о DL4J, памяти и точности

По умолчанию в DL4J (как и в ND4J) во всех массивах `INDArray` хранятся 32-разрядные числа с плавающей точкой (FP32). Поэтому `INDArray`, содержащий *N* элементов, занимает *4N* байтов. Для обучения сети можно использовать также 64-разрядные числа (FP64), что иногда повышает численную устойчивость (хотя само возникновение таких проблем указывает на другие проблемы с настройкой параметров). Однако использование типа FP64 повышает потребление памяти (в 2 раза) и снижает производительность (до 32 раз на графических процессорах потребительского класса и в 2 раза на GPU Tesla). Производительность вычислений с числами типа FP64 на CPU также снижается, хотя и в меньшей степени из-за особенностей доступа к памяти и кэширования.

Наконец, отметим, что в DL4J поддерживается формат 16-разрядных чисел с плавающей точкой (FP16) для обучения на GPU с архитектурой CUDA. Формат FP16 уменьшает потребление памяти в 2 раза по сравнению с FP32, что дает возможность увеличить размер сети, размер пакета или то и другое вместе. Однако при этом су-

щественно снижается точность вычислений, что затрудняет настройку сети. Если вы только начинаете настраивать нейронные сети или никак не можете настроить сеть правильно, то работайте с форматом FP32 и переходите на другой лишь в случае острой необходимости. Что касается производительности FP16, то она очень низкая на большинстве потребительских GPU, так что этот формат в DL4J используется только для хранения, а перед выполнением операций производится преобразование данных в формат FP32.

Столкнувшись с нехваткой памяти, мы можем предпринять следующие действия:

1. Уменьшить размер мини-пакета m.
2. Приобрести более мощное оборудование (или воспользоваться облачными службами – Azure, Amazon Web Service или Google Cloud).
3. Использовать формат FP16 вместо FP32 на GPU (но не забывайте об уменьшении точности вычислений).
4. Уменьшить размер сети.

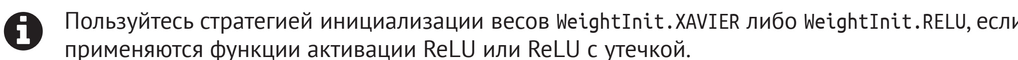
Стратегии инициализации весов

Инициализация весов может оказать существенное влияние на процесс обучения. Следует выбирать стратегию инициализации с учетом контекста задачи. Начальные веса – важнейший момент в процессе обучения любых нейронных сетей, в т. ч. глубоких.

Вообще говоря, в качестве основы выбирается следующая стратегия инициализации весов:

- смещения инициализируются нулями;
- скрытые веса инициализируются так, чтобы не было никакой симметрии между скрытыми блоками.

Чтобы избавиться от симметрии, мы включаем в процесс инициализации элемент случайности. Инициализация весов должна быть функцией числа входов (и, возможно, также выходов).



Пользуйтесь стратегией инициализации весов `WeightInit.XAVIER` либо `WeightInit.RELU`, если применяются функции активации ReLU или ReLU с утечкой.

Если веса слишком велики, то это может привести к большим выходам и большим градиентам, что, очевидно, неблагоприятно сказывается на обучении. Кроме того, нужно позаботиться об инициализации смещений. На первых этапах обучения начальные смещения видимых блоков могут играть заметную роль. Существуют различные способы инициализации смещений, но по умолчанию им обычно присваивается значение 0. Нулевые начальные смещения скрытых блоков обычно приемлемы, за исключением случаев, когда мы хотим добиться конкретной разреженности.

В этом разделе мы рассмотрим два случая инициализации весов: с функцией активации ReLU и tanh. Обычно веса инициализируются случайным образом путем выборки из распределения (как правило, нормального или равномерного) с некоторой дисперсией. В большинстве случаев можно руководствоваться следующими рекомендациями:

- для семейства функций активации ReLU (ReLU, ReLU с утечкой, рандомизированный ReLU с утечкой и т. д.) пользуйтесь стратегией инициализации весов `WeightInit.RELU`. Иногда ее называют еще инициализацией Хе, имея в виду вышедшую в 2015 году статью He et al. «Delving Deep into Rectifiers»;
- для большинства других функций активации (tanh, тождественная и т. д.) пользуйтесь стратегией `WeightInit.XAVIER`. Ее еще называют стратегией Глорота, имея в виду статью 2010 года Glorot and Bengio «Understanding the Difficulty of Training Deep Feedforward Neural Networks». Для весов связей с tanh-блоками есть и другие методы инициализации, например разреженная инициализация[8];
- смещения обычно инициализируются нулями (в DL4J это режим по умолчанию).

Не вдаваясь в детали, скажем, что стратегии ReLU и Xavier спроектированы с учетом архитектуры сети (размеры слоев) и предположений о функциях активации таким образом, чтобы выполнялись два условия:

- дисперсия значений активации постоянна для всех слоев. В частности, активации не должны становиться слишком большими или слишком малыми в более поздних слоях сети;
- дисперсия градиентов активации (а значит, и параметров) постоянна для всех слоев. Например, градиенты не должны становиться слишком большими или слишком малыми к моменту обратного распространения на более ранние слои сети.

Инициализация весов и служение двум господам

Стратегии инициализации ReLU/Xavier не могут точно оптимизировать эти два ограничения (дисперсии активаций и градиентов) в сетях со слоями разного размера (или если не выполняются допущения о функциях активации)[9].

Ортогональная инициализация весов в РНС

В главе 7 мы будем рассматривать стратегии настройки конкретных архитектур, в т. ч. LSTM. Стоит отметить, что LSTM и вентильные рекуррентные блоки (GRU) в некоторых случаях работают лучше, если применить ортогональную инициализацию весов[10].

Выбор функции активации

В табл. 6.3 приведены замечания по поводу соответствия между распределениями данных и функциями активации в сетях прямого распространения.

Сигмоидные функции встречаются в литературе по нейронным сетям чаще всего. Они были весьма популярны 20 лет назад, но сейчас практически не применяются в скрытых слоях.

---

[8] Martens, 2010. Deep learning via Hessian-free optimization // http://www.cs.toronto.edu/~jmartens/docs/Deep_HessianFree.pdf.

[9] Glorot and Bengio, 2010. Understanding the difficulty of training deep feedforward neural networks // http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[10] Saxe, McClelland and Ganguli, 2013. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks // https://arxiv.org/abs/1312.6120v3.

Таблица 6.3. Связь целевого распределения с функцией активации в выходном слое в сетях прямого распространения

Целевое распределение	Функция активации в выходном слое
Бинарное (0, 1)	Сигмоидная функция активации
Категориальное (1 из C)	Сигмоидная функция активации
Непрерывное (с ограниченным диапазоном)	Сигмоидная или tanh (с масштабированием диапазона выходов в диапазон распределения)
Положительное (верхняя граница неизвестна)	Семейство функций активации ReLU, функция softplus (или использовать логарифмическую нормировку для преобразования в непрерывное неограниченное распределение)
Непрерывное (неограниченное)	Линейная функция активации (эквивалентна отсутствию функции активации)

Сигмоида и потеря информации

По сравнению с функциями активации ReLU, сигмоиды чаще теряют информацию вследствие насыщения при прямом и обратном распространениях, что приводит к нелинейным эффектам в сети из-за нечувствительности к малым изменениям параметра[11].

У сигмоиды имеются проблемы, когда при большом по абсолютной величине входном сигнале градиент обращается в нуль. Они менее серьезны в случае мини-пакетов, но необходимо тщательно инициализировать веса, чтобы избежать насыщения.

Увядание сигмоидных функций активации

В настоящее время сигмоидным функциям предпочитают функции ReLU с утечкой, поскольку они не страдают ни от проблемы исчезающего градиента, как сигмоиды и tanh, ни от проблемы «умирающего ReLU», как обычная функция ReLU.

Мы рекомендуем не использовать сигмоиды в скрытых слоях.

Самыми популярными функциями активации в современных глубоких сетях являются ReLU (кусочно-линейные функции). Именно ReLU обычно используются в сверточных слоях СНС. Установлено, что обучение методом стохастического градиентного спуска (СГС) ускоряется, если вместо сигмоид и tanh применяются ReLU. Ко всему прочему, ReLU дешевле и с вычислительной точки зрения.

Проблема умирающего ReLU

При использовании ReLU следует проявлять осторожность, потому что некоторые блоки могут не активироваться ни для одного обучающего примера (этот феномен иногда называется «проблемой умирающего ReLU»).

Сеть может обучиться даже при большом числе «мертвых» блоков ReLU, однако эти блоки не вносят никакого вклада в выход сети, т. е. мы зря потратили время на вычисления, и эффективная емкость сети понизилась.

Поэтому рекомендуется использовать ReLU с утечкой, поскольку для этой функции градиент ненулевой при всех входных значениях.

У функции активации hardtanh имеется аналогичная проблема в области нулевого градиента, а именно когда значение аргумента меньше –1 или больше +1.

[11] http://www.deeplearningbook.org/.

Замечание о maxout-блоках

Maxout-модель – это сеть прямого распространения с maxout-блоками, которые считаются обобщением ReLU, не страдающим от проблемы умирающего блока. Использование maxout-блоков вдвое увеличивает число параметров в расчете на один нейрон, а стало быть, и общее число параметров.

Показано, что maxout-блоки обучаются в условиях, когда ReLU ведут себя плохо. Различие между блоками ReLU и maxout состоит в том, что последние кусочно-линейны, и каждому участку линейности соответствует свой вектор весов. Эта особенность maxout-блоков позволяет избегать застревания процесса обучения в тех местах, где блоки ReLU подвержены такому риску.

Сводная таблица функций активации

В табл. 6.4 перечислены рассмотренные в этом разделе функции активации.

Таблица 6.4. Список употребительных функций активации с указанием мест, где они используются

Функция	**Где используется**
Линейная	Выходной слой для регрессии
Сигмоида	• Выходной слой для бинарной классификации • Выходные значения в диапазоне [0,1] • Вышла из моды, в скрытых слоях не используется
Tanh	• Непрерывные данные, не только [−1,1] • LSTM-слои
Softmax	Выходной слой модели многоклассовой классификации
ReLU	• ОМБ • Слои СНС • Слои многослойного перцептрона

В общем случае мы рекомендуем обратить внимание на функции активации ReLU, но тщательно выбирать скорость обучения и следить за мертвыми нейронами. Для борьбы с этими явлениями можно попробовать ReLU с утечкой или функцию maxout. Альтернативой является функция активации tanh, но замечено, что на практике она дает результаты хуже, чем ReLU и maxout. Сигмоидные блоки лучше использовать только в выходном слое для однометочной классификации, а в скрытых блоках они больше не применяются.

Применение функций потерь

Благодаря функции потерь алгоритм оптимизации знает, насколько хорошо он справляется с задачей. Ранее в этой главе мы говорили о функциях потерь в контексте целей модели (например, о функциях потерь в выходном слое). Теперь сделаем несколько замечаний о других применениях функций потерь, о том, где их использование наиболее эффективно, и о возможных подводных камнях. Честно говоря, функции потерь используются только в двух местах:

- выходные слои (`LossLayer`);
- слои, поддерживающие послойное предобучение без учителя, например автокодировщики, ОМБ и VAE.

В большинстве слоев (сверточных, плотных, LSTM и других) функции потерь не используются (и не могут использоваться), потому что для них послойное предобучение невозможно.

Помимо вышеупомянутых функций потерь при классификации, встречается еще кусочно-линейная (hinge) и логистическая функции потерь. Кусочно-линейная функция потерь чаще всего используется, когда сеть следует оптимизировать для однозначной классификации, например: 0 = честная операция, 1 = мошенническая операция; такие модели принято называть классификаторами типа 0–1. Логистическая функция потерь используется, когда интересна не однозначная, а вероятностная классификация, например пометка потенциально мошеннических операций в технологическом процессе, где решение принимает человек, или предсказание вероятности перехода по рекламной ссылке, на основе которой можно тарифицировать объявление. Предсказание вероятности означает, что модель выдает число от 0 до 1.

Бинарная классификация и кусочно-линейная функция потерь
Кусочно-линейная функция потерь используется в основном для бинарной классификации. Существуют ее обобщения на многоклассовую классификацию (например, один против всех или один против одного), но они здесь не рассматриваются.

Интерпретация отладочной печати в процессе обучения

В процессе обучения программа печатает строки вида:

```
21:36:00.358 [main] INFO o.d.o.l.ScoreIterationListener
  -  Score at iteration 0
    is 0.5154157920151949
```

Последним печатается среднее значение функции потерь по примерам из текущего мини-пакета. Это число зависит от того, какая функция потерь используется в архитектуре сети. Например, для СКО порядок и характер изменения среднего будут не такими, как для отрицательного логарифмического правдоподобия.

Проще всего интерпретировать его как «мало – хорошо, много – плохо». Вообще говоря, со временем это значение убывает.

Чтобы включить отладочную печать, нужно добавить в программу строку

```
myNetwork.setListeners(new ScoreIterationListener(1));
```

Таблица 6.5. Сводная информация об использовании функций потерь

Функция потерь	Где используется	Свойства
Энтропия реконструкции	ОМБ, автокодировщики	Конструирование признаков
Квадратичная потеря	Выходной слой	Регрессия
Перекрестная энтропия	Выходной слой	Бинарная классификация
Многоклассовая перекрестная энтропия	Выходной слой	Классификация
Корень из СКО	Автокодировщик, ОМБ, выходной слой	Конструирование признаков, регрессия
Кусочно-линейная	Выходной слой	Классификация
Отрицательное логарифмическое правдоподобие	Выходной слой	Классификация

Перекрестная энтропия, логистическая функция потерь и отрицательное логарифмическое правдоподобие

Эти функции потерь в литературе и в примерах встречаются в сходных обстоятельствах. Как отмечалось в главе 2, перекрестная энтропия берет начало в теории информации, а отрицательное логарифмическое правдоподобие – в статистическом моделировании. Математически эти понятия эквивалентны, так что употребление того и другого служит лишь источником путаницы.

Скорость обучения

Скорость обучения – один из важнейших (если не *самый* важный) гиперпараметр нейронной сети. Он влияет как на устойчивость, так и на время обучения сети.

На рис. 6.1 видно, что если скорость обучения слишком велика (левый рисунок), то обучение неустойчиво (или процесс вообще не сходится). Если же скорость слишком мала (правый рисунок), то обучение происходит гораздо медленнее, чем могло бы.

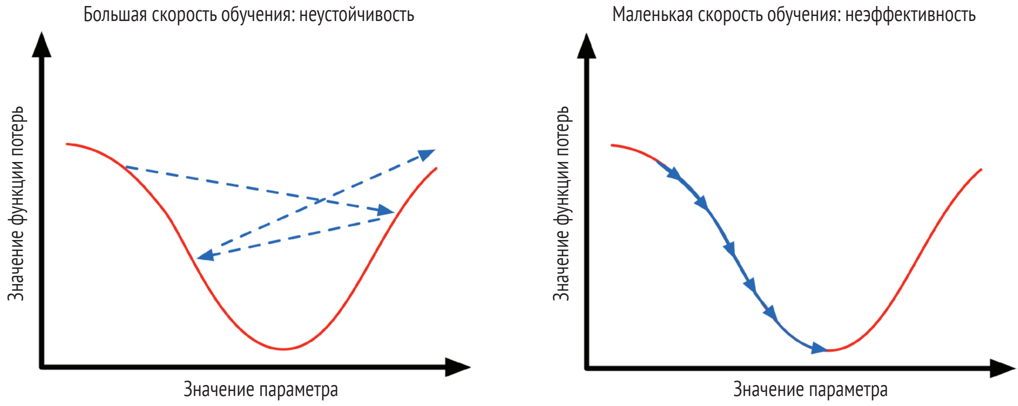


Рис. 6.1 ❖ СГС и скорость обучения

В идеале мы хотим, чтобы в начале обучения скорость была велика, а по мере приближения к сходимости постепенно уменьшалась. При использовании корректора Momentum имеет смысл медленно увеличивать величину импульса (см. следующий раздел) по мере уменьшения скорости обучения.

Импульс и некоторые корректоры

При использовании корректоров Adam или RMSProp модификация импульса не применяется.

Увеличение скорости обучения

Увеличение начального значения скорости обучения не всегда помогает, потому что, несмотря на быстрое обучение вначале, общее время обучения может оказаться даже больше.

В идеале хотелось использовать какую-нибудь нестатическую схему, позволяющую задавать скорость обучения на разных уровнях:

- глобально;
- на уровне слоя;

- на уровне нейрона;
- на уровне параметра (когда с каждым нейроном, смещением и т. д. связано несколько параметров)[12].

Мы хотим, чтобы скорость начала падать только ближе к концу обучения, и готовы потратить дополнительное время на вычисления с меньшей скоростью. Желательно, чтобы темп убывания скорости можно было задавать на уровне отдельного параметра и чтобы стратегия была не слишком агрессивной, иначе обучение будет замедляться чрезмерно.

Использование отношения обновлений к параметрам

Простой и эффективный подход к заданию скорости обучения заключается в использовании отношения обновлений к параметрам (точнее, средних абсолютных величин того и другого). Напомним, что обновления – это то, что получается после применения корректора (RMSProp, Momentum и т. д.) к градиентам, т. е. уравнение обучения имеет вид $\theta \leftarrow \theta - u$, где u – обновление. Если обозначить вектор обновлений (той же длины N, что вектор параметров) $\mathbf{u}$, а вектор параметров θ, то отношение обновлений к параметрам определяется формулой:

$$\textit{Отношение обновлений к параметрам} = \frac{\frac{1}{N}\sum_{n=1}^{N}|u_i|}{\frac{1}{N}\sum_{n=1}^{N}|\theta_i|}.$$

Это отношение измеряет, как сильно изменяются параметры относительно их текущих значений. Его можно найти в двух местах пользовательского интерфейса DL4J: на странице Overview (слева внизу) и на странице Model (первый график после выбора вершины слоя). Отметим, что в интерфейсе настройки это отношение представлено в виде десятичного логарифма, как показано на рис. 6.2.

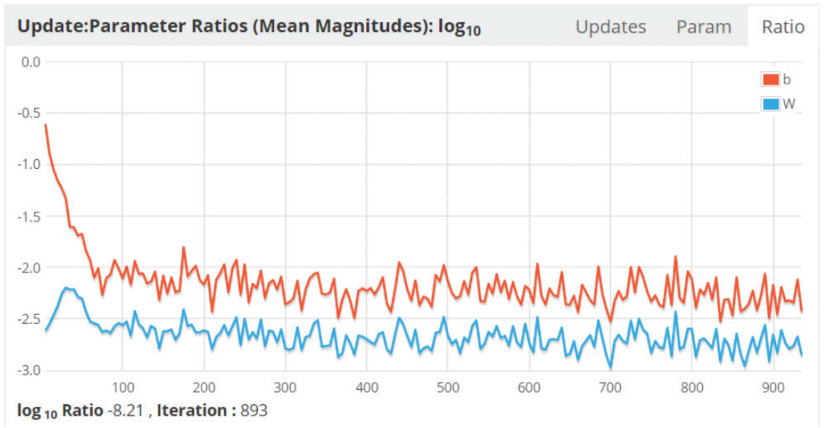


Рис. 6.2 ❖ Интерфейс настройки DL4J, показан график отношения обновлений к параметрам

[12] Bengio, 2012. Practical Recommendations for Gradient-Based Training of Deep Architectures // Muller et al., 2012. Neural Networks: Tricks of the Trade, Second Edition.

Если вы пользуетесь DL4J для наблюдения за процессом обучения (рис. 6.2), то после начала обучения перейдите по адресу http://localhost:9000/.

Использование отношения для задания скорости обучения
При задании скорости обучения мы стремимся к тому, чтобы отношение обновлений к параметрам было равно приблизительно 0.001, что соответствует значению десятичного логарифма −3 в пользовательском интерфейсе обучения DL4J.

Так, если логарифм отношения равен –4 (само отношение равно 0.0001), то следует увеличить скорость обучения, а если он равен –2 (0.01), то уменьшить[13].

Конкретные рекомендации по заданию скорости обучения

Обучение стоит начинать с относительно высокой скорости обучения и постепенно уменьшать ее со временем. Если начальная скорость слишком велика, то обычно мы наблюдаем увеличение средней потери. Оптимальная скорость обучения обычно близка (отличается не более чем в 2 раза) к наибольшей скорости, при которой процесс обучения еще не расходится.

Хорошая начальная скорость обучения
В идеале следует начинать с большой скорости обучения. Если процесс обучения расходится, то следует разделить скорость на некоторый коэффициент и попробовать снова. И продолжать так, пока процесс не сойдется.

Мы рекомендуем испытать скорости обучения [0.1, 0.01, 0.001] и посмотреть, какая дает самые устойчивые результаты. Одним из наиболее популярных начальных значений является 0.001.

Мы также рекомендуем попробовать корректоры в описанном ниже порядке.

1. Adam. В настоящее время рекомендуется как метод по умолчанию.
2. Метод импульса Нестерова.
3. RMSProp:
   - используется скользящее среднее квадратов градиентов.
   - в отличие от AdaGrad, не возникает проблем из-за монотонного убывания скорости обучения.
4. AdaGrad.

DL4J поддерживает также алгоритм AdaDelta.

За скоростью обучения все равно нужно наблюдать
Если выбрать слишком высокую скорость обучения, то обучение методом AdaGrad все равно будет неустойчивым (как и любым другим методом). Алгоритм AdaGrad не может увеличить скорость обучения. В действительности скорость обучения (для каждого параметра) в AdaGrad монотонно убывает.
AdaGrad действует как механизм снижения скорости обучения каждого веса. Но недостатком AdaGrad является тот факт, что такое монотонное убывание может оказаться чрезмерно агрессивным, и обучение остановится слишком рано.
В DL4J алгоритм AdaGrad встроен, поэтому пользователи могут меньше думать о ручной настройке скорости обучения.

[13] Можно было бы использовать и другие метрики, например отношение L2-норм. Их нет в интерфейсе обучения DL4J, но нетрудно вычислить их самостоятельно.

Метод Нестерова в DL4J

Метод импульса помогает процессу обучения избегать локальных минимумов и находить лучшие решения в ходе оптимизации[14].

Как правило, импульс близок к 0.9. Наиболее популярные значения этого гиперпараметра: [0.5, 0.9, 0.95, 0.99]. Вначале обычно задается значение 0.5, а на протяжении нескольких периодов оно закрепляется на уровне 0.9. Помимо вызова `.momentum(0.5)`, обязательно вызывайте `.updater(Updater.NESTEROVS)`. В отсутствие хотя бы одного из них метод импульса не активируется.

Если у скрытого блока много входящих связей, то обновление должно быть меньше, поскольку много изменений в одном направлении может оказать нежелательное влияние на знак градиента. В этом контексте скорость обучения следует уменьшить. Для смещений обновление может быть значительнее, но никаких негативных последствий при этом не наблюдается. Из-за всех этих нюансов настройка скорости обучения может оказаться трудной задачей.

Перечислим несколько соображений, о которых следует помнить:

- скорость обучения зависит от сети: что хорошо в одной сети, может не работать в другой;
- отношение обновлений к параметрам – всего лишь эвристика, работает она не всегда, но часто берется в качестве отправной точки;
- подходящая скорость обучения зависит (иногда сильно) от других гиперпараметров. Например, в различных корректорах (Momentum, RMSProp, AdaGrad и т. д.) нужны разные скорости обучения; после изменения начальных весов или числа, типов или размеров слоев скорость обучения приходится настраивать заново. Поэтому всегда проверяйте скорость обучения после значительных изменений сети;
- перед тем как анализировать график отношения обновлений к параметрам, дайте сети некоторое время, чтобы стабилизироваться, – скажем, от нескольких десятков до нескольких сотен итераций с момента начала обучения. Первые несколько обновлений обычно велики;
- нормально и даже рекомендуется для отношения обновления смещений выбирать начальное значение гораздо больше 0.001. Ведь все смещения обычно инициализируются значением 0.0, а раз так, то в начале работы отношения обновлений будут велики, а со временем их величина уменьшится;
- в некоторых случаях бывает необходимо задавать разные скорости обучения для разных слоев, чтобы получить правильное отношение. Но это может указывать на другие проблемы, например исчезающий или взрывной градиент, плохие начальные веса или неправильную нормировку данных. Корректоры Adam и RMSProp могут сгладить проблему, поскольку оказывают нормирующее (масштабирующее) влияние на градиенты.

Как разреженность влияет на обучение

Гиперпараметр «разреженность» используется в глубоком обучении часто. Он поощряет уменьшение числа скрытых блоков. Считается, что разреженность хо-

[14] Sutskever, 2013. Training Recurrent Neural Networks (PhD Thesis).

роша тем, что отдает предпочтение представлениям, в которых существенные факторы разделены[15].

Разреженность помогает предотвратить застревание процесса обучения, когда веса становятся большими и никак не удается вернуть их к нормальным значениям. Иногда разреженность взаимодействует с другими гиперпараметрами, определяющими размер сети. С увеличением числа скрытых блоков должен расти и коэффициент разреженности. Характер взаимодействия разреженности с размером сети зависит также от типа функции активации.

Чтобы понять, правильно ли выбрана разреженность, анализируется гистограмма средних активаций скрытых блоков. Видя ее, мы можем задать разреженность, так чтобы средние вероятности скрытых блоков были близки к целевому значению. Если вероятности тесно сгруппированы вокруг целевого значения, то разреженность следует уменьшить. Тогда она будет в меньшей степени препятствовать главной цели процесса обучения.

Разреженность и некоторые сетевые архитектуры
В DL4J разреженность неприменима к плотным и LSTM-слоям.

Целевые значения разреженности
Для получения хороших результатов обучения разреженность следует выбирать в диапазоне между 0.01 и 0.1^{-9}.

Применение методов оптимизации

Для оптимизации глубоких сетей чаще всего применяется стохастический градиентный спуск (СГС). Его довольно легко реализовать, но трудно правильно настроить и распараллелить.

Некоторые исследователи замечают, что включение дополнительных параметров в нейронную сеть не устраняет недообученность на больших наборах данных и лишь напрасно расходует ресурсы. Они указывают на СГС как на причину этого явления и предлагают использовать вместо него методы второго порядка, в т. ч.:

- метод Бройдена–Флетчера–Гольдфарба–Шанно (Broyden–Fletcher–Goldfarb–Shanno – BFGS) и BFGS с ограниченной памятью (L-BFGS);
- метод сопряженных градиентов (СГ);
- безгессианный метод Ньютона.

В табл. 6.6 приведена сводка рекомендаций по выбору метода оптимизации в зависимости от архитектуры сети.

Таблица 6.6. ***Алгоритмы оптимизации для различных архитектур***

Сеть	Метод оптимизации
ГСД	СГС
СНС	СГС (+ прореживание)
РНС	СГС, безгессианный метод

Отметим влияние размера обучающего набора на выбор метода оптимизации. Если размер набора данных невелик, то мы рекомендуем выбирать методы

[15] Le, Jaitly and Hinton, 2015. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units // https://arxiv.org/abs/1504.00941.

оптимизации второго порядка и устанавливать размер пакета равным размеру всего набора. А для больших наборов данных лучше выбрать СГС с мини-пакетами. Можно также попробовать метод второго порядка с мини-пакетом, меньшим всего набора.

***Таблица 6.7.** Сравнение методов оптимизации*

Метод	Информация первого порядка	Информация второго порядка	За	Против
СГС	Градиент	Нет	Быстро сходится, низкая стоимость обновления в расчете на один параметр	Недостаточно устойчив
L-BFGS	Градиент	Кривизна, оцененная по градиентам	Находит лучшие локальные минимумы	Обучается лучше, чем СГС, но ценой более высокой стоимости обновления в расчете на один параметр и более высокого потребления памяти
Сопряженных градиентов	Градиент	Кривизна, оцененная по градиентам	–	Более высокая стоимость обновления в расчете на один параметр, более высокое потребление памяти
СГС Безгессианный	Нет	Кривизна	Автоматически определяет размер шага, использует метод сопряженных градиентов для вычисления следующего шага	Не обобщается на все архитектуры, более высокая стоимость обновления в расчете на один параметр, более высокое потребление памяти

Несмотря на табл. 6.7, мы рекомендуем, как правило, начинать с СГС и потому в следующем разделе сосредоточимся на этом алгоритме.

Рекомендации по применению СГС

СГС хорошо работает с корректором Momentum и тщательно продуманной схемой инициализации весов. Продемонстрирована его эффективность при обучении ГСД и рекуррентных нейронных сетей[16] до уровня, не уступающего оптимизации безгессианным методом Ньютона. В больших задачах классификации мало кто может состязаться с тщательно настроенным методом СГС[17].

Ниже приведена краткая сводка рекомендаций о том, как начинать обучение глубокой сети методом СГС.

- Примените хороший метод перемешивания к входному обучающему набору[18].
- Следите за изменением процентной ошибки и частоты ошибок на контрольном наборе:

[16] Sutskever et al., 2013. On the importance of initialization and momentum in deep learning // http://www.cs.utoronto.ca/~ilya/pubs/2013/1051_2.pdf.

[17] LeCun et al., 1998. Efficient BackProp // Muller et al. 2012. Neural Networks: Tricks of the Trade Second Edition.

[18] Bottou, 2012. Stochastic Gradient Descent Tricks // Muller et al., 2012. Neural Networks: Tricks of the Trade Second Edition.

– мы хотим, чтобы ошибка обучения убывала;
– как только увидим, что частота ошибок на контрольном наборе перестает изменяться, обучение можно прекращать.

- Проверяйте различные варианты гиперпараметров на небольших подмножествах обучающих данных (особенно это относится к скорости обучения).
- Сочетайте СГС с корректорами Momentum, AdaGrad и RMSProp.
- Правильно задавайте скорость обучения: слишком много и слишком мало одинаково плохо.
- Нормируйте входные данные (это относится не только к СГС, но повторение – мать учения).

Для не слишком больших задач с вещественными выходами (аппроксимация функций, задачи управления и т. д.) отличные результаты дает метод сопряженных градиентов. С методами второго порядка обычно применяются мини-пакеты большего размера (10 000), чем в случае СГС.

СГС как самая популярная отправная точка
На практике чаще используют комбинацию СГС + (Momentum, или AdaGrad, или RMSProp и т. д.), чем методы второго порядка.

Применение распараллеливания и GPU для ускорения обучения

По мере того как цели машинного обучения становятся все более амбициозными, для их удовлетворения нужны все более крупные модели. Следовательно, приходится обучать больше параметров, а это требует больше времени. Кроме того, мы хотим обучать на больших наборах данных, чтобы модель видела более полную картину, и в какой-то момент ввод-вывод становится узким местом (см. также 12 чисел Джеффа Дина в приложении C). Если говорить об одной машине, то мы ограничены тактовой частотой современных процессоров. Компьютерные архитектуры уперлись в физические пределы мощности оборудования, так что новые прорывы возможны лишь на пути объединения мощностей нескольких процессоров с помощью различных вариантов распараллеливания. В современном машинном обучении для достижения предела последовательного обучения далеко ходить не надо[19]. Пропускная способность системы запоминающих устройств и сети одного компьютера не способна справиться с ростом объема данных. Мы должны думать об алгоритмах анализа данных, которые могут выполнять большинство шагов в распределенном режиме.

Доступ к диску
Доступ к жесткому диску характеризуется большим временем задержки. Например, при чтении со скоростью 100 МБ/с для считывания всех данных с диска объемом 2 терабайта понадобится примерно 6 часов. О различных задержках в компьютерных архитектурах см. «12 чисел Джеффа Дина» в приложении C.

---

[19] Zinkevich et al., 2011. Parallelized Stochastic Gradient Descent // http://martin.zinkevich.org/publications/nips2010.pdf.

Онлайновое обучение и параллельные итеративные алгоритмы

Благодаря произошедшему вот уже десять лет назад переходу от пакетных алгоритмов обучения к онлайновым удалось справиться с ростом наборов данных. Но если данные не помещаются на одном диске, то обработать их с помощью СГС за разумное время невозможно из-за накладных расходов на ввод-вывод. В мире больших данных мы стремимся как можно меньше перемещать данные, поэтому ищем способы распараллелить выполнение программ.

В недавних работах традиционно последовательные процессы распараллелены. Для этого несколько ядер выполняют локальные вычисления, а затем результаты их работы объединяются[20]. Когда набор данных перестает помещаться на нескольких дисках, мы начинаем переосмысливать дизайн систем хранения и обработки. Именно это стало побудительным массивом для проектирования MapReduce и Google File System в компании Google в начале 2000-х годов (а затем работы Дуга Каттинга и Майкла Карафелла над системой Hadoop в середине 2000-х).

Распараллеливать программы вообще и алгоритмы обучения в частности можно разными способами. Любая параллельная программа состоит из одновременно выполняемых процессов. Различия в формах параллелизма связаны с тем, как именно процесс разлагается на части, которые могут параллельно выполняться в разных местах. Основные два способа – *параллелизм на уровне задач* и *параллелизм на уровне данных*.

Параллелизм на уровне задач

В случае параллелизма на уровне задач (его еще называют *функциональным параллелизмом*) мы разбиваем работу на несколько задач, выполнение которых планируется на различных процессорах. Процессорами могут быть как локальные потоки, так и ядра на другой физической машине.

Параллелизм на уровне данных

В этом случае одна и та же функция применяется к различным частям набора данных. Эти задачи выполняются в различных потоках (локально или на удаленной машине распределенного кластера), что превращает параллелизм на уровне данных в разновидность параллелизма на уровне задач. Вычисление распределяется между несколькими процессорами в распараллеленной вычислительной среде. Процессорами могут быть потоки, выполняемые одним ядром, потоки на разных ядрах или ядра физически отдельных машин в кластере.

При распараллеливании итеративных алгоритмов типа СГС мы можем эффективно использовать распараллеливание между локальными потоками с помощью техники усреднения параметров[21]. Но с ростом объема данных возникают проблемы из-за копирования данных в параллельные процессоры. Первое, что приходит в голову, – перейти к системе, способной обрабатывать большие данные, например Hadoop.

---

[20] Dean and Ghemawat, 2004. MapReduce: Simplified Data Processing on Large Clusters // https://static.googleusercontent.com/media/research.google.com/ru//archive/mapreduce-osdi04.pdf.

[21] Agarwal, Chapelle, Dudik and Langford, 2011. A Reliable Effective Terascale Linear Learning System // http://hunch.net/~vw/; McDonald, Hall and Mann, 2010. Distributed training strategies for the structured perceptron // https://static.googleusercontent.com/media/research.google.com/ru//pubs/archive/36266.pdf.

Большие данные?

Ныне термин «большие данные» постоянно на слуху во всем, что касается маркетинга ПО уровня предприятия. Он стал настолько вездесущим, что его даже употребил президент, а уж на страницах *Wall Street Journal* встречается чуть ли не ежедневно. Но интересно, что большинство людей затрудняется определить, что же он означает. Мы приверженцы практического определения: большие данные обрабатываются там, где хранятся, за счет того, что вычисление переносится ближе к ним. Вокруг этой идеи построена технология MapReduce. Многие компании и производители инструментальных средств заявляют, что умеют обрабатывать большие данные, но, говоря начистоту, если мы перемещаем данные перед обработкой, то называть их большими уже нельзя.

Сколько времени занимает сканирование петабайта?
Убедительным примером ограничений больших данных является тот факт, что после определенного момента перемещение данных становится безнадежным предприятием. Например, простая операция линейного сканирования или копирования одного петабайта данных при скорости, характерной для типичного диска потребительского класса (40 МБ/с), занимает 310 дней.

Apache Hadoop (http://hadoop.apache.org/) – замечательное средство для хранения и обработки больших объемов данных, таких как журналы веб-сервера или показания датчиков, испытанное на самых разных предприятиях, работающих с петабайтными данными. Исторически в дистрибутивы Hadoop[22] включался каркас MapReduce, распараллеливание в котором основано на обработке больших объемов данных в том месте, где они хранятся. Вычисления (задачи) перемещаются на компьютеры, где находятся блоки данных, и выполняются локально, чтобы максимизировать производительность физического диска.

Что такое MapReduce?

MapReduce – технология распараллеливания традиционных пакетных программ, в основе которой лежит идея функций map и reduce, применяемых в функциональном программировании. Задача-распределитель (map), исполняемая на компьютере, где хранится блок данных, чтобы сократить время ввода-вывода и передачи по сети, берет пары ключ-значение и раздает их различным редукторам для обработки (reduce). Hadoop прозрачно обрабатывает отказы, хотя традиционно в параллельном программировании эта задача возлагалась на программиста. Однако MapReduce плохо адаптируется к итеративным алгоритмам из-за проблем со временем инициализации одного прохода по данным.

Проблемы применения MapReduce к итеративным алгоритмам
Если требуется выполнить 100 проходов по данным, а фаза планирования MapReduce занимает 30 секунд на один проход, то накладные расходы составят 100 × 30 = 3000 секунд. Эти 50 минут накладных расходов перевешивают длительность многих процессов обучения. Такая динамика не позволяет считать MapReduce хорошей системой для распараллеливания итеративных алгоритмов.

[22] См.: Cloudera // https://www.cloudera.com/; Hortonworks // https://hortonworks.com/.

В последние годы методы параллельной оптимизации привлекают все большее внимание как способ масштабирования алгоритмов машинного обучения, и мы видим разнообразные методы ускорения. СГС изучался во многих научных работах[23] с целью найти способы его распараллеливания без привлечения MapReduce. Мы также наблюдаем развитие каркасов с открытым исходным кодом (Vowpal Wabbit[24] и Apache Spark[25]) для распараллеливания итеративных алгоритмов – центральной темой в них является усреднение параметров.

Распараллеливание СГС в DL4J

Еще в одной публикации Google[26] описывается система *Downpour* для распараллеливания СГС. В ней используется алгоритм AdaGrad, большое число реплик модели и алгоритм Sandblaster (параллельный вариант L-BFGS). Downpour (и ее составные части) легла в основу архитектуры распараллеливания в DL4J, изображенной на рис. 6.3.

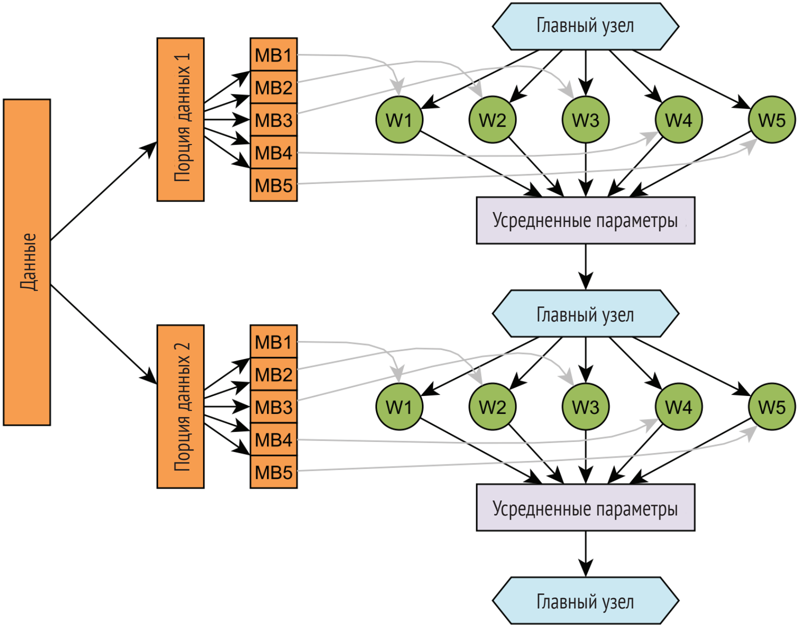


Рис. 6.3 ❖ Стратегия распараллеливания на основе усреднения параметров в DL4J

Одной из важнейших работ, подсказавшей стратегию распараллеливания в DL4J, стала работа Джеффа Дина и его группы в Google над системой SandBlaster.

[23] Ngiam et al., 2011. On optimization methods for deep learning // http://cs.stanford.edu/~jngiam/papers/LeNgiamCoatesLahiriProchnowNg2011.pdf.

[24] http://hunch.net/~vw/.

[25] http://spark.apache.org/.

[26] Dean et al., 2012. Large Scale Distributed Deep Networks // https://research.google.com/archive/large_deep_networks_nips2012.html.

Кто такой Джефф Дин?

Джефф Дин – заслуженный исследователь в компании Google, где он работал над некоторыми базовыми системами. Джефф пришел в Google в 1999 году и занимает должность старшего научного сотрудника в группе Knowledge Group. Как было сказано выше, Джефф возглавлял проектирование и реализацию систем DistBelief и SandBlaster (ICML 2012). DistBelief – крупномасштабная распределенная система для обучения глубоких сетей, которая применялась для распознавания изображений, распознавания речи, обработки естественного языка и других задач машинного обучения. Под руководством Джеффа разработаны и другие оказавшие большое влияние проекты в области машинного обучения и распределенных систем:

- пять поколений систем обхода и индексирования веба и обслуживания запросов;
- первая версия продукта AdSense for Content;
- дизайн и реализация MapReduce;
- дизайн и реализация BigTable;
- дизайн и реализация Spanner.

Малоизвестный факт:

> Джефф Дин пишет непосредственно двоичный код. А уже потом – исходный код в качестве документации для других разработчиков.

DL4J поддерживает разные среды выполнения (например, AWS, многопоточную работу, Spark-YARN поверх Hadoop), но во всех используется стратегия усреднения параметров (на сегодняшний день). Эта методика считается общепринятой в литературе по распараллеливанию итеративных алгоритмов, и мы проектировали библиотеку на основе ряда общих паттернов, которые затем перенесли в каждую среду выполнения.

Сервер параметров

В настоящее время ведутся работы по добавлению в проект сервера параметров, но пока усреднение параметров является достойным подходом ко многим задачам параллельного глубокого обучения.

Параллельное выполнение СГС

На верхнем уровне поток выполнения можно описать следующим образом. Каждому исполнителю выдается порция набора данных, и он применяет к ней собственный мини-пакетный алгоритм обучения. После обработки каждого мини-пакета исполнитель отправляет вычисленный вектор параметров главному мастеру, чтобы тот произвел усреднение параметров на *супершаге* процесса обработки.

Управление усреднением параметров

В DL4J частота усреднения настраивается. Мы считаем, что усреднение параметров «на каждый мини-пакет» неэффективно. Лучше задавать усреднение «на 5–20 мини-пакетов».

Затем главный узел отправляет новый глобальный вектор параметров всем исполнителям, чтобы они могли обработать следующий мини-пакет из своей порции данных. Это продолжается, пока не будет выполнено *N* проходов по всему набору данных (подробнее см. главу 9, посвященную Spark).

Проиллюстрируем эту идею на примере параллельного обучения. Пусть имеется 10 параллельных исполнителей, каждому из которых выдана порция из

100 записей. Каждый исполнитель разбивает эту порцию на мини-пакеты по 10 записей. Исполнители параллельно осуществляют обучение на каждом пакете, пока не обработают все свои 100 записей. Благодаря распараллеливанию общая скорость обучения кардинально возрастает, хотя обучение на уровне одного исполнителя происходит относительно медленно. Эксперименты, описанные в литературе и проведенные с открытыми реализациями, показывают, что скорость растет приблизительно линейно с увеличением числа машин.

Настройка параллельного обучения производится несколько иначе, чем последовательного.

При параллельном обучении два главных гиперпараметра – размер мини-пакета и регуляризация. Рассмотрим их внимательно, поскольку они могут влиять на результаты не так, как при последовательном обучении.

- Мини-пакеты. Если размер пакета при параллельном обучении слишком велик, то могут наблюдаться мини-пакеты с выбросами. Тогда время сходимости возрастает, потому что процессу обучения становится труднее переварить информацию, содержащуюся в выбросах.
- Регуляризация. Распараллеленная модель выигрывает при решении задач с большей дисперсией (малой постоянной регуляризации). Основное отличие обучения с усреднением параметров от последовательного – снижение накладных расходов на ввод-вывод и извлечение выгоды из локальности данных (в реализациях, которые перемещают исполнителей на компьютер, где хранится блок данных). Еще один побочный эффект состоит в том, что усреднение воздействует на вектор параметров как своего рода регуляризирующая функция.

Графические процессоры

Метод усреднения параметров – не единственный способ ускорить итеративный алгоритм. Рассмотрим еще один подход, основанный на использовании графических процессоров (GPU), в которых имеется специализированное оборудование для выполнения векторных математических операций. Для графических карт, которыми оснащаются современные ПК, пиковая пропускная способность памяти в несколько раз выше, чем у современных CPU.

Современные GPU имеют более 1000 ядер, поддерживающих технологию CUDA[27].

Интересное свойство GPU состоит в том, что они могут одновременно выполнять тысячи потоков, а выполнение задач линейной алгебры графическими ядрами возможно с небольшими накладными расходами. Благодаря этому классу мелкоструктурного параллелизма GPU являются привлекательным механизмом для ускорения крупномасштабных матричных вычислений, характерных для машинного обучения.

Мы хотим осуществлять доступ к памяти, так чтобы степень параллелизма была выше. GPU предлагают вариант, при котором операции доступа к памяти объединяются и оборудование может выполнять их параллельно, так что эффективная скорость доступа оказывается в несколько раз выше, чем при доступе

---

[27] https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units#GeForce_10_series.

к ЗУПВ со стороны CPU. Таким образом, узким местом становится обмен данными между оперативной памятью CPU и GPU. Чтобы было понятнее, скажем, что в большинстве случаев время такого обмена даже превосходит время перемножения больших матриц на GPU (так, собственно, умножение матриц размера 1000 × 1000 занимает лишь 0.5 процента от общего времени выполнения операции). Мы сможем повысить эффективность использования оборудования, если будем копировать за раз больше данных в память и выполнять несколько операций большим пакетом.

В приложении C отмечена желательность объединения операций с памятью в пакет. Обеспечив эффективное копирование данных на GPU, мы сможем воспользоваться преимуществами, которые дает одновременная работа тысяч потоков над отдельными блоками обучающих данных. Эти соображения и определяют архитектуру линейно-алгебраических операций в DL4J, обеспечивающую ускоренное вычисление большого числа параметров нейронной сети. Чтобы еще больше ускорить вычисления, мы создали библиотеку ND4J, которая упрощает перенос векторных вычислений в разные среды выполнения линейной алгебры и на разные GPU.

ND4J: адаптируемый движок выполнения

DL4J с самого начала проектировалась с пониманием того, что работать придется на CPU, GPU или в параллельной инфраструктуре. Поэтому интерфейсы ND4J со средой выполнения легко заменяются, что обеспечивает гибкость выбора.

Подробнее о конфигурировании GPU, на котором выполняется ND4J, написано в приложении E.

Управление периодами и размером мини-пакета

В главе 2 мы познакомились с идеей мини-пакета. Было показано, что разделение обучающих данных на мини-пакеты позволяет обучать сеть более эффективно. Размер мини-пакета может варьироваться от десяти примеров до всего входного набора.

Та же идея позволяет производить векторное вычисление некоторых линейно-алгебраических операций (конкретно перемножения матриц). И тогда векторные вычисления можно перенести на GPU, если они присутствуют в системе.

Введем два важных термина, относящихся к управлению обучением:

- *период* – так называется полный проход по всему входному набору данных. Нередко для достижения сходимости приходится обучать сеть в течение нескольких периодов;
- *размер мини-пакета* – это число примеров (или векторов), передаваемых за один раз алгоритму обучения. Это выгоднее, чем передавать по одному обучающему примеру.

Зависимость скорости обучения модели от размера пакета обычно описывается U-образной кривой. Это означает, что вначале время обучения уменьшается с ростом размера пакета. Но после того как размер пакета достигнет некоторой пороговой величины, время обучения начинает возрастать.

По мере возрастания размера мини-пакета вычисленные градиенты становятся более гладкими, но время их вычисления увеличивается.

В идеале каждый обучающий мини-пакет должен содержать примеры из каждого класса, чтобы уменьшить выборочную погрешность при вычислении оценки градиента для всего набора.

Компромиссы при определении размера мини-пакета

В принципе (после соответствующей настройки), нейронная сеть может обучиться при любом размере мини-пакета. Но на практике размер выбирается с учетом следующих факторов:

- требований к памяти;
- вычислительной эффективности;
- эффективности оптимизации.

Требования к памяти рассматривались выше, и снова говорить о них мы не будем.

Что касается *вычислительной эффективности*, DL4J (и другие современные библиотеки глубокого обучения) распараллеливают обучение на уровне таких математических операций, как умножение матриц и операции над векторами (сложение, скалярное произведение и т. д.). Это значит, что при слишком малом размере пакета возможности оборудования задействуются не полностью (особенно это относится к GPU), а при слишком большом вычисления становятся неэффективными – мы усредняем градиенты по всем примерам из мини-пакета, поэтому в конце концов стоимость добавления новых градиентов начинает перевешивать выгоду.

Максимальная производительность (это особенно важно для GPU) достигается, когда размер пакета кратен 32 (https://svail.github.io/rnn_perf/) или хотя бы 16, 8, 4 или 2, если выбрать размер, кратный 32, невозможно (или приводит к слишком большим изменениям с учетом других требований). Причина проста: доступ к памяти и общая конструкция оборудования оптимизированы для работы с массивами, размер которых равен степени двойки.

При задании размера слоя также следует стремиться к степеням двойки. Лучше задавать размер 128, а не 125, 256, а не 250 и т. д.

В части *эффективности оптимизации* следует отметить, что мы не можем выбирать размер мини-пакета, полностью абстрагировавшись от других гиперпараметров, в т. ч. скорости обучения: чем больше мини-пакет, тем более гладкими (а значит, более верными) получаются градиенты, что при надлежащей настройке означает более быстрое обучение при заданном числе обновлений параметров. Но, конечно, на вычисление каждого обновления параметров будет уходить больше времени. Увеличение размера мини-пакета иногда может выручить в особо трудных случаях, например когда набор данных зашумлен или не сбалансирован.

Так какой же размер мини-пакета выбрать? На практике для обучения на CPU считается разумным размер от 32 до 256, а для обучения на GPU – от 32 до 1024. Обычно значение из этого диапазона вполне приемлемо для сетей умеренного размера, но для больших сетей (когда полное время обучения слишком велико для экспериментов) требуется предварительное тестирование.

Однако для больших сетей максимальный размер пакета в любом случае ограничен требованиями к памяти.

При распределенном обучении довольно часто используют мини-пакеты меньшего размера для каждого исполнителя, работающего на разделяемом оборудовании (например, когда обучение производится на Spark с несколькими исполнителями на одной машине).

Наконец, не забывайте, что число обновлений параметров в одном периоде уменьшается при увеличении мини-пакета, поскольку это просто общее число примеров в обучающем наборе, поделенное на размер мини-пакета.

Связь между размером мини-пакета и числом периодов
Если увеличить размер мини-пакета вдвое, то и число периодов нужно увеличить вдвое, чтобы число обновлений параметров осталось тем же самым.

О применении регуляризации

Некоторые виды регуляризации способствуют уменьшению весов, которые слишком быстро становятся чрезмерно большими (например, по норме L1 и L2). В учебниках по машинному обучению снижение весов рассматривается как регуляризация. В других видах регуляризации, например прореживании (dropout) и прореживании связей (dropconnect), для противодействия переобучению применяются другие методы, помимо уменьшения больших весов. Вообще, регуляризация применяется в обучении не только для предотвращения переобучения.

Регуляризация может дать более эффективное представление модели благодаря использованию методов, которые препятствуют чрезмерному росту параметров. Подбор правильной комбинации параметров регуляризации обычно производится вручную, но имеются также автоматизированные методы: «случайный поиск» или «поиск на сетке». В следующем разделе мы кратко опишем все методы регуляризации и объясним, как они влияют на процесс обучения в целом.

Априорная функция как регуляризатор

Применение априорных функций – распространенная в машинном обучении техника регуляризации вектора параметров. Для снижения весов обычно применяются априорные функции L1 или L2. Иногда эти функции комбинируются[28], особенно при обучении глубоких сетей. В табл. 6.8 показаны, какие априорные функции использовать в зависимости от типа моделируемых данных.

***Таблица 6.8.** Рекомендации по использованию априорных функций*

Априорная функция	Где используется
L1	Разреженные модели
L2	Плотные модели

В литературе описано одновременное использование регуляризаторов L1 и L2 с разными параметрами. В тех случаях, когда используется ранняя остановка, L2-регуляризация вообще ни к чему, потому что ранняя остановка решает ту же задачу более эффективно[29]. L1-регуляризация полезна для выделения признаков.

---

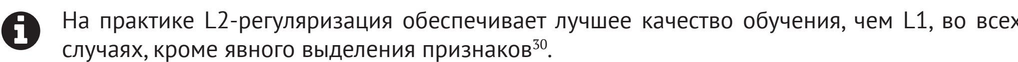
[28] Zou and Hastie, 2005. Elastic Net // http://users.stat.umn.edu/~zouxx019/Papers/elasticnet.pdf.

[29] Bengio, 2012. Practical Recommendations for Gradient-Based Training of Deep Architectures // Muller et al., 2012. Neural Networks: Tricks of the Trade, Second Edition.

На практике L2-регуляризация применяется чаще. Вообще:

- L2 сильнее штрафует за большие веса, но не заставляет малые веса обращаться в 0;
- L1 слабее штрафует за большие веса, но сводит малые веса к 0 (или близко к 0), так что результирующий вектор весов может оказаться разреженным.

L1 и L2-регуляризацию можно сочетать в одной сети.



На практике L2-регуляризация обеспечивает лучшее качество обучения, чем L1, во всех случаях, кроме явного выделения признаков[30].

В следующем разделе мы увидим, что результаты моделирования могут зависеть также от объема используемых данных.

Регуляризация по максимальной норме

При такой форме регуляризации мы задаем верхнюю границу L2-нормы входного вектора весов для каждого скрытого блока. Это не то же самое, что штрафование за квадрат длины всего вектора весов, как обычно делается в случае L2-нормы. Метод, основанный на ограничениях, а не на стратегии штрафования, не дает весам чрезмерно расти, каким бы большим ни было обновление входящих весов.

Регуляризация по максимальной норме дает возможность начать с высокой скорости обучения вкупе со стратегией ее убывания (например, AdaGrad), что позволяет производить более полный поиск в пространстве весов по сравнению с другими методами с меньшей скоростью обучения. Продемонстрировано, что этот подход хорошо работает с СГС в глубоких нейронных сетях даже без прореживания.

Прореживание

Прореживание – эффективный и вычислительно недорогой метод регуляризации, применимый к моделям разных типов. Его идея заключается в том, чтобы исключать из сети некоторые блоки в процессе ее обучения. Прореживание работает почти во всех архитектурах и хорошо сочетается с СГС. Технически это делается путем временного обнуления активаций блоков. Нейроны входного слоя прореживаются с некоторой вероятностью (сохранения или обнуления активации) – от 0.5 до 1.0.

Иногда входной слой не прореживается вовсе, особенно для зашумленных или разреженных наборов данных.

Скрытые слои прореживаются с вероятностью 0.5. Случайное исключение нейронов предотвращает коадаптацию детекторов, что улучшает обобщаемость модели на зарезервированных данных.

Выходные слои и прореживание

К выходным слоям прореживание обычно не применяется.

В общем случае полезный эффект дают почти все значения коэффициента прореживания, кроме совсем уж крайних. Часто задают коэффициент 0.5, который хоро-

[30] Li, Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition (Course Notes) // http://cs231n.stanford.edu/.

шо работает для широкого диапазона сетей и целей. Прореживание во всех скрытых слоях эффективнее, чем в каком-то одном скрытом слое. Кроме того, прореживание обычно приводит к более редким активациям скрытых блоков даже в отсутствие других методов регуляризации, что порождает разреженные представления[31].

В DL4J, чтобы применить прореживание (с вероятностью 0.5), нужно включить в код конфигурирования сети такую строку:

```
.regularization(true).dropout(0.5).
```

Вероятность прореживания

В DL4J вероятность прореживания (задаваемая с помощью метода `dropout(double)`) определена как вероятность сохранения активации. То есть `dropout(0.7)` означает, что 70% активаций сохраняются, а 30% обнуляются. Вызов `dropout(0.0)` означает «выключить прореживание в этом слое».

Согласно оригинальной работе Сриваставы с соавторами, эффект прореживания не сильно зависит от вероятности. В большинстве случаев достаточно просто положить ее равной 0.5. Но если сеть очень велика (относительно объема обучающих данных), то имеет смысл понизить вероятность сохранения активации (это соответствует более сильной регуляризации).

Еще несколько замечаний о прореживании:

- на практике прореживание часто сочетают с другими методами регуляризации, в т. ч. с L2-регуляризацией;
- вообще говоря, следует избегать прореживания в первом слое сети (т. е. выключать его вызовом `dropout(0.0)`): это воспрепятствует вымыванию важных частей входного набора данных.

Прореживание и рекуррентные нейронные сети в DL4J

В рекуррентных нейронных сетях прореживание обычно применяется только к входящим в слой активациям, а не к рекуррентным активациям внутри слоя. Применение прореживания к рекуррентным активациям может помешать сети обучиться временным зависимостям, поэтому в DL4J оно не применяется.

Проблемы в связи с прореживанием

Как показывают исследования, прореживание не столь эффективно, когда количество обучающих примеров исчисляется десятками миллионов. Показано также, что прореживание увеличивает время обучения в два-три раза по сравнению с той же архитектурой без прореживания. Градиентный шум при включенном прореживании выше, чем в стандартном СГС.

Для борьбы с градиентным шумом можно использовать более высокий параметр импульса (не 0.9, а от 0.95 до 0.99). Иногда это приводит к росту весов, но с этим можно сладить при помощи регуляризации по максимальной норме[32].

Прореживание связей

Прореживание связей (DropConnect)[33] – это просто временное обнуление некоторых весов. Оно связано с прореживанием, но не является его частным случаем, поскольку применяется к другим объектам.

---

[31] Srivastava et al., 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting // http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf.

[32] Srivastava et al., 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting // http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf.

[33] https://cs.nyu.edu/~wanli/dropc/.

Другие вопросы, связанные с регуляризацией

- *Стохастический пулинг*[34] – эта техника регуляризации применяется в СНС для построения ансамблей СНС. Каждая сеть занимается своим пространственным расположением каждой карты признаков.
- *Состязательное обучение*[35] – мы можем создать входные данные, которые намеренно немного искажают обучающие. В результате получится модель, которая с высокой степенью уверенности дает неправильные ответы. Показано, что на практике это уменьшает переобучение и порождает модели, устойчивые к неправильным примерам.
- *Обучение по плану*[36] – в этом случае обучение начинается с простых примеров, а со временем сложность примеров возрастает. Такая стратегия приводит к ускорению обучения и сходимости к лучшим решениям и в то же время оказывает регуляризирующее воздействие на сеть. Применялась с переменным успехом.

Распараллеливание и регуляризация

Следует стремиться к исключению из весов шума, образовавшегося в процессе обучения, в т. ч. вследствие использования различных скоростей обучения. Один из способов исключить шум из окончательных весов – усреднить веса по нескольким обновлениям. Этого можно добиться благодаря эффекту усреднения параметров в режимах параллельного выполнения, описанных ниже в этой главе.

Иногда при распараллеливании мы вообще не используем априорные функции в качестве регуляризатора, поскольку усреднение параметров само по себе дает нужный эффект.

Дисбаланс классов

В машинном обучении часто приходится иметь дело с обучающими наборами данных, в которых имеет место значительный дисбаланс классов, т. е. количество примеров из различных классов сильно различается. Обычно так происходит, когда интересующее нас событие случается гораздо реже, чем не случается, например если мы хотим предсказать, щелкнет ли кто-нибудь по рекламному объявлению на странице сайта.

PhysioNet и прогнозирование смерти в палате интенсивной терапии

Набор данных PhysioNet Challenge, упомянутый в главе 1, – прекрасный пример несбалансированного набора данных. Получить достаточно данных о таком (к счастью) редком событии, как смерть в палате интенсивной терапии, – серьезная проблема.

Если обучать модель на данных, которые на 99 процентов отрицательны и лишь на 1 процент положительны, то в большинстве методов обучения модель научится предсказывать доминирующий класс (отрицательный). Поначалу мы, возможно, и не заподозрим ничего плохого, потому что среднее число правильных ответов

[34] Zeiler and Fergus, 2013. Stochastic Pooling for Regularization of Deep Convolutional Neural Networks // http://www.matthewzeiler.com/iclr2013-2/.

[35] Goodfellow, Shlens and Szegedy, 2014. Explaining and Harnessing Adversarial Examples // https://arxiv.org/abs/1412.6572.

[36] Bengio et al., 2009. Curriculum Learning // http://www.machinelearning.org/archive/icml2009/papers/119.pdf.

исключительно велико, и мы можем просто счесть модель на удивление верной. Мы оказались в такой ситуации, потому что в процессе обучения стремились минимизировать ошибку и достигли 99-процентной верности, всегда предсказывая отрицательный класс.

При построении моделей в ситуации, когда данные не сбалансированы, рекомендуется не оптимизировать процент правильных предсказаний, а обратить больше внимания на такие оценки, как F-мера или площадь под кривой (Area Under the Curve – AUC). Модель иногда должна давать положительные предсказания, но хорошо бы, чтобы это было исключительно по делу.

Замечание о калибровке модели

В задачах, где имеет место значительный дисбаланс классов, последствия неправильного предсказания классов часто неравнозначны. Например, если не распознать серьезного заболевания, хотя пациент действительно болен (ложноотрицательный результат), то пациент умрет. В данном случае не заметить заболевание куда хуже, чем диагностировать заболевание, которого нет. С этой точки зрения, правильное оценивание модели чрезвычайно важно для практических приложений машинного обучения.

Для решения проблемы оценивания модели применяется метод *калибровки модели*. Обычно калибровка основана на построении такого преобразования, которое дает лучшие оценки значений или вероятностей. В табл. 6.9 приведена сводка методов калибровки.

Таблица 6.9. Таксономия проблем калибровки[37]

Тип	Задача	Проблема	Глобальная/ локальная	Что калибруется?
CD	Классификация	Ожидаемое распределение классов отличается от реального	Глобальная или локальная	Предсказания
CP	Классификация	Ожидаемая (оценочная) вероятность правильного предсказания отличается от реальной	Локальная	Вероятность / уверенность
RD	Регрессия	Ожидаемый результат отличается от реального среднего результата	Глобальная или локальная	Предсказания
RP	Регрессия	Ожидаемый (оценочный) доверительный интервал ошибки или функция плотности вероятности слишком узкие или слишком широкие	Локальная	Вероятность / уверенность

Как отмечается в оригинальной статье, в случае типов CD и RD для калибровки результатов нужно модифицировать предсказания. В этой книге мы не будем развивать эту тему, но посчитали необходимым упомянуть ее для самостоятельного изучения.

Есть два основных способа учета дисбаланса в процессе обучения:

- более редкая выборка из больших классов или более частая из меньших;
- назначение обучающим примерам весов в соответствии с взвешенной функцией потерь.

Рассмотрим оба способа.

[37] Bella et al., 2009. Calibration of Machine Learning Models.

Методы выборки из классов

Первый подход к устранению дисбаланса классов – выбирать не все примеры из большего класса, а так, чтобы обучающий набор оказался сбалансированным (50% положительных и 50% отрицательных примеров). Недостатком этого метода является тот факт, что мы отбрасываем хорошие данные, а достоинством – что обучение становится более эффективным.

Одно из решений – применить *апостериорное масштабирование*. Это означает, что мы обучаем сеть как обычно, но после обучения масштабируем результаты. Получается хуже, чем при использовании других методов.

Другое решение – *вероятностная выборка*. В этом случае мы сначала случайным образом (с некоторой вероятностью) выбираем класс, а затем – случайный пример из этого класса.

И последнее решение – *избыточная выборка*, когда мы дублируем примеры из низкочастотных классов, чтобы уравнять их число с примерами из других классов[38].

Взвешенные функции потерь

Другой способ учета дисбаланса классов в процессе обучения состоит в том, чтобы назначать меньшие веса обучающим примерам из большего класса. Это позволяет выправить несбалансированный набор данных и построить модель, способную лучше распознавать более редкие метки при предъявлении незнакомых примеров.

Проиллюстрируем эту идею небольшим фрагментом кода. Ниже создается массив INDArray для хранения весов классов, используемых во взвешенной функции потерь:

```
INDArray weights = Nd4j.create(new double[]{0.1, 0.4, 1.0});
```

Этот массив передается конструктору функции потерь в строке, выделенной полужирным шрифтом:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(1)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.learningRate(learningRate)
.updater(Updater.NESTEROVS).momentum(0.9)
.list()
.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
    .weightInit(WeightInit.XAVIER)
    .activation(Activation.RELU)
    .build())
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
    .weightInit(WeightInit.XAVIER)
    .activation(Activation.SOFTMAX)
    .lossFunction(new LossNegativeLogLikelihood(weights))
    .nIn(numHiddenNodes).nOut(numOutputs).build())
.pretrain(false).backprop(true).build();
```

[38] В глубоком обучении с несколькими периодами между различными вариантами выборки практически нет разницы, одна при использовании метода опорных векторов и случайных лесов разница есть – и весьма существенная.

Смысл взвешенной функции потерь – в том, чтобы сильнее штрафовать за неправильные предсказания редких классов. В идеале это должно увести сеть от плохих локальных минимумов, в которых всегда предсказываются самые частые классы.

Об относительных весах
Вес 1.0 означает отсутствие взвешивания (например, {1.0, 1.0, 1.0}). Если все веса равны 1.0, то массив весов можно вообще не задавать. Но мы хотим, чтобы у часто встречающихся классов веса были меньше.

Борьба с переобучением

Переобучением называется встречающийся в машинном обучении феномен, когда модель так хорошо подгоняется к обучающему набору данных, что плохо работает для примеров, которых раньше не видела. Говорят, что модель не обобщается на больший набор данных.

Если модель просто запоминает случайные закономерности в обучающих данных, то она будет плохо работать для незнакомых примеров. Если модель так хорошо обучилась на предложенных данных, что плохо обобщается на другие, то ее полезность снижается. Переобученная модель уловила случайные корреляции, которые ничем не помогут при оценке ранее не предъявлявшихся данных.

Повышение обобщаемости
При обучении моделей нашей целью является обобщение информации, содержащейся в обучающем наборе данных, таким образом, чтобы модель хорошо работала на других данных из похожих источников.

Любой процесс машинного обучения в той или иной мере склонен к переобучению, и мастерство заключается в том, чтобы вовремя прекратить обучение, пока модель не потеряла способности к обобщению. При моделировании сложных наборов данных с помощью глубокого обучения мы неизбежно используем сеть с большим числом параметров. Компромисс состоит в том, чтобы добавлять параметры в темпе, позволяющем моделировать сложные наборы, но не слишком быстро, чтобы не получить переобученную модель.

Для обнаружения переобученности мы можем оценить качество модели на зарезервированном тестовом наборе. Важно, чтобы тестовые данные не использовались для обучения, поскольку в таком случае мы ничего не узнаем о том, как сеть ведет себя на незнакомых данных. Из стратегий противостояния переобучению отметим следующие:

- повышение степени регуляризации (L1, L2, прореживание, прореживание связей);
- ранняя остановка обучения;
- увеличение размера набора данных;
- уменьшение размера сети.

Первым шагом в борьбе с переобучением обычно является регуляризация. Может помочь регуляризация по норме L1 или L2 (это обсуждалось выше). Прореживание тоже часто используется и, как правило, оказывается эффективной мерой регуляризации нейронных сетей.

Переобучение и число параметров
Если количество параметров искусственно ограничено, то модель окажется неверной. Если же число параметров слишком велико, то модель на первый взгляд кажется верной, но она переобучена, поэтому верность на зарезервированных данных будет не такой, как на обучающих.

Использование статистики сети из интерфейса настройки

В состав DL4J входит средство сбора статистики сети, позволяющее наглядно и в реальном времени представить, что сеть делает. Анализируя текущее состояние (и историю) активаций, градиентов и обновлений, мы можем выявить и устранить недостатки в обучении или конфигурации.

Сконфигурировать пользовательский интерфейс (UI) настройки просто:

```
UIServer uiServer = UIServer.getInstance();
StatsStorage statsStorage = new InMemoryStatsStorage();
uiServer.attach(statsStorage);
int listenerFrequency = 1;
myNetwork.setListeners(new StatsListener(statsStorage, listenerFrequency));
```

О том, как конфигурировать объект `StatsListener` для сохранения статистики, как изменить порт UI и как отправить результаты удаленному UI, см. в онлайновой документации. Обратите внимание на переменную `listenerFrequency`, которая определяет частоту сбора информации (об активациях, обновлениях и т. д.). Чтобы снизить накладные расходы, связанные со сбором статистики, это число следует увеличить. Вполне разумно выглядит сбор информации после каждых 10 итераций или даже еще реже (если мы просто хотим понаблюдать за поведением прилично настроенной сети на протяжении длительного времени).

UI обучения в DL4J состоит из нескольких страниц, первая из которых называется Overview и содержит сводные данные, представленные в виде четырех разделов (рис. 6.4):

- слева вверху: оценка модели в зависимости от итерации (и ее скользящее среднее);
- справа вверху: сводная информация о сети;
- слева внизу: отношения обновлений к параметрам;
- справа внизу: зависимости активаций, градиентов и обновлений от времени.

Здесь слово «оценка» относится к сумме функции потерь (СКО, отрицательное логарифмическое правдоподобие и т. п.) и регуляризирующих членов (например, L1 или L2). Вторая страница UI посвящена самой модели (рис. 6.5). Она позволяет наглядно представить структуру модели и получить детальные сведения о каждом слое.

Страница Model состоит из двух разделов. Слева показана структурная схема сети, на которой изображены уровни (и вершины в случае графа вычислений – ComputationGraph). Справа – сводная таблица (конфигурация слоев) и несколько графиков, в т. ч.:

- зависимость отношения обновлений к параметрам от времени для каждого типа параметров (имеются также вкладки для показа средних величин параметров и средних величин обновлений);

- зависимость активаций слоев от времени (среднее и среднее ± 2 стандартных отклонения);
- гистограммы для каждого типа параметров для одного слоя;
- гистограммы обновлений (для каждого типа параметров) для одного слоя;
- зависимость скоростей обучения от времени для каждого параметра (она будет плоской, если не используется стратегия изменения скорости обучения).

Рис. 6.4 ❖ Средство просмотра статистики сети в DL4J

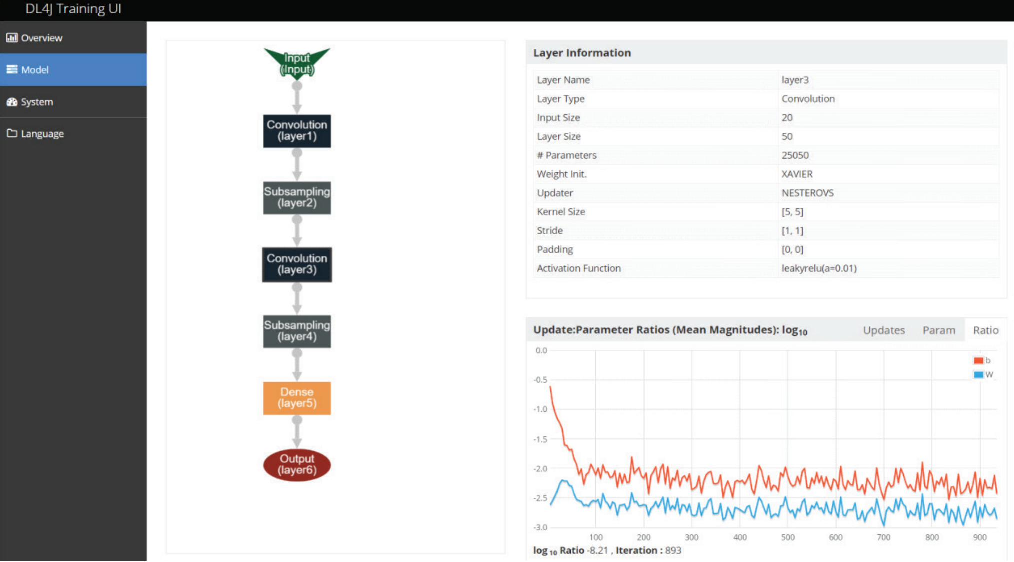


Рис. 6.5 ❖ Страница Model пользовательского интерфейса

Останов сервера пользовательского интерфейса

Закончив изучение, мы можем остановить UI, вызвав метод `UIServer.getInstance().stop();` или просто сняв JVM вручную. Но до тех пор сервер UI будет продолжать работу, даже если обучение уже завершилось.

Есть еще инструмент для вычисления потери (или верности) на тестовом наборе, т. е. данных, на которых обучение не производилось. Это полезнее для выявления переобученности, тогда как UI чаще используется для обнаружения и исправления проблем, относящихся к оптимизации. Обычно настройку начинают с UI, а уже потом вычисляют потерю или верность на тестовом наборе.

Обнаружение неудачной инициализации весов

Стратегии инициализации ReLU и Xavier спроектированы с учетом архитектуры сети (размеров слоев) и предположений о функции активации, так чтобы обеспечить две вещи:

- постоянство дисперсии активаций сети для всех слоев (т. е. активации не должны слишком сильно увеличиваться или уменьшаться в последующих слоях сети);
- постоянство градиентов активации (и, следовательно, параметров) для всех слоев, т. е. градиенты не должны оказаться слишком большими или слишком малыми к моменту обратного распространения на предшествующие уровни сети.

К сожалению, в общем случае невозможно гарантировать одновременное выполнение обоих требований. Особенно если имеется ситуация, когда соседние слои отличаются по размеру (например, за слоем с 1024 нейронами следует слой с 10 нейронами или наоборот) или когда функции активации существенно отличаются от предположений, принятых в стратегиях инициализации Xavier/ReLU (например, не совпадают с ReLU, tanh, тождественной функцией или чем-то подобным), – тогда, возможно, придется прибегнуть к другой схеме инициализации. Это особенно актуально для глубоких сетей. Если же число слоев сети невелико, то можно смириться с неоптимальной инициализацией.

Существует два способа обнаружить, что веса плохо инициализированы. Во-первых, можно использовать график стандартного отклонения (логарифмического) активаций на странице Overview, показанный на рис. 6.6.

Для построения этого графика была взята стратегия инициализации весов ReLU (с функцией активации ReLU) и дисперсия начальных весов уменьшена втрое. При такой (намеренно плохой) инициализации активации становятся все меньше и меньше и к последнему слою практически обращаются в нуль. Это можно обнаружить также, глядя на графики зависимости активаций в слое от номера итерации, показанные на странице Model (рис. 6.7).

Активации могут также чрезмерно возрастать, если начальные веса слишком велики. Это тоже можно обнаружить, глядя на графики на рис. 6.7.

В общем случае желательно, чтобы логарифм стандартного отклонения активаций в скрытых слоях был порядка 1.0.

При интерпретации этих графиков важно помнить об одной вещи: результат плохой нормировки данных может выглядеть так же, как результат неудачно-

го выбора начальных весов. Обычно разобраться в этом можно, поглядев непосредственно на графики активации; например, мы видим, что на рис. 6.7 стандартное отклонение входных данных равно 1 ($\log_{10}(1) = 0$). Это хороший масштаб для входных данных. Следовательно, заметив необычно большие или необычно малые активации, нужно проверить, все ли в порядке с нормировкой данных.

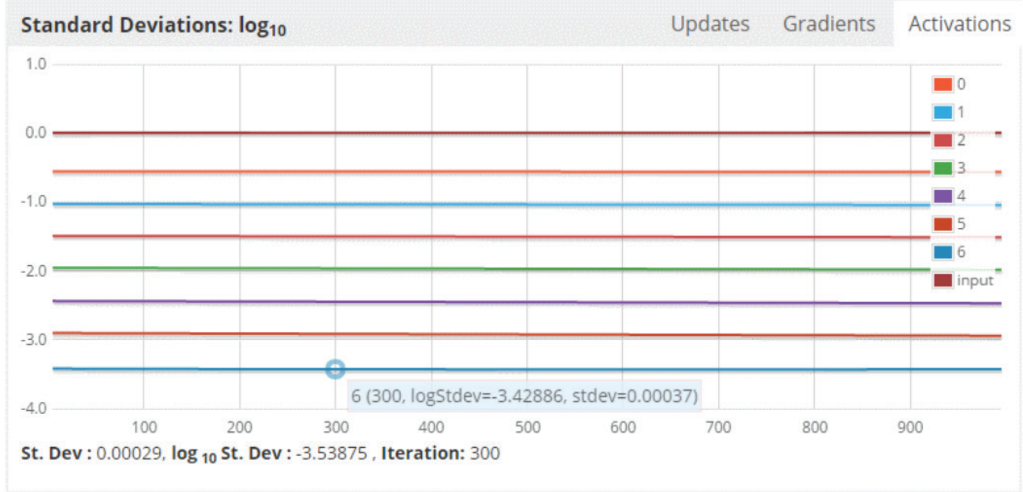


Рис. 6.6 ❖ График логарифма стандартного отклонения активаций

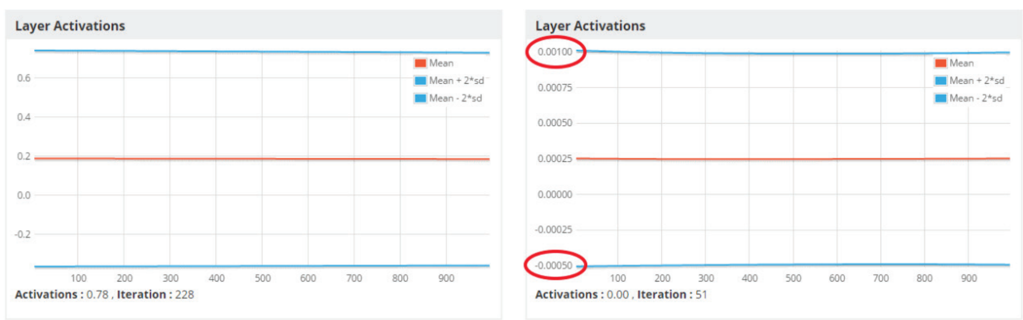


Рис. 6.7 ❖ Графики активации в слоях

Обнаружение неперемешанных данных

Для обучения нейронной сети обычно используются мини-пакеты, т. е. небольшие подмножества обучающих данных (как правило, от 32 до 1024 примеров). Чтобы результаты обучения были лучше, данные следует перемешать, т. е. предъявлять в случайном порядке, а не сначала все данные одного класса, потом все данные другого класса в одном мини-пакете. Если обучающие данные перемешаны хорошо, то картина обучения должна выглядеть, как показано на рис. 6.8.

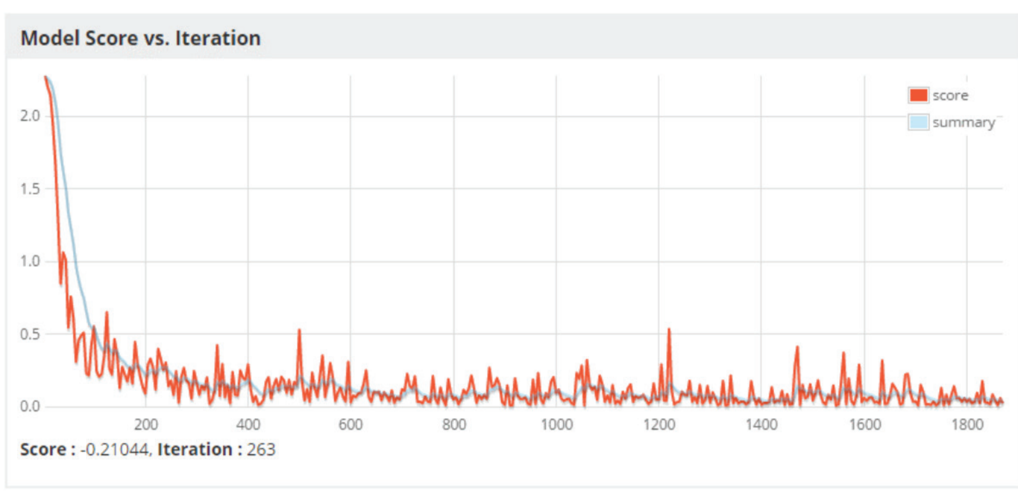


Рис. 6.8 ❖ График оценки обучаемой модели

Чтобы проиллюстрировать, как плохое перемешивание может негативно сказаться на результате обучения, рассмотрим уже упоминавшийся ранее набор MNIST. Допустим, что требуется обучить сеть распознавать цифры, но примеры предъявляются не в случайном порядке (как было бы естественно), а в порядке возрастания цифр, т. е. сначала идут мини-пакеты, содержащие только 0, за ними – только 1 и т. д.

На рис. 6.9 показано, что при таких плохо перемешанных данных оценка модели изменяется со временем совсем иначе.

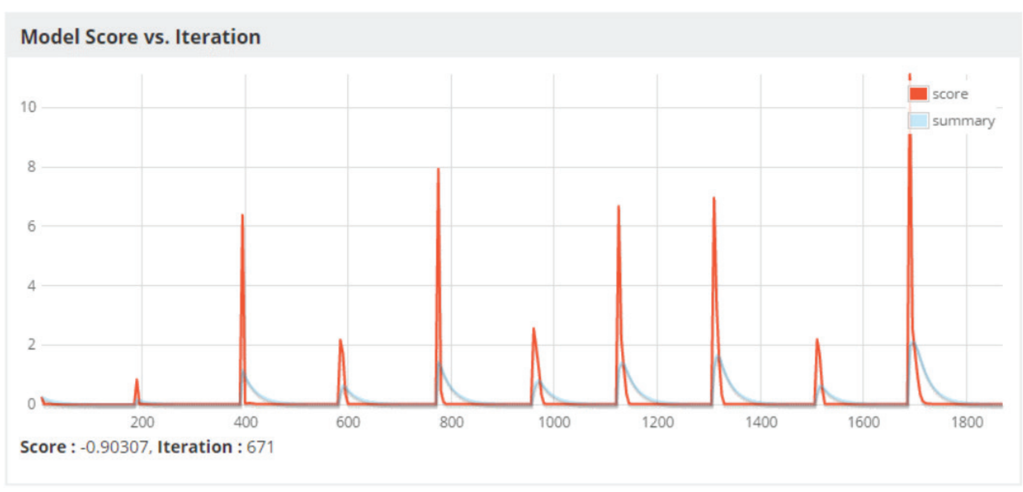


Рис. 6.9 ❖ График оценки обучаемой модели с пиками

На перемешанных данных эта модель обучилась бы вполне достойно, и график был бы таким, как на рис. 6.8. А так мы видим периодические высокие пики, соответствующие переходу от одной цифры к другой. Запомните: всякий раз, увидев

такие периодические пики, надо задуматься о качестве подготовки обучающих данных. Как правило, нужно внести небольшие исправления в процедуру подготовки данных – и все наладится.

Замечание о порядке следования примеров в мини-пакете
Отметим, что перемешивать данные в пределах одного мини-пакета необязательно, поскольку градиенты все равно усредняются по всем примерам из мини-пакета и только потом применяются к модели.

Обнаружение проблем, относящихся к регуляризации

Как и в случае скорости обучения, задание слишком большого или слишком малого параметра регуляризации по норме L1 или L2 может негативно сказаться на результатах обучения. Если параметр слишком велик, то веса будут сдвигаться к 0, а если слишком мал, то это чревато переобучением или неустойчивостью процесса обучения. Обнаружить завышение параметра L1/L2 можно, воспользовавшись вкладкой **Param** на странице **Model**. Здесь показаны средние абсолютные величины параметров (рис. 6.10).

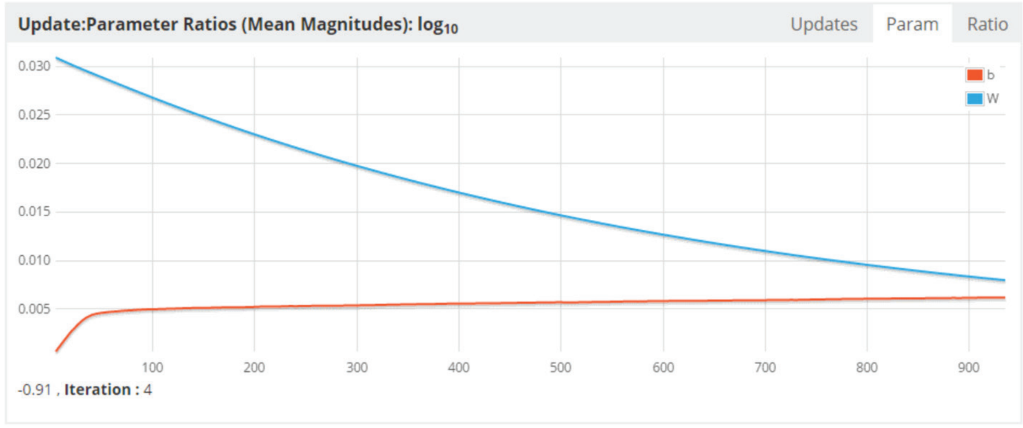


Рис. 6.10 ❖ Веса, стремящиеся к нулю

Мы видим, что вначале средние абсолютные веса велики, но с течением времени сходят на нет, что свидетельствует о чрезмерно агрессивной регуляризации. Поэтому следует уменьшить параметр регуляризации L1/L2.

Тот же инструмент позволяет обнаружить недостаточную регуляризацию. Мы хотим, чтобы гистограмма параметров в слое (Layer Parameters Histogram) имела правильную колоколообразную форму, как показано на рис. 6.11.

Если регуляризация недостаточна, то часто бывает так, что относительно небольшая часть параметров принимает несообразно большие значения, и гистограмма параметров в слое выглядит, как на рис. 6.12.

На рис. 6.12 не сразу понятно, что происходит, потому что в данном случае лишь немногие параметры принимают большие значения (обратите внимание, что большая часть столбиков гистограммы едва заметна). В экстремальных случаях самые большие параметры по мере продолжения обучения стремятся к $\pm\infty$.

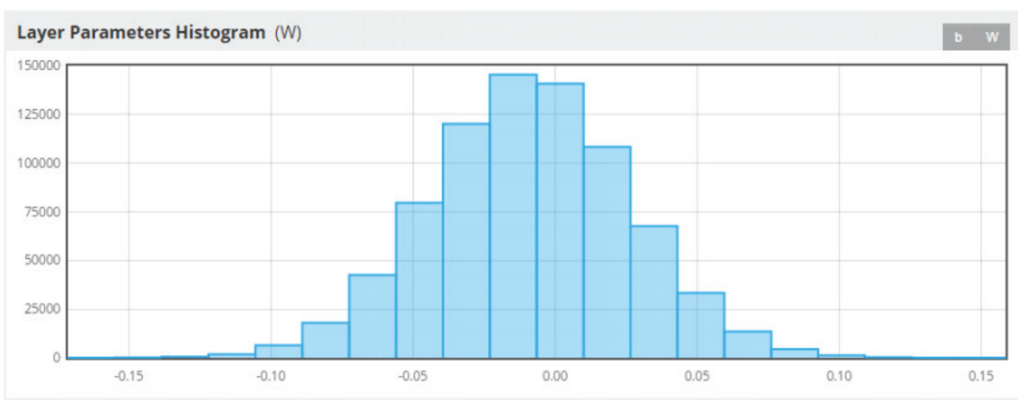


Рис. 6.11 ❖ Гистограмма параметров в слое с нормально распределенными весами

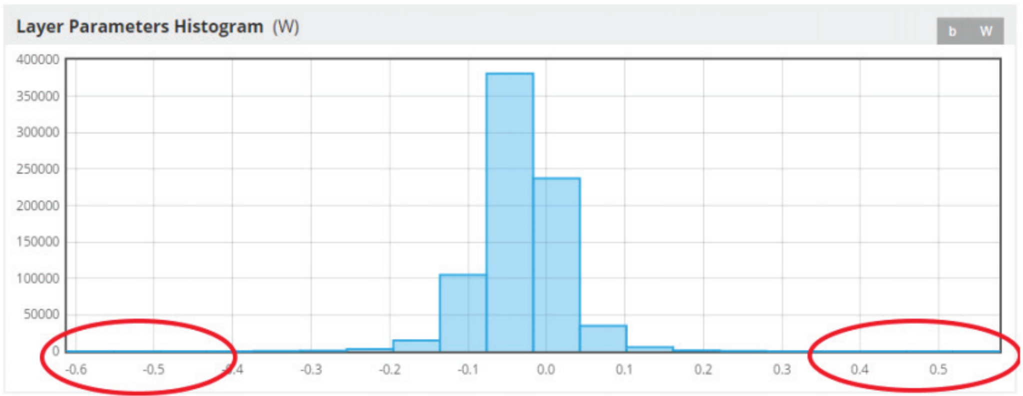


Рис. 6.12 ❖ Гистограмма параметров в слое с небольшим количеством больших весов

Глава 7

Настройка глубоких сетей с конкретной архитектурой

> В данный момент тебе нужно знать только, что Галактика гораздо сложнее, чем ты можешь подумать, даже если с самого начала будешь думать, что она чертовски сложна.
>
> – *Дуглас Адамс. «Автостопом по Галактике»*

В этой главе мы продолжим начатый в главе 6 разговор о настройке глубоких сетей и рассмотрим конкретные архитектуры:

- сверточные нейронные сети (СНС);
- рекуррентные нейронные сети (РНС);
- глубокие сети доверия (ГСД).

Поскольку компьютерное зрение – одно из самых популярных приложений глубокого обучения, начнем с вопроса о настройке архитектуры СНС.

Сверточные нейронные сети (СНС)

К СНС применимы как общие принципы проектирования сетей, так и специфические, относящиеся исключительно к сверточной архитектуре. В этом разделе мы рассмотрим только последние, связанные со сверточными и пулинговыми слоями.

Как было описано в главе 4, на стадии свертки используется много фильтров для обучения различным признакам, присутствующим во входных данных слоя. Выход этой стадии преобразуется функцией активации – блоком линейной ректификации (ReLU).

Детекторная стадия

В литературе по СНС иногда выделяют детекторную стадию в отдельный слой или этап архитектуры СНС. Эта стадия включает только функцию активации для слоя, в DL4J мы рассматриваем функцию активации как часть соответствующего слоя. Поэтому детекторную стадию мы будем считать частью сверточного слоя.

Между соседними сверточными слоями обычно вставляются пулинговые слои, цель которых – уменьшение пространственных размеров (ширины и высоты) представления данных, а значит, и числа параметров, которые необходимо рас-

считывать. Сокращение числа параметров заодно препятствует переобучению. Большинство других аспектов настройки, в частности стратегии настройки выходного слоя, уже было рассмотрено в главе 6.

Общие архитектурные паттерны сверточных сетей

Часто между каждыми двумя пулинговыми слоями располагается один сверточный слой, т. е. последовательность слоев имеет вид:

- входной слой;
- сверточный слой;
- пулинговый слой;
- сверточный слой;
- пулинговый слой;
- полносвязный слой;
- возможно, еще один полносвязный слой.

Иногда встречается несколько сверточных слоев подряд, а за ними пулинговый.

В крупных сетях даже рекомендуется размещать два сверточных слоя перед каждым пулинговым, поскольку это позволяет сети обучиться большему числу сложных признаков до того, как выход будет подвергнут понижающей передискретизации в пулинговом слое.

Чтобы лучше разобраться в этом архитектурном паттерне, вернемся к примеру сети LeNet из главы 5. Впервые опубликованная в 1998 г. Яном Лекуном, она является собой пример одной из самых известных сверточных архитектур. Показано, что она успешно моделирует набор данных MNIST (см. раздел «Моделирование рукописных цифр с помощью СНС» главы 5). Перечислим слои этой сети:

- входной слой;
- сверточный слой: 20 фильтров [5 × 5];
- max-пулинговый слой: [2 × 2];
- сверточный слой: 50 фильтров [5 × 5];
- max-пулинговый слой: [2 × 2];
- полносвязный слой.

В примере 7.1 эта конфигурация представлена в виде кода на Java (см. пример из главы 5). Для краткости мы включили только код конфигурирования, чтобы можно было сопоставить его с архитектурой сети LeNet.

Пример 7.1 ❖ Архитектура сети LeNet в виде конфигурации модели на Java в DL4J

```
/*
    Конструирование нейронной сети
 */
log.info("Построение модели....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(iterations) // Число итераций обучения
        .regularization(true).l2(0.0005)
        .learningRate(.01)
        .weightInit(WeightInit.XAVIER)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .list()
```

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
        // nIn и nOut задают глубину. Здесь nIn равно nChannels, а nOut –
        // количество применяемых фильтров
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
.layer(2, new ConvolutionLayer.Builder(5, 5)
        // Отметим, что в последующих слоях nIn задавать не нужно
        .stride(1, 1)
        .nOut(50)
        .activation(Activation.IDENTITY)
        .build())
.layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
.layer(4, new DenseLayer.Builder().activation(Activation.RELU)
        .nOut(500).build())
.layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
    .NEGATIVELOGLIKELIHOOD)
        .nOut(outputNum)
        .activation(Activation.SOFTMAX)
        .build())
.setInputType(InputType.convolutionalFlat(28,28,1))
.backprop(true).pretrain(false).build();
```

В примере 7.1 за каждым сверточным слоем следует пулинговый. Сеть завершается полносвязным слоем `DenseLayer`, за которым расположен выходной слой с функцией активации softmax. В данном случае softmax используется для распознавания одной из десяти рукописных цифр в обучающем и тестовом наборах. Обратите внимание еще на один метод:

```
.setInputType(InputType.convolutionalFlat(28,28,1))
```

Он выполняет несколько действий:

- добавляет препроцессоры, которые отвечают за переход между сверточным или пулинговым слоем и плотным слоем;
- осуществляет дополнительную проверку конфигурации;
- там, где необходимо, устанавливает значение `nIn` (число входных нейронов или глубину входа в случае СНС) для каждого слоя, исходя из размера предыдущего слоя:
  - значения, заданные вручную с помощью `InputType`, не переопределяются;
  - допустимо использование и для слоев других типов (РНС, МСП и т. д.), а не только для СНС.

Если говорить о проектировании сверточных архитектур, то в общем случае мы должны стремиться к включению нескольких меньших сверточных слоев вместо

одного слоя с большим рецептивным полем. Это позволит сети улавливать больше нелинейной динамики в данных с более выразительными признаками.

Для первого сверточного слоя глубина входа (`nIn`) должна соответствовать данным, а число фильтров (`nOut`) является свободным параметром. Говоря о «числе блоков», обычно имеют в виду `nOut`. В идеале мы хотели бы, чтобы размер входного слоя делился на степень двойки, большую 1, чтобы иметь достаточно информации для последующей субдискретизации и конструирования признаков. Так, в СНС для моделирования набора данных CIFAR-10 (см. главу 2) размер входного слоя равен 32. Как правило, в сверточных слоях используется функция активации ReLU.

Количество каналов во входных данных задается с помощью метода `.nIn()`:

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
    // nIn и nOut задают глубину. Здесь nIn равно nChannels, а nOut –
    // количество применяемых фильтров
    .nIn(nChannels)
```

Этот параметр можно задавать по-разному в зависимости от вида входных данных, но обычно он равен 1 для монохромных изображений и 3 – для цветных в формате RGB. Конечно, если на входе мы имеем не изображения, а какое-то многомерное представление, то эта величина может быть гораздо больше.

Несколько замечаний о размере первого сверточного слоя

В общем случае количество блоков во входном слое зависит от величины шага и размера фильтра.

Интересный пример дает сеть AlexNet[1]: размер входа 224 × 224 × 3, размер ядра 11 × 11 (необычно большой для современных сетей).

На практике данные можно кадрировать или масштабировать под размер, подходящий для выбранной конфигурации сети. По возможности лучше уменьшать, а не увеличивать размер изображения.

После первого сверточного слоя идут чередующиеся сверточные и пулинговые слои. В некоторых архитектурах, например VGGNet, бывает несколько (скажем, два) сверточных слоя, а за ними один пулинговый.

Вообще говоря, для первых сверточных слоев СНС берут фильтры относительно большого размера и постепенно уменьшают его в последующих слоях.

Не существует «волшебной» последовательности сверточных слоев

На момент написания этой книги не существовало какой-то одной волшебной сверточной архитектуры, оптимальной для всех задач моделирования изображений. Мы рекомендуем начинать с заведомо успешной архитектуры, например LeNet, VGGNet, Inception[2] или AlexNet, и адаптировать ее к своей задаче, варьируя последовательности слоев и гиперпараметры.

Выходной слой настраивается в соответствии с рекомендациями из главы 6.

---

[1] Krizhevsky et al., 2012. ImageNet Classification with Deep Convolutional Neural Networks // https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[2] Szegedy et al., 2015. Going Deeper with Convolutions // https://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf.

Конфигурирование сверточных слоев

Нам нужно задать пространственную организацию сверточного слоя, а затем выбрать число фильтров.

От того, как заданы гиперпараметры сверточного слоя, зависит, сколько будет нейронов в выходной области и как они организованы. Перечислим основные гиперпараметры:

- размер фильтра;
- шаг;
- режим дополнения нулями.

Например, код для построения сверточного слоя с фильтрами 5×5 выглядит следующим образом:

```
ConvolutionLayer convLayer = new ConvolutionLayer.Builder()
    .kernelSize(5,5).stride(1,1).padding(2,2)
    .name("first_layer")
    .nOut(out)
    .biasInit(bias)
    .build();
```

В этом примере размер фильтра (`.kernelSize`) задан равным 5×5, а шаг и режим дополнения – соответственно (1,1) и (2,2). Метод `.name()` задает имя сверточного слоя (`"first_layer"`), а метод `.nOut(out)` – количество фильтров. Выход слоя несколько сложнее, чем просто количество блоков. Ниже мы рассмотрим, как вычисляется размер выходной области.

Задание шага применения фильтров

С увеличением шага рецептивные поля выходных нейронов меньше перекрываются, и пространственные размеры выходной области уменьшаются. В примере выше было показано, как задается размер шага:

```
.stride(1,1)
```

Это означает, что при каждом применении фильтр сдвигается на один пиксель вправо, а по достижении конца строки – на один пиксель вниз (подробнее см. главу 4). Если бы размер шага был равен 2, то фильтр сдвигался бы сразу на два пикселя.

Шаг больше 2

На практике шаги, большие 2, встречаются редко, особенно если размер фильтра мал. А шагов, больших размера фильтра, вообще следует избегать.

По мере увеличения размера шага уменьшается размер выходной области. Было показано, что сверточные слои лучше всего работают при малом размере шага (например, 1). Это оставляет заботы об агрегировании пулинговым слоям, позволяя сверточным сосредоточиться на преобразовании входных данных с учетом глубины.

Дополнение нулями

В некоторых компонентах определенных архитектур (например, Inception) мы хотим сохранить пространственные размеры входной области. Для этого применяется дополнение входной области нулями до нужного размера. При этом заодно сохраняется входная информация на границах области.

Выбор числа фильтров

Каждый фильтр ищет во входных данных что-то свое. По мере возрастания сложности и вариативности данных нам требуется больше фильтров для улавливания релевантных признаков. На рис. 7.1 показаны фильтры, обученные на первом сверточном слое сети.

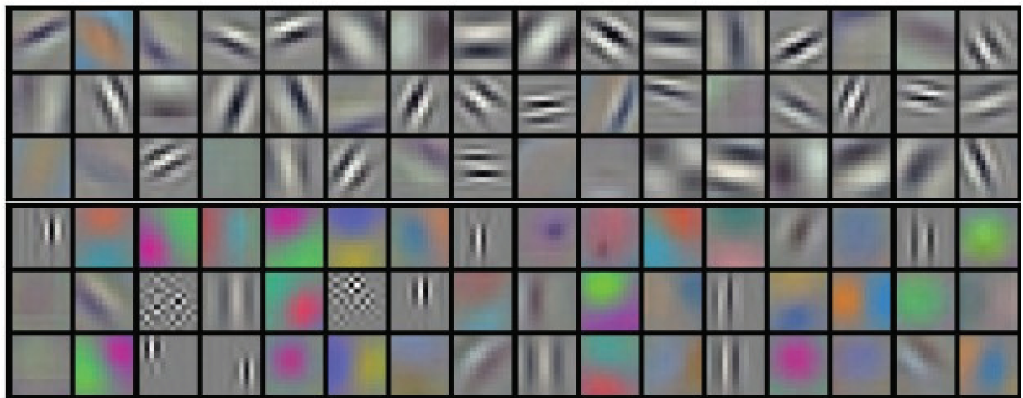


Рис. 7.1 ❖ Визуализация 96 обученных фильтров размера 11 × 11 × 3[3]

При выборе числа фильтров для сверточного слоя следует проявлять благоразумие, потому что вычисление значения активации одного фильтра обходится дороже, чем в традиционных многослойных нейронных сетях. В DL4J количество фильтров задается путем вызова метода `nOut(int)`.

По мере продвижения от ранних слоев СНС к более поздним размер карты активаций уменьшается. В слоях, близких к входному, фильтров обычно меньше. А с приближением к выходному слою их число возрастает.

Никакой специальной эвристики для задания количества фильтров в сверточном слое не существует, но отметим, что в некоторых архитектурах больших СНС начинают с 64 фильтров и заканчивают 1024[4]. В табл. 7.1 приведены счетчики фильтров для трех наиболее распространенных архитектур СНС.

Таблица 7.1. ***Счетчики фильтров в разных сетях***

СНС	Количество фильтров в последовательных слоях
LeNet	20, 50
AlexNet	96, 256, 384, 384, 256
VGGNet	64, 128, 256, 512, 512

О стоимости вычислений и сохранении данных

Хотелось бы, чтобы стоимость вычислений была приблизительно одинакова во всех слоях, т. е. чтобы произведение числа признаков на число пиксельных позиций в каждом слое

---

[3] Krizhevsky, Sutskever and Hinton, 2012. ImageNet Classification with Deep Convolutional Neural Networks // https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[4] http://josephpcohen.com/w/visualizing-cnn-architectures-side-by-side-with-mxnet/.

было примерно одинаково. Это означает, что по мере конструирования новых признаков по входным данным мы постепенно теряем сами входные данные вследствие субдискретизации в пулинговых слоях. С приближением к выходному слою сети информация о входных данных сохраняется, но в другой форме (в виде признаков).

Задание размера фильтра

Небольшие фильтры в сочетании с большим числом сверточных слоев обычно работают лучше[5], чем большие фильтры в сочетании с малым числом слоев[6].

По мере роста размера фильтра растет и стоимость вычислений, поскольку мы должны обсчитывать области большего размера, накрываемые фильтром. Свертка с большими фильтрами (например, 5×5 или 7×7) оказывается несоразмерно дорогой. Например, свертка с *n* фильтрами размера 5×5 на сетки, содержащей *m* фильтров, в 25/9 = 2.78 раза дороже, чем свертка с таким же числом фильтров размера 3×3.

Мы хотим, чтобы размер фильтра был заведомо меньше размера входной области, но все же достаточно велик для улавливания релевантных признаков. В сверточных слоях рекомендуется использовать небольшие фильтры (3×3 или 5×5) с шагом 1[7]. Для сравнения, GoogLeNet (https://arxiv.org/abs/1409.4842)[8] – сравнительно сложная СНС, но в ней применяются фильтры размера 1×1, 3×3 и 5×5.

Еще одна популярная сеть, VGGNet[9], обладает рядом интересных свойств, принесших ей успех. Фильтр в сверточном слое имеет размер 3×3 – минимальный фильтр, еще способный улавливать признаки в нескольких пикселях.

Размер фильтра и число соседних слоев

Три соседних сверточных слоя с фильтрами 3×3 и пулинговый слой аналогичны одному сверточному слою с фильтром 7×7.

Увеличение числа сверточных слоев и фильтров позволяет внести больше регуляризации в процесс обучения. Чем больше слоев, тем больше нелинейных преобразований и тем меньше необходимое число параметров.

Еще один подход к выбору последовательности слоев и формы фильтров состоит в том, чтобы начать с успешной архитектуры, например VGGNet или GoogLeNet, и делать выводы из наблюдаемых по результатам экспериментов паттернов[10].

---

5 Simonyan and Zisserman, 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition // https://arxiv.org/abs/1409.1556.

6 В 2011 году Сиресан (Ciresan) ввел идею небольших фильтров, но тогда сети не были такими большими, как сейчас. Гудфеллоу применил аналогичный подход в 2014 году для распознавания номеров домов на табличках. Этот же подход был применен в сети GoogleLeNet (Сегеди), которая победила в соревновании ImageNet 2014 года.

7 Примером могут служить изображения из набора MNIST размера 28×28, для которых в первом слое сети используются фильтры размера 5×5.

8 Ее называют также «Inception» (точка отсчета) за то, что она задала новый стандарт классификации и распознавания в соревновании ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC 2014).

9 Simonyan and Zisserman, 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition // https://arxiv.org/abs/1409.1556.

10 Размер фильтров в таких сетях, как AlexNet, по современным стандартам великоват. Как обычно, необходимо тестировать и не забывать о том, что мы можем постфактум изменить размер изображений (например, кадрировать или масштабировать), чтобы повысить качество сети.

Предостережение по поводу размера фильтра из статьи о сети Inception v3

Авторы отмечают:

> Приведенные выше результаты позволяют предположить, что свертка с фильтрами размера больше 3 × 3 в общем случае может оказаться не слишком полезной, поскольку ее всегда можно свести к последовательности сверточных слоев с фильтрами 3 × 3[11].

Режим свертки и вычисление пространственного размера выходной области

Приведем общую формулу вычисления пространственного размера выходной области как функции от размера входной области:

```
Размер выходной области = (W - F + 2P)/S + 1.
```

В табл. 7.2 описаны входящие в формулу величины.

Таблица 7.2. Величины, определяющие размер выходной области

Величина	Описание
W	Размер входной области
F	Размер рецептивного поля нейронов сверточного слоя
S	Шаг
P	Дополнение нулями

Результат вычисления по этой формуле должен быть целым числом; если это не так, то конфигурация неправильна. Исправить ее можно, включив дополнение нулями, выбрав другой размер фильтра или воспользовавшись режимом усечения (см. ниже). Следует иметь в виду, что сверточные слои сохраняют пространственный размер входной области, а пулинговые уменьшают его вследствие понижающей передискретизации.

Запоминание входных областей

Если мы используем шаг больше 1 или не дополняем нулями входные данные сверточных слоев, то следует следить за входными областями в сверточной сети. Необходимо, чтобы все шаги и фильтры были уравновешены.

В DL4J производится ряд проверок входных данных, которые обнаруживают многие недопустимые конфигурации. Но чтобы знать, как исправить проблему (или что правильно в каждом слое), мы все равно должны следить за входными областями. Отметим также, что существуют такие конфигурации, для которых никакое дополнение не помогает.

В DL4J перечисление `ConvolutionMode` определяет, как именно должны выполняться операции свертки в сверточных и пулинговых слоях при данном размере входа и конфигурации сети (конкретно величине шага, режиме дополнения нулями и размере фильтра).

В текущей версии поддерживаются три режима (https://github.com/deeplearning4j/deeplearning4j/blob/master/deeplearning4j-nn/src/main/java/org/deeplearning4j/nn/conf/ConvolutionMode.java):

1) строгий (`Strict`);

---

[11] Szegedy et al., 2015. Rethinking the Inception Architecture for Computer Vision // https://arxiv.org/abs/1512.00567v3.

2) усечения (`Truncate`);
3) конгруэнтный (`Same`).

В режиме `Strict` выходной размер по каждому измерению для сверточного и пулингового слоев вычисляется по формуле

outputSize = (inputSize – kernelSize + 2 * padding) / stride + 1.

Если outputSize – не целое число, то на этапе инициализации сети или во время прямого прохода будет возбуждено исключение.

В режиме `Truncate` выходной размер для сверточного и пулингового слоев вычисляется так же, как в режиме `Strict`. Если outputSize – целое число, то режимы `Strict` и `Truncate` совпадают. В противном случае выходной размер округляется с недостатком до целого числа.

Последствия округления

Основное последствие заключается в том, что в случае округления виден граничный эффект, поскольку часть входных данных вдоль того или иного измерения (ширины или высоты) не используется как входы, поэтому некоторые значения активации теряются. Это может сказаться на верхних уровнях сети (если отброшенные активации составляют заметную часть входных данных), особенно когда размер фильтра или величина шага велики.

Режим `Same` отличается от `Strict` и `Truncate` в трех отношениях:

- дополняющие значения не задаются, а вычисляются на основе размера входа, размера фильтра и величины шага;
- размеры выхода вычисляются иначе (см. ниже), чем в режимах `Strict` и `Truncate`. В частности, когда шаг равен 1, размеры входа и выхода совпадают;
- вычисленные дополняющие значения могут различаться для верхний/нижней и левой/правой границ (если они действительно различаются, то на правой или нижней границе может оказаться на одну строку или столбец больше, чем на верхней или левой).

Конфигурирование пулинговых слоев

Функции пулинга заменяют часть выходного слоя в некоторой области некоторым агрегатом находящихся в ней нейронов. Благодаря этому представление модели оказывается в большей степени инвариантным к небольшим сдвигам параллельным переносам входных данных. У самих пулинговых слоев нет параметров, поскольку они вычисляют фиксированную функцию входа. Дополнение нулями в пулинговых слоях обычно не используется.

Инвариантность к небольшим изменениям благодаря пулингу

Инвариантность модели к небольшим изменениям или к локальным параллельным переносам – полезное свойство при обучении на изображениях.

Концептуально пулинг можно рассматривать как бесконечно сильное априорное распределение, когда функция, обученная сверточным слоем, должна быть инвариантна к малым изменениям. С точки зрения статистической эффективности это дает гораздо более сильную сверточную сеть.

Обычно в качестве операции понижающей передискретизации задается max-пулинг с указанием формы (ширина и высота области агрегирования, например 2 × 2):

```
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
```

Следует иметь в виду, что если увеличивать размер области пулинга слишком быстро, то мы рискуем потерять чрезмерно много информации.

На практике в пулинговых слоях встречается max-пулинг двух видов:

- размер рецептивного поля 3, шаг 2 (пулинг с перекрытием);
- размер рецептивного поля 2, шаг 2 (более распространенный вариант).

Если рецептивное поле больше, то теряется слишком много информации.

В последнее время max-пулинг стал популярнее других разновидностей, например пулинга усреднением или по норме L2, поскольку он дает лучшие результаты.

Перенос обучения

Редко бывает, что модель на основе СНС обучается со случайными начальными весами, поскольку для этого требуются большая вычислительная мощность и много времени. В главе 4 мы видели, что одной из причин популярности глубокого обучения стало наличие высококачественных больших наборов данных типа ImageNet (1,2 миллиона помеченных обучающих изображений). Хотя подобные эталонные обучающие наборы данных стали лучше, не так уж часто удается заполучить большой размеченный набор для выполнения крупномасштабного обучения СНС на изображениях. Один из способов преодолеть эту сложность – начать с предварительно обученной СНС, которая дает хорошие результаты, а затем продолжить ее обучение на более специфичном наборе изображений. Такой подход называется переносом обучения (transfer learning)[12].

Альтернатива обучению с чистого листа

Было продемонстрировано, что в первых нескольких слоях СНС обучается общим визуальным признакам, а затем постепенно конструирует на их основе признаки, характерные для конкретного набора данных. Эти ранние признаки, похожие на фильтры Габора и цветовые пятна, являются «строительными блоками» машинного зрения. Отметим два случая, когда перенос обучения особенно полезен:

- точная настройка существующей модели;
- использование существующей сверточной модели для выделения признаков.

В обоих методах используется предобученная модель (как правило, на наборе ImageNet), а различие заключается в последнем этапе настройки.

- *Точная настройка существующей сверточной модели.* В этом случае последний «классифицирующий слой» предобученной СНС заменяется чем-то, специфичным для конкретного набора данных. В одних вариантах продолжается обратное распространение через все слои, а в других обновляются

[12] Yosinski et al., 2014. How transferable are features in deep neural networks? // https://arxiv.org/abs/1411.1792.

только слои, близкие к выходному, поскольку признаки, обученные в более ранних слоях, являются общими для всех типов машинного зрения, так что обновлять их нет особой необходимости. Что же касается более поздних слоев, то они заняты комбинированием низкоуровневых признаков способами, зависящими от задачи, и потому важнее для предметно-ориентированного обучения.

- *Использование существующей сверточной модели для выделения признаков.* В этом случае мы убираем последний полносвязный слой (выходной) и рассматриваем всю остальную СНС как «экстрактор признаков» для меньшего набора данных. Это интересно, потому что набор, на котором производилось предобучение, скажем ImageNet, дает на выходе 1000 различных классов и, скорее всего, не годится для более узкой задачи.

Обучение на наборе ImageNet с чистого листа
Обучение СНС на наборе данных ImageNet может занять от двух до трех недель при использовании нескольких GPU. На практике исследователи часто выкладывают обученные модели в открытый доступ, чтобы их можно было использовать в качестве отправной точки для обучения других моделей. В проекте Caffe тоже имеется модель zoo[13], из которой можно загрузить много предобученных моделей.

Когда имеет смысл рассматривать перенос обучения

Подумать о применении переноса обучения имеет смысл в следующих ситуациях:

- обучающий набор данных мал;
- обучающий набор данных имеет общие визуальные признаки с эталонным набором.

Замечание о скорости обучения при переносе обучения
Если перенос обучения применяется для точной настройки, а набор данных невелик, то мы рекомендуем уменьшить скорость обучения, поскольку предобученные веса, вероятно, уже достаточно хороши.

Рекуррентные нейронные сети

Рекуррентные нейронные сети в чем-то похожи на другие архитектуры, в частности СНС, но у них есть свои проблемы и свои гиперпараметры. Как и для большинства нейронных сетей, не существует точного руководства по мгновенному нахождению наилучших параметров. Нередко при выбранных гиперпараметрах процесс обучения «застревает» или даже ведет себя хуже, чем при предыдущих значениях. Еще раз напоминаем, что настройка сетей всегда производится методом проб и ошибок.

DL4J и сети с долгой краткосрочной памятью
В настоящее время DL4J поддерживает самый популярный вид РНС: модель с долгой краткосрочной памятью (LSTM). Для других вариантов РНС ведется активная разработка.
DL4J поддерживает также двунаправленные LSTM-сети.
Следующим по популярности вариантом РНС являются вентильные рекуррентные блоки (GRU) и «обычные» РНС.

[13] https://github.com/BVLC/caffe/wiki/Model-Zoo.

Входные данные и входной слой сети

Входными данными для канонической нейронной сети прямого распространения традиционно является либо один одномерный вектор, либо двумерная матрица, содержащая мини-пакет обучающих векторов. В рекуррентной нейронной сети имеется третье измерение, соответствующее времени. В DL4J такие входные данные представляются следующими параметрами (рис. 7.2):

- количество примеров;
- размер входа (число столбцов);
- длина временного ряда.

Конструирование объемных входных данных для РНС требует больше усилий, чем для традиционных сетей, таких как многослойные перцептроны. Выше в этой книге мы уже упоминали три измерения входных данных:

- размер мини-пакета;
- количество столбцов вектора на каждом временном шаге;
- количество временных шагов.

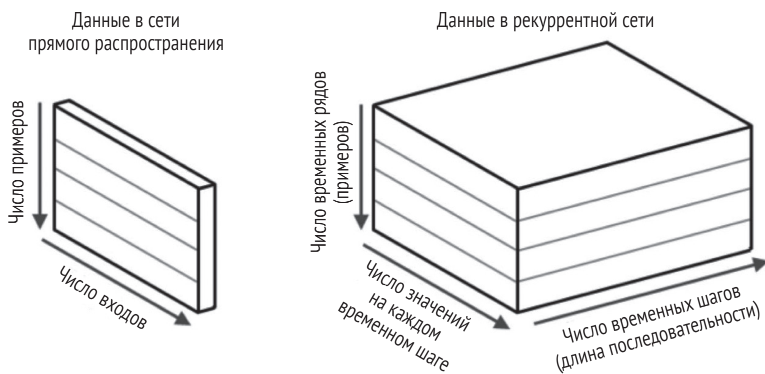


Рис. 7.2 ❖ Представление трехмерного входа рекуррентной нейронной сети

Стандартизация

Вообще говоря, в любой нейронной сети полезно стандартизовать входные данные (привести их к нулевому среднему и единичной дисперсии). Тогда данные оказываются удобнее для применения стандартных функций активации[14].

Благодаря стандартизации связь между входными данными и метками становится максимально простой и локализованной. Но это можно делать только для вещественных входных данных – и ни в коем случае не для категориальных.

Существует несколько подходов к обучению и настройке рекуррентных нейронных сетей.

Выходные слои и RnnOutputLayer

У выходных данных РНС есть три измерения:

- размер мини-пакета (число примеров);

[14] Graves, 2012. Supervised Sequence Labelling with Recurrent Neural Networks (PhD Thesis) // http://www.cs.toronto.edu/~graves/phd.pdf.

- размер выхода (число столбцов);
- длина временного ряда.

Поскольку матрицы в DL4J представляются объектами класса INDArray, каждое значение описывается тремя индексами (i, j, k), перечисленными в табл. 7.3.

Таблица 7.3. Интерпретация переменных во входной матрице РНС

Переменная	Описание
i	Индекс примера в мини-пакете
j	Индекс столбца
k	Индекс временного шага

В DL4J во многих РНС для решения задач регрессии и классификации последний слой имеет тип RnnOutputLayer. Класс RnnOutputLayer выполняет следующие действия:

- вычисление оценки;
- вычисление ошибки (расхождения между предсказанным и фактическим значениями) при заданной функции потерь.

Функционально слой типа RnnOutputLayer похож на стандартный слой OutputLayer канонической сети прямого распространения, но умеет работать с трехмерными выходными данными. Слой RnnOutputLayer конфигурируется так же, как другие слои. Для задач классификации последним слоем в сети MultiLayerNetwork делается RnnOutputLayer, как в примере 7.2.

Пример 7.2 ❖ Конфигурирование слоя RnnOutputLayer

```
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
    .activation(Activation.SOFTMAX)
    .weightInit(WeightInit.XAVIER)
    .nIn(prevLayerSize)
    .nOut(nOut)
    .build())
```

Чтобы лучше понять, как этот слой работает, вернитесь к примеру рекуррентной нейронной сети из главы 5, в котором мы классифицировали показания датчиков.

Обучение сети

Обучение РНС может оказаться вычислительно накладным. В качестве основного алгоритма оптимизации РНС рекомендуется выбирать СГС. Исследования показывают, что неплохие результаты дает также безгессианный метод Ньютона[15].

Инициализация весов

Инициализация весов очень важна для обучения РНС. Установлено, что хорошие начальные веса приводят к созданию скрытых блоков, способных передавать информацию на большие расстояния при моделировании долгосрочных зависимостей[16].

---

[15] Graves, 2012. Supervised Sequence Labelling with Recurrent Neural Networks (PhD Thesis) // http://www.cs.toronto.edu/~graves/phd.pdf.

[16] Sutskever, 2013. Training Recurrent Neural Networks (PhD Thesis) // http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf.

Для большинства слоев РНС мы рекомендуем стратегию инициализации Ксавье[17], как показано в следующем фрагменте:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
        .learningRate(0.1)
        .rmsDecay(0.95)
        .seed(12345)
        .regularization(true)
        .l2(0.001)
        .weightInit(WeightInit.XAVIER)
```

Обратное распространение во времени

Если рекуррентная сеть имеет дело с длинными последовательностями, содержащими много временных шагов, то хорошо работает усеченный алгоритм обратного распространения во времени (BPTT) – обобщение стандартного алгоритма обратного распространения на РНС (см. главу 4). Усеченный BPTT уменьшает вычислительную сложность обновления каждого параметра сети. В примере кода ниже мы видим, как просто задать этот алгоритм в DL4J:

```
.backpropType(BackpropType.TruncatedBPTT)
    .tBPTTForwardLength(100)
    .tBPTTBackwardLength(100)
```

В результате во время обучения будет использоваться усеченный BPTT с заданными длинами на прямом и обратном проходах. Сделаем еще несколько замечаний об использовании BPTT в DL4J:

- по умолчанию в DL4J используется полный BPTT, усеченный алгоритм следует задавать явно;
- длина обычно выбирается из диапазона от 50 до 200 временных шагов (конечно, значение зависит от приложения);
  - количество временных шагов на прямом и обратном проходах обычно задается одинаковым;
  - количество временных шагов на обратном проходе может быть меньше, чем на прямом, но ни в коем случае не больше.

Количество временных шагов в усеченном алгоритме BPTT должно быть меньше или равно длине входных временных рядов.

Регуляризация

Самый распространенный способ регуляризации РНС – прореживание. Оно применяется только к подмножеству связей в LSTM (к нерекуррентным связям). Показано, что такой вариант прореживания действительно полезен[18]. В DL4J прореживание в РНС производится только для входных связей и активаций, но не для рекуррентных связей.

Стандартное прореживание и рекуррентные нейронные сети
Показано, что стандартное прореживание плохо работает в РНС, поскольку рекуррентность усиливает шум, который мешает обучению.

---

[17] Многообещающей является также ортогональная инициализация весов.

[18] Zaremba, Sutskever and Vinyals, 2014. Recurrent Neural Network Regularization // https://arxiv.org/pdf/1409.2329v4.pdf.

РНС более чувствительны к скорости обучения и импульсу, чем многослойные перцептроны[19].

Отладка типичных проблем в LSTM

Хотя настройка нейронных сетей действительно во многом производится методом проб и ошибок, с течением времени мы начинаем замечать некоторые закономерности, позволяющие ускорить процесс. Мы часто видели, как люди пытаются настроить сеть, меняя параметры наугад, но это не лучший путь. Иногда градиенты оказываются чрезмерно велики (такие «взрывные градиенты» отчетливо видны на графике средней абсолютной величины обновлений). Побочным эффектом взрывных градиентов является тот факт, что значения функции потерь возрастают на протяжении нескольких мини-пакетов. Реальная опасность состоит в том, что при большом градиенте мы начинаем вносить большие изменения в параметры, что может негативно отразиться на уже обученных признаках. Чтобы решить эту проблему, применяется усечение или перенормировка градиентов. Взрывные градиенты – типичная проблема в обычных рекуррентных нейронных сетях.

Взрывные и исчезающие градиенты

Известно, что РНС страдают не только от взрывных, но и от «исчезающих» градиентов. Эта проблема возникает, когда градиент становится слишком мал, что затрудняет моделирование долгосрочных зависимостей (10 шагов и более) в структуре входного набора данных.

Самый эффективный способ борьбы с исчезающим градиентом[20] в РНС – использовать LSTM-сеть, поддерживаемую DL4J. Есть и другие способы:

- сочетание коротких и длинных путей в развернутом графе потока (нейронные сети с временным запаздыванием);
- блоки с утечкой и иерархия различных единиц времени;
- GRU;
- применение улучшенных методов оптимизации;
- усечение градиента (для взрывных градиентов);
- регуляризация потока информации;
- штрафы по норме L1 или /L2.

Дополнение и маскирование

Под дополнением понимается добавление нулей в конец временных рядов, которые короче самого длинного временного ряда в мини-пакете. В результате обучающие данные принимают вид прямоугольной матрицы, что необходимо для обучения на временных рядах переменной длины. Под маскированием понимается использование двух дополнительных массивов, которые показывают, какие значения во входных и выходных данных присутствовали изначально, а какие были добавлены в процессе дополнения.

[19] Sutskever, 2013. Training Recurrent Neural Networks (PhD Thesis) // http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf.

[20] LSTM-сети решают проблему исчезающего градиента, но ничем не могут помочь в борьбе с взрывным градиентом.

Дополнение и маскирование позволяют DL4J поддержать следующие варианты обучения рекуррентной сети:

- один ко многим;
- многие к одному;
- временные ряды переменной длины в одном мини-пакете.

На рис. 7.3 приведены примеры каждого варианта.

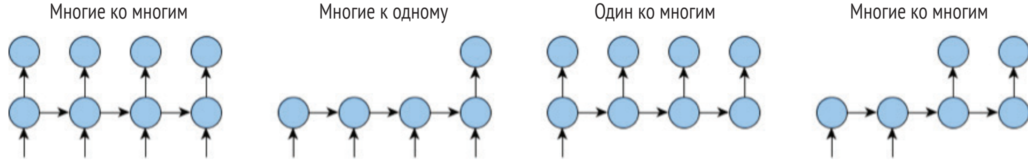


Рис. 7.3 ❖ Различные варианты обучения рекуррентной нейронной сети

Дополнение и маскирование позволяют работать с РНС, в которых не все входы или выходы присутствуют на каждом временном шаге.

Полезность дополнения и маскирования
Если бы не дополнение и маскирование, то можно было бы работать только с вариантом обучения «многие ко многим», когда все входные записи имеют одинаковую длину и любой вход и выход присутствует на всех временных шагах.

Применение дополнения и маскирования к объемным входным данным

При обучении РНС в DL4J задаются следующие параметры, определяющие форму мини-пакета:

- размер мини-пакета;
- размер входа;
- длина временного ряда.

Массив дополнений учитывает следующие параметры для входа и выхода:

- размер мини-пакета;
- длина временного ряда.

Элемент этого двумерного массива равен 1 или 0 в зависимости от того, присутствовал данный временной шаг в исходной последовательности (до ее дополнения) или нет. Массивы масок для входных и выходных данных хранятся отдельно.

Если все элементы массива масок равны 1, то его можно вообще не задавать. В случае «многие к одному» мы имеем единственный массив масок – только для выхода. В DL4J массивы масок создаются на этапе импорта данных (например, итератором `SequenceRecordReaderDatasetIterator`) и представлены в объекте `DataSet`. Объект `MultiLayerNetwork` в DL4J знает, как на этапе обучения работать с массивами масок, если они существуют.

Применение маскирования для оценивания и скоринга

Массивы масок используются в РНС при оценивании верности модели. В случае «многие к одному» мы имеем один выход для каждого входного примера, и при оценивании модели это нужно учитывать.

Классификация и класс Evaluation

Массивы масок выхода при оценивании передаются объекту `Evaluation`, как показано в примере 7.3.

Пример 7.3 ❖ Настройка оценивания

```
Evaluation.evalTimeSeries(INDArray labels, INDArray predicted, INDArray outputMask)
```

В табл. 7.4 описаны аргументы метода `evalTimeSeries`.

Таблица 7.4. ***Описание аргументов***

Аргумент	Описание	Размерность
`labels`	Истинные результаты для обучающих данных	3
`predicted`	Предсказанные сетью результаты	3
`outputMask`	Массив масок выхода	2

Заметим, что массив масок входа для оценивания не нужен, потому что на этом этапе важны только выходы модели.

Оценивание новых данных с помощью MultiLayerNetwork

Массивы масок можно использовать также для скоринга моделей. Скоринг отличается от описанного выше оценивания модели, поскольку мы вычисляем значение функции потерь, а не показатели типа верности или F-меры. Но в обоих случаях мы хотим применять оценивание и скоринг к реально присутствующим, а не дополненным временным шагам. Для скоринга временных рядов применяется класс `MultiLayerNetwork`:

```
MultiLayerNetwork.score(DataSet)
```

Как и в предыдущем примере, если `DataSet` содержит массив масок выхода, то он автоматически применяется при скоринге сети.

Варианты архитектуры рекуррентных сетей

Иногда пользователь хочет скомбинировать слои из разных архитектур для моделирования различных аспектов данных, например в случае видео, когда слои LSTM-сети и СНС комбинируются для моделирования последовательности изображений (видеокадров). Для подобных случаев DL4J предлагает такие препроцессорные классы, как `CnnToRnnPreProcessor` и `FeedForwardToRnnPreprocessor`.

Препроцессоры РНС можно добавить вручную, но во многих случаях сеть добавляет их автоматически.

Ограниченные машины Больцмана

Ограниченные машины Больцмана (ОМБ) – это тип нейронной сети, применяемый для обучения присутствующим в наборе данных признакам без учителя. Они отображают входные данные в скрытое состояние, а затем пытаются реконструировать вход по скрытому состоянию. В этом отношении они похожи на другие модели, обучаемые без учителя, например на автокодировщики (шумоподавляющие,

сжимающие и вариационные), хотя процедура обучения ОМБ (сопоставительное расхождение) совершенно не такая, как для автокодировщиков.

Обычно ОМБ применяются в следующих случаях:

- обучение без учителя признакам в ГСД;
- реконструкция данных;
- рекомендательные системы.

Набор данных MNIST

Рассмотрим, к примеру, обучение на наборе данных MNIST, когда выходной слой состоит из 784 нейронов. Для скрытого слоя мы зададим меньшее число нейронов. Скажем, в первом скрытом слое будет 500 нейронов (примерно две трети от числа нейронов в видимом входном слое), а во втором – 250 нейронов.

Теперь подумаем, как конфигурировать скрытые слои для произвольных задач, а не только для MNIST.

Скрытые блоки и моделирование доступной информации

Это делается не так, как в дискриминантном машинном обучении, когда мы налагаем ограничение: параметров модели должно быть столько, сколько битов необходимо для описания метки.

Информация и метки примеров

Метки содержат лишь несколько бит информации, поэтому объем работы невелик. В некоторых случаях, когда мы используем больше параметров, чем имеется обучающих примеров, построенная модель оказывается сильно переобученной (но это зависит и от регуляризации).

Количество бит во входном векторе может быть на несколько порядков больше, чем нужно для представления метки. В этом случае входной вектор данных содержит гораздо больше информации, доступной для обучения, чем метка.

Пользуясь исходной информацией, которая неявно содержится во входных данных, мы можем более эффективно выделять признаки и (или) более эффективно производить классификацию на последующих этапах конвейера (если сеть используется как часть ГСД). Нас всегда должно интересовать, сколько информации доступно для использования. От этого зависит количество скрытых блоков.

Простой пример

Количество обучающих изображений: 10 000
Количество пикселей в одном изображении: 1000
Предлагаемое количество глобально связанных скрытых блоков: 1000

Как видим, количество скрытых (глобально связанных) блоков в этом примере тесно связано с объемом информации во входном векторе (в данном случае в изображении). В сетях на основе ОМБ, которые не являются глобально связанными и не используют схемы разделения весов, количество блоков может быть больше. Рассмотрим эвристические соображения о выборе количества скрытых блоков в ОМБ.

Выбор числа скрытых блоков

Выбирая число скрытых блоков для ОМБ, мы должны прежде всего думать о том, как избежать переобучения. При таком ограничении имеется эвристическая оценка: умножить число видимых блоков на 0.75. Если целевая разреженность очень мала, то число скрытых блоков можно и увеличить. Для случаев, когда входной набор данных содержит много очень похожих примеров, число параметров можно уменьшить.

Типы блоков

Настройку ОМБ и ГСД мы обычно начинаем с бинарных (или логистических блоков). Если данные плохо моделируются такими блоками, то есть и другие варианты как скрытых, так и видимых блоков ОМБ, а именно:

- мультиномиальные видимые блоки;
- гауссовы видимые блоки;
- биномиальные блоки;
- ReLU.

Мультиномиальные блоки применяются для тематического моделирования, построения рекомендательных систем и классификаторов на основе ОМБ (в последнем случае мы должны добавить выходной слой классификации после предобучения без учителя). В отличие от бинарных видимых блоков, мультиномиальные используют функцию softmax вместо логистической функции, а также скрытые блоки Бернулли.

При распознавании изображений и речи бинарные видимые блоки дают плохое представление признаков. В этом случае предпочтительны гауссовы блоки. Но в скрытых слоях мы хотим оставить бинарные блоки, поскольку гауссовы блоки там и тут приводят к неустойчивому обучению.

В приложениях для распознавания изображений и речи применяются гауссовы блоки, потому что логистические ведут себя плохо. Для таких блоков нужна меньшая скорость обучения, чем для бинарных. В некоторых случаях они демонстрируют неустойчивость в процессе обучения. При работе с разреженными данными рекомендуются биномиальные блоки.

Для непрерывных данных используются блоки ReLU. Количество параметров для ReLU и бинарных блоков примерно одинаково, но ReLU более выразительны. Для лучшей устойчивости во время обучения (https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf) в блоках ReLU рекомендуется задавать меньшую скорость обучения, чем для бинарных. В табл. 7.5 приведены рекомендации по выбору типа блоков для различных входных данных.

Таблица 7.5. Типы блоков ОМБ

Тип данных	Тип видимых блоков	Тип скрытых блоков
Текст	Гауссовы (погружения слов), бинарные (в случае мешка слов)	ReLU (погружения слов), бинарные (в случае мешка слов)
Звуковые / временные ряды	Гауссовы (для непрерывных данных), бинарные (для 0–1)	ReLU (для непрерывных данных), бинарные (для 0–1)
Изображения / видео	Гауссовы (если нулевое среднее и единичная дисперсия), бинарные (для 0–1)	ReLU (если нулевое среднее и единичная дисперсия), бинарные (для 0–1)

Регуляризация в ОМБ

Регуляризация – важная тема машинного обучения, поскольку с ее помощью мы управляем величинами весов. В случае ОМБ основная причина применения регуляризации состоит в том, что иногда на ранних этапах обучениях большие веса и значения ассоциированных скрытых блоков «застревают» в неизменном состоянии. Регуляризация помогает «разблокировать» такие блоки в процессе обучения.

В случае ОМБ нужно задать целевую разреженность для активаций скрытых бинарных блоков, т. е. вероятность того, что бинарный скрытый блок окажется активным. Как правило, этот коэффициент гораздо меньше 1.0.

При обучении ОМБ коэффициент стоимости весов для штрафования больших весов по норме L2 рекомендуется выбирать из диапазона от 0.01 до 0.00001. В ОМБ мы обычно не применяем коэффициент стоимости весов к скрытым и видимым смещениям, потому что их меньше и, соответственно, меньше шансы переобучения. Иногда мы хотели бы, чтобы смещения росли, поэтому применение коэффициента стоимости весов не слишком осмысленно. В случае ОМБ рекомендуется начинать с коэффициента 0.0001. Небольшие различия в стоимости весов, скорее всего, не скажутся на качестве процесса обучения. Показано также[21], что применение прореживания к ОМБ весьма эффективно.

Глубокие сети доверия

Обучение ГСД состоит из двух этапов: предобучение и точная настройка. На этапе предобучения из входных данных выделяются признаки верхнего уровня, чтобы лучше инициализировать веса сети прямого распространения для последующей точной настройки. Для обучения признакам на этапе предобучения в ГСД используются последовательности ОМБ. В главе 4 было подробно описано взаимодействие между ГСД и ОМБ; если вы забыли, прочитайте еще раз этот отрывок.

Обычно мы хотим инициализировать веса ГСД небольшими случайными значениями, имеющими нормальное распределение с нулевым средним и стандартным отклонением порядка 0.01. Если взять большие начальные значения, то поначалу обучение пойдет быстрее, то конечная модель будет хуже (https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf). Значения скрытых смещений обычно можно задавать равными 0.

При обучении нейронных сетей вообще и ГСД в особенности скорость обучения варьируется от 0.001 до 0.1. Если скорость обучения слишком велика, то резко возрастает ошибка реконструкции и веса. В большинстве случаев результатом является плохая модель.

Визуальное задание скорости обучения

В случае ГСД для задания скорости обучения имеет смысл взглянуть на гистограмму обновлений весов и гистограмму весов. Обновления должны отличаться от весов примерно в 10^{-3} раз.

Применение импульса

Благодаря импульсу процесс обучения может при некоторых условиях сдвигать параметры не в направлении самого крутого спуска. Это означает, что будут исследованы соседние области пространства поиска с противоположно направленными градиентами еще до полного поворота назад. Поэтому направление обучения изменяется более гладко без неустойчивых колебаний.

Мы можем настроить импульс для методики сопоставительного расхождения, используемой при обучении ОМБ на этапе предобучения. Рекомендуем начинать со значения 0.5 и увеличивать его примерно до 0.9, после того ошибка реконструкции стабилизируется. Если ошибка реконструкции колеблется слишком сильно,

[21] Srivastava et al., 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting // http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf.

уменьшайте скорость обучения вдвое, пока не будет достигнута стабилизация. По мере того как ошибка реконструкции нарастает или мы приближаемся к условию остановки, импульс будет спадать, стремясь к 0.0.

Применение регуляризации

Говоря о регуляризации ОМБ в предыдущем разделе, мы назвали основные причины ее использования (это относится также к предобучению ГСД).

Предобучение ОМБ без учителя можно рассматривать как форму регуляризации. Оно сводится к наложению ограничений на ту область пространства параметров, которую может исследовать алгоритм. Есть факты, указывающие на то, что предобучение без учителя играет роль зависящего от данных регуляризатора, когда число нейронов или параметров достаточно. Однако в некоторых случаях оно негативно сказывается на способности к обобщению, например когда обучающий набор данных относительно мал (меньше 100 000 примеров).

Разреженность и ГСД

Иногда изучение признаков, которые редко бывают активны, – хороший способ повысить качество модели. Мы можем задать целевую разреженность – вероятность активации двоичных скрытых блоков. Как показывают исследования, для ОМБ целевую разреженность следует выбирать в диапазоне от 0.01 до 0.1. Скорость убывания коэффициента должна быть между 0.9 и 0.99.

Прореживание

Когда для предобучения применяется прореживание[22], рекомендуется задавать небольшую скорость обучения без ограничений на веса, чтобы не потерять детекторы признаков, обнаруженные с помощью предобучения. Прореживание эффективно также на этапе точной настройки ГСД.

MNIST и прореживание

В опубликованном отчете о применении прореживания к набору данных MNIST задание 50-процентного прореживания в скрытых слоях повысило качество модели по сравнению с отсутствием прореживания. 20-процентное прореживание входных блоков принесло дальнейшее улучшение. Соотношение 20:50 часто оказывается оптимальным.

Задание числа скрытых блоков

Основной вопрос при определении числа скрытых блоков ГСД – сколько бит нужно взять для описания каждого входного обучающего вектора, чтобы получить хорошую модель?

В этом контексте главная подстерегающая нас опасность – переобучение, управление которым мы уже обсуждали ранее. Чтобы определить число скрытых блоков в следующем скрытом слое, возьмите их число в предыдущем и сделайте число параметров на порядок меньше.

[22] Hinton et al., 2012. Improving neural networks by preventing co-adaption of feature detectors // https://arxiv.org/pdf/1207.0580v1.pdf.

Глава 8

Векторизация

> Нью-Йорк, Старый Иосиф, Альбукерке, Сантьяго
> Ревет и трясется колымага, считающая себя машиной.
> А если кто спросит, куда подевался этот бродяга,
> Ответьте – он ищет конец линии белой и длинной.
>
> – *Стэрджил Симпсон. «Длинная белая линия»*

Введение в векторизацию в машинном обучении

В этой главе приведены рекомендации по векторизации данных разных типов в машинном обучении. Возникает вопрос, с чего вдруг мы решили писать о векторизации в книге, посвященной глубокому обучению. Главным образом потому, что в книгах по машинному обучению основное внимание уделяется алгоритмам, а на описание полного жизненного цикла добычи данных не остается места. Мы хотим поскорее приступить к экспериментам с данными, а в результате тратим куда больше времени, чем предполагали, на специализированную векторизацию текстовых данных.

В нашей практике работы с корпоративными заказчиками приходилось сталкиваться с ситуациями, когда мы собирались говорить о реализации методов классификации текстов, а в результате скатывались к долгим беседам об основах преобразования текста в векторное представление. В компаниях накопилось много данных в простых форматах, например электронных таблиц, которые можно экспортировать в CSV-файл, но в итоге-то они все равно должны быть преобразованы в векторную форму. Нам также приходится объяснять, что существуют мириады способов векторизовать текстовые данные. В зависимости от располагаемого инструментария и требуемого алгоритма классификации пользователю, даже для того, чтобы только подступиться к векторизации текста, иногда приходится писать кучу кода, не имеющего отношения к статистическому моделированию как таковому.

Исходные данные и автоматизированное обучение признакам

В главе 3 мы говорили, что главная особенность глубокого обучения – переход от ручного конструирования признаков к их автоматизированному обучению. Однако же нам все-таки необходимо предварительно преобразовать исходные данные к виду, с которым могут работать инструменты. Для этого и предназначены методы векторизации данных.

Со временем стало ясно, что без методов векторизации (наряду с самим процессом обработки данных) нет никакой науки о данных, но им почему-то уделяется непропорционально мало внимания. Обдумывая план книги, мы сочли

необходимым дать хорошее представление о векторизации с точки зрения построения моделей глубокого обучения, не отвлекаясь на громоздкие упражнения в сопутствующем программировании. И есть еще одна сторона: при переходе от теории к практике мы хотим, чтобы векторы было легко обрабатывать, чтобы из них сразу получалось много данных для подачи на вход глубоких моделей. Мы хотим, чтобы вы приступили к моделированию данных как можно скорее, для этого и пытаемся навести этот мост.

Обработка канонических наборов данных для машинного обучения – лучший пример того, как неудобно устроен этот процесс. Векторизация может занять от нескольких часов до нескольких дней – в зависимости от вашего опыта и навыков программирования. Это мешает новым пользователям приступить к работе со статическими моделями.

Зачем нужно векторизовать данные?

В машинном обучении и науке о данных требуется анализировать данные самых разных видов. Ключевое требование – умение представить данные в виде вектора чисел (иногда – в виде многомерного массива чисел). В нейронных сетях входные данные тоже должны быть представлены в виде векторов и матриц, поскольку они не умеют работать непосредственно с текстом, графами и другими представлениями.

Есть много подходов к векторизации и много видов предварительной обработки, характеризуемых разной степенью эффективности выходных моделей. Часто трудоемкость моделирования зависит от того, насколько хорошо векторизованы входные данные. Входные данные могут представать в разном виде:

- табличные данные в формате CSV;
- текстовые документы;
- изображения;
- звуковые данные;
- видеоданные;
- последовательные данные.

Ниже приведен пример табличного формата CSV, в котором представлен набор данных об ирисах (https://archive.ics.uci.edu/ml/datasets/Iris) из репозитория Калифорнийского университета в Ирвайне:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
```

А вот пример текстового документа – отрывок из детской книжки «Go, Dog, Go!»:

```
Go, Dogs. Go!
Go on skates
or go by bike.
```

Данные разные, но те и другие нужно как-то привести к векторной форме, пригодной для машинного обучения. Мы хотим, чтобы входные данные алгоритма были представлены в виде разреженного вектора:

```
1.0 1:0.7500000000000001 2:0.4166666666666663 3:0.702127659574468
  4:0.5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.45833333333333326 2:0.3333333333333336 3:0.8085106382978723
  4:0.7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.3333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872
  4:0.5652173913043479
1.0 1:0.916666666666667 2:0.6666666666666667 3:0.7659574468085107
  4:0.5652173913043479
0.0 1:0.08333333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149
  4:0.5217391304347826
0.0 2:0.7500000000000002
```

Процесс преобразования исходных данных в такую форму состоит из двух этапов:

1) векторизация;
2) нормировка.

Мы рассмотрим эти этапы в последующих разделах, а также обсудим векторизацию и нормировку в трех основных разделах глубокого обучения:

- последовательные данные;
- изображения;
- текстовые данные.

Мы выделили именно эти типы данных из-за их тесной связи с глубоким обучением. В этой книге было показано, что рекуррентные нейронные сети – эксперты в обработке последовательных данных, поэтому будет полезно посмотреть, как такие данные векторизуются. Глубокое обучение доказало свою ценность и в области анализа изображений с помощью сверточных нейронных сетей (СНС), отсюда и наш интерес к векторизации изображений. А методы векторизации текста напрямую связаны с их использованием в методе Word2Vec.

Глубокое обучение и изменение природы подготовки данных

Глубокое обучение доказало, что можно не уделять столько времени предварительным шагам конструирования признаков. Мы все-таки должны привести данные к векторной форме, но глубокое обучение отлично справляется с выделением признаков и понижением размерности в процессе обучения структуре, присутствующей в наборе данных.

В прошлом десятилетии многие специалисты по машинному обучению предпочитали выделять из данных признаки вручную, опираясь на знание предметной области, т. е. глубокое понимание того, как данные были сгенерированы или как взаимодействие источников данных с реальным миром порождает определенные следствия.

Ниже в этой главе мы увидим, что набор данных с небольшим количеством атрибутов, возможно, и подходит для ручного выделения признаков, но к крупным наборам данных, состоящим из текстовых документов, изображений или звуковых файлов, необходим алгоритмический подход. В других случаях мы поручаем общему методу векторизации автоматически генерировать векторы в основном потому, что эта задача оказывается трудной из-за сложности исходных данных (большого объема текста). Примерами алгоритмической векторизации могут служить ядерное хэширование, частота терма – обратная частота документа (TF-IDF) и Word2Vec.

Вот на что нужно обратить внимание в первую очередь при построении моделей на основе исходных данных:

- с какого рода данными мы имеем дело?
- какого типа модель мы хотим обучить на этих данных?
- каким будет наш подход к векторизации данных?
  - мы собираемся кодировать признаки вручную или применить алгоритмический подход?
  - работать с исходным текстом трудно и долго, что будем делать?

В зависимости от типа исходных данных в процессе векторизации нужно принимать во внимание разные вещи. Так, для табличных данных хорошо известно, как производить векторизацию столбцов разного типа. Но как быть с векторизацией многомерных временных рядов, находящихся в нескольких файлах? Именно такого рода вопросы решаются при разработке архитектуры конвейера машинного обучения в современных приложениях, и в этой главе мы разберемся, как эти конвейеры реально устроены. Начнем с векторизации табличных данных.

Извлечение, преобразование, загрузка

Во многих средствах бизнес-аналитики (business intelligence – BI) и отчетности, применяемых в современных корпоративных системах, имеется стадия извлечения, преобразования и загрузки данных (Extract, Transform, Load – ETL). В системе Apache Hadoop эта аббревиатура тоже встречается сплошь и рядом. На этой стадии мы объединяем несколько наборов данных, отфильтровываем ненужные столбцы, применяем различные преобразования и загружаем данные туда, где их ожидает найти другое приложение. Большая часть процесса векторизации выполняется на стадии ETL конвейера.

Стратегии обработки табличных исходных данных

Табличные данные – например, экспортированные из таблицы реляционной базы данных – могут быть разной формы и размера, но каждый столбец таблицы считается «атрибутом» данных. Мы будем использовать определения атрибутов, применяемые в науке о данных. В учебниках по статистике обычно определяются четыре атрибута:

- номинальные;
- порядковые;
- интервальные;
- относительные.

Кратко рассмотрим эти типы.

Номинальные

Номинальные атрибуты еще называют *перечислимыми*, *категориальными* или *дискретными* (например, «ясно», «облачно» и «дождь»). Слово «категориальные» означает, что у атрибута конечное число значений, или «категорий». Значениями являются символы, играющие роль меток. Термин «номинальные» происходит от латинского слова, означающего «имя». Номинальные атрибуты никак не связаны друг с другом, и не предполагается, что они как-то упорядочены.

Для номинальных столбцов предлагается использовать унитарное представление. При этом каждое значение представляется вектором, длина которого равна числу потенциальных значений в столбце. Ровно один элемент этого вектора, соответствующий данному значению столбца, будет равен 1.0, а остальные – 0.0. В общем векторе признаков тогда появляются столбцы, соответствующие таким унитарным векторам.

В табл. 8.1 приведены примеры двух записей, первая соответствует значению «солнечно», вторая – «дождь».

Таблица 8.1. ***Пример унитарного представления части вектора признаков***

[Другие столбцы]	Ясно	Облачно	Дождь	[Другие столбцы]
...	1.0	0.0	0.0	...
...	0.0	0.0	1.0	...

Порядковые

Порядковые значения отличаются от номинальных только тем, что для них определено отношение порядка. У порядковых значений есть ранг, определяющий их относительный порядок, но понятие расстояния не определено. Порядковые значения можно сравнивать, но математические операции для них не имеют смысла.

Примерами могут служить «горячо», «тепло», «прохладно» (что считать большим, а что меньшим, не важно, лишь бы принятый порядок применялся единообразно). Еще пример: «низкий», «средний», «высокий». Такие значения преобразуются в целые числа (например, прохладно = 0, тепло = 1 и т. д.), но в коде представляются числами с плавающей точкой.

Различие между порядковыми и номинальными данными не всегда очевидно. В некоторых старых системах добычи данных рассматривались только номинальные и порядковые типы. Если речь идет о выходном векторе, то рекомендуется унитарное представление. Столбцы входных векторов лучше преобразовать в значения на вещественной шкале.

Интервальные

Интервальные значения упорядочены и измеряются в фиксированных единицах. Примером может служить дата или год. Интервальные данные можно сравнивать, но сложение и вычитание для них бессмысленны. Интервальные значения уже являются числовыми, поэтому преобразовывать ничего не надо, но может потребоваться нормировка.

Относительные

Относительные значения измеряются относительно какой-то начальной точки и рассматриваются как вещественные числа. В этом контексте математические

операции имеют смысл. Относительные значения уже числовые, поэтому преобразование не требуется. Но к ним можно применять методы нормировки.

Конструирование признаков и методы нормировки

Векторизация существует не первый день, и специалисты-практики выработали приемы построения канонических «плоских» векторов. Они часто встречаются в конвейерах для таких методов машинного обучения, как логистическая регрессия, случайные леса и т. д.

В классическом варианте мы создаем вектор фиксированной длины n (где $n - 1$ – количество признаков, а в последнем элементе хранится значение метки) и записываем в каждую позицию некоторое значение в соответствии с эвристикой, придуманной для представления данных. Это подходит для таких сетей, как многослойный перцептрон (в которых мини-пакеты *не* используются). Но в глубоком обучении мы часто создаем более сложные многомерные матричные структуры для представления входных данных. В этой главе мы познакомимся с продвинутыми методами создания входных тензоров для рекуррентных нейронных сетей и четырехмерных тензоров для СНС. А в этом разделе займемся конструированием отдельных признаков во входных тензорах и применением к ним нормировки.

Процесс векторизации заключается в выборе атрибутов и нахождении позиции вектора признаков, в которую записать признак. Это делается по-разному в зависимости от того, с чем мы работаем: с CSV-файлом, текстом, изображением или временным рядом. Нам в любом случае необходимо выбрать те части данных, с которыми мы собираемся работать, и поместить их во входной вектор, т. е. выполнить процедуру преобразования n атрибутов в m признаков, которая называется *конструированием признаков* (feature engineering). Обычно исходные данные необходимо преобразовать к виду, пригодному для моделирования. Исходные данные могут представать в разной форме:

- простой текст – документ в текстовом файле;
- файл, в каждой строке которого находится один твит;
- временные ряды в специальном двоичном формате;
- предварительно обработанные наборы данных, содержащие как числовые, так и строковые атрибуты;
- файлы изображений;
- звуковые файлы.

В зависимости от условий и источника данных атрибуты могут записываться в числовом или строковом виде. В большинстве случаев значения атрибутов – числа, целые или вещественные.

Очистка данных и ETL

Очистка данных отнимает немалое время. Обычно это делается на стадии ETL с помощью скриптов или задач Hadoop. В некоторых случаях мы заменяем отсутствующие атрибуты специальными значениями. Но даже если в каждом столбце набора данных все значения присутствуют, это не освобождает нас от обязанности внимательно изучить данные. Неправильные значения могут вкрасться по самым разным причинам. Бывает, что данные собирались в течение длительного времени, и в неко-

торых столбцах присутствуют не относящиеся к делу, потенциально неверные значения. Исследование статистики набора данных и построение графиков и диаграмм – очень полезная практика.

Мы можем применить приемы конструирования признаков для ручной векторизации табличных данных или при работе с более сложными данными, например временными рядами или изображениями. К этим приемам относятся:

- непосредственное использование значений атрибута в неизмененном виде;
- нормировка атрибута для создания признака;
- бинаризация признаков;
- понижение размерности.

Копирование признаков

Для создания векторов из исходных данных необходимо отобрать признаки, наиболее релевантные модели. Отбор признаков долгое время считался ключом к построению успешной модели. Количество признаков в созданном векторе обычно не совпадает с количеством атрибутов в исходных данных. Нередко исходные данные соединяются с другими наборами, а затем для порождения окончательного вектора признаков выбирается подмножество получившегося денормализованного представления.

Самый простой способ порождения признака – копирование атрибута, который уже является числовым и принадлежит правильному диапазону. Увы, такое встречается редко. Чаще приходится применять к атрибуту то или иное преобразование.

Обработка отсутствия значений

В исходных данных часто отсутствуют некоторые значения, и это обозначается специальным значением, не принадлежащим допустимому диапазону (например, –1 там, где могут встречаться только положительные числа, или 0 там, где должны быть ненулевые значения). Отсутствующие значения номинальных атрибутов обычно представляются пробелом или минусом. Значения могут отсутствовать по разным причинам, и, чтобы разобраться в них, нужно хорошо понимать механизм порождения данных источником. Рекомендуется предварительно изучить статистику и свойства набора данных, обращая особое внимание на аномалии. Для этого можно построить графики различных атрибутов и посмотреть, нет ли каких-нибудь странностей, например атрибутов с нулевым значением.

Ниже описаны основные методы обработки отсутствия значений:

- отфильтровать записи, в которых присутствуют не все значения (предостережение: если отсутствие значений подчиняется каким-то закономерностям, то в результате данные могут оказаться смещенными!);
- заменить отсутствующие значения нулями (предостережение: иногда это нормально, а иногда вносит шум в данные);
- подставить вместо отсутствующих значений самое часто встречающееся значение в столбце (есть и другие варианты подстановки).

Нормировка

В процессе нормировки уже преобразованные в векторную форму данные масштабируются и приводятся к определенному диапазону, например [0, 1], [–1, 1] и т. п. Нормировка важна, поскольку она может повлиять на активацию в нейронной сети. Если входы слишком велики, то и значения активации будут большими, что может негативно отразиться на результатах обучения. Наоборот, если входы слишком малы, то значения активации тоже будут малыми, и это повлияет на вычисление градиентов.

Для некоторых гиперпараметров предполагается, что данные нормированы

Некоторые проектные решения, например выбор схемы инициализации весов (скажем, стратегия Ксавье), принимаются в предположении, что входные данные нормированы.

В результате нормировки мы хотим получить более согласованные данные. Для этого применяются два преобразования:

- центрирование;
- масштабирование.

Оба направлены на то, чтобы облегчить процесс обучения. Центрирование означает, что значения признака располагаются вокруг некоторой начальной точки, чаще всего среднего значения. В процессе масштабирования значения признака умножаются на такой коэффициент, чтобы дисперсия всего набора данных стала равной 1 или чтобы максимальное абсолютное значение стало равно 1.

Эти базовые преобразования применяются в следующих методах нормировки:

- стандартизация;
- минимаксное масштабирование;
- обеление;
- метод главных компонент (Principal Component Analysis – PCA).

Под стандартизацией понимается применение центрирования и масштабирования таким образом, чтобы среднее значение данных стало равно 0, а дисперсия – 1. Минимаксное масштабирование считается вариантом базового масштабирования и повышает эффективность алгоритма обучения. Метода главных компонент и обеления мы кратко коснемся ниже.

Недостатки нормировки

Во многих практических наборах данных имеются выбросы. При нормировке регулярные данные масштабируются на меньший интервал, о чем следует помнить.

Краткая справка: среднее и дисперсия

При выполнении преобразований векторов часто используются статистические характеристики признаков, в т. ч. выборочное среднее (или просто «среднее») и выборочная дисперсия (или просто «дисперсия»).

Средним признака (или столбца набора данных) называется среднее арифметическое всех его значений. Среднее N вещественных чисел $x_1, x_2, \ldots x_N$ вычисляется по формуле:

$$\mu = \frac{1}{N}\sum_n x_n.$$

Дисперсия измеряет разброс значений признака относительно среднего:

$$\sigma^2 = \frac{1}{N-1}\sum_n (x_n - \mu)^2.$$

где μ – определенное выше выборочное среднее.

Стандартизация. Чтобы стандартизировать[1] столбец признаков, мы вычитаем из всех значений некоторую величину (минимум, максимум, медиану и т. п.), а затем делим на масштабный коэффициент (дисперсию, стандартное отклонение, длину диапазона и т. п.). Мы хотим, чтобы масштабированные признаки обладали свойствами стандартного нормального распределения:

- $\mu = 0$ (среднее равно 0);
- $\sigma = 1$ (стандартное отклонение равно 1).

Чтобы добиться этого, мы должны сначала вычислить среднее и стандартное отклонение для каждого признака. Затем из каждого признака вычитается его среднее, а результат делится на стандартное отклонение:

$$z = \frac{x-\mu}{\sigma}.$$

Это называется *распределением с нулевым средним и единичной дисперсией*. В результате значения признака оказываются центрированы относительно 0 со стандартным отклонением 1.0 и располагаются в диапазоне [−1, 1].

Приведение к распределению с нулевым средним и единичной дисперсией – самый популярный метод стандартизации. Обычно он применяется к плотным векторам в нейронных сетях.

Стандартизация весьма положительно отражается на таких методах оптимизации, как стохастический градиентный спуск (СГС). Из-за различия в масштабе признаков некоторые параметры могут обучаться быстрее прочих. Стандартизация полезна также для методов, в которых сходство признаков вычисляется на основе какой-то метрики.

Преобразуя атрибуты в признаки, нужно также учитывать разреженность вектора (количество нулей в нем). Если данные разрежены, то признаки рекомендуется нормировать, так чтобы они попадали в интервал (0, 1), т. е. могли интерпретироваться как вероятности. Если же вектор плотный, то лучше привести данные к распределению с нулевым средним и единичной дисперсией, а на последнем шаге произвести масштабирование, так чтобы все значения попали в диапазон [−1, 1].

Основное различие между предварительной обработкой разреженных и плотных данных состоит в том, что для плотных данных мы сначала выполняем приведение к нулевому среднему и единичной дисперсии, а затем производим масштабирование, а для разреженных ограничиваемся масштабированием. Понятно, что в тех случаях, когда алгоритм требует, чтобы данные принадлежали диапазону (0, 1) (пример: ОМБ), мы должны произвести масштабирование на этот диапазон.

[1] В литературе стандартизация иногда называется вычислением *z-оценки*.

Производите стандартизацию с осторожностью

Наивное применение стандартизации к разреженному вектору – не всегда здравая идея. Иногда лучше минимаксное масштабирование (см. следующий раздел). В разреженном случае мы хотим, чтобы после нормировки нуль оставался нулем, а стандартизация этого не гарантирует.

Стандартизация дает более широкий диапазон

Диапазон [−1, 1] дает более широкое представление данных, чем [0, 1], характерный для базовой нормировки, поскольку в представлении с плавающей точкой содержится больше информации.

Минимаксное масштабирование. В случае минимаксного масштабирования мы приводим каждый признак к фиксированному диапазону (часто встречается диапазон [0, 1]). Простейшее минимаксное масштабирование вычисляется по формуле:

$$X_{norm} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}.$$

По сравнению со стандартизацией, мы получаем распределение с меньшим стандартным отклонением. Минимаксное масштабирование более чувствительно к выбросам, поскольку единственный выброс (в большую или меньшую сторону) влияет на максимальное или минимальное значение, а стало быть, и на результат масштабирования. В случае стандартизации эта проблема не так серьезна.

Минимаксное масштабирование часто применяется при обработке изображений, когда яркость пикселей следует привести к определенному диапазону (например, в формате RGB значения принадлежат диапазону [0, 255], а должны быть приведены к диапазону [0.0, 1.0]).

Унификация терминологии

В литературе минимаксное масштабирование иногда называют просто «нормировкой» и считают стандартизацию другим методом. В этой книге оба метода рассматриваются как варианты нормировки.

Обеление и метод главных компонент. В результате статистического обеления[2] (или просто обеления) получаются данные с единичной ковариационной матрицей. В процессе обеления производится также декорреляция данных, что способствует эффективному поиску оптимального их представления. Своим названием обеление обязано тому, что входные данные преобразуются в вектор белого шума. Известная проблема состоит в том, что в некоторых ситуациях обеление увеличивает присутствующий в данных шум.

Метод главных компонент[3] преобразует набор данных (возможно, коррелированных) в набор линейно некоррелированных величин. В нем используется ортогональное преобразование для выделения главных компонент.

---

[2] Kessy, Lewin and Strimmer, 2015. Optimal whitening and decorrelation // https://arxiv.org/abs/1512.00809.

[3] Pearson, 1901. On Lines and Planes of Closest Fit to Systems of Points in Space.

Понижение размерности

Понижение размерности часто применяется в ситуациях, когда требуется алгоритмически найти атрибуты, наиболее важные для модели. Интересно, что в глубоком обучении, а конкретно в глубоких сетях доверия, на этапе предобучения представление данных ищется с помощью обучения слоев ОМБ. Мы также видим, что автокодировщики представляют данные более короткими числовыми векторами. Это сжатое представление можно использовать для классификации или в алгоритмах поиска по сходству в качестве входных данных.

Применение нормировки в РНС и СНС. Если модель предполагает наличие некоторой структуры во входных данных, то эта структура не должна исчезнуть в результате нормировки. При работе с вариантами рекуррентных нейронных сетей, например с LSTM-сетями, при вычислении среднего, стандартного отклонения, максимума и минимума требуется рассматривать все временные шаги.

В контексте СНС среднее, стандартное отклонение и прочие показатели следует вычислить для всех пикселей изображений, обычно отдельно по каждому каналу.

Нормировка в регрессионных моделях. С точки зрения нормировки регрессию следует рассматривать как особый случай. Если в случае классификации мы обычно нормируем признаки (входные данные), то в случае регрессии часто нормируются метки (выходные значения).

К регрессионным моделям применяются те же базовые принципы и методы нормировки, например минимаксное масштабирование и стандартизация.

Почему для регрессии нормируются выходы?

Рассмотрим, к примеру, среднеквадратическую ошибку в качестве функции потерь. Если значения изменяются в диапазоне от 0 до миллиона, то и ошибка, и вычисленные градиенты будут очень велики. Но после нормировки меток (выходных значений) мы будем предсказывать уже не то, что нас в действительности интересует, поэтому требуется ввести поправку на нормировку. Это возможно в случае минимаксного масштабирования и стандартизации, т. к. обе эти операции взаимно однозначны.

Ниже приведены уравнения обратной нормировки:

```
(обратная стандартизация)
origScaleOutput = netOutput * sigma + mu
(обратное минимаксное масштабирование)
origScaleOutput = netOutput * (xMax-xMin) + xMin
```

Бинаризация

Иногда алгоритм моделирования требует, чтобы данные имели многомерное распределение Бернулли, и это надо учитывать при конструировании признаков из атрибутов.

В таком случае можно применить *бинаризацию признаков*, т. е. сравнение числовых признаков с пороговой величиной. Если признак больше порога, то он заменяется на 1, иначе на 0.

Применение библиотеки DataVec для ETL и векторизации

Мы уже сказали, что специалист-практик должен уметь выполнять ETL, векторизацию и нормировку. Нужно думать не только о том, какого рода конвейер машинного обучения построить, но и как его практически реализовать. Учитывая, что большинство операций соединения и ETL требуют одного прохода по набору данных, для ускорения работы следует обеспечить горизонтальную масштабируемость.

Скорость нужна не только при обработке очень больших данных
То, что объем входных данных не исчисляется терабайтами, не означает, что можно пренебречь масштабируемостью этапа ETL в конвейере машинного обучения. Начальство или заказчики часто спрашивают, можно ли получить ответ поскорее, а решением зачастую является горизонтальное масштабирование. Такие инструменты, как Apache Hadoop, Apache Spark, DL4J и DataVec, в том или ином виде предлагают современные средства построения масштабируемых конвейеров.

Библиотека DataVec может работать в локальном режиме, а также допускает горизонтальное масштабирование на Apache Spark и Apache Hadoop. В примерах из главы 5 мы не раз использовали ND4J API напрямую с целью создания объектов `DataSet`, нужных библиотеке DL4J. Приятной особенностью DataVec является тот факт, что она умеет автоматически порождать векторизованные объекты `DataSet` из некоторых стандартных типов данных. В DataVec включены также средства для сбора статистики о векторизованных данных с последующей их нормировкой.

Выше уже было сказано, что в машинном (и глубоком) обучении векторизация чаще всего применяется к табличным данным. В локальном режиме DataVec для этого нужно написать такой код:

```
RecordReader reader = new CSVRecordReader( numLinesToSkip, delimiter );
InputSplit inputSplit = new FileInputSplit( file );
reader.initialize( inputSplit );

// Создать DataSetIterator. Здесь предполагается задача классификации:

int minibatchSize = 10;     // Число примеров в каждом мини-пакете
int labelIndex = 7;         // Индекс столбца метки
int numClasses = 5;         // Число классов (различных меток)
DataSetIterator iterator =
    new RecordReaderDataSetIterator(
        reader, minibatchSize, labelIndex, numClasses );
```

Этот код мы уже видели в главе 5, когда создавали модель многослойного перцептрона с входными данными в формате CSV. DataVec позволяет задавать различные читатели записей, объединять их с итераторами `DataSet` и работать с данными в конвейере моделирования на основе DL4J. DataVec умеет выполнять следующие операции:

- соединение;
- фильтрацию;
- нормировку;
- стандартизацию.

Далее в этой главе мы обсудим другие важные типы данных и объясним, как с ними работает DataVec.

Дополнительные сведения о DataVec см. в приложении F, написанном Алексом Блэком.

Векторизация изображений

Изображения – еще один богатый источник информации. При хранении в компьютере изображение представляется в виде массива пикселей, организованного, как показано в табл. 8.2. Значение пикселя определяет его яркость или цвет.

***Таблица 8.2.** Пиксели изображения, организованные в виде строк и столбцов*

	Столбец-1	**Столбец-2**	**...**	**Столбец-*m***
Строка-1	p11	p21		p-*m*-1
Строка-2	p12	p22		p-*m*-2
...	...	...		...
Строка-*n*	p-1-*n*	p-2-*n*		p-*m*-*n*

Количество бит на пиксель определяет, сколько цветов может представить один пиксель. Однобитовые пиксели способны представить только двуцветные изображения (обычно черно-белые). Восьмибитовые пиксели встречаются чаще и представляют полутоновые изображения с градацией яркости (чем больше значение пикселя, тем он ярче, т. е. ближе к белому). Для представления цветных изображений следует задать отдельно яркости красной, зеленой и синей компонент (в предположении, что используется цветовое пространство RGB), т. е. каждый пиксель на самом деле представляется тройкой чисел. Из распространенных графических форматов отметим JPG, PNG и GIF.

Графические форматы

Графический формат определяет способ хранения данных изображения и степень сжатия. Векторизовать изображения часто бывает проще, чем текст.

Работа с видеоданными

Векторизация видеоданных складывается из векторизации изображений и временных рядов. Видеоданные – это ряд изображений с временными метками. В процессе векторизации мы должны отслеживать изображение во времени. При решении задачи следует определиться с тем, что считать отдельным вектором (один кадр или последовательность соответственных векторов изображения в нескольких кадрах).

Представление изображений в DL4J

В контексте глубокого обучения векторизация изображений интересует нас прежде всего для построения СНС. В DL4J изображение представляется трехмерным тензором, хранящимся в объекте `INDArray` (подробнее см. приложение E). Устройство этого представления показано на рис. 8.1.

Ранее в примерах кода мы видели, что при создании обучающих мини-пакетов изображений на самом деле создаются четырехмерные тензоры, имеющие следующие измерения:

1) размер мини-пакета;
2) глубину;
3) высоту;
4) ширину.

Здесь ширина и высота соответствуют размерам изображения, а глубина равна числу каналов (3 для формата RGB).

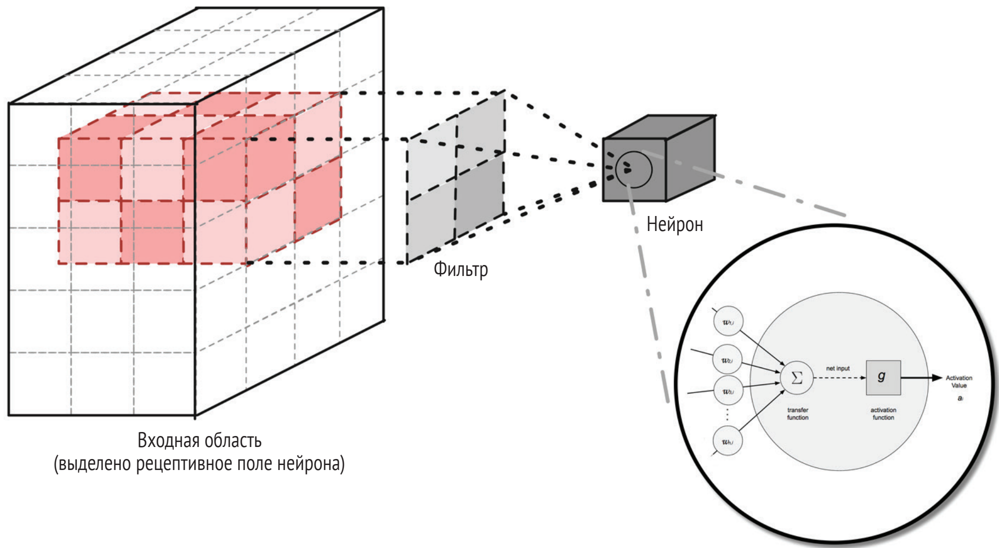


Рис. 8.1 ❖ Трехмерные входные области в сверточных сетях

Четырехмерные мини-пакеты изображений – стандарт в СНС
Четырехмерный формат данных всегда применяется в DL4J при работе с СНС и изображениями.

В случае векторизации данных для многослойного перцептрона их надо сериализовать, оставив всего два измерения (размер мини-пакета, depth × height × width). Чтобы понять, как производится такая векторизация, вернемся к описанному выше представлению в виде сетки и вытянем прямоугольную матрицу $m \times n$ в линейный массив размера $m \times n$, как показано в табл. 8.3.

Таблица 8.3. ***Сериализованные данные изображения***

	Столбец-1	**Столбец-2**	**...**	**Столбец-*m*-×-*n***
Строка-1	пиксель1	пиксель2		пиксель-*m*-×-*n*

Физически изображение состоит из последовательности значений с плавающей точкой, описывающих яркость и оттенок, но логически мы рассматриваем их как матрицу пикселей $M \times N$. Чтобы представить изображение в виде вектора, пригодного для алгоритмов машинного обучения, нужно расположить элементы матрицы в линейном порядке.

Каждое изображение в мини-пакете должно быть преобразовано в массив длины $N \times M$, образующий одну строку матрицы, как показано в табл. 8.4. Это действие выполняет класс `CnnToFeedForwardPreprocessor`, и производится оно автоматически при вызове метода `.setInputType(InputType.convolutional(height,width,depth))` на этапе конфигурирования сети. Метод `.setInputType()` понимает, что мы пытаемся подать 4-мерный тензор плотному слою, который ожидает получить двумерный массив, и пользуется препроцессором для сериализации.

Таблица 8.4. Мини-пакет изображений, представленный в виде двумерного массива

	Столбец-1	Столбец-2	...	Столбец-*m*-×-*n*
Изображение-1	пиксель1	пиксель2		пиксель-*m*-×-*n*
Изображение-2	пиксель1	пиксель2		пиксель-m-×-n
Изображение-3	пиксель1	пиксель2		пиксель-m-×-n

Нормировка данных изображения с помощью DataVec

Самая важная часть векторизации при моделировании изображений – вычленить из графического файла пиксели и представить их в виде вектора, понятного алгоритму обучения. Мы должны читать пиксели один за другим, производить необходимые преобразования или нормировку, а затем помещать результат в соответствующую позицию выходного вектора. Часто мы берем все множество пикселей изображения и преобразуем его непосредственно в вектор, попутно подвергая пиксели некоторым преобразованиям. Но есть и более тонкие методы, когда выделяются части изображения и представляются в виде отдельных векторов.

Класс `ImageRecordReader` в составе DataVec читает данные изображения и выполняет стандартные операции, например кадрирование. Он поддерживает различные графические форматы, в т. ч.:

- JPG;
- GIF;
- PNG;
- TIFF;
- BMP.

Эффективная реализация операций загрузки и преобразования изображений предоставляется библиотеками JavaCV и OpenCV. В приведенном ниже фрагменте кода класс `ImageRecordReader` используется для построения объекта `DataSetIterator`, который позволяет DL4J обучаться непосредственно на данных изображения:

```
ImageRecordReader reader =
    new ImageRecordReader(
        outputHeight, outputWidth, inputNumChannels, labelMaker );
reader.initialize( inputSplit );

// И наконец, создаем DataSetIterator:

int minibatchSize = 10;     // Число примеров в мини-пакете
int labelIndex = 1;         // Для ImageRecordReader всегда 1
int numClasses = 3;         // Число классов (различных меток)

DataSetIterator iterator =
    new RecordReaderDataSetIterator(
        reader, minibatchSize, labelIndex, numClasses );
```

Подобная функциональность упрощает работу с DL4J, поскольку мы можем сосредоточиться на построении конвейера ETL и векторизации, не отвлекаясь на такие мелкие детали, как выделение пикселей из графического формата. Класс `ImageRecordReader` будет подробно рассмотрен в приложении F вместе с методами преобразования, нормировки и стандартизации данных изображения.

Векторизация последовательных данных

Последовательные данные похожи на табличные с тем отличием, что в одном столбце может быть несколько значений. Примером может служить последовательность показаний датчика температуры, установленного в определенном месте, снятых в разные моменты времени. Если добавить к каждому показанию временную метку, то мы получим «временной ряд». Обычно данные во временном ряде изменяются через равные промежутки времени. В мире финансов мы встречаем временные ряды[4] ежедневно обновляемых цен закрытия акций на Нью-Йорской фондовой бирже. Другой пример – показания регистратора вектора параметров в интеллектуальной энергосистеме (https://openpdc.codeplex.com/), который 30 раз в секунду измеряет угол сдвига фаз и напряжение.

Одно из самых распространенных приложений – обработка журналов. Веб-серверы, сотовые телефоны, устройства считывания кредитных карт – все эти системы порождают запись в журнале всякий раз, как выполняют некоторое действие в ответ на запрос. Это бесценный источник последовательных данных и временных рядов. Данные не всегда удобно хранить в традиционной РСУБД, поэтому они хранятся на дисках сетей хранения данных или в файловой системе Apache HDFS. Интернет вещей, о котором сейчас так много толкуют, в значительной мере и состоит из получения и обработки временных рядов от различных датчиков. Есть и другие, не столь очевидные, источники последовательных данных, например геномные последовательности и последовательности обычных символов (предложения).

Многие современные средства машинного обучения плохо поддерживают последовательные данные, так что специалистам-практикам приходится вручную кодировать сложные конвейеры ETL для их обработки. Такие данные могут быть сильно зашумлены, поэтому необходимы нормировка и стандартизация. Далее в этом разделе мы рассмотрим некоторые практические способы ETL и векторизации последовательных данных, предлагаемые библиотекой DataVec.

Основные виды источников последовательных данных

Последовательные данные могут принимать различный вид в зависимости от того, как они были сгенерированы, где и в каком формате хранятся. Вот несколько типичных примеров:

- строки журналов;
- один CSV-файл или временной ряд;
- несколько CSV-файлов или временных рядов.

В каждом случае данные могут быть организованы по-разному. Имеются сущности (например, «пользователь» или иной агент, действия которого мы отслеживаем) и *N* значений в каждом элементе последовательности. Таким образом, для каждой сущности мы имеем один или более столбцов. На диске данные тоже могут храниться по-разному:

[4] Подробнее о временных рядах на практике можно прочитать в статье по адресу http://www.cloudera.com/blog/2011/03/simple-moving-average-secondary-sort-and-mapreduce-part-1/.

- все последовательности в одном файле – каждая строка представляет собой множество разделенных запятыми показаний (несколько столбцов) из одного источника;
- последовательности находятся в нескольких файлах – в каждом файле хранятся показания отдельного источника.

В варианте с одним файлом данные иногда хранятся построчно, когда каждая строка представляет данные из одного источника, генерирующего единственный столбец. В этом случае строка содержит также идентификатор источника.

Если каждый источник на каждом шаге генерирует несколько столбцов временного ряда, то данные часто устроены как набор CSV-файлов, в которых каждая строка соответствует одному временному шагу и содержит один или несколько столбцов, представляющих различные значения, зарегистрированные на этом шаге.

Следует также учитывать, сколько столбцов измеряется на каждом шаге временного ряда. Например, если мы измеряем только напряжение в энергосистеме, то на каждом шаге порождается лишь один столбец. Если же на каждом шаге измеряются напряжение и температура, то необходим формат файла, способный представить два столбца в одном элементе последовательности.

Все эти (и многие другие) факторы определяют процесс ETL, в котором последовательные данные подвергаются векторизации, нормировке и стандартизации. В конечном итоге вся эта информация собирается в один объект (например, `DataSet`), понятный используемым средствам моделирования. DL4J поддерживает обработку последовательных данных посредством DataVec.

Векторизация последовательных данных с помощью DataVec

Мы должны загружать исходные последовательные данные с диска (локального или из распределенной файловой системы типа HDFS). Это подразумевает решение следующих задач:

- разбор формата файла;
- соединение или преобразование данных;
- сопоставление меток (если присутствуют) с признаками для релевантных источников последовательных данных;
- выполнение необходимой нормировки или стандартизации;
- создание окончательного объекта `DataSet` с правильно подготовленными данными.

В идеале формат входного файла уже соответствует какому-то имеющемуся в DataVec (или в Spark, если мы пользуемся этой средой выполнения) читателю записей. В конечном итоге мы хотим построить мини-пакет последовательных данных с одним или несколькими столбцами в тензорном формате, который представлен классом `NDArray` из библиотеки ND4J. Нас интересует трехмерный тензор, структура которого изображена справа на рис. 8.2.

Мы хотим создать трехмерный тензор со следующими свойствами:

- первое измерение – размер мини-пакета;
- второе измерение – столбцы признаков;
- третье измерение – значения столбца на каждом временном шаге.

Вручную выполнить все операции ETL, а затем упаковать значения в объект NDArray довольно сложно. Поэтому мы рекомендуем всюду, где есть возможность, пользоваться классами чтения записей и форматами, уже имеющимися в DataVec.

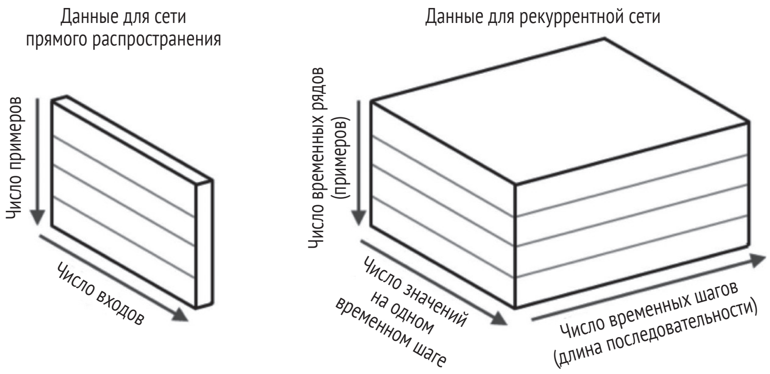


Рис. 8.2 ❖ Представление последовательных данных для рекуррентных нейронных сетей

Пользовательские форматы файлов

Если для некоторого типа входных данных в DataVec нет подходящего читателя, то придется написать код построения тензора самостоятельно. Сложность этой задачи зависит от характера данных.

Рассмотрим сначала некоторые устаревшие варианты векторизации временных рядов, а затем перейдем к приемам преобразования последовательностей в тензоры, встроенным в DataVec.

Преобразование временного ряда в один вектор

В большинстве методов машинного обучения мы стремимся представить каждый пример в виде вектора или строки матрицы. Одна строка представляет один вектор, а группа строк – мини-пакет обучающих векторов, как показано в левой части рис. 8.2.

Основной недостаток этого подхода – утрата временного аспекта данных. До появления рекуррентных нейронных сетей это было не так важно, потому что алгоритмы машинного обучения в любом случае не умели учитывать последовательность значений (или временные метки). Приходилось изобретать экзотические способы создания векторов вручную. Иногда это эффективно, но в принципе такой путь ведет к большому техническому долгу с точки зрения стабильности в конвейере машинного обучения.

Символьное агрегирование

Существует несколько способов представить временные ряды в виде плоских векторов. Один из них относится к зашумленным временным рядам и состоит в применении фильтра нижних частот в качестве шага предварительной обработки. Метод символического агрегирования[5] (Symbolic Aggregated approXimation – SAX), разработанный группой д-ра Имонна Кио (Eamonn Keogh) в Калифорнийском университете в Риверсайде, великолепно устраняет шум. SAX – это символическое представление временного ряда, обладающее уникальными свойствами. Временной ряд T длины n представляется в d-мерном пространстве. По существу, это фильтр нижних частот

[5] http://www.cs.ucr.edu/~eamonn/SAX.htm.

с нижней границей, выраженной в терминах евклидова расстояния. SAX реализует понижение размерности исходного временного ряда, что очень кстати в алгоритмах обучения.

На основе SAX был разработан метод индексируемого символического агрегирования (indexable Symbolic Aggregate approXimation – iSAX), обладающий дополнительными возможностями. iSAX реализует «расширяемое хэширование» и представление с переменным разрешением, благодаря чему обеспечиваются быстрый точный поиск и сверхбыстрый приближенный поиск. Таким образом, iSAX дает большую гибкость, когда нужно представить различные распределения значений, и предлагает интересную схему индексирования временных рядов, полезную во многих приложениях, в т. ч. для быстрого приближенного поиска.

Преобразование последовательных данных в объект DataSet в локальном режиме

При построении РНС (и LSTM-сетей) мы хотим породить трехмерный тензор, представленный объектом `DataSet`. Для иллюстрации рассмотрим случай моделирования данных, в которых последовательность, относящаяся к каждой сущности, находится в отдельном файле и имеется соответствующий файл для меток каждой сущности, например:

- файл последовательных данных для сущности 0: *train/features/0.csv*;
- файл меток для сущности 0: *train/labels/0.csv*.

В данном случае каждое измерение состоит только из одной колонки, поэтому временной ряд является одномерным. Мы можем воспользоваться классом DataVec `CSVSequenceRecordReader` для чтения записей, а затем классом `NumberedFileInputSplit` для обработки схемы именования файлов в указанном каталоге. Класс `SequenceRecordReaderDataSetIterator` порождает объекты `DataSet`, которые пользуются объектами чтения записей и разбиения входных данных на порции:

```
// У нас есть 450 обучающих файлов с признаками:
// от train/features/0.csv до train/features/449.csv

SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
    .getAbsolutePath() + "/%d.csv", 0, 449));

SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
    .getAbsolutePath() + "/%d.csv", 0, 449));

int miniBatchSize = 10;
int numLabelClasses = 6;

DataSetIterator trainData =
    new SequenceRecordReaderDataSetIterator(trainFeatures,
    trainLabels, miniBatchSize, numLabelClasses,
    false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

// Нормировать обучающие данные
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainData);        // Собрать статистику по обучающим данным
trainData.reset();
```

```
// Использовать собранную статистику для нормировки на лету.
// Все объекты DataSet, возвращенные итератором по 'trainData', будут нормированы
trainData.setPreProcessor(normalizer);
```

Здесь вышеупомянутые объекты используются для создания объекта `DataSetIterator`, который порождает мини-пакеты объектов `DataSet`. Эти мини-пакеты содержат правильно подготовленные данные из исходных файлов (и файлов меток).

В библиотеке DataVec есть и другие классы для чтения записей:

- `CSVNLinesSequenceRecordReader` – читает последовательности в формате CSV. Каждая последовательность содержит *N* соседних строк из одного файла;
- `RegexSequenceRecordReader` – предназначен для чтения файлов журналов, разбираемых с помощью регулярных выражений.

Что касается форматов имен файлов, то выше был продемонстрирован класс `NumberedFileInputSplit`, который понимает, как читать пронумерованные файлы в каталоге. Очевидно, что есть много других способов разместить файлы на диске, для обработки которых мы можем создать подклассы класса `FileSplit`.

Далее мы рассмотрим вопрос о создании пользовательских объектов `DataSet` из последовательных данных в случае, когда в DataVec нет подходящего класса. А затем (в главе 9) перейдем к исполнению DataVec на платформе Spark.

Построение пользовательских объектов DataSet из последовательных данных

Для обучения LSTM-сети в DL4J необходимо следующее:

- входной массив NDArray, содержащий обучающие данные;
- массив NDArray, содержащий метки;
- два объекта NDArray, описывающих маски.

Из этих четырех массивов конструируется объект `DataSet`, необходимый DL4J:

```
DataSet d = new DataSet( input, labels, mask_in, mask_labels );
```

Хитрость в том, чтобы поместить данные в нужное место в каждом из массивов NDArray. Проиллюстрируем на примере:

```
// Выделить память: { размер мини-пакета, число столбцов, число временных шагов }
INDArray input  =
    Nd4j.zeros(new int[]{ miniBatchSize, inputColumnCount, maxTimestepLength });
INDArray labels =
    Nd4j.zeros(new int[]{ miniBatchSize, outputColumnCount, maxTimestepLength });
INDArray mask   = Nd4j.zeros(new int[]{ miniBatchSize, maxTimestepLength });

for (int miniBatchIndex = 0; miniBatchIndex < miniBatchSize; miniBatchIndex++) {

    for ( int curTimestep = 0; curTimestep < endTimestep; curTimestep++ ){
    // input -> задать индекс столбца для текущего временного шага
    input.putScalar(new int[]{ miniBatchIndex, columnIndex, curTimestep }, 1.0);

    // Другие столбцы ....

    // Теперь задать маски, помещая 1.0 для тех временных шагов, на которых есть данные
    mask.putScalar(new int[]{ miniBatchIndex, curTimestep }, 1.0);

    // labels -> задать индекс столбца для текущего временного шага
    labels.putScalar(new int[]{ miniBatchIndex, nextValue, timestep }, 1.0);

  }
```

```
}

INDArray mask2 = Nd4j.zeros(new int[]{ miniBatchSize, maxLength });
Nd4j.copy(mask, mask2);
return new DataSet(input,labels, mask, mask2);
```

Очевидно, что к моменту выполнения этого кода ETL и векторизация исходных данных уже должны быть произведены. Сейчас мы просто размещаем данные внутри тензора. Сделаем несколько замечаний относительно этого кода:

- для конструирования входных матриц необходимо использовать библиотеку ND4J;
- два вложенных цикла – типичный способ обойти и инициализировать трехмерную структуру данных;
- внутри цикла мы обращаемся к `INDArray.putScalar()` для задания отдельных значений.

Обратите внимание, как инициализируется матрица меток. Когда предсказывается символ, как в примере с шекспировскими текстами в главе 5, мы помещаем значение следующего символа в матрицу меток. В модели классификации последовательных данных (например, классификации аномалий в журналах) класс следует помещать в матрицу меток на каждом временном шаге. Структурирование данных – в немалой степени искусство, существует несколько стратегий работы с данными в ситуации, когда временные шаги неодинаковы. Решение о том, как производить совмещение временных шагов, мы оставляем на ваше усмотрение.

Маскирование в тензорах

В DL4J маски используются как для обучающих данных, так и для меток.

Маске присваивается значение 1.0 на каждом временном шаге, где имеются обучающие данные, а на всех остальных шагах – 0.0. Обычно для входных данных и меток маска одна и та же.

Обычно такой код встречается, когда мы обходим тензор в цикле и на каждом временном шаге задаем маску. Выглядит это так:

```
INDArray mask = Nd4j.zeros(new int[]{ miniBatchSize, maxLength });
...
for (...) {
    ...
    mask.putScalar(new int[]{ miniBatchIndex, timeStep }, 1.0);
```

Обратите внимание, что в тензоре масок задаются только индекс мини-пакета и индекс временного шага. Задавать, в каких конкретно столбцах имеются данные на этом временном шаге, не нужно.

Векторизация текста

На первый взгляд, работать с текстом очень неудобно, поскольку в документе или в его части может быть любое число слов, а количество слов в разных документах корпуса не совпадает. Если количество «атрибутов» в разных документах различно, то простейший метод конструирования признаков (скопировать значения атрибутов) работать не будет. Нам нужны методы, преобразующие переменное

число атрибутов в постоянное число признаков. Как мы скоро увидим, такие методы есть – и не один. В этом разделе мы рассмотрим несколько применявшихся ранее методов векторизации, а затем сравним их с более современными, например Word2Vec. Начнем с векторной модели текста (Vector Space Model – VSM[6]).

Модель VSM – распространенный способ векторизации текстовых документов. Каждому слову в ней сопоставляется целое число – его номер. Выделяется достаточно большой массив, в каждом элементе которого хранится количество вхождений в документ слова с номером, равным индексу этого элемента. Обычно размер этого массива меньше, чем словарь корпуса документов, так что налицо стратегия векторизации.

В моделировании текста можно выделить несколько этапов.

1. Сегментация предложений. В зависимости от ситуации можно перейти сразу к разбиению на лексемы.
2. Разбиение на лексемы. На этом этапе выделяются отдельные слова.
3. Стемминг. Нахождение основы слов (факультативно).
4. Лемматизация[7]. Объединение различных грамматических форм леммы (факультативно).
5. Удаление стоп-слов (факультативно).
6. Векторизация. Результат процесса преобразуется в массив чисел с плавающей точкой.

В VSM предполагается, что слова – это ортогональные измерения, как если бы это были независимые координаты x и y. Правда, в случае текста это не совсем так, потому что для слов можно говорить о совместной встречаемости. Взять, к примеру, название команды «Волонтеры из Теннесси». Вероятность встретить слова «волонтеры» и «Теннесси» вместе выше, чем в случае, если бы эти слова были действительно независимыми. Для векторизации текста есть разные стратегии, и в следующих разделах мы обсудим некоторые из них – от простых к более сложным.

Мешок слов

Под мешком слов понимается список слов вместе со счетчиками вхождения. Это простейшая векторная модель, но количество столбцов в ней может оказаться очень большим, поскольку самих слов много. Обычно для упрощения алгоритма мы нормируем счетчики слов в документе, т. е. вектор, представляющий документ, содержит вероятности вхождения слов. Модель мешка слов дает упрощенное представление документа. Группа слов, или документ, представляется мешком, или мультимножеством входящих слов. Грамматика и порядок слов игнорируются, но количество вхождений каждого слова запоминается. Эта техника векторизации часто применяется в обработке естественного языка (ОЕЯ), классификации документов и в информационном поиске[8].

---

[6] https://nlp.stanford.edu/IR-book/html/htmledition/the-vector-space-model-for-scoring-1.html.

[7] В некоторых языках стемминг является облегченной формой лемматизации.

[8] Одно из первых упоминаний модели мешка слов встречается в статье Зеллига Харриса 1954 года о «дистрибутивной структуре». Эта модель также была предложена Салтоном и Макгиллом в 1983 году под названием «представление документа без сохранения порядка в виде частот слов из словаря».

В табл. 8.5 представлены векторы различных слов в документах. Если слову «apple» соответствует индекс 0, то всякий раз, как в документе встречается это слово, мы увеличиваем значение в элементе вектора с индексом 0. То есть имеет место упражнение на подсчет различных слов. В результате документ представляется вектором признаков, основанным на частоте вхождения в него слов. В такой форме количество вхождений сохраняется, но информация о порядке слов теряется. Это представление эквивалентно гистограмме распределения частот слов.

Таблица 8.5. Модель векторного пространства

	T_1	T_2	...	T_t
D_1	w_{11}	w_{21}	...	w_{t1}
D_2	w_{12}	w_{22}	...	w_{t2}
...	...	...		...
D_n	w_{1n}	w_{2n}	...	w_{tn}

Модель мешка слов часто называют вектором *частот термов*. На основе частот термов можно построить и более изощренные схемы векторизации, например TF-IDF. В этой схеме частота слова в документе умножается на обратную частоту его встречаемости во всем корпусе документов. В других вариантах модели мешка слов элементами вектора могут быть только 0 и 1, показывающие, встречается слово в документе или нет. Для построения модели мешка слов требуется несколько проходов по набору данных, что в случае больших наборов может оказаться накладно.

Недостатком модели мешка слов является и то, что она не улавливает фразы и выражения из нескольких слов. Кроме того, без специальной предобработки не учитывается неправильное написание слов, а также варианты написания одного слова, которые мы хотели бы считать одинаковыми. Оба недостатка устраняют N-граммы и другие методы предварительной обработки[9].

TF-IDF

Метод TF-IDF[10] исправляет некоторые проблемы, свойственные модели мешка слов. В разных документах одни и те же слова встречаются с разной частотой. Величина TF-IDF корректирует частоту слова с учетом редкости.

TF-IDF в числовом виде оценивает важность встречающегося в документе слова относительно всей коллекции документов. Зачастую TF-IDF выступает в роли весового коэффициента в информационном поиске и анализе текстов. Значение TF-IDF возрастает пропорционально числу вхождений слова в документ, но в то же время снижается пропорционально частоте встречаемости слова во всем корпусе. Поэтому TF-IDF правильно учитывает слова, которые в каком-то одном документе кажутся важными, но на самом деле являются часто употребляемыми, так что не могут служить значимыми маркерами.

Полезность TF-IDF связана с тем, что он учитывает тот факт, что некоторые слова встречаются чаще других. Величина TF-IDF равна произведению частоты

---

[9] Еще один вариант векторизации на основе модели мешка слов – хэширование, позволяющее отобразить каждое слово на индекс в векторе фиксированного размера.

[10] https://nlp.stanford.edu/IR-book/html/htmledition/tf-idf-weighting-1.html.

терма (term frequency – TF) на обратную частоту документа (inverse document frequency – IDF). Ее вычисление начинается с вычисления частоты терма.

Веса термов

В поисковых системах TF-IDF часто используется в качестве механизма оценивания и ранжирования документов по релевантности запросу пользователя. Эффективно также применение в качестве фильтра стоп-слов. Для стоп-слов вес TF-IDF мал, а термам, которые редко встречаются во всем корпусе, назначается большой вес.

У таких важных слов обычно велико значение TF и IDF, поэтому произведение того и другого оказывается большим. Слова, у которых TF-IDF относительно велико, можно считать определяющими тему документа.

Помимо поисковых систем, показатель TF-IDF очень полезен для целей автоматического реферирования и классификации текстов. Процесс векторизации точнее, чем в модели мешка слов, но требует больше вычислительных ресурсов, поскольку мы должны подсчитывать частоту вхождения слов не только в одном документе, но и во всем корпусе. Это требует более одного прохода по набору данных, а также нескольких этапов предварительной обработки.

TF

В простейшей форме TF определяется как число вхождений слова в документ. В более сложных вариантах счетчики нормируются на длину документа. Длина документов обычно различается; чтобы учесть это, мы делим частоту терма на длину документа (общее число термов в нем), что дает более верную меру. Если обозначить t – терм, а d – документ, то

$$tf_{t,d} = \frac{count(t)}{count(alltermsind)}.$$

IDF

Следующий шаг – определить, сколько информации несет слово, для чего следует вычислить его IDF[11]. Мы хотим изменить, часто или редко встречается слово, и включить эту информацию в вектор признаков. Интуиция подсказывает, что больший вес следует приписать термам, которые встречаются в немногих документах корпуса[12].

Чтобы определить «обратную частоту документа», мы сначала делим общее число документов в корпусе на число документов, содержащих слово. Но в такой форме значение IDF не идеально, потому что маскирует влияние TF на окончательный вес терма. Чтобы сгладить эту проблему, обычно берут логарифм этой величины (в табл. 8.6 приведены определения переменных в формуле).

$$idf_t = \log\left(\frac{N}{df_t}\right).$$

[11] https://nlp.stanford.edu/IR-book/html/htmledition/inverse-document-frequency-1.html.

[12] Понятие IDF было введено в 1972 году Кареном Шпарком под названием «специфичность терма».

Таблица 8.6. Члены IDF

Переменная	Описание
df_t	Частота документа – количество документов, содержащих терм t
N	Общее число документов в корпусе

Вычисление полной оценки TF-IDF

Таким образом, вес TF-IDF слова принимает вид:

$$tfidf_{t,d} = tf_{t,d} \times idf_t.$$

Большой вес будет у слов, имеющих относительно высокую частоту TF и относительно низкую частоту встречаемости во всем корпусе документов. Часто встречающиеся термы отфильтровываются, потому что у них низкая TF и высокая частота встречаемости во всех документах. Легко видеть, что значение IDF всегда больше или равно 0, поскольку аргумент логарифма больше или равен 1. Понятно также, что если терм встречается во многих документах, то отношение под знаком логарифма близко к 1. Поэтому IDF оказывается близко к 0, а значит, и TF-IDF близко к 0. На практике оценка TF-IDF более эффективна, чем мешок слов, но ее можно еще улучшить за счет применения *N-грамм*.

В DL4J реализация[13] TF-IDF входит в состав средств обработки текста. Ниже приведен простой пример ее использования в программе:

```
File rootDir = new ClassPathResource("tripledir").getFile();
LabelAwareSentenceIterator iter = new LabelAwareFileSentenceIterator(rootDir);
TokenizerFactory tokenizerFactory = new DefaultTokenizerFactory();

TfidfVectorizer vectorizer = new TfidfVectorizer.Builder()
        .setMinWordFrequency(1)
        .setStopWords(new ArrayList<String>())
        .setTokenizerFactory(tokenizerFactory)
        .setIterator(iter)
        .build();

vectorizer.fit();
```

Удаление стоп-слов

Типичный шаг предварительной обработки при вычислении TF-IDF – удаление стоп-слов. Поскольку стоп-слова встречаются очень часто, они могут существенно повлиять на веса TF-IDF. Поэтому для достижения более верной оценки сходства между двумя векторами документов стоп-слова удаляются. Примерами стоп-слов в английском языке могут служить такие артикли и союзы:

- a;
- an;
- who;
- the;
- what.

---

[13] В проекте Apache Lucene также имеется высококачественная реализация TF-IDF, используемая во многих программах на базе JVM (https://github.com/apache/lucene-solr/blob/master/lucene/core/src/java/org/apache/lucene/search/similarities/TFIDFSimilarity.java).

N-граммы

N-граммой называется последовательность *N* соседних элементов текста. С помощью *N*-грамм в процессе векторизации можно учесть, как часто некоторые группы слов встречаются совместно. В базовой реализации TF-IDF информация о соседстве слов не принимается во внимание, поэтому теряется важная информация, содержащаяся, например, в названиях тип «Wall Street» или «Coca Cola».

N-граммы применяются, когда имеется последовательность элементов, например слов, и мы хотим предсказать следующее слово. Зная, что некоторые слова с большей вероятностью встречаются в определенных контекстах, мы можем воспользоваться этой информацией. *N*-граммы как раз и дают возможность смоделировать контекст в виде соседних слов. *N*-граммы применяются в ОЕЯ и в распознавании речи. В последнем случае слова моделируются как последовательности *N*-грамм. В задаче идентификации языка текста в роли *N*-грамм выступают последовательности символов (или графем – «букв алфавита»). При разборе текстового документа *N*-граммами считаются последовательности *N* соседних слов.

Помните, что если *N* слишком велико, то алгоритмы на основе *N*-грамм становятся вычислительно накладными. Объем словаря очень сильно возрастает даже при $N = 5$, так что старайтесь выбирать значение поменьше.

Ядерное хэширование

Для векторизации текста применяется также метод ядерного хэширования. Создание признаков в процессе векторизации – дело непростое, и мы рады воспользоваться любой возможностью его облегчить. Несколько проходов по данным, как при вычислении TF-IDF, вызывает трудности, когда данных много. Ядерное хэширование применяется, чтобы векторизовать данные за один проход – «своевременно». Правда, при этом возможно возникновение коллизий между словами. Этот метод имеет смысл использовать для векторизации текста непосредственно перед подачей на вход алгоритма обучения.

Сравнение Word2Vec и векторной модели

В главе 5 мы говорили, что Word2Vec определяет сходство слов, обучаясь контексту, в котором слова встречаются. Word2Vec создает векторы, являющиеся распределенными числовыми представлениями признаков-слов (контекста), а точнее – список погружений слов (векторов), в котором каждое слово представлено одним вектором.

Благодаря Word2Vec мы можем уловить семантику слов, выявив положение слова в большом количестве предложений и связав это положение со значением слова. Это позволяет запомнить во внутреннем представлении слова в нейронной сети, созданной Word2Vec, скрытую информацию, присутствующую во входных данных.

Векторная арифметика в Word2Vec

Семантика и связи слова пространственно кодируются в порождаемых погружениях слов. Побочным эффектом этого являются такие полезные свойства, как возможность выполнять осмысленные арифметические операции над векторами.

Напротив, модели векторного пространства (например, TF-IDF) строятся только на основе статистики совместной встречаемости слов во входных документах.

Это открывает интересные возможности, например сравнение двух документов по сходству, но не позволяет представить семантические или синтаксические связи между словами в документе.

Векторное представление методом Word2Vec открыло новые подходы к пониманию исходного текста[14], построению рекомендательных систем[15] и моделированию графов[16]. Примеры кода см. в главе 5.

Работа с графами

Векторизация графовых структур – трудная проблема науки о данных, относящаяся к конструированию конвейеров обработки. Мы не можем рассмотреть ее во всей полноте, но обсудим систему Node2Vec (см. также главу 5).

Node2Vec[17] позволяет сгенерировать вектор для каждой вершины графа. Эти векторы можно использовать так же, как векторы слов (см. предыдущий раздел), для решения таких задач, как классификация вершин и выведение существования ребра между двумя вершинами.

Node2Vec является обобщением другого метода работы с графами – DeepWalk[18], в котором производится случайное блуждание по графу с целью сбора локальной информации о его структуре. Это позволяет DeepWalk обучаться латентным представлениям, присутствующим в графе, рассматривая маршруты блуждания как эквиваленты предложений. Наконец, отметим работу[19], посвященную графовым СНС (https://tkipf.github.io/graph-convolutional-networks/). Об этом стоит знать, хотя в настоящее время DL4J таких СНС не поддерживает.

Матрицы смежности

Векторизация графов, представленных в виде матрицы смежности, – метод, который часто встречается в машинном обучении[20]. Идея в том, чтобы представить конечный граф квадратной матрицей, строки и столбцы которой соответствуют вершинам графа. На пересечении i-й строки и j-го столбца находится 1, если вершины i и j соединены ребром, в противном случае – 0. Главная диагональ матрицы смежности заполнена нулями, потому что никакая вершина не может быть соединена ребром сама с собой. В этой книге мы больше рекомендуем методы, способные оперировать структурой графа непосредственно, а не различные варианты построения матрицы смежности.

---

[14] Nay, 2016. Gov2Vec: Learning Distributed Representations of Institutions and Their Legal Text // http://aclweb.org/anthology/W16-5607.

[15] Barkan and Koenigstein, 2016. Item2Vec: Neural Item Embedding for Collaborative Filtering // https://arxiv.org/abs/1603.04259.

[16] Grover and Leskovec, 2016. node2vec: Scalable Feature Learning for Networks // https://arxiv.org/abs/1607.00653.

[17] Там же.

[18] Perozzi, Al-Rfou and Skiena, 2014. DeepWalk: Online Learning of Social Representations // https://arxiv.org/abs/1403.6652.

[19] Niepert, Ahmed and Kutzkov, 2016. Learning Convolutional Neural Networks for Graphs // https://arxiv.org/abs/1605.05273.

[20] Для векторизации графов также используется структура данных «список смежности». См.: Introduction to Algorithms (Second ed.), MIT Press and McGraw-Hill, раздел 22.1 «Представления графов».

Глава 9

Глубокое обучение и DL4J на платформе Spark

Ten years on the road, making one night stand
Speeding my young life away
Tell me one more time just so I'll understand
Are you sure Hank done it this way?
Did old Hank really do it this way?

– Уэйлон Дженнингс.
«Are You Sure Hank Done It This Way»

Введение в использование DL4J совместно с Spark и Hadoop

За минувшие десять лет появились две ключевые технологии, используемые в центрах обработки данных: Apache Hadoop и Apache Spark. Именно Hadoop стал эпицентром роста и развития хранилищ данных. Spark добился того, что технология MapReduce стала основным средством выполнения параллельных итеративных алгоритмов поверх Hadoop.

DL4J поддерживает горизонтальное масштабирование обучения сети на платформе Spark, что позволяет значительно сократить время обучения, а также противостоять линейному росту времени обучения с увеличением объема входных данных.

В облако!
Такие платформы, как Amazon Web Services (AWS), Google Cloud и Microsoft Azure, позволяют создать и настроить кластер Spark по запросу всего за несколько долларов. DL4J работает в большинстве публичных облаков[1], так что практики могут гибко решать, где и как реализовать процесс глубокого обучения.

Spark[2] – это универсальный движок параллельной обработки, который может работать самостоятельно в кластере Apache Mesos[3] или в кластере Hadoop с при-

---

1 В настоящее время вариант Spark, предлагаемый компанией DataBrick, не поддерживается, но в будущем это может измениться.

2 http://spark.apache.org/.

3 http://mesos.apache.org/.

менением каркаса Hadoop YARN (Yet Another Resource Negotiator). Он умеет работать с данными в распределенной файловой системе Hadoop (Hadoop Distributed File System – HDFS), пользуясь форматами, включенными в состав Hadoop. В Spark применяется кэширование часто используемых данных в памяти с помощью надежных распределенных наборов данных (Resilient Distributed Dataset – RDD). Spark также позволяет программисту абстрагироваться от особенностей параллельной обработки и сосредоточиться на алгоритме. В этой книге нас будут интересовать те аспекты пакетной обработки в Spark, которые находят применение в параллельных итеративных алгоритмах, в т. ч. стохастического градиентного спуска (СГС).

Далее представлены основные компоненты задачи Spark.

- Приложение. Собираемый нами JAR-файл задачи Spark представляет приложение Spark. Это может быть одна задача, несколько сцепленных задач или интерактивный сеанс.
- Драйвер Spark. Организует контекст Spark и преобразует приложение в ориентированный граф заданий, для которых планируется выполнение в кластере. С каждым приложением Spark связан только один драйвер.
- Мастер приложения Spark. При выполнении Spark поверх связки Hadoop+YARN мастер приложения согласовывает с YARN выделение ресурсов кластера. В каждом приложении Spark есть только один мастер.
- Исполнитель Spark. Исполнитель выполняет несколько заданий в одном экземпляре виртуальной машины Java (JVM), работающем на одном локальном хосте с исполнителем. Драйвер Spark инструктирует постоянно работающих исполнителей о том, какие задания они должны выполнить. На одном хосте может работать несколько исполнителей. В кластере могут быть сотни, а то и тысячи исполнителей, распределенных между узлами. Все они одновременно выполняют различные приложения Spark.
- Задание Spark. Представляет единицу работы исполнителя над частью распределенного набора данных (RDD).
- RDD. Отказоустойчивая коллекция элементов, которые могут обрабатываться параллельно.

RDD

RDD – основное понятие в Apache Spark. Это отказоустойчивый набор элементов, допускающий параллельную обработку. Операции над RDD пишутся на языке высокого уровня, а затем компилируются, так что программист может сосредоточиться на алгоритме и стоящей перед ним задаче, не отвлекаясь на управление распределенной системой.

Существует два способа создания RDD в программе на базе Spark:

- распараллелить существующую в приложении коллекцию;
- сослаться на набор данных во внешней системе хранения.

Поддерживаются такие внешние системы хранения, как HDFS, HBase (https://hbase.apache.org/), Cassandra (http://cassandra.apache.org/), и другие, совместимые с форматом Hadoop `InputFormat`. На практике чаще всего используется HDFS.

Apache Hadoop – это написанный на Java набор средств параллельной обработки (в частности, MapReduce) и распределенная файловая система HDFS. Он ос-

нован на опубликованном проекте технологии Google MapReduce[4] и Google File System (GFS). Изначально Hadoop проектировался для параллельного построения инвертированного индекса для поисковой системы Apache Lucene[5] в рамках проекта Apache Nutch[6]. У истоков проекта Hadoop стояли Дуг Хантинг и Майк Кафарелла. В конечном итоге он привел к демократизации инфраструктуры поисковых систем, а потом и к революции в технологии построения хранилищ данных. В январе 2008 года Hadoop отпочковался от проекта Nutch и стал отдельным проектом Apache верхнего уровня.

Затем проект был взят под крыло компанией Yahoo!, которая внедрила эту технологию в собственные разработки и выделила для нее остро необходимые инженерные ресурсы. В апреле 2008-го компания объявила о новом достижении: терабайтный массив данных был отсортирован на кластере из 910 узлов Hadoop за 209 секунд[7]. К 2009 году многие компании, в т. ч. last.fm, Facebook, *New York Times* и даже Управление ресурсами бассейна реки Теннесси[8], увидели в Hadoop практический способ использования недорогого стандартного оборудования для распараллеливания обработки больших массивов данных[9]. В настоящее время Hadoop используется многими компаниями из списка Fortune 500, и не только. Apache Hadoop стал современной инфраструктурой хранилищ данных и стандартом де-факто для корпоративных сред выполнения.

DL4J, Spark и Hadoop

Библиотека DL4J с самого начала проектировалась как полноправный гражданин мира Hadoop. Она может работать на автономном узле Spark, на Spark в кластере Mesos или на Spark в кластере Hadoop в связке с YARN. Для экспериментов с DL4J имеет смысл использовать автономный узел Spark, а при эксплуатации на большом предприятии лучше организовать кластер – Mesos или Hadoop+YARN.

Запуск Spark из командной строки

Обычно задачи Spark запускаются из командной строки с различными параметрами, которые описаны ниже.

spark-submit

Задачи запускаются в кластере с помощью bash-скрипта `spark-submit`. Вот как выглядит командная строка для запуска задач в кластере Hadoop (CDH[10], HDP[11]):

```
spark-submit --class [class name] --master yarn [jar name]
    [job options]
```

---

[4] Dean and Ghemawat, 2008. MapReduce: Simplified Data Processing on Large Clusters // https://static.googleusercontent.com/media/research.google.com//en/archive/mapreduce-osdi04.pdf.

[5] https://lucene.apache.org/.

[6] https://nutch.apache.org/.

[7] http://sortbenchmark.org/YahooHadoop.pdf.

[8] См.: http://cnet.co/2uytyct и http://bit.ly/2uSutEa.

[9] https://gigaom.com/2009/11/10/the-google-android-of-the-smart-grid-openpdc/.

[10] https://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_running_spark_apps.html.

[11] https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.5.3/bk_spark-component-guide/content/ch_tuning-spark.html.

где `class name` – полное имя класса; `jar name` – полный путь к JAR-файлу. Это файл, который часто называет «uber jar», содержит все необходимое для выполнения задачи; `job options` – различные параметры задачи Spark.

Например:

```
spark-submit --class io.skymind.spark.SparkJob --master yarn
    /tmp/Skymind-SNAPSHOT.jar /user/skymind/data/iris/iris.txt
```

Здесь мы выполняем класс `io.skymind.spark.SparkJob`, находящийся в JAR-файле `Skymind-SNAPSHOT.jar`. У задачи всего один параметр, определяющий местонахождение входного файла.

Иногда требуется изменить некоторые конфигурационные свойства задачи на этапе ее выполнения. Для этого можно либо включить дополнительные флаги в командную строку, либо задать их в специальном конфигурационном файле, как показано ниже:

```
spark.master     spark://mysparkmaster.skymind.com:7077
spark.eventLog.enabled    true
spark.eventLog.dir        hdfs:///user/spark/eventlog
# Задать объем памяти для исполнителя spark
spark.executor.memory     2g
spark.logConf             true
```

Удобно записывать конфигурационные параметры в такой текстовый файл, чтобы было проще управлять задачей. Обычно этот файл помещают в один каталог с JAR-файлом задачи и ссылаются на него из командной строки.

Безопасность в Hadoop и Kerberos

Kerberos – стандартная система аутентификации корпоративного уровня. Она защищает от таких атак, как перехват данных аутентификации. Кроме того, она устраняет угрозу имперсонации, поскольку учетные данные не передаются по сети в открытом виде. (Если вопросы безопасности вас не интересуют, переходите прямо к разделу «Конфигурирование и настройка Spark»).

Основные дистрибутивы Hadoop, в т. ч. CDH и HDP, поддерживают аутентификацию через Kerberos.

Сертифицирована для работы с дистрибутивами Hadoop

Компания Skymind (осуществляющая коммерческую поддержку DL4J) гарантирует, что все новые версии DL4J сертифицированы для работы с последними версиями CDH и HDP.

Учетные данные для Kerberos могут храниться в каталоге LDAP или Active Directory.

Для запуска Spark в YARN-кластере с поддержкой а Kerberos нужно:

1) вручную загрузить JAR-файл со сборкой Spark в HDFS;
2) инициализировать Kerberos из командной строки.

Загрузка сборки Spark. Сначала JAR-файл со сборкой Spark загружается в HDFS:

```
/user/spark/share/lib
```

Этот JAR-файл находится в локальной файловой системе, обычно в каталоге

```
/usr/lib/spark/assembly/lib
```

или в случае CDH:

```
/opt/cloudera/parcels/CDH/lib/spark/assembly/lib
```

При выполнении задачи Spark в HDP библиотека загружается в HDFS, поэтому у пользователя, запустившего задачу, должны быть права для записи в HDFS[12].

Замечание о Kerberos и Spark
Если попытаться запустить задачу Spark в кластере с поддержкой Kerberos, не загрузив вручную JAR-файл в HDFS, то выполнение завершится ошибкой. Ошибка возникнет при выполнении команды загрузки файла (выполняемой как часть задачи), поскольку поддержка Kerberos ограничена.

Инициализация Kerberos. Для инициализации Kerberos введите команду

```
kinit [user-name]
```

Будет предложено ввести пароль. После инициализации Kerberos мы можем убедиться, что мандат Kerberos получен:

```
klist
```

В результате будет выведена примерно такая информация, из которой видно, что мандат Kerberos действителен:

```
[skymind@sandbox ~]$ klist

Ticket cache: FILE:/tmp/krb5cc_1025

Default principal: skymind@HORTONWORKS.COM


Valid starting Expires Service principal

07/05/16 20:39:08 07/06/16 20:39:08 krbtgt/HORTONWORKS.COM@HORTONWORKS.COM
     renew until 07/05/16 20:39:08
```

Конфигурирование и настройка Spark

Spark может работать на различных распределенных платформах или локально на одной машине. В этой главе будем считать, что Spark выполняется в кластере Hadoop+YARN или в кластере Mesos. Эти системы должны быть знакомы корпоративным пользователям, работающим с дистрибутивами Cloudera CDH или Hortonworks HDP, а также пользователям системы управления кластером Apache Mesos.

Spark работает по-разному в зависимости от того, является система распределенной или локальной, и от того, работает ли процесс драйвера Spark во время выполнения задачи. Понимание базовых принципов работы позволит без лишнего напряжения перейти от локального выполнения к выполнению длительных задач в кластере, которые можно оставить на ночь, прервав сеанс связи с кластером.

У каждого приложения есть процесс драйвера, который может работать в приоритетном (клиентском) или фоновом (кластерном) режиме. Если снять приоритетного клиента, то задача остановится, потому что пропал локальный процесс,

---

[12] https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.5.3/bk_spark-component-guide/content/run-sample-apps.html.

управляющий работой в кластере. Но мы можем снять фонового клиента и позволить задаче Spark продолжить работу, потому что ее контроллер работает на другом хосте. Этот активный драйвер используется для управления потоком задач и планирования заданий Spark.

В клиентском режиме процесс драйвера работает на локальной машине, на которой запущен, а в кластерном режиме – на удаленном узле самого кластера.

Выполнение Spark в кластере Mesos

Возможно, вы предпочитаете запускать Spark в распределенном режиме без Hadoop, но с поддерживаемым диспетчером кластера Apache Mesos. Mesos позволяет пользователю абстрагировать процессоры, память, устройства хранения и другие вычислительные ресурсы от физических машин и рассматривать кластер как единый логический пул ресурсов, позволяющий эффективно реализовать отказоустойчивые многоарендаторные системы.

Мастер Mesos заменяет мастер Spark в роли диспетчера кластера. В этом режиме Mesos определяет, какие машины за какие задания отвечают, когда драйвер создает задачу Spark и подлежащие планированию задания. В кластере Mesos исполняется много коротких задач, и Mesos управляет ими таким образом, чтобы дать другим каркасам и заданиям возможность работать в кластере (обеспечивает многоарендаторность).

Есть два режима работы Spark поверх Mesos:

- клиентский режим;
- кластерный режим.

В клиентском режиме каркас Spark запускается непосредственно на клиентской машине и ждет, когда пользователь введет команду или захочет выполнить программу. Пользователь будет видеть сообщения драйвера прямо на своем экране.

Для запуска Spark поверх Mesos в клиентском режиме[13] необходимо:

1) задать в файле *spark-env.sh* переменные среды, относящиеся к Mesos;
2) передать в контекст SparkContext URL-адрес кластера Mesos.

В кластерном режиме[14] драйвер Spark запускается на каком-то хосте кластера, а пользователь наблюдает за результатами задачи в веб-интерфейсе Mesos. В этом режиме необходимо запустить диспетчер `MesosClusterDispatcher` кластера с помощью скрипта *sbin/startmesos-dispatch.sh*, указав URL-адрес мастера Mesos. Для запуска задачи можно использовать тот же скрипт *spark-submit.sh*, что и раньше, но нужно будет включить URL-адрес мастера, как показано в примере ниже:

```
./bin/spark-submit \
  --class io.skymind.spark.mesos.MyTestMesosJob \
  --master mesos://210.181.122.139:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /tmp/mySparkJob.jar \
  1000
```

[13] https://spark.apache.org/docs/latest/running-on-mesos.html#client-mode.

[14] https://spark.apache.org/docs/latest/running-on-mesos.html#cluster-mode.

Если вы хотите узнать больше о совместной работе Mesos и Spark, загляните на страницу «Running Spark on Mesos» по адресу https://spark.apache.org/docs/latest/running-on-mesos.html. Там вы найдете полный перечень параметров для конфигурирования задач Spark.

Можно также запустить Spark и Mesos вместе с кластером Hadoop на одном и том же наборе физических машин. В этом случае задачи Spark, работающие в кластере Mesos, смогут обращаться к данным в системе HDFS, задавая полные URL-адреса файлов.

Выполнение Spark поверх YARN

YARN – это API, который позволяет полноценно использовать инфраструктуру Hadoop приложениям общего назначения, а не только ориентированным на MapReduce. В настоящее время реализация Spark поверх YARN поддерживает два режима выполнения задач:

- yarn-client;
- yarn-cluster.

При работе с приложениями YARN важны понятия мастера приложения (`ApplicationMaster`) и диспетчера узла (`NodeManager`).

YARN, ApplicationMaster и NodeManager

Мастер приложения (ApplicationMaster) отслеживает потребление ресурсов и выполнение конкретной задачи в кластере Hadoop. Диспетчер узла (NodeManager) выделяет контейнеры YARN, в которых выполняются задания, входящие в состав задачи. Мы хотим не столько нагрузить вас жаргоном распределенных систем, сколько сообщить о контексте, охватывающем все, что происходит в производственном кластере Hadoop.

В случае Spark внутри контейнеров YARN работают исполнители Spark, а задача Spark координируется мастером приложения Spark. Вы должны знать о существовании этих процессов, чтобы лучше понимать, как, когда и где они выполняют задачи Spark.

Каждый исполнитель Spark работает в контейнере YARN[15]. Spark размещает несколько заданий в одном контейнере, что на несколько порядков уменьшает время инициализации. Рассмотрим оба режима более пристально:

- yarn-client – в этом режиме драйвер Spark исполняется на машине, с которой была запущена задача. Часто это ноутбук разработчика, подключенный к кластеру по сети. Драйвер Spark взаимодействует с мастером приложения Spark в кластере Hadoop и передает команды, которые должны быть выполнены в качестве заданий исполнителями, работающими в контейнерах YARN;
- yarn-cluster – процесс драйвера Spark работает удаленно на машине кластера Hadoop+YARN (HDP[16], CDH). Мастер приложений, от которого YARN ожидает входных данных для координации задачи, также исполняет процесс драйвера. Если задача Spark запущена таким образом, то пользователь может остановить свой клиент или терминальный сеанс, а задача все равно завершится.

[15] Не путайте с контейнером Linux, это разные понятия.

[16] См. http://bit.ly/2tQmuuh, а также об удаленном запуске задач Spark из NiFi.

Сравнение режимов выполнения Spark

В табл. 9.1 приведены характеристики работы Spark в различных режимах в кластере с поддержкой YARN. Эту таблицу составил Сэнди Райза (Sandy Ryza) для блога Cloudera Engineering.

Таблица 9.1. ***Различные режимы выполнения Spark в кластере с поддержкой YARN***

	YARN-кластер	**YARN-клиент**	**Автономно**
Где работает драйвер?	Мастер приложений	Клиент	Клиент
Кто запрашивает ресурсы?	Мастер приложений	Мастер приложений	Клиент
Кто запускает процессы исполнителей?	Диспетчер узла YARN	Диспетчер узла YARN	Рабочий процесс Spark
Службы сохранения	Менеджер ресурсов YARN и диспетчеры узлов	Менеджер ресурсов YARN и диспетчеры узлов	Мастер и рабочие процессы Spark
Поддержка оболочки Spark	Нет	Да	Да

Когда какой режим выполнения Spark использовать

Во время интерактивной отладки задач Spark в YARN-кластере удобно использовать режим yarn-client. А если имеется долго работающая или планируемая производственная задача Spark, то больше подойдет режим yarn-cluster.

Зачем выполнять Spark в кластере YARN или Mesos?

Каркас YARN позволяет пользоваться всеми преимуществами среды выполнения Hadoop не только приложениям на основе технологии MapReduce. Выполнение Spark поверх YARN открывает следующие возможности:

- централизованно конфигурировать один пул ресурсов кластеров и динамически разделять его между различными одновременно работающими приложениями и каркасами;
- пользоваться преимуществами планировщика yarn для повышения уровня конкурентности;
- динамически выделять исполнителей и контролировать, сколько их работает в кластере над одним приложением Spark;
- поддержку Kerberos[17].

Способность динамически разделять кластер между несколькими приложениями – важное свойство системы масштаба предприятия. Это означает, что мы можем одновременно в одном и том же кластере выполнять запросы Impala, задачи MapReduce и приложения Spark и при этом разумно управлять тем, сколько ресурсов получает каждое приложение. Возможность использовать планировщик YARN или Mesos позволяет одновременно выполнять в кластере произвольные запросы и запланированное производственное приложение, гарантируя, что последнему будет выделено достаточно ресурсов для удовлетворения соглашения об уровне обслуживания (Service Level Agreement – SLA). Все это возможно, потому что YARN – как и Mesos – знает, как конкурентно управлять несколькими каркасами, чтобы все получали ресурсы предсказуемым и контролируемым образом.

В многоарендаторной среде с различными типами рабочих нагрузок в одном кластере YARN и Mesos – лучшие друзья DevOps.

[17] Mesos поддерживает все то же самое, кроме Kerberos.

Общее руководство по настройке Spark

На верхнем уровне для настройки задачи Spark важны два аспекта: процессор и память. Мы можем задать количество исполнителей, а также количество ядер и объем памяти, доступные каждому исполнителю.

Mesos и GPU

В кластере Mesos есть также возможность управлять графическими процессорами (GPU), в YARN она пока отсутствует.

Задание числа исполнителей

Число исполнителей приложения задается в командной строке или в конфигурационном файле. В командной строке для этого служит флаг:

```
--num-executor
```

А в конфигурационном файле – свойство:

```
spark.executor.instances
```

Это свойство можно задать в файле *spark-defaults.conf* или в объекте `SparkConf` с помощью API.

Задание числа процессорных ядер

Каждому исполнителю приложения Spark доступно одинаковое количество ядер. Оно задается в командной строке или в конфигурационном файле. В командной строке для этого служит флаг:

```
--executor-cores
```

А в конфигурационном файле – свойство:

```
spark.executor.cores
```

задаваемое в файле *spark-defaults.conf* или в объекте `SparkConf` с помощью API.

Ядра и параллельно выполняемые задания исполнителей

Задавая количество ядер, доступных в пакетном режиме выполнения Spark, мы тем самым определяем, сколько заданий может выполняться одновременно.

Задание объема памяти

Объем памяти, доступной исполнителю, задается в командной строке с помощью флага:

```
--executor-memory
```

Объем можно задавать в мегабайтах (суффикс m) или в гигабайтах (суффикс g). То же самое можно сделать с помощью свойства в конфигурационном файле:

```
spark.executor.memory
```

Существует еще два параметра для более точного управления памятью:

- `spark.shuffle.memoryFraction`;
- `spark.storage.memoryFraction`.

Они описывают, как память распределяется между операциями преобразования и сохранения данных.

- `spark.shuffle.memoryFraction` – определяет, какая часть кучи Java используется для агрегирования и когрупп в процессе перемешивания. По умолчанию значение равно 0.2, и мы рекомендуем оставить его хотя бы поначалу;
- `spark.storage.memoryFraction` – определяет, какая часть кучи Java используется для кэширования RDD. По умолчанию значение равно 0.6.

У исполнителя Spark могут также быть накладные расходы, для управления которыми служит следующий параметр.

- `spark.yarn.executor.memoryOverhead` – определяет, сколько памяти вне кучи Java выделяется каждому исполнителю. Эта память используется для накладных расходов виртуальной машины, интернированных строк и других платформенных надобностей. По умолчанию значение равно (executorMemory * 0.10), но не менее 384 МБ.

Spark и выделение ресурсов для контейнеров YARN

При работе со Spark поверх YARN следует учитывать объем ресурсов, доступных YARN[18]. Наибольший интерес представляют следующие свойства YARN:

- `yarn.nodemanager.resource.memory-mb` – этот параметр задает максимальный объем памяти, доступной контейнерам на каждом хосте кластера Spark;
- `yarn.nodemanager.resource.cpu-vcores` – этот параметр задает максимальное количество ядер, доступных контейнерам на каждом хосте кластера Spark. Запрос 10 ядер для исполнителей задачи Spark транслируется в запрос 10 виртуальных ядер у YARN.

Запросы памяти для исполнителей в YARN

Параметр `spark.executor.memory` управляет размером кучи исполнителя, но JVM может также использовать память вне кучи. Чтобы узнать, сколько всего памяти запрашивается у YARN, нужно сложить величины `spark.executor.memory` и `spark.yarn.executor.memoryOverhaul`. Это показано на рис. 9.1.

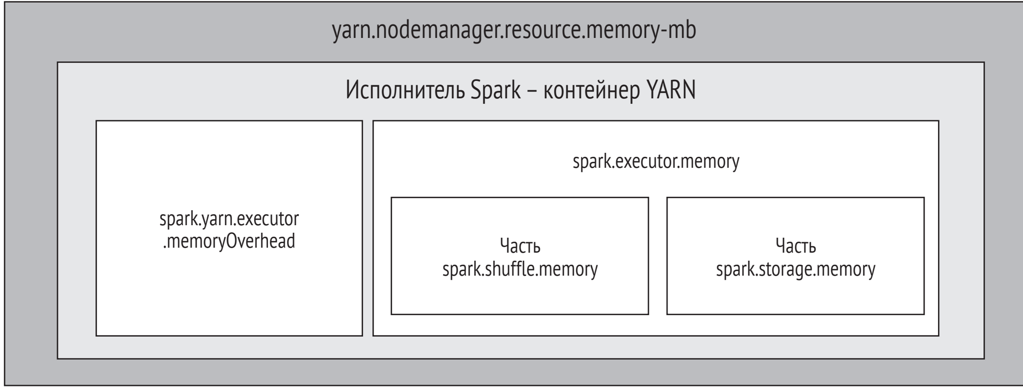


Рис. 9.1 ❖ Иерархия памяти для Spark поверх YARN

[18] Отметим также, что YARN может ограничивать ресурсы, запрашиваемые Spark, поскольку процесс Spark работает в контейнере YARN.

Будьте осторожны при запросе памяти

Если сумма `spark.yarn.executor.memoryOverhead` и `spark.executor.memory` больше параметра YARN `yarn.nodemanager.resource.memory-mb`, то задача не запустится, потому что YARN не выделит ей затребованных ресурсов контейнера.

По умолчанию значение `spark.shuffle.memoryFraction` равно 0.2, а значение `spark.shuffle.safetyFraction` равно 0.8. Объем памяти, доступной каждому заданию исполнителя, вычисляется по формуле:

```
( spark.executor.memory * spark.shuffle.memoryFraction
 * spark.shuffle.safetyFraction ) / spark.executor.cores
```

Надеемся, что приведенного краткого обзора настройки Spark и YARN достаточно, чтобы приступить к работе. Существует еще много настраиваемых параметров, но они выходят за рамки этой книги.

Для дополнительного чтения

Интересную информацию о настройке Spark можно найти в блоге Сэнди Райза (Sandy Ryza) по адресу http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/.

Spark, JVM и сборка мусора

Тема настройки JVM для работы Spark обширна и сложна. Мы остановимся только на нескольких важных вопросах, а остальное оставим для самостоятельного изучения.

Как быть, если сборка мусора становится менее эффективной или приостанавливается. Следует проверять, что приложение Spark эффективно использует выделенную память. Если RDD занимает очень много памяти, то у исполнителя останется меньше памяти для работы, и производительность снизится.

Если вы видите, что сборка мусора производится слишком часто, значит, Spark использует память недостаточно эффективно. Сгладить проблему можно, если явно удалять кэшированные RDD, когда необходимость в них отпала.

Выбор сборщика мусора. Вообще говоря, для исполнителей Spark рекомендуется выбирать сборщик мусора G1 (https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html).

Настройка задач DL4J для Spark

Рассмотрев основные механизмы настройки задачи Spark, мы можем перейти к вопросу о настройке задач, написанных с применением DL4J для Spark. Как и раньше, есть три основных фактора:

- количество рабочих процессов (исполнителей);
- объем памяти, доступный каждому рабочему процессу;
- количество процессорных ядер, доступных каждому рабочему процессу.

Рассмотрим, как они влияют на обучение глубокой модели.

Задание числа исполнителей

Увеличивая число исполнителей (рабочих процессов), мы разбиваем данные на большее число частей, так что на долю каждого исполнителя приходится меньше записей. Это уменьшает общее время обучения.

Число исполнителей и уменьшение рентабельности

После определенного порога добавление новых исполнителей приводит к уменьшению рентабельности. После того как весь набор данных разбит на мини-пакеты и число исполнителей равно числу мини-пакетов, дальнейшее их наращивание бессмысленно, поскольку исполнителям будет нечего обрабатывать.

Задание объема памяти для исполнителей

Мини-пакет обучающих примеров передается ND4J в виде матрицы векторизованных данных, подлежащих обработке. Тем самым оборудование используется более эффективно. Выбор размера мини-пакета важен, потому что влияет не только на качество обучения, но и на объем памяти, потребляемой исполнителем[19].

Чем больше примеров в мини-пакете, тем больше памяти выделяется исполнителю. Если общий объем памяти ограничен, то мини-пакеты не следует делать слишком большими.

Эвристические правила использования памяти

Для данных с плавающей точкой (самый типичный случай): 4 байта на одно значение плюс накладные расходы. Вот два примера:

- набор MNIST (небольшой): 28 × 28 = 784 → ~3 КБ (плюс 10 меток) на каждый пример;
- временные ряды с 256 входами, 1000 временных шагов → 1 МБ на пример для признаков (плюс метки).

Производительность параллельных вычислений

Распараллеливание итеративных алгоритмов обучения отражается на сходимости, хотя и не сильно. Пользователь DL4J может задать размер мини-пакета, и программа оптимально распределит нагрузку между исполнителями Spark. Это оптимальное разбиение входных данных, как в Hadoop. Система максимизирует количество доступных ей процессоров, автоматически осуществляя горизонтальное масштабирование. При выполнении DL4J в таких средах, как сам YARN или Spark поверх Hadoop, исполнитель получает часть набора данных, зависящую от параметра Hadoop InputSplit (или размера блока данных HDFS), что заодно обеспечивает хороший баланс дисковых операций ввода-вывода между исполнителями.

На практике усреднение параметров играет роль регуляризатора. Мы рекомендуем начинать с увеличения числа периодов обучения на 10 процентов, чтобы учесть дополнительную регуляризацию. Хотя число периодов немного возрастает, время завершения каждого периода уменьшается кратно числу исполнителей Spark.

В некоторых режимах (например, если усреднение производится сравнительно редко) количество периодов должно быть больше. Обучение с усреднением по каждому мини-пакету (не рекомендуется из соображений производительности) должно быть аналогично локальному (с размером мини-пакета, равным общему числу примеров для всех исполнителей на одной итерации).

Подготовка проекта Maven для Spark и DL4J

Подготовка POM-файла для проекта Maven – ключ к созданию проекта DL4J, а также задачи Hadoop или Spark. Apache Maven позволяет собрать все необходимые

[19] Breuel, 2015. The Effects of Hyperparameters on SGD Training of Neural Networks // https://arxiv.org/abs/1508.02788.

файлы и зависимости в один JAR-файл[20]. В этом разделе мы дадим ряд рекомендаций по написанию POM-файлов для проектов на основе DL4J. Приложение DL4J + Spark имеет следующие зависимости:

- DL4J;
- ND4J;
- DataVec;
- DL4J-Spark.

Для разных версий Spark файл *pom.xml* строится по-разному. Для поддержки базового взаимодействия с Hadoop следует добавить такую зависимость:

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>
```

Переменная `hadoop.version` задается в секции свойств файла *pom.xml* и зависит от дистрибутива Hadoop. О ее допустимых значениях можно прочитать на сайтах поставщиков Hadoop – CDH (https://www.cloudera.com/documentation/enterprise/release-notes/topics/cdh_vd_cdh5_maven_repo_57x.html) и HDP (http://repo.hortonworks.com/index.html#welcome)[21].

Для использования DL4J совместно со Spark следует включить зависимость `deeplearning4j-spark`:

```
<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>dl4j-spark_${scala.binary.version}</artifactId>
    <version>${dl4j.version}</version>
</dependency>
```

scala.binary.version

Обратите внимание на свойство Maven `${scala.binary.version}` в предыдущем фрагменте. Оно должно быть равно 2.10 или 2.11 – в соответствии с используемой версией Spark.

Переменная Maven `spark.version` зависит от дистрибутива Hadoop или Spark, а переменная `scala.binary.version` – от версии Spark. В табл. 9.2 приведены их краткие описания.

Таблица 9.2. Переменные в файле pom.xml

Переменная Maven	Описание
`hadoop.version`	Описывает используемый дистрибутив или версию Hadoop (например, CDH, HDP и т. д.)
`scala.binary.version`	Зависит от используемой версии Spark
`spark.version`	Зависит от дистрибутива Hadoop или Spark

Платформенная зависимость

Если вы собираете проект на одной платформе, а развертываете на другой, то задавайте вместо этого свойство `nd4j-native-platform`. При этом будут включены двоичные файлы для всех платформ.

[20] https://maven.apache.org/plugins/maven-assembly-plugin/.

[21] О семантическом версионировании Hadoop см. страницу по адресу https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/Compatibility.html.

Например, если проект собирается на Macbook Pro, а запускаться будет на кластере Spark, узлы которого работают под управлением ОС RedHat Linux, то этому свойству нужно присвоить значение RedHat.

Шаблон секции зависимостей в файле pom.xml

В этом разделе мы покажем, как должна выглядеть секция зависимостей в файле *pom.xml*. Сначала представим шаблон этой секции, а затем покажем, как задавать переменные в зависимости от дистрибутива Hadoop.

Пример 9.1 ❖ Файл pom.xml для проекта на базе DL4J

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-native-platform</artifactId>
            <version>${nd4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-api</artifactId>
            <version>${nd4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-library</artifactId>
            <version>${scala.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>

    <!-- Зависимости от Spark и Scala  -->
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-mllib_${scala.binary.version}</artifactId>
        <version>${spark.version}</version>
        <scope>${spark.scope}</scope>
    </dependency>

    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
        <version>${scala.version}</version>
        <scope>${spark.scope}</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_${scala.binary.version}</artifactId>
        <version>${spark.version}</version>
        <scope>${spark.scope}</scope>
    </dependency>
```

```
<!-- Зависимости от Deeplearning4j -->
<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-core</artifactId>
    <version>${dl4j.version}</version>
</dependency>

<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>dl4j-spark_${scala.binary.version}</artifactId>
    <version>${dl4j.version}</version>
</dependency>

<dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-kryo_${scala.binary.version}</artifactId>
    <version>${nd4j.version}</version>
</dependency>

<dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-native-platform</artifactId>
    <version>${nd4j.version}</version>
</dependency>

<!-- Зависимости от DataVec -->
<dependency>
    <groupId>org.datavec</groupId>
    <artifactId>datavec-api</artifactId>
    <version>${datavec.version}</version>
</dependency>

<dependency>
    <groupId>org.datavec</groupId>
    <artifactId>datavec-spark_${scala.binary.version}</artifactId>
    <version>${datavec.version}</version>
</dependency>

<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.7.1</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>

<!-- hadoop-mapreduce-client-app -->

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-app</artifactId>
```

```
        <version>${hadoop.version}</version>
        <scope>${spark.scope}</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-mapreduce-client-common</artifactId>
        <version>${hadoop.version}</version>
        <scope>${spark.scope}</scope>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
    </dependency>

    <dependency>
        <groupId>joda-time</groupId>
        <artifactId>joda-time</artifactId>
        <version>2.7</version>
    </dependency>

    <dependency>
        <groupId>org.apache.mrunit</groupId>
        <artifactId>mrunit</artifactId>
        <version>1.1.0</version>
        <classifier>hadoop2</classifier>
    </dependency>

    <!-- JCommander для разбора аргументов -->
    <dependency>
        <groupId>com.beust</groupId>
        <artifactId>jcommander</artifactId>
        <version>${jcommander.version}</version>
    </dependency>

</dependencies>
```

Управление размером JAR-файла

Одна из главных причин появления гигантских JAR-файлов – включение зависимостей, которые и так уже присутствуют на платформе, где производится развертывание. За счет их исключения иногда удается сократить размер JAR-файла от 50 до 80 процентов, что ускоряет компиляцию, перемещение и выполнение задач. Включением зависимостей управляет тег <scope>:

```
<scope>provided</scope>
```

Вот пример зависимости, в которой задана область видимости:

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>provided</scope>
</dependency>
```

Сериализация

По умолчанию в Spark используется сериализация Java, но обычно это не лучшее решение. В примере 9.1 выше мы исправили это упущение, включив зависимость Kryo:

```
<dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-kryo_${scala.binary.version}</artifactId>
    <version>${nd4j.version}</version>
</dependency>
```

Мы еще вернемся к данной зависимости, когда будем обсуждать типичные проблемы, связанные с ND4J и Spark.

Теперь обратимся к настройке для конкретных дистрибутивов Hadoop.

Настройка POM-файла для CDH 5.X

В этом разделе указаны номера версий компонентов, необходимых для сборки задачи Spark CDH 5.x.

Пример 9.2 ❖ Файл pom.xml для сборки задачи CDH5.x

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <slf4j.version>1.7.5</slf4j.version>
    <jackson.version>2.5.1</jackson.version>

    <hadoop.version>2.6.0-cdh5.5.2</hadoop.version>

    <!-- Версии компонентов DL4J -->
    <nd4j.version>0.7.2</nd4j.version>
    <dl4j.version>0.7.2</dl4j.version>
    <datavec.version>0.7.2</datavec.version>

    <scala.binary.version>2.10</scala.binary.version>
    <scala.version>2.10.4</scala.version>
    <spark.version>1.3.1</spark.version>

</properties>
```

Номера версий nd4j.version, dl4j.version и datave.version должны быть одинаковы в любом проекте на основе DL4J.

Настройка POM-файла для HDP 2.4

В этом разделе показано, как донастроить файл *pom.xml* для задачи Spark с дистрибутивом HDP 2.4, т. е. задать версии компонентов.

Пример 9.3 ❖ Файл pom.xml для сборки задачи HDP 2.4

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <slf4j.version>1.7.5</slf4j.version>
    <jackson.version>2.4.4</jackson.version>
    <jcommander.version>1.27</jcommander.version>

    <!-- Версия HDP 2.4. Модифицировать, если используется другая -->
    <hdp.version>2.4.0.0-169</hdp.version>
    <hadoop.version>2.7.1</hadoop.version>
```

```
    <spark.version>1.6.0</spark.version>
    <spark.scala.version>2.10</spark.scala.version>

    <!-- Версии компонентов DL4J -->
    <nd4j.version>0.7.2</nd4j.version>
    <dl4j.version>0.7.2</dl4j.version>
    <datavec.version>0.7.2</datavec.version>

    <scala.binary.version>2.10</scala.binary.version>
    <scala.version>2.10.4</scala.version>

</properties>
```

Отметим, что свойства в примере 9.3 используются системой сборки для построения JAR-файла артефактов Maven. В самом коде проекта на основе DL4J эти переменные не используются.

Отладка Spark и Hadoop

В этом разделе мы рассмотрим некоторые типичные проблемы, которые периодически возникают при использовании Spark поверх Hadoop.

Иногда, если исполнитель запрашивает больше памяти, чем есть в контейнере, драйвер Spark сообщает:

```
WARN TaskSchedulerImpl: Initial job has not accepted any resources; check your
cluster ui to ensure that workers are registered and have sufficient memory

ПРЕДУПРЕЖДЕНИЕ TaskSchedulerImpl: начальная задача не приняла никаких ресурсов;
проверьте в UI кластера, что рабочие процессы зарегистрированы и располагают
достаточной памятью.
```

Часто проблему можно сгладить, уменьшив объем памяти или количество ядер, запрашиваемое задачей.

На что смотреть при отладке Spark

В табл. 9.3 перечислены основные порты, позволяющие посмотреть, что делает Spark, и произвести отладку.

Таблица 9.3. *Основные порты для отладки Spark*

Веб-служба	Номер порта
UI сервера истории задач YARN	19888
UI диспетчера ресурсов YARN	8088
UI веб-интерфейса истории задач Spark	18080

Типичные проблемы при работе с ND4J

Далее мы рассмотрим несколько типичных проблем, встречающихся при работе ND4J на платформе Spark, а также их решения.

ND4J и библиотека сериализации Kryo

Kryo – библиотека сериализации, которая чаще всего используется совместно Apache Spark. Она повышает производительность благодаря уменьшению времени сериализации объектов.

SerDe

Аббревиатура «SerDe» означает «сериализация и десериализация». В случае Spark сериализации подвергаются данные и функции. Spark лишь устанавливает механизм сериализации, и по умолчанию таковым является стандартная сериализация Java, что удобно, но неэффективно.

В Hadoop реализованы собственные механизмы SerDe (объекты Writable), а для преобразования файлов в объекты Writable и наоборот применяются форматы ввода и вывода. Spark нуждается в форматах ввода-вывода для работы с данными, хранящимися в HDFS.

Однако Kryo испытывает трудности при работе со структурами данных вне кучи, применяемыми в ND4J[22]. Чтобы использовать сериализацию Kryo для ND4J на платформе Apache Spark, необходимо дополнительное конфигурирование. Если Kryo сконфигурирована неправильно, то возможны исключения `NullPointerException` для некоторых полей INDArray вследствие некорректной сериализации.

Для использования Kryo добавьте зависимость `nd4j-kryo` и настройте Spark, так чтобы использовался регистратор Kryo для ND4J:

```
SparkConf conf = new SparkConf();
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
conf.set("spark.kryo.registrator", "org.nd4j.Nd4jRegistrator");
```

Если Kryo сконфигурирована неправильно, то при использовании классов DL4J `SparkDl4jMultiLayer` и `SparkComputationGraph` в журнал записывается предупреждение.

jnind4j и java.library.path

Иногда встречается следующее сообщение об ошибке:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no jnind4j in
    java.library.path
```

Параллельное выполнение DL4J на платформе Spark

DL4J поддерживает обучение нейронных сетей в кластере Spark с целью ускорения. Для Spark определены классы, аналогичные `MultiLayerNetwork` и `ComputationGraph`.

- `SparkDl4jMultiLayer` – обертка вокруг `MultiLayerNetwork`.
- `SparkComputationGraph` – обертка вокруг `ComputationGraph`.

Поскольку это обертки вокруг стандартных классов для одной машины, процесс конфигурирования сети (т. е. создание объекта `MultiLayerConfiguration` или `ComputationGraphConfiguration`) идентичен в обоих случаях. Но распределенное обучение на платформе Spark все же отличается от локального в двух отношениях: загрузка данных и подготовка к обучению (требуется дополнительное конфигурирование специально для кластера).

Типичная последовательность обучения сети в кластере Spark выглядит так:

1. Создать собственный класс для обучения. Обычно для этого нужно:
   – определить конфигурацию сети (`MultiLayerConfiguration` или `ComputationGraphConfiguration`), как для обучения на одной машине;
   – создать экземпляр класса `TrainingMaster`: он описывает, как реально будет происходить распределенное обучение (см. ниже);

---

[22] Это проблема не столько ND4J, сколько Spark, но с практической точки зрения о ней все равно нужно знать.

   - создать экземпляр `SparkDl4jMultiLayer` или `SparkComputationGraph`, пользуясь объектами конфигурации сети и TrainingMaster;
   - загрузить обучающие данные. Существуют разные методы загрузки, каждый со своими компромиссами, детали изложены в документации;
   - вызвать подходящий вариант метода `fit` объекта `SparkDl4jMultiLayer` или `SparkComputationGraph`;
   - сохранить обученную сеть или воспользоваться ей.
2. Собрать JAR-файл, готовый для передачи Spark.
   - Если используется Maven, то для этого можно выполнить команду `mvn package -DskipTests`.
3. Выполнить скрипт *spark-submit.sh*, указав конфигурацию своего кластера.

Обучение на платформе Spark на одной машине

Для обучения на одной машине можно использовать локальный режим работы Spark, хотя это не рекомендуется (из-за накладных расходов на синхронизацию и сериализацию). Лучше рассмотреть следующие подходы:

- для системы с одним CPU/GPU использовать стандартное обучение с помощью классов `MultiLayerNetwork` или `ComputationGraph`;
- для системы с несколькими CPU/GPU использовать `ParallelWrapper`. Функционально это эквивалентно запуску Spark в локальном режиме, но накладные расходы ниже (а значит, обучение проходит быстрее).

В текущей версии DL4J для обучения сети используется процедура усреднения параметров. В будущие версии, возможно, будут добавлены другие подходы к распределенному обучению.

Процесс обучения сети методом усреднения параметров концептуально очень прост.

1. Мастер (драйвер Spark) берет начальную конфигурацию сети и параметры.
2. Данные разбиваются на порции, исходя из конфигурации объекта `TrainingMaster`.
3. Порции данных перебираются. Для каждой порции:
   - передать конфигурационные параметры (а для корректоров Momentum/RMSProp/AdaGrad также состояние корректора) от мастера каждому исполнителю;
   - обучить каждый исполнитель на его порции данных;
   - усреднить параметры (и, если нужно, состояние корректора) и вернуть результат усреднения мастеру.
4. После завершения обучения у мастера будет копия обученной сети.

Теперь посмотрим, как все это выглядит в коде.

Минимальный пример обучения на платформе Spark

Ниже продемонстрированы основные концепции, на основе которых мы далее построим полные примеры работы со Spark.

```
JavaSparkContent sc = ...;
    JavaRDD<DataSet> trainingData = ...;
    MultiLayerConfiguration networkConfig = ...;

    // Создать экземпляр TrainingMaster
    int examplesPerDataSetObject = 1;
```

```
TrainingMaster trainingMaster =
    new ParameterAveragingTrainingMaster
        .Builder(examplesPerDataSetObject)
        .(другие конфигурационные параметры)
        .build();

// Создать экземпляр SparkDl4jMultiLayer
SparkDl4jMultiLayer sparkNetwork =
    new SparkDl4jMultiLayer(sc, networkConfig, trainingMaster);

// Обучить сеть:
sparkNetwork.fit(trainingData);
```

Класс `TrainingMaster` в DL4J – абстракция (интерфейс), позволяющая использовать с `SparkDl4jMultiLayer` и `SparkComputationGraph` различные реализации обучения. В настоящее время существует только одна реализация, `ParameterAveragingTrainingMaster`, основанная на описанном выше процессе усреднения параметров. Для создания объекта мы пользуемся построителем:

```
TrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(int dataSetObjectSize)
        ... (конфигурационные параметры)
        .build();
```

В классе `ParameterAveragingTrainingMaster` определен ряд конфигурационных параметров, управляющих процессом обучения. Рассмотрим некоторые из них:

- `dataSetObjectSize` (обязательный параметр) – передается конструктору построителя и определяет количество примеров в каждом объекте `DataSet`. Запомните:
  - если обучение производится на предварительных обработанных объектах `DataSet`, то этот параметр задает размер именно таких объектов;
  - если обучение производится непосредственно на строках `String` (например, на CSV-данных, попадающих в `RDD<DataSet>` в результате ряда шагов), то этот параметр обычно равен 1;
- `batchSizePerWorker` – задает размер мини-пакета для каждого исполнителя. Аналог размера мини-пакета, используемого при обучении на одной машине. Иначе говоря, это количество примеров, после обработки которых исполнитель производит обновление параметров;
- `averageFrequency` – задает частоту усреднения и перераспределения параметров в терминах количества мини-пакетов размера `batchSizePerWorker`. Следует помнить о трех вещах:
  - небольшой период усреднения (например, `averagingFrequency=1`) может оказаться неэффективен (накладные расходы на сетевой обмен и инициализацию слишком велики по сравнению со временем вычислений);
  - большой период усреднения (например, `averagingFrequency=200`) может привести к снижению качества обучения (параметры, вычисленные каждым исполнителем, сильно различаются);
  - период усреднения через каждые 5–10 мини-пакетов обычно безопасен.
- `workerPrefetchNumBatches` – исполнители Spark умеют осуществлять асинхронную предвыборку нескольких мини-пакетов (объектов `DataSet`), чтобы не ждать загрузки данных. Если этот параметр равен 0, то предвыборка не

производится. Разумным значением часто является 2. Маловероятно, что значительно большие значения окажутся полезны (но памяти при этом потребляется больше);

- `saveUpdater` – в DL4J алгоритмы Momentum, RMSProp и AdaGrad называются корректорами (updaters). В большинстве из них хранится история, или состояние. Если параметр `saveUpdater` равен `true`, то состояние корректора (на каждом исполнителе) усредняется и возвращается мастеру вместе с параметрами, а мастер раздает текущее состояние исполнителям. Это увеличивает время и объем трафика, но результаты обучения иногда улучшаются. Если `saveUpdater` равен `false`, то каждый исполнитель отбрасывает состояние корректора и инициализирует его заново;
- `repartition` – этот параметр определяет, когда производить переразбиение данных на разделы. Класс `ParameterAveragingTrainingMaster` выполняет операцию `mapPartitions`, поэтому количество разделов (и значений в каждом разделе) существенно влияет на полноту использования возможностей кластера. Однако переразбиение – не бесплатная операция, т. к. некоторые данные приходится копировать по сети. Параметр может принимать следующие значения:
  - `Always` (по умолчанию). Всегда переразбивать данные, чтобы гарантировать правильность числа разделов;
  - `Never`. Никогда не переразбивать данные, даже если они сильно разбалансированы;
  - `NumPartitionsWorkersDiffers`. Переразбивать, только если число разделов не равно числу исполнителей (общему числу процессорных ядер). Отметим, однако, что даже если оба числа равны, это еще не гарантирует, что число объектов `DataSet` в каждом разделе оптимально: одни разделы могут быть больше, другие меньше;
- `repartitionStrategy` – стратегия переразбиения. Может принимать следующие значения:
  - `SparkDefault`. Стандартная стратегия переразбиения, применяемая Spark. Каждому объекту в начальном RDD случайным образом и независимо от остальных назначается один из *N* RDD. Поэтому разделы могут быть сбалансированы не оптимально, и это особенно проблематично (из-за случайных колебаний выборки) для небольших RDD, например состоящих из предварительно обработанных объектов `DataSet` с частым усреднением;
  - `Balanced`. Специальная стратегия переразбиения, определенная в DL4J. Идея в том, чтобы гарантировать, что каждый раздел сбалансирован (в терминах количества объектов) лучше, чем в режиме `SparkDefault`. Но на практике это требует дополнительной операции подсчета, и в некоторых случаях (прежде всего в малых сетях или в сетях с небольшим объемом вычислений на один мини-пакет) получаемая выгода перевешивается дополнительными накладными расходами на поиск лучшего переразбиения.

Рекомендации по использованию DL4J API для Spark

Чтобы получить максимум преимущества использования кластера Spark+Hadoop, придерживайтесь следующих рекоменаций:

- уменьшайте размер JAR-файла;
- тщательно настраивайте кластер;
- оптимизируйте конвейер ETL/векторизации;
- настраивайте параметры JVM.

В идеале задача должна быть как можно меньше. Главный посыл горизонтального масштабирования – «переносить вычисления ближе к данным», и мы хотим, чтобы по кластеру перемещался настолько компактный JAR-файл, насколько это возможно.

Кластер, в котором используется Spark, должен быть оптимально настроен и оснащен хорошей распределенной файловой системой.

Что такое «хорошая распределенная файловая система»?

Существует несколько распределенных файловых систем. Следует поискать такую, которая заведомо работает с тем комплектом распределенных компонентов, который вы собираетесь использовать.

Мы не согласны с тем, что настройка и сопровождение инфраструктуры распределенных систем не входят в обязанности специалиста по анализу данных. Проект машинного обучения, абстрагирующийся от сложностей инфраструктуры, – синоним проекта, обреченного на неудачу. Придерживаясь этой точки зрения, мы рекомендуем брать распределенную файловую систему, обладающую следующими свойствами:

- заведомо надежна;
- поддерживает интеграцию с Kerberos;
- хорошо протестирована;
- масштабируется;
- работает с другими выбранными вами компонентами.

Очень часто, особенно в контексте Spark и Hadoop, это означает выбор HDFS.

Серьезно настроенные корпоративные пользователи предпочитают иметь хорошо сопровождаемый кластер Hadoop+Spark, работающий на современном дистрибутиве Hadoop. Мы рекомендуем CDH5 или HDP 2.4.

Вынесите ETL и векторизацию за пределы главного цикла обучения. Выполнение больших задач в кластерах обходится дорого, поэтому они должны быть максимально эффективными. В идеале хотелось бы сохранять и загружать серализованные объекты `DataSet`, а не заниматься снова и снова преобразованием в RDD.

Наконец, оптимизация JVM занимает далеко не последнее место. Убедитесь, что JVM настроена так, чтобы не было долгих пауз на сборку мусора, – и всем станет легче жить.

Пример многослойного перцептрона на платформе Spark

В этом примере мы вновь вернемся к набору данных MNIST для классификации рукописных цифр, но на этот раз воспользуемся многослойным перцептроном и обучим его с применением Spark.

Основное отличие от примеров в главе состоит в распараллеливании обучения, все остальное очень похоже.

Нужно также учитывать, что теперь мы работаем с HDFS и что конвейер ETL и векторизации следует строить в расчете на горизонтальное масштабирование.

Для этого можно использовать либо функции Spark (как мы вскоре увидим), либо библиотеки ETL, например DataVec. Но в этом примере мы будем использовать встроенный в библиотеку код, специализированный под формат набора MNIST, так особых проблем не возникнет. А упомянули мы об этом, имея в виду использование в других проектах, где имеются данные в формате CSV или просто текстовые, для которых требуется сложная предварительная обработка.

Осторожнее с однопроцессным кодом ETL

При работе с большими наборами данных следует с осторожностью подходить к построению конвейеров ETL. Spark позволяет смешивать однопроцессный код на Java с функциями Spark, и, проявив небрежность, мы можем получить код, который не масштабируется под размер входного набора. В примерах применения DataVec в репозитории на GitHub показано, как эффективно строить конвейер ETL для связки Spark и DL4J.

Пример 9.4 ❖ Набор данных Saturn для многослойного перцептрона на платформе Spark

```
public class MnistMLPExample {
    private static final Logger log = LoggerFactory.getLogger(MnistMLPExample.class);

    @Parameter(names = "-useSparkLocal", description =
        "Usespark local (helper for testing/running without spark submit)",
            arity = 1)
    private boolean useSparkLocal = true;

    @Parameter(names = "-batchSizePerWorker", description =
        "Number of examples to fit each worker with")
    private int batchSizePerWorker = 16;

    @Parameter(names = "-numEpochs", description = "Number of epochs for training")
    private int numEpochs = 15;

    public static void main(String[] args) throws Exception {
        new MnistMLPExample().entryPoint(args);
    }

    protected void entryPoint(String[] args) throws Exception {
        // Обработка аргументов в командной строке
        JCommander jcmdr = new JCommander(this); try {
            jcmdr.parse(args);
        } catch (ParameterException e) {
            // Если заданы недопустимые параметры, напечатать сообщение
            jcmdr.usage();
            try { Thread.sleep(500); } catch (Exception e2) { }
            throw e;
        }

        SparkConf sparkConf = new SparkConf();
        if (useSparkLocal) {
            sparkConf.setMaster("local[*]");
        }
        sparkConf.setAppName("DL4J Spark MLP Example");
        JavaSparkContext sc = new JavaSparkContext(sparkConf);

        // Загрузить данные в память, затем распараллелить
        // Вообще говоря, это не очень хороший подход, но он простой,
        // поэтому мы решили использовать его в примере
```

```
DataSetIterator iterTrain =
    new MnistDataSetIterator(batchSizePerWorker, true, 12345);
DataSetIterator iterTest =
    new MnistDataSetIterator(batchSizePerWorker, true, 12345);
List<DataSet> trainDataList = new ArrayList<>();
List<DataSet> testDataList = new ArrayList<>();
while (iterTrain.hasNext()) {
    trainDataList.add(iterTrain.next());
}
while (iterTest.hasNext()) {
    testDataList.add(iterTest.next());
}

JavaRDD<DataSet> trainData = sc.parallelize(trainDataList);
JavaRDD<DataSet> testData = sc.parallelize(testDataList);


// ----------------------------------
// Сконфигурировать и обучить сеть
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(12345)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
        .activation(Activation.LEAKYRELU)
        .weightInit(WeightInit.XAVIER)
        .learningRate(0.02)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .regularization(true).l2(1e-4)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(28 * 28).nOut(500).build())
        .layer(1, new DenseLayer.Builder().nIn(500).nOut(100).build())
        .layer(2, new OutputLayer.Builder(LossFunctions.LossFunction
            .NEGATIVELOGLIKELIHOOD)
            .activation(Activation.SOFTMAX).nIn(100).nOut(10).build())
        .pretrain(false).backprop(true)
        .build();

// Конфигурация для обучения на Spark: описание параметров см. по адресу
// http://deeplearning4j.org/spark
TrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(batchSizePerWorker)    // Каждый объект DataSet по умолчанию
                                    // содержит 32 примера
    .averagingFrequency(5)
    .workerPrefetchNumBatches(2)    // Асинхронная предвыборка: 2 примера
                                    // на каждого исполнителя
    .batchSizePerWorker(batchSizePerWorker)
    .build();

// Создать сеть Spark
SparkDl4jMultiLayer sparkNet = new SparkDl4jMultiLayer(sc, conf, tm);

// Обучение
for (int i = 0; i < numEpochs; i++) {
    sparkNet.fit(trainData);
    log.info("Конец периода {}", i);
```

```
        }

        // Оценивание (распределенное)
        Evaluation evaluation = sparkNet.evaluate(testData);
        log.info("***** Оценивание *****");
        log.info(evaluation.stats());

        // Удалить временные файлы, они больше не нужны
        tm.deleteTempFiles(sc);

        log.info("***** Конец примера *****");
    }
}
```

Далее мы разберем этот код по частям и обсудим отличия версии для Spark.

Для сборки проекта задачи Spark перейдите в корневой каталог проекта и выполните команду

```
mvn package
```

JAR-файл задачи Spark будет создан в подкаталоге ./target. Его нужно скопировать на компьютер-шлюз, чтобы кластер Spark выполнил задачу, не забыв указать в командной строке параметры, описанные выше в этой главе.

Конфигурирование архитектуры МСП для Spark

В примере 9.4 архитектура сети похожа на ту, что мы видели в главе 5. Вот она:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(12345)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
        .activation(Activation.LEAKYRELU)
        .weightInit(WeightInit.XAVIER)
        .learningRate(0.02)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .regularization(true).l2(1e-4)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(28 * 28).nOut(500).build())
        .layer(1, new DenseLayer.Builder().nIn(500).nOut(100).build())
        .layer(2, new OutputLayer.Builder(LossFunctions.LossFunction
            .NEGATIVELOGLIKELIHOOD)
            .activation(Activation.SOFTMAX).nIn(100).nOut(10).build())
        .pretrain(false).backprop(true)
        .build();
```

Здесь мы по-прежнему используем СГС для оптимизации параметров сети. В многослойном перцептроне из главы 5 был всего один скрытый слой, а в этом примере есть еще один. Мы также используем в качестве функции активации блок линейной ректификации (ReLU) с утечкой, а не просто ReLU, как раньше. Все остальное, включая и выходной слой, ничем не отличается от предыдущего примера. Это наглядно показывает, что простые на первый взгляд изменения конфигурации позволяют моделировать совершенно другие данные. Хитрость в том, чтобы путем подбора гиперпараметров отыскать эти простые изменения.

Одинаковая архитектура сети для локального обучения и обучения на Spark
DL4J позволяет разработать сеть на подмножестве данных на локальной машине, а затем перенести архитектуру на платформу Spark и смоделировать полный набор данных.

Распределенное обучение и оценивание модели

Обратите внимание на два важных отличия от локальных примеров, приведенных в главе 5:

- появление класса `ParameterAveragingTrainingMaster`;
- другой класс обертки: `SparkDl4jMultiLayer`.

Это единственные крупные изменения по сравнению с примером из главы 5. Они занимают не много строчек кода, так что переход от локального обучения к обучению в кластере Spark, как видим, вовсе не труден. Приведем еще раз код, в котором производится усреднение параметров:

```
TrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(batchSizePerWorker)    // Каждый объект DataSet по умолчанию содержит 32 примера
    .averagingFrequency(5)
    .workerPrefetchNumBatches(2)    // Асинхронная предвыборка: 2 примера на каждого
                                    // исполнителя
    .batchSizePerWorker(batchSizePerWorker)
    .build();
```

А также код, в котором используется обертывающий класс `SparkDl4jMultiLayer`, производный от `MultiLayerNetwork`:

```
// Создать сеть Spark
SparkDl4jMultiLayer sparkNet = new SparkDl4jMultiLayer(sc, conf, tm);

// Обучение
for (int i = 0; i < numEpochs; i++) {
    sparkNet.fit(trainData);
    log.info("Конец периода {}", i);
}
```

Эта обертка работает в тандеме с `TrainingMaster` и реализует распределенное обучение модели, позволяя нам абстрагироваться от деталей. С помощью класса `SparkDl4jMultiLayer` мы можем контролировать исполнителей почти так же, как при обучении на локальной машине.

Как видим, конструктор класса `SparkDl4jMultiLayer` принимает три параметра:

1) контекст Spark;
2) конфигурацию сети;
3) объект `ParameterAveragingTrainingMaster`.

В локальном варианте этого примера класс `MultiLayerNetwork` используется для моделирования данных. А эта обертка хороша тем, что выглядит почти как `MultiLayerNetwork` и ее можно точно так же использовать в цикле `for` по периодам.

Наконец, мы вычисляем и протоколируем F-меру сети:

```
// Оценивание (распределенное)
Evaluation evaluation = sparkNet.evaluate(testData);
log.info("***** Оценивание *****");
log.info(evaluation.stats());
```

Этот пример – хорошая иллюстрация того, как легко и изящно производится обучение в производственном безопасном кластере Spark с помощью DL4J. Помимо нескольких дополнительных строчек, в которых настраивается Spark и `TrainingMaster`, вам и делать-то ничего не надо.

Создание и выполнение задачи Spark

Перейдите в корневой каталог примера и с помощью Maven соберите JAR-файл задачи, выполнив команду:

```
maven package
```

JAR-файл будет создан в подкаталоге *./target/*.

Скопировав JAR-файл задачи на нужную машину, введите следующую команду:

```
spark-submit --class org.deeplearning4j.examples.feedforard.MnistMLPExample
  --num-executors 3 --properties-file ./spark_extra.props
  ./dl4j-examples-1.0-SNAPSHOT.jar
```

Она выводит на консоль кучу диагностической информации о ходе обучения. Из аргументов командной строки наибольший интерес представляет флаг `--properties-file`, указывающий, где находится файл с конфигурационными параметрами Spark. Таким образом, мы можем записать параметры в файл и не набирать их каждый раз при запуске команды. Еще отметим флаг `--num-executors`, который задает число исполнителей Spark, в данном случае он равен `3`.

Порождение текстов в стиле Шекспира с помощью Spark и LSTM-сети

Вернемся к примеру долгой краткосрочной памяти (LSTM) из главы 5 и посмотрим, как, модифицировав код, реализовать ту же модель в кластере Spark[23]. Мы не станем подробно рассматривать код загрузки данных, потому что хотим сосредоточиться на отличиях архитектуры и процесса обучения.

Пример 9.5 ❖ Основной метод обучения LSTM-сети для порождения текстов в стиле Шекспира

```java
protected void entryPoint(String[] args) throws Exception {
    // Обработка аргументов командной строки
    JCommander jcmdr = new JCommander(this);
    try {
        jcmdr.parse(args);
    } catch (ParameterException e) {
        // Если заданы недопустимые параметры, напечатать сообщение
        jcmdr.usage();
        try {
            Thread.sleep(500);
        } catch (Exception e2) {
        }
        throw e;
```

[23] https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-spark-examples/dl4j-spark/src/main/java/org/deeplearning4j/rnn/SparkLSTMCharacterExample.java.

```
}

Random rng = new Random(12345);
int lstmLayerSize = 200;    // Число блоков в слое GravesLSTM
int tbpttLength = 50;       // Длина усеченного алгоритма обратного распространения
                            // во времени, т. е. обновление параметров производится
                            // через каждые 50 символов
int nSamplesToGenerate = 4;     // Сколько примеров порождать
                                // после каждого периода обучения
int nCharactersToSample = 300;              // Длина одного примера в символах
String generationInitialization = null;     // Факультативная инициализация символов;
                                            // если null, то берется случайный символ
// Эта строка используется для инициализации LSTM последовательностью
// символов, которую следует продолжить. Начальные символы должны принадлежать
// множеству CharacterIterator.getMinimalCharacterSet()

// Задать конфигурацию сети:
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
        .learningRate(0.1)
        .rmsDecay(0.95)
        .seed(12345)
        .regularization(true)
        .l2(0.001)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.RMSPROP)
        .list()
        .layer(0, new GravesLSTM.Builder().nIn(CHAR_TO_INT.size())
            .nOut(lstmLayerSize).activation(Activation.TANH).build())
        .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
            .activation(Activation.TANH).build())
        .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
            .activation(Activation.SOFTMAX)    // MCXENT + softmax для классификации
            .nIn(lstmLayerSize).nOut(nOut).build())
        .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
            .tBPTTBackwardLength(tbpttLength)
        .pretrain(false).backprop(true)
        .build();

// -------------------------------------------------------------
// Относящаяся к Spark часть конфигурации
/* Как часто усреднять параметры (количество мини-пакетов).
   Слишком частое усреднение может замедлить обучение (накладные расходы
   на синхронизацию и сериализацию), а слишком редкое может привести к тому,
   что сеть не сойдется) */
int averagingFrequency = 3;

// Задать конфигурацию и контекст Spark
SparkConf sparkConf = new SparkConf();
if (useSparkLocal) {
    sparkConf.setMaster("local[*]");
}
sparkConf.setAppName("LSTM Character Example");
JavaSparkContext sc = new JavaSparkContext(sparkConf);
```

```
    JavaRDD<DataSet> trainingData = getTrainingData(sc);

    // Настроить объект TrainingMaster, управляющий обучением в кластере Spark.
    // В данном случае используется стандартный метод усреднения параметров.
    // Подробное описание параметров см. на странице
    // https://deeplearning4j.org/spark#configuring
    int examplesPerDataSetObject = 1;
    ParameterAveragingTrainingMaster tm = new ParameterAveragingTrainingMaster
        .Builder(examplesPerDataSetObject)
        .workerPrefetchNumBatches(2)    // Асинхронная предвыборка двух мини-пакетов
        .averagingFrequency(averagingFrequency)
        .batchSizePerWorker(batchSizePerWorker)
        .build();
    SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, conf, tm);
    sparkNetwork.setListeners(Collections.<IterationListener>singletonList(new
        ScoreIterationListener(1)));

    // Выполнить обучение, затем получить от сети и напечатать примеры
    for (int i = 0; i < numEpochs; i++) {
        // Выполнить один период обучения. В конце периода мы имеем копию обученной сети
        MultiLayerNetwork net = sparkNetwork.fit(trainingData);

        // Выбрать несколько символов из сети (делается локально)
        log.info("Выбираются символы из сети при заданной инициализации \"" +
            (generationInitialization == null ? "" : generationInitialization) +
                "\"");
        String[] samples = sampleCharactersFromNetwork(generationInitialization,
            net, rng, INT_TO_CHAR,
            nCharactersToSample, nSamplesToGenerate);
        for (int j = 0; j < samples.length; j++) {
            log.info("----- Пример " + j + " -----");
            log.info(samples[j]);
        }
    }

    // Удалить временные файлы, они больше не нужны
    tm.deleteTempFiles(sc);

    log.info("\n\nExample complete");
}
```

Этот код отличается от прежнего в двух отношениях:

- появление класса `ParameterAveragingTrainingMaster`;
- другой класс обертки: `SparkDl4jMultiLayer`.

Таким образом, переход от локального обучения к обучению в кластере Spark вовсе не труден. Чтобы еще раз убедиться в этом, посмотрим, как здесь определяется LSTM-сеть.

Конфигурирование архитектуры LSTM-сети

Интересно, что следующий далее фрагмент кода полностью идентичен коду в случае локального обучения из главы 5.

```
// Задать конфигурацию сети:
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
```

```
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
        .learningRate(0.1)
        .rmsDecay(0.95)
        .seed(12345)
        .regularization(true)
        .l2(0.001)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.RMSPROP)
        .list()
        .layer(0, new GravesLSTM.Builder().nIn(CHAR_TO_INT.size())
            .nOut(lstmLayerSize).activation(Activation.TANH).build())
        .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
            .activation(Activation.TANH).build())
        .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
            .activation(Activation.SOFTMAX)      // MCXENT + softmax для классификации
            .nIn(lstmLayerSize).nOut(nOut).build())
        .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
            .tBPTTBackwardLength(tbpttLength)
        .pretrain(false).backprop(true)
        .build();
```

Мы отмечаем этот факт, потому что с точки зрения эксплуатации это огромное преимущество. Мы можем взять конфигурацию сети, разработанной локально, и обучить ее с помощью Spark в кластере Hadoop или Mesos.

Отметим также, что в этом примере есть два скрытых слоя типа `GravesLSTM`, оба с функцией активации tanh. И выходной слой такой же, как и раньше, – `RnnOutputLayer` с функцией потерь MCXENT и функцией активации softmax. Напомним (см. главу 6), что функция softmax используется, когда требуется предсказать один из нескольких классов, а в данном случае мы предсказываем следующий символ (один из многих). Далее мы рассмотрим отличия этого кода от примера из главы 5.

Обучение, наблюдение за ходом работы и интерпретация результатов

Выше в этой главе мы уже обсуждали основные особенности перехода на платформу Spark, рассматривая следующий обобщенный фрагмент кода:

```
// Создать экземпляр TrainingMaster
int examplesPerDataSetObject = 1;
TrainingMaster trainingMaster = new ParameterAveragingTrainingMaster
        .Builder(examplesPerDataSetObject)
        .(other configuration options)
        .build();

// Создать экземпляр SparkDl4jMultiLayer
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, networkConfig,
    trainingMaster);
```

А ниже показано, как эта идея претворяется в модифицированном примере LSTM, где используется конкретный подкласс `ParameterAverageTrainingMaster` класса `TrainingMaster`, а конфигурация сети обернута объектом класса `SparkDl4jMultiLayer`, учитывающим нюансы работы в кластере Spark.

Ранее в этой главе мы уже рассмотрели параметры `TrainingMaster` и обертки Spark, поэтому не будем повторяться.

```
int examplesPerDataSetObject = 1;
ParameterAveragingTrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(examplesPerDataSetObject)
    .workerPrefetchNumBatches(2)    // Асинхронная предвыборка двух мини-пакетов
    .averagingFrequency(averagingFrequency)
    .batchSizePerWorker(batchSizePerWorker)
    .build();
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, conf, tm);
sparkNetwork.setListeners(Collections.<IterationListener>singletonList(new
    ScoreIterationListener(1)));

// Выполнить обучение, затем получить от сети и напечатать примеры
for (int i = 0; i < numEpochs; i++) {
    // Выполнить один период обучения. В конце периода мы имеем копию обученной сети
    MultiLayerNetwork net = sparkNetwork.fit(trainingData);

    // Выбрать несколько символов из сети (делается локально)
    log.info("Выбираются символы из сети при заданной инициализации \"" +
        (generationInitialization == null ? "" : generationInitialization) +
            "\"");
    String[] samples = sampleCharactersFromNetwork(generationInitialization,
        net, rng, INT_TO_CHAR,
        nCharactersToSample, nSamplesToGenerate);
    for (int j = 0; j < samples.length; j++) {
        log.info("----- Пример " + j + " -----");
        log.info(samples[j]);
    }
}
```

Помимо нескольких строк, необходимых для переноса примера на Spark, мы имеем такой же, как и раньше, цикл обучения модели, в котором используется все тот же метод `.fit()`. Взаимодействие с файловыми системами (локальной или HDFS), передача модели Spark по сети и прочие функции – все это скрыто под капотом.

Моделирование набора MNIST с помощью сверточной нейронной сети в кластере Spark

Вернемся к примеру сверточной нейронной сети (СНС) из главы 5 и модифицируем его для работы в кластере Spark. Пример 9.6 похож на программу моделирования набора MNIST, приведенную выше в этой главе, но теперь мы перешли на архитектуру СНС, более подходящую для изображений.

Пример 9.6 ❖ Моделирование набора MNIST в кластере Spark с помощью СНС

```
public class MnistExample {
    private static final Logger log = LoggerFactory.getLogger(MnistExample.class);

    public static void main(String[] args) throws Exception {

        // Создать контекст Spark и загрузить данные в память
        SparkConf sparkConf = new SparkConf();
        sparkConf.setMaster("local[*]");
```

```
sparkConf.setAppName("MNIST");
JavaSparkContext sc = new JavaSparkContext(sparkConf);

int examplesPerDataSetObject = 32;
DataSetIterator mnistTrain = new MnistDataSetIterator(32, true, 12345);
DataSetIterator mnistTest = new MnistDataSetIterator(32, false, 12345);
List<DataSet> trainData = new ArrayList<>();
List<DataSet> testData = new ArrayList<>();
while(mnistTrain.hasNext()) trainData.add(mnistTrain.next());
Collections.shuffle(trainData,new Random(12345));
while(mnistTest.hasNext()) testData.add(mnistTest.next());

// Получить обучающие данные. Отметим, что в реальных задачах
// использовать parallelize не рекомендуется
JavaRDD<DataSet> train = sc.parallelize(trainData);
JavaRDD<DataSet> test = sc.parallelize(testData);


// Сконфигурировать сеть (как стандартную сеть в DL4J)
int nChannels = 1;
int outputNum = 10;
int iterations = 1;
int seed = 123;

log.info("Построение модели....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.xterations(iterations) // количество итераций обучения
.regularization(true).l2(0.0005)
.learningRate(.01)// .biasLearningRate(0.02)
.weightInit(WeightInit.XAVIER)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.NESTEROVS).momentum(0.9)
.list()
.layer(0, new ConvolutionLayer.Builder(5, 5)
        // nIn и nOut задают глубину. Здесь nIn равно nChannels, а nOut –
        // число применяемых фильтров
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
.layer(2, new ConvolutionLayer.Builder(5, 5)
        // Отметим, что в последующих слоях задавать nIn не нужно
        .stride(1, 1)
        .nOut(50)
        .activation(Activation.IDENTITY)
        .build())
.layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
```

```
        .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
                .nOut(500).build())
        .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
            .NEGATIVELOGLIKELIHOOD)
                .nOut(outputNum)
                .activation(Activation.SOFTMAX)
                .build())
        .setInputType(InputType.convolutionalFlat(28,28,1)) // См. примечание ниже
        .backprop(true).pretrain(false).build();

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

        // Создать многослойную сеть с заданной конфигурацией для Spark
        ParameterAveragingTrainingMaster tm =
            new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetObject)
                .workerPrefetchNumBatches(0)
                .saveUpdater(true)
                .averagingFrequency(5)             // Каждый исполнитель выполняет обучение
                                                   // на 5 мини-пакетах, затем усредняет
                                                   // и перераспределяет параметры
                .batchSizePerWorker(examplesPerDataSetObject)    // Количество примеров
                                                                 // в одном мини-пакете
                .build();

        SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, net, tm);

        // Обучить сеть
        log.info("--- Начинается обучение сети ---");
        int nEpochs = 5;
        for(int i=0; i<nEpochs; i++ ){
            sparkNetwork.fit(train);
            System.out.println("----- Период " + i + " завершен -----");

            // Оценить с помощью Spark:
            Evaluation evaluation = sparkNetwork.evaluate(test);
            System.out.println(evaluation.stats());
        }

        log.info("***************Конец примера*******************");
    }
}
```

Далее мы разберем этот код по частям.

Конфигурирование задачи Spark и загрузка набора данных MNIST

Для работы со Spark необходимо создать объекты `SparkConf` и `JavaSparkContext`. Именно здесь указывается, должна ли задача выполняться в локальном режиме или в кластере[24].

[24] Если запустить пример как есть, то он будет выполняться локально, поскольку это режим работы Spark по умолчанию. Примечание: локальный режим Spark следует использовать только для разработки или тестирования. Для параллельного обучения на одной машине (например, оснащенной несколькими GPU) используйте класс `ParallelWrapper` (это быстрее, чем использовать Spark в локальном режиме).

При работе со Spark данные обычно хранятся в RDD. Ниже показано, как получить исходный набор MNIST от итератора, как в главе 5, и преобразовать данные в экземпляры класса JavaRDD.

```
DataSetIterator mnistTrain = new MnistDataSetIterator(32, true, 12345);
DataSetIterator mnistTest = new MnistDataSetIterator(32, false, 12345);
List<DataSet> trainData = new ArrayList<>();
List<DataSet> testData = new ArrayList<>();
while(mnistTrain.hasNext()) trainData.add(mnistTrain.next());
Collections.shuffle(trainData,new Random(12345));
while(mnistTest.hasNext()) testData.add(mnistTest.next());

JavaRDD<DataSet> train = sc.parallelize(trainData);
JavaRDD<DataSet> test = sc.parallelize(testData);
```

Мы собираем все данные MNIST в памяти, а затем с помощью контекста Spark создаем объекты JavaRDD.

Конфигурирование и обучение СНС LeNet

В данном примере конфигурируется та же СНС LeNet, что и в главе 5.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(iterations) // количество итераций обучения
    .regularization(true).l2(0.0005)
    .learningRate(.01)// .biasLearningRate(0.02)
    .weightInit(WeightInit.XAVIER)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list()
    .layer(0, new ConvolutionLayer.Builder(5, 5)
            // nIn и nOut задают глубину. Здесь nIn равно nChannels, а nOut -
            // число применяемых фильтров
            .nIn(nChannels)
            .stride(1, 1)
            .nOut(20)
            .activation(Activation.IDENTITY)
            .build())
    .layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
            .kernelSize(2,2)
            .stride(2,2)
            .build())
    .layer(2, new ConvolutionLayer.Builder(5, 5)
            // Отметим, что в последующих слоях задавать nIn не нужно
            .stride(1, 1)
            .nOut(50)
            .activation(Activation.IDENTITY)
            .build())
    .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
            .kernelSize(2,2)
            .stride(2,2)
            .build())
    .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
```

```
            .nOut(500).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
            .NEGATIVELOGLIKELIHOOD)
            .nOut(outputNum)
            .activation(Activation.SOFTMAX)
            .build())
    .setInputType(InputType.convolutionalFlat(28,28,1))
    .backprop(true).pretrain(false).build();
```

Мы используем тот же класс `MultiLayerConfiguration` и те же слои, что в примере обучения сети без Spark. Основное отличие – распараллеливание цикла обучения в кластере Spark – мы увидим в следующем разделе.

Обучение модели DL4J в кластере Spark похоже на то, что мы делали на одной машине, но есть и различия. В приведенном ниже фрагменте обратите внимание на отличие кода обучения от примера из главы 5.

```
// Создать многослойную сеть с заданной конфигурацией для Spark
ParameterAveragingTrainingMaster tm =
    new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetObject)
        .workerPrefetchNumBatches(0)
        .saveUpdater(true)
        .averagingFrequency(5)
        .batchSizePerWorker(examplesPerDataSetObject)
        .build();

SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, net, tm);

// Обучить сеть
log.info("--- Начинается обучение сети ---");
int nEpochs = 5;
for(int i=0; i<nEpochs; i++ ){
    sparkNetwork.fit(train);
    System.out.println("----- Период " + i + " завершен -----");

    // Оценить с помощью Spark:
    Evaluation evaluation = sparkNetwork.evaluate(test);
    System.out.println(evaluation.stats());
}
```

Основные отличия (как и в двух предыдущих примерах):

- использование класса `ParameterAveragingTrainingMaster`;
- использование класса `SparkDl4jMultiLayer`.

В остальном изменения минимальны.

Приложение A

Что такое искусственный интеллект?

Купер: А скажи-ка, ТАРС, какой у тебя параметр честности?
ТАРС: 90 процентов.
Купер:90 процентов?
ТАРС: Абсолютная честность не является ни самым дипломатичным, ни самым безопасным способом общения с эмоциональными существами.
Купер: 90 процентов, значит.

– *Сцена из фильма «Интерстеллар»*

Искусственный интеллект (ИИ) как дисциплина так же стар, как сама философия. Со временем он развивался, однако мы все никак не можем определить его место в обществе, не говоря уже об экзистенциальных последствиях для всего человечества. Одни из самых ярких строк о зарождении ИИ принадлежат перу Памелы Маккордак, написавшей, что ИИ начался с «античного желания уподобиться богам»[1].

Маккордак писала высокоинтеллектуальную прозу, а многие современные маркетологи, хоть и играют на таких же духоподъемных материях, на практике продвигают в жизнь куда более скромные функции. Глубокое обучение возникает в контексте обсуждения ИИ настолько регулярно, что становится трудно отделить одно от другого.

Мы добавили это приложение, потому что специалисту-практику часто приходится беседовать с заказчиками, руководителями и менеджерами о том, чем им может быть полезно глубокое обучение и каково его место в общем ландшафте ИИ. Мы расскажем как об истории дисциплины ИИ (чтобы иметь контекст), так и о наших дискуссиях с заказчиками и коллегами. Мы хотим снабдить вас, практика, средствами для ведения беседы на тему глубокого обучения и формирования реалистичных ожиданий у участников проекта, чтобы они могли занять позицию, с которой будет легче внедрять глубокое обучение. Короче говоря, вокруг ИИ циркулирует слишком много слухов, но в конце концов рынок все расставит по своим местам.

Мы хотели также, чтобы это приложение оказалось нескучным и будило воображение практика или исследователя, не позволяя ему, однако, оторваться от

[1] McCorduck, 2004. Machines Who Think (2nd ed.).

реальности. Мы дадим основные определения, совершим краткий экскурс в историю ИИ, а затем заглянем вперед и попробуем предположить, куда все это может завести. Надеемся, что поможем читателю не наступать на грабли и не дать заглохнуть своим проектам в области машинного обучения – для этого нужно ответственно ставить цели и не завышать ожидания.

Положение на данный момент

Тема этой книги, глубокое обучение, в СМИ и маркетинге постоянно сопровождается словами «искусственный интеллект». Определения не устоялись, поэтому очень трудно обсуждать эти вопросы с практиками и сторонами, заинтересованными во внедрении. Отделы маркетинга цепляют ИИ к каждой новой волне потребительского ажиотажа. Вот несколько тем из недавнего прошлого:

- интеллектуальные энергосистемы;
- облако;
- большие данные.

Работая в подобных предметных областях или в области глубокого обучения, мы должны отделять реальность от маркетинговой шелухи. Для этого нужно знать историю предмета и опираться на твердые определения. Начнем с того, что мы понимаем под глубоким обучением, а затем перейдем к определению ИИ.

Определение глубокого обучения

В главах 1 и 3 мы предложили рабочее определение глубокого обучения, сводящееся к описанию нейронных сетей со следующими свойствами:

- больше нейронов, чем в сетях предыдущего поколения;
- более сложные способы соединения слоев;
- «кембрийский» взрыв вычислительной мощности, доступной для обучения;
- автоматическое обучение признакам.

Эти сети могут выполнять те же функции, что и другие модели машинного обучения (регрессия, классификация), но также доказали свою полезность при решении следующих задач:

- порождающее моделирование (например, порождение изображений и текста);
- распознавание речи;
- распознавание изображений.

Еще одна особенность, внесшая немалый вклад в распространение глубокого обучения, – способность автоматически обучаться признакам (в противовес их ручному конструированию), ничего не понимая в предметной области. Это стало причиной появления многих новых приложений и возбуждает воображение людей, далеких от технологии. Но само по себе глубокое обучение не обладает такими высокоуровневыми функциями, как «автоматическое определение самого интересного вопроса к набору данных», не говоря уже об операциях, требующих наличия сознания.

Определение искусственного интеллекта

История ИИ изобилует мифами, выдумками и не в меру рьяными маркетологами, стремящимися присосаться к известиям о новейших технологиях. Чтобы опреде-

лить ИИ, нам необходим контекст, включающий историю изучения интеллекта, современную аргументацию и развитие дисциплины со временем.

Действуя в соответствии с этим планом, рассмотрим, как эта дисциплина зародилась и развивалась на протяжении последних 60 лет.

Изучение интеллекта

Формально изучение интеллекта началось в 1956 г. в Дартмуте, но на самом деле ему уже по меньшей мере 2000 лет. Предметом изучения является попытка понять, что такое разумное существо вообще и такие его частные проявления, как:

- зрение;
- обучение;
- запоминание;
- рассуждение.

Эти функции считаются составными частями интеллекта, их понимание позволило бы приблизиться к ответу на вопрос, что такое интеллект. Следы изучения интеллекта присутствуют во всей истории человечества, нужно только поискать. Вот несколько сложившихся со временем компонентов исследования интеллекта:

- философия (400 до н. э.) – философы первыми высказали предположение, что разум – это механическое устройство, кодирующее знания в какой-то форме внутри мозга;
- математика – математики разработали основные идеи манипулирования логическими высказываниями, а также заложили основы рассуждений об алгоритмах;
- психология – эта область науки развивает идею о том, что мозг людей и животных способен обрабатывать информацию;
- информатика – разрабатывает оборудование, структуры данных и алгоритмы, необходимые для понимания основных механизмов работы мозга.

Сегодняшние исследования и применения ИИ стоят на этом фундаменте. В типичной работе по ИИ исследуется либо поведение, либо мышление смоделированной интеллектуальной системы. Мы видим это в таких приложениях, как машинное обучение, системы накопления знаний и игры (например, шахматы и го).

Однако достижения в области ИИ имеют ограничения. По-прежнему не существует хороших моделей высших функций мозга, в т. ч. сознания. Науке еще только предстоит понять, где в мозгу находится сознание. В связи с этим возникает вопрос, а действительно ли сознание является функцией мозга, но споры на эту тему мы оставим философам и специалистам по информатике.

Для самостоятельного изучения

Одним из лучших (если не лучшим) сочинением на тему ИИ является книга Stuart Russell, Peter Norvig «Artificial Intelligence: A Modern Approach»[2]. Мы всячески рекомендуем ее для получения более полного представления о глубине и истории развития ИИ.

Когнитивный диссонанс и современные определения

Имея дело с такой темой, как ИИ, которая неразрывно связана со многими ключевыми вещами, от которых зависит функционирование общества, мы естествен-

---

[2] *Рассел С., Норвиг П.* Искусственный интеллект. Современный подход. М.: Вильямс, 2017.

но испытываем когнитивный диссонанс, пытаясь найти, на что же опереться. Бо Кронин[3] пишет:

> Подобно интернету вещей, Web 2.0 и большим данным, ИИ обсуждается во многих различных контекстах людьми с разной мотивацией и образованием: учеными, бизнесменами, журналистами и технологами. Неудивительно, что смысл ИИ, как и всех прочих расплывчатых технологий, трудно зафиксировать – каждый видит то, что хочет видеть.

Отчасти проблема разных точек зрения на определение интеллекта состоит в тесном переплетении с определением того, что такое сознание, и с другими философскими и религиозными вопросами (например, «что есть душа?»). Тут мы вступаем на заминированную территорию – всякие дебаты по поводу определения души обставлены сложностями. Лучшие на сегодняшний день карты определения интеллекта можно сравнить с областью, которая на старых глобусах обозначалась надписью «тут обитают драконы». Но раз мы не понимаем, что такое естественный интеллект, определить искусственный становится еще труднее.

Д-р Джейсон Бэлбридж[4], рассуждая на тему ИИ и машинного обучения, так писал о противоречивом восприятии понятий:

> Независимо от технических нюансов определения ИИ, я практически уверен, что когда неспециалист слышит выражение «искусственный интеллект», он представляет себе небиологические объекты, которые общаются с людьми, как мы общаемся между собой.
> Люди не думают ни об экспертных системах, способных анализировать проблему из сложной предметной области и предлагать интересный план действий, ни об алгоритмах машинного обучения, которые находят чарующие закономерности в грудах данных.
> В общественном сознании легко ликвидируется разрыв между этими двумя совершенно разными уровнями технического и научного воплощения ИИ.

Далее д-р Бэлбридж переходит к различию между глубоким обучением и полной искусственной моделью биологического мозга:

> Несмотря на достигнутый прогресс, мы – хорошо это или плохо – все еще далеки от создания сознающих себя машин. Глубокое обучение возникло из изучения того, как функционируют нейроны человека, но, насколько мне известно, искусственные нейронные сети до сих пор не имеют ничего общего с архитектурой разума существ из плоти и крови.

Нам с таким трудом даются эти определения, потому что они действительно сложны и затрагивают много аспектов, рассматриваемых с разных точек зрения. Сделаем еще один шаг к формулировке определения, попытавшись сегментировать проблему и разбить сегменты на более простые части.

Франсуа Шолле (Francois Chollet) недавно сделал такое знаменательное замечание в Твиттере[5]:

> Искусственный интеллект – плохо определенная штука, которой многие приписывают совершенно нереальные способности. Этот верный путь к беде.

---

[3] https://www.oreilly.com/ideas/ais-dueling-definitions.

[4] https://www.peoplepattern.com/post.html#!/ai-not-yet-but-machine-learning-is-here-and-now.

[5] https://twitter.com/fchollet/status/734919468908875776.

И в следующем твите[6]:

> Отчасти проблема в том, что некоторые компании и журналисты накручивают публику, размывая грань между научной фантастикой и реальностью. Продается же.

Шолле далее пишет[7], что мы должны «определить», о чем идет речь:

> Говоря «ИИ», *определите*, о чем вы говорите. Явно укажите, что он может, а чего не может. Избегайте аналогий с мозгом.

Это здравый совет, и индустрия должна как следует постараться и дать четкие определения.

Чем ИИ не является. Люди, заявляющие, что машинное обучение и есть ИИ, оказывают всей компьютерной науке дурную услугу. Машинное обучение – это классификация и регрессия, оно ни в коей мере не претендует на роль всезнающей, осознающей себя системы, способной помочь жаждущему читателю в решении его маркетинговой проблемы. Как заметил Франсуа Шолле, (пока) лучше избегать аналогий с мозгом.

Нередко маркетологи преподносят ИИ как приложение, знающее ответы на все вопросы. Это не так, по крайней мере в обозримом будущем.

Смена ориентиров. Психологи по привычке с презрением относятся к метафоре человеческого мозга как компьютера. В эссе 2016 года Роберт Эпштейн писал[8]:

> Сколько бы ни старались когнитивные психологи, они никогда не отыщут, где в мозгу находится копия 5-й симфонии Бетховена – как и копии слов, картин, грамматических правил или любых внешних раздражителей.

К сожалению, д-р Эпштейн не видел визуализаций фильтров CHC, приведенных в главе 4. Главный аргумент в его статье звучит так:

> Ваш мозг не обрабатывает информацию, не извлекает знания и не хранит воспоминания. Короче говоря, ваш мозг – не компьютер.

Это не новая мысль, за последние 60 лет она многократно встречалась в дисциплинах, не имеющих отношения к информатике. Рассел и Норвиг пишут в своей книге об ИИ:

> В общем и целом сообщество интеллектуалов предпочитало верить, что «машина никогда не сможет сделать X».

А далее приводят примеры, как исследователи ИИ систематически демонстрировали один X за другим. Таким образом, изучение ИИ и соответствующие определения долгое время страдали от «смены ориентиров» – представлений индустрии о том, что же все-таки понимать под «искусственным интеллектом».

Сегментирование определений ИИ. Полезно будет выписать различные существующие в настоящее время точки зрения на ИИ. Бо Кронин в своем эссе[9] приводит четыре группы определений:

---

[6] https://twitter.com/fchollet/status/734919888876097536.

[7] https://twitter.com/fchollet/status/734920203478274048.

[8] https://aeon.co/essays/your-brain-does-not-process-information-and-it-is-not-a-computer.

[9] https://www.oreilly.com/ideas/ais-dueling-definitions.

- ИИ как собеседник:
  - HAL, Siri, Cortana, Watson;
  - ИИ – виртуальный партнер по общению;
  - ограниченная способность к рассуждениям;
- ИИ как андроид:
  - машина, имеющая форму гуманоида;
  - ИИ в механическом воплощении;
  - примерами могут служить Терминатор и C-3PO;
  - похож на собеседника, но в гуманоидном теле;
- ИИ как механизм рассуждений:
  - пионеров ИИ привлекали более утонченные и возвышенные задачи: игра в шахматы, нахождение логических доказательств и планирование сложных действий;
  - все еще не может справиться с задачами, которые легко решают дети;
- ИИ как анализатор больших данных:
  - сравнительно недавнее определение;
  - многие стали говорить о построении «моделей ИИ».

Взглянем на эти сегментированные определения критически.

Критические замечания о сегментах. ИИ как собеседник и ИИ как анализатор больших данных – недавние определения, появившиеся вследствие включения методов машинного обучения в коммерческие продукты. ИИ как собеседник может выполнять простые действия, опираясь на распознавание речи. Применяя сочетание машинного (или глубокого) обучения для преобразования голоса в текст и технологий обработки естественного языка (ОЕЯ), собеседник определяет, чего хочет пользователь. У ИИ как собеседника ограниченные способности к рассуждению, потому что он обычно полагается на отдельную систему, на вход которой подаются результаты распознавания речи и ОЕЯ. Зачастую эта система не сложнее классической экспертной системы с базой правил.

Хотя поначалу пользователей может заинтересовать беседа с системой, и они даже могут поверить в ее «интеллект», довольно быстро ограничения взаимодействия становятся очевидны. В конечном счете ИИ как собеседник – это хорошо сконструированная комбинация методов машинного обучения, которая со временем становится достаточно качественной для включения в полезные потребительские продукты.

ИИ как андроид – интересное воплощение концепции, но, по сути дела, оно опирается на комбинацию подсистем машинного обучения – точно так же, как собеседник. ИИ как механизм рассуждения – классическая реализация ИИ, но в последние годы интерес к ней с точки зрения интеграции с промышленными продуктами остается на постоянном уровне. Она по-прежнему является важным компонентом интеллектуальных систем, объединяющим различные части для создания нового качества, как в примере с собеседником.

«ИИ как анализатор больших данных» – спорное использование термина, набравшее популярность в последние несколько лет (2010–2015). Нередко маркетологи создают новый образ продукта, обзывая применение машинного обучения к пользовательским данным «искусственным интеллектом». Хуже того, иногда в эту категорию помещают продукты, реализующие обычные функции бизнес-

аналитики. Машинное (или глубокое) обучение само по себе не должно считаться разновидностью ИИ. Но это полезная составная часть интеллектуальной системы.

Сдержанный маркетинг глубокого обучения

Следует проявлять сдержанность, называя модель машинного обучения (глубокую или нет) «искусственным интеллектом». Неумеренное расхваливание может привлечь инвестиции на начальном этапе, но в конечном счете повредит проекту.

Пятое амбициозное определение ИИ. Пытаясь ответить на вопрос «что такое искусственный интеллект», полезно задать другой вопрос: «что могло бы раз и навсегда положить конец спорам о том, что такое искусственный интеллект?»

Если бы нам предъявили совершенный, наделенный сознанием, осознающий себя интеллект, который понимает наш мир (и данные) гораздо лучше любого человека, то, вероятно, мы назвали бы его «настоящим Искусственным Интеллектом». Или чужаком.

К сожалению, погоня за миражами несет с собой груз нереалистичных ожиданий, который неизменно погребает под собою отрасль, каким бы ощутимым ни был прогресс.

Зимы ИИ

ИИ пережил много взлетов и падений интереса и финансирования. Причиной падения всегда были нереалистичные и неумеренные обещания, за которыми предсказуемо следовали разочаровывающие результаты. Такие периоды получили названия «зимы ИИ». Им сопутствовало уменьшение финансирования научных исследований, снижение интереса со стороны венчурных фондов и клеймо позора на любых попытках маркетинга с упоминанием «искусственного интеллекта».

В результате настоящие технические достижения (например, распознавание речи или оптическое распознавание символов) включались в продукты под другой маркой.

Первая зима ИИ (1974–1980). Предвестником первой зимы ИИ стало невыполнение обещаний по поводу машинного перевода. В 1970-х годах интерес к коннекционизму (нейронным сетям) угас, а исследования по распознаванию речи, которым сулили оглушительный успех, так и не оправдали ожиданий.

В 1973 году агентство DARPA сократило ассигнования на научные исследования в области ИИ. В отчете Лайтхилла, опубликованном в Великобритании, эта область исследований была подвергнута резкой критике, и финансирование было урезано еще сильнее.

Вторая зима ИИ (конец 1980-х годов). В конце 1980 – начале 1990-х годов чрезмерные ожидания возлагались на экспертные системы и LISP-машины. Но все обернулось крахом. Фонд Стратегической инициативы в области компьютерной техники отказался продолжать финансирование по истечении этого цикла. Программа разработки компьютеров пятого поколения также закончилась провалом.

Общие черты зим ИИ

Обеим зимам свойственны общие черты. Все начиналось с хвастливых обещаний будущих успехов, даваемых отраслью. Когда условия созревают, мы наблюдаем, как рекламная кампания достигает пика и деньги, выделяемые на исследования

в академических и отраслевых институтах, устремляются на финансирование ИИ. Немногие реальные проекты, стоящие на прочном технологическом фундаменте, хотя бы частично достигают заявленных целей и решают конкретные задачи. Однако большая часть рекламных обещаний остается невыполненной, и наступает пора избавления от иллюзий.

Зима убивает слабых.

Но несколько интересных приложений переживает эту пору и под другой маркой («распознавание речи») включается в другие проекты в виде дополнительных усовершенствований, обычно проходящих по разряду «латентного интеллекта». Так происходило не раз:

- информатика;
- машинное обучение;
- системы на основе базы знаний;
- управление бизнес-правилами;
- системы обработки знаний;
- интеллектуальные системы;
- вычислительный интеллект.

Изменение названия может быть отчасти связано с тем, что авторы считают свою дисциплину фундаментально отличающейся от ИИ. Правда и то, что новое название привлекает финансирование, избавившись от клейма ложных обещаний, ассоциируемых со словами «искусственный интеллект».

Приведем интересную историю о конференции по ИИ, которая состоялась в 1980-х годах:

> На конференции Роджер Шэнк и Марвин Мински – два ведущих исследователя в области ИИ, переживших первую зиму, – предупредили деловое сообщество, что энтузиазм по поводу ИИ в 1980-х годах вышел из-под контроля и что разочарование неминуемо. Три года спустя начался коллапс индустрии ИИ, в которую были вложены миллиарды долларов.

Чем подпитывается интерес к ИИ сегодня?

В наши дни интерес к ИИ зиждется на трех китах:

1) резкий скачок в технологиях компьютерного зрения в конце 2000-х;
2) волна интереса к большим данным, поднявшаяся в начале 2010-х;
3) достижения в применении глубокого обучения, продемонстрированные крупнейшими технологическими компаниями.

В 2006 году группа Джеффри Хинтона из Торонтского университета опубликовала основополагающую работу по глубоким сетям доверия[10]. Это заронило в отрасль творческую искру, потенциально способную улучшить состояние дел. В последующие десять лет ведущие журналы захлестнул вал работ по глубокому обучению. В результате верность начала расти во многих областях, помимо компьютерного зрения, и очень скоро глубокое обучение стало преобладать в ландшафте прикладного машинного обучения.

Крупные интернет-компании, в т. ч. Google, Facebook и Amazon, следят за публикациями в ведущих журналах в поисках новых идей. Они заметили работы Яна

---

[10] Hinton, Osindero and Teh, 2006. A fast learning algorithm for deep belief nets // https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf.

Лекуна, Хинтона и других ученых и приступили к реализации этих идей в своих продуктах. Новые приложения (например, улучшенное распознавание лиц или интеллектуальный помощник Alexa, разработанный компанией Amazon) обрели широкое признание в СМИ, пишущих на темы технологий.

В середине 2000-х технологии хранения и ETL, разработанные крупными интернет-компаниями с Западного побережья США, стали превращаться в проекты с открытым исходным кодом, например Hadoop и MongoDB.

Занимаясь системами хранения и ETL, интернет-компании (Google, Yahoo! и другие), попутно разработали новые методы машинного и глубокого обучения, позволяющие в полной мере воспользоваться новыми большими наборами данных.

Традиционные корпорации из списка *Fortune 500* начали внедрять большие онлайновые распределенные системы в начале 2010-х годов, чтобы хранить постоянно растущие транзакционные наборы данных. Эти корпорации обычно следуют за компаниями с Западного побережья с отставанием на 5–10 лет. Закономерно, что у компаний из списка *Fortune 500* возник интерес к глубокому обучению и к системам, которые позволили бы традиционному бизнесу оправдать инвестиции в большие данные.

Комбинация трех вышеупомянутых факторов с такими крупными публичными успехами, как системы Watson (победившая в игре *Jeopardy*[11]), AlphaGo (победившая человека в игре го) и беспилотные автомобили, созданные Google, создает условия, при которых энтузиазм настолько велик, что не замечает трудностей предстоящего пути.

Вокруг все шире распространяющегося ИИ наблюдается прилив радостных ожиданий. Увы, в этих циклах прилив рано или поздно сменяется отливом. Существуют реальные приложения, в которых используются сложные наборы данных для глубокого обучения. Перечислим некоторые из них:

- здравоохранение (например, прогнозирование срока пребывания пациента в больнице);
- розничная торговля (анализ покупательских привычек);
- телекоммуникации и финансовые сервисы (например, поиск признаков мошенничества в банковских транзакциях).

Кое-что из этого мы рассматривали в книге. Практику, внедряющему идеи глубокого обучения и ИИ, мы рекомендуем искать подобные реальные задачи, стоящие на «твердой почве». Мы говорим о «твердой почве», потому что хотелось бы, чтобы у наших коллег-практиков было, на чем утвердиться, когда прилив схлынет.

Зима не за горами

Глубокое обучение как таковое нашло опору в реальности. Это инфраструктура для лидирующей в отрасли методологии моделирования нейронных сетей на сложных типах данных. Само по себе оно не удовлетворяет пятому амбициозному определению ИИ, так что на этот счет особо переживать не надо.

Нас интересуют системы, которые в 2016 году подавались как «искусственный интеллект для X», хотя по существу это обычное применение машинного обуче-

---

[11] Аналог «Как стать миллионером». – *Прим. перев.*

ния. AlphaGo стала выдающимся достижением в играх, но на примере DeepBlue и Chess мы видели, что прогресс в умении играть в игры не всегда влечет за собой успех в коммерческих приложениях[12].

К сожалению, отделы маркетинга идут по той же дорожке, которая привела к двум предыдущим зимам ИИ. Но, как случалось и раньше, угли настоящих достижений будут согревать истинных энтузиастов и целеустремленных исследователей во время близящейся третьей зимы ИИ.

[12] Однако *Jeopardy* не выглядит «решенной проблемой».

Приложение B

RL4J и обучение с подкреплением

Рубен Фижель

Введение

Мы начнем это приложение с введения в *обучение с подкреплением*, а затем подробно опишем глубокие Q-сети (DQN), получающие на входе пиксели. И в заключение приведем пример использования библиотеки RL4J.

Обучение с подкреплением – захватывающая область машинного обучения. Ее смысл заключается в нахождении эффективной стратегии в заданной среде. Неформально это очень похоже на *условные рефлексы по Павлову*: за определенное поведение назначается вознаграждение, и со временем агенты обучаются повторять это поведение, чтобы увеличить вознаграждение.

Марковский процесс принятия решений

Формально окружающие условия описываются *марковским процессом принятия решений* (Markov Decision Process – MDP). За этим пугающим названием стоит всего лишь следующая пятерка:

- множество состояний S (например, в шахматах состояние – это расположение фигур на доске);
- множество возможных действий A (в шахматах множество всех допустимых ходов во всех возможных позициях, например e4–e5);
- условное распределение P(s′|s,a)$P(s'|,a)$ следующего состояния при условии текущего состояния и действия. В детерминированной среде, например в шахматах, существует только одно состояние s′ с вероятностью 1, а все остальные имеют вероятность 0. Но в стохастической (с элементами случайности, как при бросании монеты) среде распределение не такое простое;
- функция вознаграждения при переходе из состояния s в s′: $R(s,s')$ (например, в шахматах она может быть равна +1 для последнего хода, приведшего к победе, −1 для последнего хода, приведшего к поражению, и 0 в остальных случаях);
- коэффициент обесценивания γγ, выражающий предпочтение текущим вознаграждениям по сравнению с будущими (идея, широко распространенная в финансах[1]).

---

[1] https://en.wikipedia.org/wiki/Discounted_utility.

Использование множества действий A_S

Обычно удобнее использовать множество действий A_S, т. е. множество ходов, допустимых в данном состоянии, а не полное множество A. По определению A_S – множество таких элементов A, что P(s′|s,a) > 0.

Под марковским свойством (см. рис. B.1) понимается отсутствие памяти. В любом состоянии прошлая история (состояния, посещенные ранее) не должна оказывать влияния на будущие переходы и вознаграждения, значение имеет только текущее состояние.

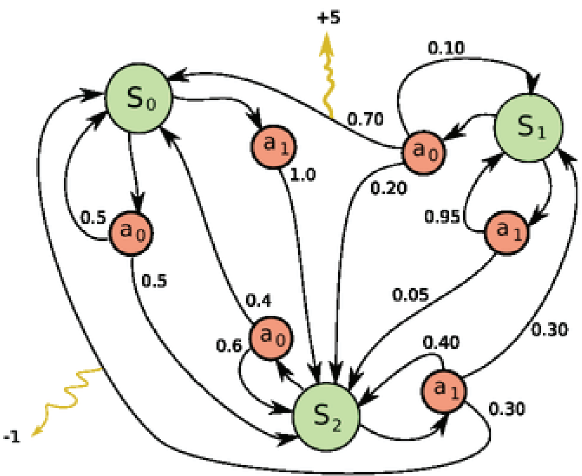


Рис. B.1 ❖ Схема марковского процесса принятия решений

Терминология

Прежде чем двигаться дальше, дадим определения некоторых терминов:

- конечные (терминальные) состояния – состояния, в которых нет ни одного возможного действия, называются конечными, или терминальными;
- эпизодом называется полная последовательность состояний от начального к конечному:

  $s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_n;$

- накопительным вознаграждением называется обесцененная сумма вознаграждений, полученных на протяжении эпизода:

  $R = \sum_{t=0}^{n} \gamma^t r_{t+1};$

- политикой называется стратегия выбора агентом действия в каждом состоянии. Обозначается буквой π;
- оптимальной называется теоретическая политика, которая максимизирует математическое ожидание накопительного вознаграждения. Из определения математического ожидания и закона больших чисел следует, что при достаточном количестве эпизодов такая политика дает максимальное

среднее накопительное вознаграждение. Точное вычисление оптимальной политики может оказаться нерешаемой задачей.

Цель обучения с подкреплением – обучить агента таким образом, чтобы он выбирал политику, максимально близкую к оптимальной.

Различные варианты

Qui peut le plus peut le moins
(кто способен на большее, тот способен и на меньшее).

Безмодельное обучение

Условное распределение и функция вознаграждения составляют модель среды. В игре в нарды модель известна (переход определяется броском костей, и каждое вознаграждение в результате перехода можно предсказать, не реализуя его, поскольку можно вычислить новую позицию). В алгоритме TD-gammon этот факт используется для обучения V-функции (см. раздел «Q-обучение» ниже).

Некоторые алгоритмы обучения с подкреплением могут работать без модели. Тем не менее для обучения наилучшей стратегии им дополнительно необходимо обучиться модели в процессе обучения. Это называется *безмодельным обучением с подкреплением*. Безмодельные алгоритмы очень важны, потому что в эту категорию попадает подавляющее большинство реальных сложных задач. Кроме того, отсутствие модели – это просто дополнительное ограничение. Это более мощный класс алгоритмов, поскольку он является надмножеством обучения с подкреплением на основе модели.

Наблюдаемое состояние

Вместо доступа к истинному состоянию нам может быть предоставлен доступ только к частичному наблюдению состояния. Та же идея лежит в основе *скрытой марковской цепи*. Например, наше поле зрения дает лишь очень ограниченное наблюдение полного состояния Вселенной (положение и энергия каждой элементарной частицы во Вселенной). По счастью, частичное наблюдаемое состояние может быть сведено к полному, если воспользоваться историей (состоянием становится совокупность всех предыдущих состояний).

Тем не менее всю историю обычно не аккумулируют. Либо учитываются последние h наблюдений (с помощью скользящего окна), либо используется рекуррентная нейронная сеть, которая обучается тому, что хранить в памяти, а что забыть (так работает долгая краткосрочная память – LSTM).

Допуская некоторую вольность речи ради совместимости с существующими обозначениями, историю (даже усеченную) также будем называть «состоянием» и обозначать S_t.

Однопользовательские и состязательные игры

Однопользовательская игра естественно переводится на язык MDP. Состояния представляют те моменты, в которые игру контролирует игрок. Наблюдения этих состояний – вся накопленная между состояниями информация (например, сколько кадров разделяют кадры контроля). Действие – это все возможные

команды, находящиеся в распоряжении игрока (в игре *Doom*: вверх, вправо, влево, стрелять и т. д.).

Обучение с подкреплением можно применить также к состязательным играм, заставив агента играть с самим собой. Часто в этом случае существует равновесие Нэша[2], когда в ваших интересах играть так, как будто вам противостоит идеальный игрок. Это имеет смысл, например, в шахматах. В заданной позиции хороший ход против мастера будет хорошим и против начинающего. Каким бы ни был текущий уровень агента, играя против самого себя, агент все же собирает информацию о качестве своих предыдущих ходов (они считаются хорошими в случае выигрыша и плохими в случае проигрыша).

Разумеется, информация – в контексте нейронных сетей градиент – будет более высокого качества, если агент с самого начала играл против очень хорошего агента. Но удивительно, что агент все же может повысить свой уровень, играя против самого себя, т. е. агента того же уровня. Именно такой метод обучения использовался в системе AlphaGo[3] (созданном компанией DeepMind агенте для игры в го, победившем чемпиона мира). Политика была первоначально обучена на наборе данных, состоявшем из партий мастеров. Затем обучение с подкреплением в играх с самим собой использовалось для повышения уровня. В итоге агент стал лучше, чем политика, которой он обучился на исходном наборе данных, и сумел победить лучшего из мастеров. Для вычисления окончательной политики команда AlphaGo использовала градиент политики в сочетании с поиском по дереву методом Монте-Карло, применяя мощную вычислительную технику.

Этот случай несколько отличается от обучения на пикселях. Во-первых, поскольку размерность входа не так велика, многообразие[4] гораздо ближе к своему пространству погружения. Тем не менее сверточный слой все-таки использовался, чтобы эффективно воспользоваться локальностью некоторых позиций на части доски. Во-вторых, обучение AlphaGo не безмодельное (оно детерминированное).

Q-обучение

> Я не любитель теории вероятностей, я ненавидел ее с того самого момента, как наш дорогой друг Макс Борн дал ей путевку в жизнь. Ибо благодаря ей складывалось впечатление, будто в принципе все легко и просто, все приглажено, тогда как истинные проблемы при этом оставались скрытыми.
>
> – *Эрвин Шрёдингер*

От политики к нейронным сетям

Наша цель – обучиться оптимальной политике π^*, которая максимизирует следующую функцию:

$$E[R_0] = E\left[\sum_{t=0}^{n} \gamma^t r_{t+1}\right].$$

2 https://ru.wikipedia.org/wiki/Равновесие_Нэша.

3 https://deepmind.com/research/alphago/.

4 http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/.

Введем в рассмотрение вспомогательную функцию:

$V_\pi(s) = E\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n r_n | s_t = s$, политика, преследуемая в каждом состоянии, обозначаемая $\pi\}$.

Это ожидаемое накопительное вознаграждение, получаемое при начальном состоянии s и политике π. Рассмотрим оракул

$V_{\pi^*}(s)$.

Он дает функцию V оптимальной политики. Зная ее, мы можем найти оптимальную политику, определив такую политику, которая из всех действий, возможных в текущем состоянии, выбирает то, что максимизирует математическое ожидание $V_{\pi^*}(s)$. Это жадное поведение. Оптимальной является жадная политика относительно V_{π^*}:

$\pi^*(s)$ выбирает a такое, что $a = \mathrm{argmax}_a[E_\pi(r_t + \gamma V(s_{t+1}) | s_t = s, a_\mathrm{t} = a)]$.

Внимательный читатель заподозрит, что здесь что-то не так. В безмодельном случае мы не можем предсказать состояние s_{t+1} по s_t, потому что игнорируем модель переходов. Даже при наличии оракула наша модель все равно не вычислима!

Чтобы разрешить эту досадную проблему, введем еще одну вспомогательную функцию – Q-функцию:

$Q_{\pi^*}(s, a) = E_\pi[r_t + \gamma V_{\pi^*}(s_{t+1}) | s_t, a_t = a]$.

При жадном определении политики имеем:

$V_\pi(s_t) = \max_a Q_\pi(s_t, a)$.

Теперь предположим, что вместо оракула для V мы имеем оракул для Q. Тогда π^* можно переопределить следующим образом:

$\pi^*(s)$ выбирает a такое, что $a = \max_a[Q_{\pi^*}(s, a)]$.

Вот теперь нет никаких невычислимых математических ожиданий, и все хорошо.

Но на самом деле мы просто переместили математическое ожидание внутрь оракула. И, увы, в реальном мире оракулов не существует.

Трюк заключается в том, что мы свели абстрактное понятие политики к числовой функции, которая благодаря математическому ожиданию может быть относительно «гладкой» (непрерывной). К счастью, в нашем распоряжении имеется инструмент для аппроксимации таких сложных функций: нейронные сети.

Нейронные сети – универсальные аппроксиматоры функций. С их помощью можно аппроксимировать любую непрерывно дифференцируемую функцию. Однако они застревают в локальных экстремумах, и многие доказательства сходимости, применяемые в теории обучения с подкреплением, перестают работать, когда мы вводим в рассмотрение нейронные сети. Дело в том, что их обучение не является ни детерминированным, ни ограниченным. Тем не менее в большинстве случаев при правильно подобранных гиперпараметрах нейронные сети оказываются на удивление мощным инструментом[5]. Использование глубокого

[5] http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

обучения в сочетании с обучением с подкреплением называется *глубоким обучением с подкреплением*.

Перебор политик

Сейчас наши знания о машинном обучении, да и простой здравый смысл подсказывают, что в описанном подходе чего-то не хватает. Нейронные сети могут аппроксимировать функции при наличии меток. К несчастью, у оракула ничего не спросишь, поэтому добывать метки придется иным путем.

Вот тут-то и вступает в игру волшебный метод Монте-Карло, опирающийся на повторные случайные выборки для вычисления оценки. (Знаменитый пример[6] – вычисление числа π.)

Если случайным образом продолжать игру, начиная с заданного состояния, то лучшие состояния должны в среднем получить большее вознаграждение (согласно закону больших чисел[7]). Поэтому, ничего не зная о среде, мы можем собрать кое-какую информацию об ожидаемом значении состояния. Например, при игре в покер игрок, получивший лучшие карты, выигрывает чаще, чем игроки, у которых карты хуже, даже если каждое решение принимается случайно. Поиск по дереву методом Монте-Карло также основан на этом свойстве (шокирует, да?). Это фаза исследования, которая ведет к обучению без учителя и позволяет выделять осмысленные метки.

Более формально, пусть даны политика π, состояние s и действие a. Тогда, чтобы получить аппроксимацию функции $Q(s, a)$, мы производим выборку из нее согласно определению:

$$Q_\pi(s, a) = E[r_t + \gamma r_{t+1} + \ldots + \gamma^n r_n \mid s_t = s, a_t = a].$$

Проще говоря, мы можем получить метку для $Q_\pi(s, a)$, сыграв достаточное количество партий с начальным состоянием s, согласно политике π.

На рис. B.2 показано агрегирование сигналов.

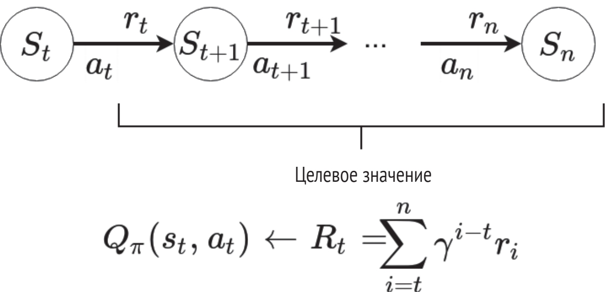


Рис. B.2 ❖ Один сигнал

Собственно обучение производится методом стохастического градиентного спуска (СГС) на пакетах с метками. В качестве функции потерь мы берем средне-

---

6 http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html.

7 https://ru.wikipedia.org/wiki/Закон_больших_чисел.

квадратическую ошибку (потерю по норме L2), скорость обучения α и применяем СГС (на пакете размера 1) в виде:

$$[Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha[R_t - Q_\pi(s_t, a_t)].$$

Здесь (s_t, a_t) – вход, $Q_\pi(s_t, a_t) + \alpha[R_t - Q_\pi(s_t, a_t)]$ метка (целевое значение).

Даже при использовании СКО в формуле нет квадратов, потому что функция потерь применяется позже к разности ожидаемого выхода $Q_\pi(s_t, a_t)$ и метки $\alpha[R_t - Q_\pi(s_t, a_t)]$.

Много раз повторяем выборку из π:

$$Q_\pi(s_t, a_t) \leftarrow E_\pi[R_t] = E_{s_t, a_t, \ldots, s_n \sim \pi}\left[\sum\nolimits_{i=t}^{n} \gamma^{i-t} r_i\right].$$

На рис. B.3 иллюстрируется сходимость к истинному математическому ожиданию.

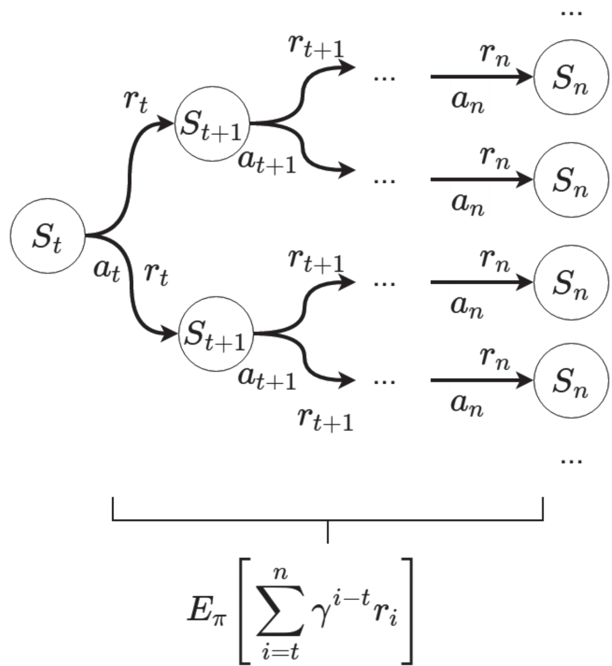


Рис. B.3 ❖ Много сигналов

Теперь мы можем спроектировать наивный прототип нашего алгоритма обучения (код написан на Scala, но для его понимания владение Scala не требуется).

Пример B.1 ❖ Прототип RL4J на Scala

```
// Случайным образом инициализированная нейронная сеть
val neuralNet: NeuralNet

// Повторять MaxEpoch периодов
for (t <- (1 to MaxEpoch))
    epoch()
```

```
def epoch() = {

    // Случайным образом выбрать состояние и действие
    val state = randomState
    val action = randomAction(state)

    // Перейти в новое состояние, инициализировать вознаграждение
    var (new_state, accuReward) = transitition(state, action)

    // Играть до достижения конечного состояния, накапливая вознаграждение
    accuReward += playRandomly(state)


    // Применить СГС к входу и меткам!
    fit((state, action), accuReward)
}

// Специфика MDP, вернуть новое состояние и вознаграждение
def transition(state: State, action: Action): (State, Double)

// Вернуть случайное состояние, выбранное из всего пространства состояний
def randomState: State

// Играть до перехода в конечное состояние
def playRandomly(state): Double = {
    var s = state
    var accuReward = 0
    var k = 0
    while (!s.isTerminal) {
        val action = randomAction(s)
        val (state, reward) = transition(s, action)
        accuReward += Math.pow(gamma, k) * reward
        k += 1
        s = state
    }
    accuReward
}

// Выбрать случайное действие из множества доступных в данном состоянии
def randomAction(state: State): Action =
    oneOf(state.available_action)

// Вспомогательная функция, выбор одного из многих
def oneOf(seq: Seq[Action]): Action =
    seq.get(Random.nextInt(seq.size))

// Как это делалось бы в DL4J
def fit(input: (State, Action), label: Double) =
    neuralNet.fit(toTensor(input), toTensor(label))


// Вернуть INDArray от ND4J
def toTensor(array: Array[_]): Tensor =
    Nd4j.create(array)
```

Тут есть целый ряд проблем: работать-то программа будет, но чудовищно неэффективно. Мы должны сыграть полную игру с *n* состояниями и *n* действиями для нахождения единственной метки, которая, возможно, не особенно значима (если интересные траектории трудно найти, действуя наугад).

Исследование и использование

Если исследовать среду наугад, то процесс в конце концов сойдется к оптимальной политике, но гарантируется это только по истечении почти бесконечного времени: мы должны хотя бы один раз посетить каждую возможную траекторию. (Траекторией называется упорядоченный список всех посещенных состояний и действий, выбранных в одном эпизоде.) Учитывая, сколько существует состояний и ветвлений, нужно признать, что сделать это невозможно. Именно из-за количества ветвлений игра го считается намного более трудной, чем шахматы. В реальном мире мы не располагаем бесконечным временем (а время – как известно, деньги).

Таким образом, мы должны использовать то, чему обучились в прошлом, чтобы сфокусировать исследование на наиболее перспективных траекториях. Добиться этого можно разными способами, один из них – ε-жадное исследование. Идея довольно проста: эта политика выбирает случайное действие с вероятностью ε или наилучшее действие, диктуемое текущей политикой, с вероятностью $(1 - \varepsilon)$. Обычно ε со временем убывает, чтобы после достаточно длительного исследования отдать предпочтение использованию.

По мере появления новой информации фактические *Q*-функции становятся более верным отражением политики, и исследование сосредоточивается на лучших траекториях. Политика, основанная на новой *Q*-функции, становится все лучше (поскольку верность *Q* увеличивается), и ε-жадное исследование находит лучшие траектории. Будучи сфокусированной на лучших траекториях, наша *Q*-функция может исследовать еще лучшие и должна обновить свои результаты в соответствии с новой политикой. Как показывает рис. B.4, этот итеративный цикл сходится к оптимальной политике. Неудивительно, что он называется *перебором политик*. К сожалению, для сходимости может потребоваться бесконечное время, а если *Q* аппроксимируется нейронной сетью, то сходимость даже не гарантируется. Но отсутствие формального доказательства компенсируется впечатляющими результатами.

Этот алгоритм требует также, чтобы выборка состояний производилась «правильно»: в ней должны быть пропорционально отражены репрезентативные состояния, типичные для игры (или, по крайней мере, для игры того уровня, на который мы рассчитываем обучить агента).

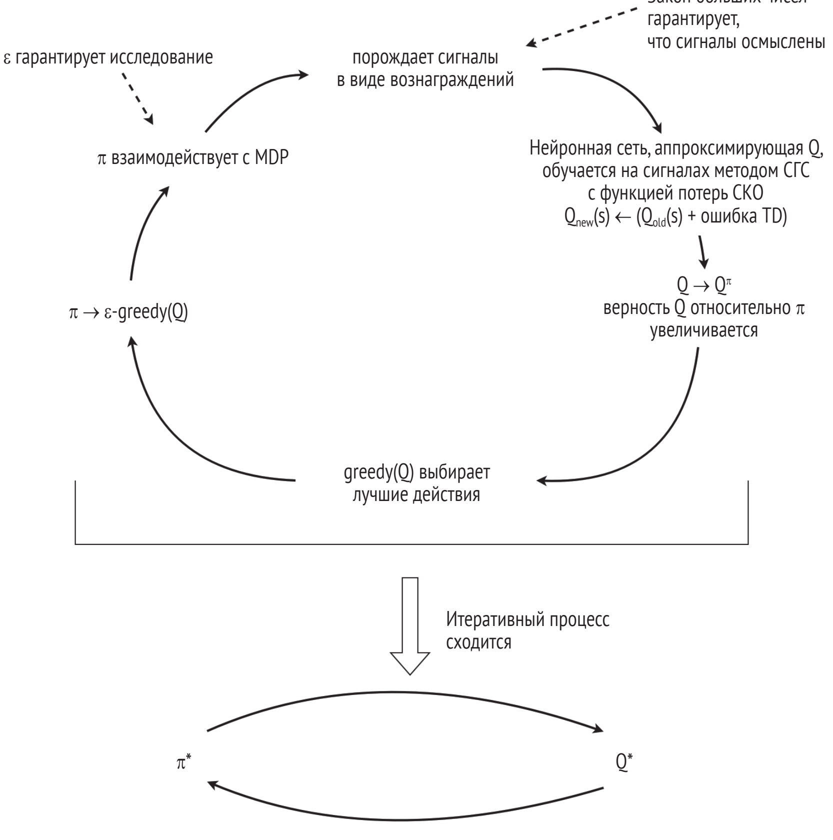


Рис. В.4 ❖ Перебор политик

Уравнение Беллмана

Мы можем преобразовать уравнение *Q*-функции в уравнение Беллмана:

$$Q_\pi(s,a) = E[r_t + \gamma r_{t+1} + \ldots + \gamma^n r_n \mid s_t = s, a_t = a]$$
$$= E[r_t + \gamma r_{t+1} + V(s_{t+1}) \mid s_t = s, a_t = a] = E[r_t + \gamma r_{t+1} + \ldots + \gamma \max_{a'} Q(s_{t+1}, a')]$$
$$\mid s_t = s, a_t = a].$$

Как и в методе Монте-Карло, можно произвести несколько обновлений *Q*. СКО:

$$Q_\pi(s_t,a_t) \leftarrow Q_\pi(s_t,a_t) + \alpha\left[\left(\underbrace{\underbrace{r_t + \max_a Q_\pi(s_{t+1},a)}_{\text{целевое значение}} - Q_\pi(s_t,a_t)}_{\text{ошибка TD}}\right)\right].$$

Здесь ошибка TD – это временно́е различие. Действительно, мы вычисляем разность между ожидаемым значением аппроксимации *Q* в будущем плюс реализованное вознаграждение и ее текущим значением, вычисленным нейронной сетью.

Уравнение Беллмана имеет смысл только при задании граничных условий. Если s – конечное состояние, то:

$$V(s) = 0,$$

и для любого a:

$$Q(s_{t-1}, a) = r_t.$$

Для состояний, близких к конечным, сходимость наступает быстрее, потому что в цепочке они ближе к «истинной» метке – известным граничным условиям. В случае применения обучения с подкреплением к го или шахматам переходам, ведущим в проигранную позицию, назначается значение –1, ведущим в выигранную позицию – значение +1, а всем остальным – значение 0. Значения Q-функции размазываются путем поиска точки в интервале между двумя экстремумами [–1, 1]. Переход со значением Q-функции, близким к 0, ведет к сбалансированной позиции, а со значением, близким к 1, – почти наверняка к победе.

Может показаться удивительным, что ходам присваиваются не только значения –1 и 1 (поскольку отклонение от оптимальной траектории должно бы быть фатальным). Интересный аспект вычисления значений Q состоит в том, что во многих играх или марковских процессах принятия решений никакая ошибка сама по себе не является фатальной. К поражению приводит накапливание ошибок. ИИ полон таких жизненных уроков. Кроме того, пространство ожидаемых накопленных вознаграждений гораздо более гладкое, чем часто думают. Одно из возможных объяснений заключается в том, что математическое ожидание обладает эффектом усреднения, ведь оно не что иное, как взвешенное среднее, в котором в роли весов выступают вероятности. К тому же, поскольку $\gamma < 1$, эффекты отдаленного прошлого не дают заметного вклада. Правда, заманчиво уметь вычислять шансы на выигрыш для любого перехода?

Если произвести достаточно большую выборку переходов вблизи конечных состояний, то Q-обучение может сойтись. Невероятная способность глубокого обучения с подкреплением состоит в обобщаемости обучения с посещенных состояний на непосещенные. Сеть должна понимать, что данная позиция является сбалансированной или выигрышной, даже если никогда не видела ее прежде. Объясняется это тем, что сеть абстрагирует паттерны и понимает силу действия, ориентируясь на предъявленный ранее паттерн (например, стрелять во врага, распознав его форму).

Офлайновое и онлайновое обучение с подкреплением

О различиях между офлайновым и онлайновым обучениями с подкреплением см. замечательную статью Кофзора по адресу https://kofzor.github.io/Reinforcement_Learning_101/#comparing-reinforcement-learning-algorithms.

Выборка начальных состояний

В однопользовательской игре (например, в играх компании Atari) необязательно хорошо играть в каждой ситуации (хотя если это так, значит, мы достигли очень хорошего уровня обобщаемости). Мы должны научиться играть эффективно лишь в тех состояниях, которые встречает наша политика. Следовательно, мы можем произвести выборку легко достижимых состояний, поиграв с текущей полити-

кой из начального состояния. Это позволяет делать выборку прямо из эпизода, сыгранного агентом.

Реализация Q-обучения

Теперь мы можем спроектировать наивный прототип *Q*-обучения. Код представлен в примере B.2.

Пример B.2 ❖ Наивный прототип Q-обучения, написанный на Scala

```
def epoch() = {

    // Выборка из пространства начальных состояний (часто всего одно состояние)
    var state = initState

    // Пока не достигнуто конечное состояние, играть эпизод
    // и при каждом переходе обновлять значение Q
    while(!state.isTerminal)  {

        // Выбрать действие из eps-жадной политики
        val action = epsilonGreedyAction(state)

        // Взаимодействие со средой
        val (nextState, reward) = transition(state, action)

        // Обновление Q
        update(state, action, reward, nextState)

        state = nextState
    }
}

// Обновление Q производится, как описано выше
def update(state: State, action: Action, reward: Double, nextState: State) = {
    val target = reward + maxQ(nextState)
    fit((state, action), target)
}

// Реализация eps-жадной политики
def epsilonGreedyAction(state: State) = {
    if (Random.float() < epsilon)
        randomAction(state)
    else
        maxQAction(state)
}

// Получить максимальное значение Q
def maxQ(state: State) =
    actionsWithQ(state).maxBy(_._2)._2

// Получить действие, соответствующее максимальному значению Q
def maxQAction(state: State) =
    actionsWithQ(state).maxBy(_._2)._1

// Вернуть список действий и значений Q их переходов по состоянию
def actionsWithQ(state: State) = {
    val stateActionList = available_actions.map(action => (state, action))
    available_actions.zip(neural_net.output(toTensor(state_action_list)))

def initState: State
```

Моделирование Q(s,a)

Вместо того чтобы подавать на вход нейронной сети комбинацию a и состояния, мы подаем только состояние, а на выходе получаем значение Q для каждого возможного действия. Это имеет смысл, только когда во всем эпизоде доступны одни и те же действия (иначе выходной слой зависел бы от состояния). Мы можем взять в качестве выхода полное множество действий A и игнорировать невозможные действия (в некоторых статьях целевое значение для невозможных действий полагают равным 0).

Воспроизведение опыта

Применение нейронной сети для аппроксимации Q наталкивается на одну проблему. Переходы сильно коррелированы, поэтому полная дисперсия перехода уменьшается. В конце концов, все они выбраны из одного и того же эпизода. Представьте, что вам требуется обучиться решению задачи вообще без памяти (даже краткосрочной); тогда вы по необходимости оптимизировали бы обучение, основываясь на последнем эпизоде.

Исследовательская группа Google DeepMind использовала воспроизведение опыта (experience replay), т. е. скользящее окно последних N переходов (в оригинальной статье N равно миллиону) с глубокой Q-сетью. Таким образом, ей удалось существенно повысить качество игры на компьютерах Atari. Вместо того чтобы обновлять значение Q на основе последнего перехода, мы сохраняем переход в окне опыта и производим обновление на основе пачки случайно выбранных переходов из того же окна.

Функция `epoch()` принимает вид:

```
def epoch() = {

    // Выборка из пространства начальных состояний (часто всего одно состояние)
    var state = initState

    // Пока не достигнуто конечное состояние, играть эпизод
    // и при каждом переходе обновлять значение Q
    while(!state.isTerminal)  {

        // Выбрать действие из eps-жадной политики
        val action = epsilonGreedyAction(state)

        // Взаимодействие со средой
        val (nextState, reward) = transition(state, action)

        // Сохранить переход (окно воспроизведения опыта – кольцевой буфер)
        expReplay.store(state, action, reward, nextState)

        // Пакетное обновление Q
        updateFromBatch(expReplay.getBatch())

        state = nextState
    }
}
```

Сжатие

Библиотека ND4J, содержащая реализацию тензоров для DL4J, формально не поддерживает тип `uint8`. Но пиксели полутонового изображения кодируются именно с такой точностью. Чтобы не тратить слишком много памяти, массивы `INDArray` сжимаются в `uint8`.

Сверточные слои и предварительная обработка изображений

Сверточные слои прекрасно обнаруживают локальные паттерны в изображениях. В случае пиксельных изображений они используются как процессоры, понижающие размерность входа путем сведения его к многообразию. Имея подходящее многообразие наблюдений, становится гораздо проще принимать решения.

Обработка изображений

Можно подать на вход нейронной сети само изображение в формате RGB, но тогда сеть должна будет обучиться также этому дополнительному паттерну. Похоже, в мозгу имеются механизмы для комбинирования цветов (на наше счастье!). Таким образом, с подобной предварительной обработкой вполне можно смириться.

На рис. B.5 показано, что видим мы.

Рис. B.5 ❖ Кадр из игры Doom

А на рис. B.6 показано, что видит нейронная сеть.

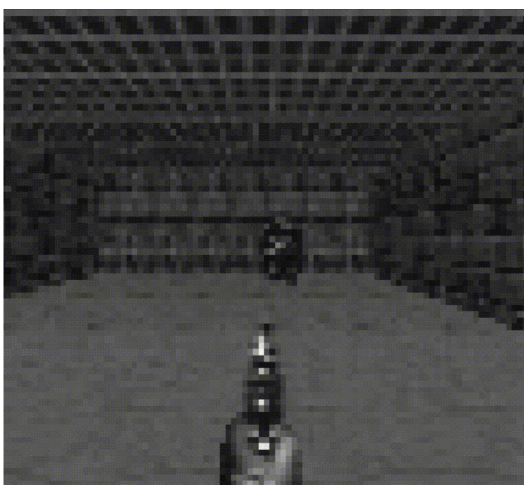


Рис. B.6 ❖ Тот же кадр, обработанный для подачи на вход нейронной сети

Размер изображения уменьшен до 84 × 84. По мере роста размера входных данных растут потребление памяти и объем вычислений в сверточных слоях. Для того чтобы хорошо играть, мелкие детали изображения не нужны. Многие из них вообще введены только из соображений эстетики. Масштабирование до разумного размера ускоряет обучение.

Пропуск кадров

В оригинальной статье обрабатывается только один из четырех кадров. Для следующих трех повторяется последнее действие. Это ускоряет работу примерно в 4 раза без существенной потери информации. Игры Atari не рассчитаны на покадровую точность игры, так что большинство действий имеет смысл сохранять по меньшей мере на протяжении четырех кадров.

Обработка истории

Чтобы сообщить нейронной сети информацию о текущем импульсе, последние четыре кадра (с учетом пропуска кадров мы выбираем один кадр из каждых четырех) собираются в четыре канала, как показано на рис. B.7. Эти четыре кадра представляют историю (см. раздел «Наблюдаемое состояние» выше).

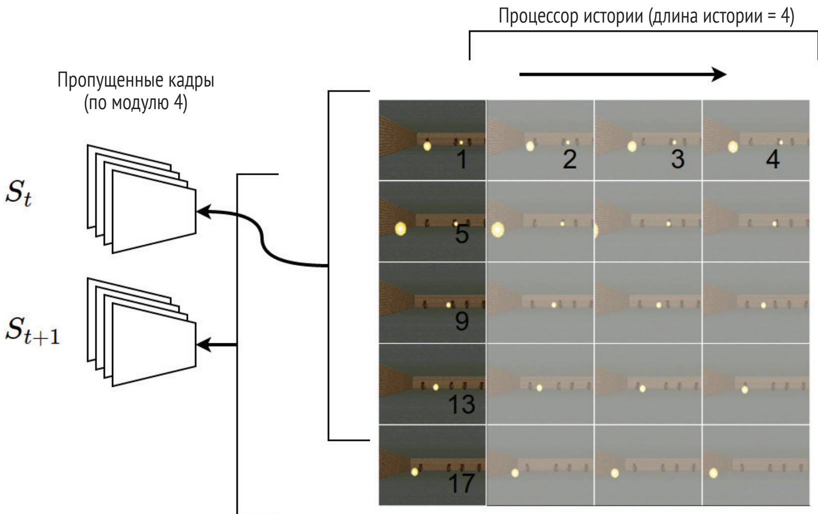


Рис. B.7 ❖ Обработка истории и стопка кадров

Первые кадры истории можно сгенерировать с помощью политики выбора случайного действия или использовать воспроизведение, которое ничего не делает[8].

Двойное Q-обучение

Идея двойной глубокой *Q*-сети состоит в том, что сеть замораживается после каждых *M* обновлений (жесткое обновление) или сглаживается путем усреднения (target = target * (smooth) + current * (1–smooth)) после каждого обновления (мягкое обновление). Это делает обучение более устойчивым благодаря использова-

[8] Чтобы вычисление было справедливым, можно использовать случайный старт.

нию значения Q в формуле ошибки TD, которая становится менее восприимчивой к «дерганью». Формула обновления Q принимает вид:

$$Y_{\text{target}} = r_t + \gamma^*(Q_{\text{target}}(s_t + 1, \text{argmax}_a Q(s_t + 1, a))).$$

Ограничение

Ошибку TD можно ограничить с двух сторон, так чтобы выбросы в процессе обновления не оказывали большого влияния на обучение.

Масштабирование вознаграждений

Масштабирование вознаграждений с приведением значений Q к диапазону $[-1, 1]$ (по аналогии с нормировкой) может кардинально повысить эффективность обучения. Это важный гиперпараметр, и пренебрегать им не следует.

Приоритетное воспроизведение

Идея приоритетного воспроизведения[9] состоит в том, что не все переходы одинаково важны. Отсортировать их можно, например, по ошибке TD. Действительно, большая ошибка TD коррелирует с высоким уровнем информации (в смысле – сюрприз). Такие переходы следует выбирать чаще прочих.

График, визуализация и среднее значение Q

Для визуализации и отладки процесса или метода обучения подкреплением полезно видеть, как протекает обучение агента в динамике. Для этой цели я разработал контрольную панель webapp-rl4j[10] (рис. B.8).

Самое важное – отслеживать накопительное вознаграждение (рис. B.9). Таким образом мы проверяем, становятся ли агенты лучше. Важно отметить, что рисунок отражает ε-жадную стратегию, а не политику, выведенную непосредственно из аппроксимации Q.

Можно также поинтересоваться потерей (оценкой качества нейронной сети) и средним значением Q (рис. B.10).

В отличие от классического обучения с учителем, потеря необязательно уменьшается, потому что обучение воздействует на метки!

При использовании с целевой сетью будут видны разрывы, поскольку вычисление различных целевых сетей производится не непрерывно. Относительно одной целевой сети потеря должна уменьшаться. Средние значения Q должны гладко сходиться к значению, пропорциональному среднему ожидаемому вознаграждению.

---

[9] https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html#prio.

[10] https://github.com/rubenfiszel/webapp-rl4j.

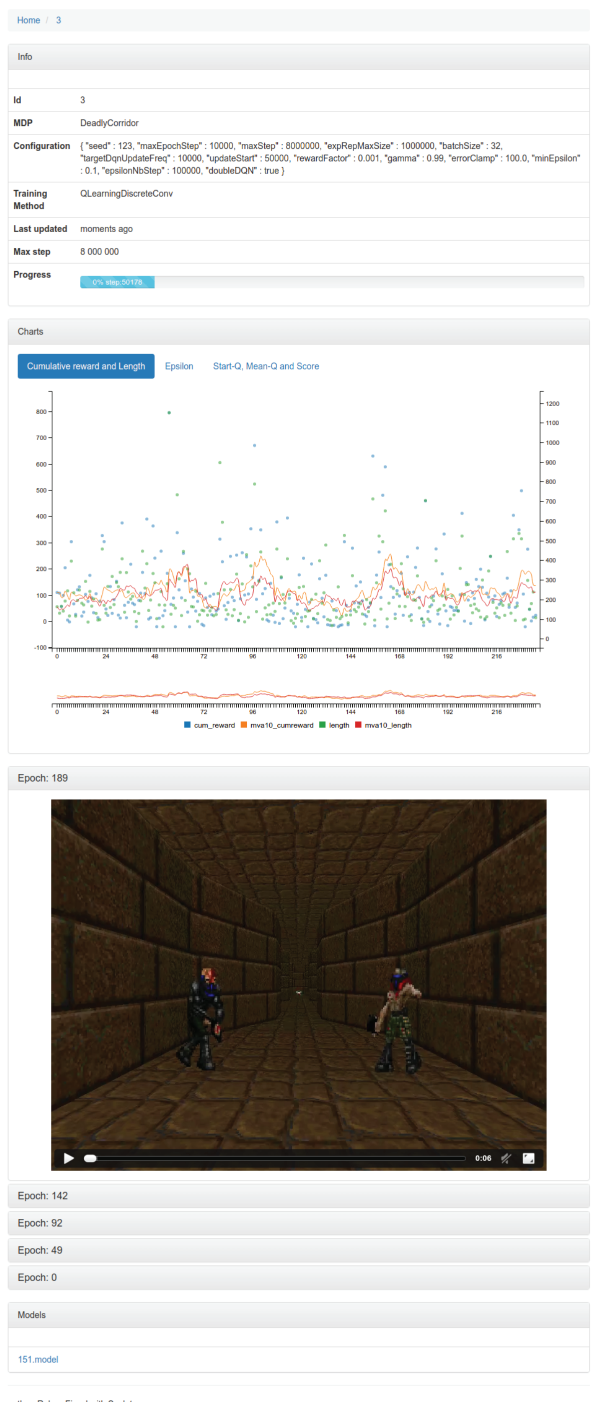


Рис. В.8 ❖ Снимок экрана webapp-rl4j

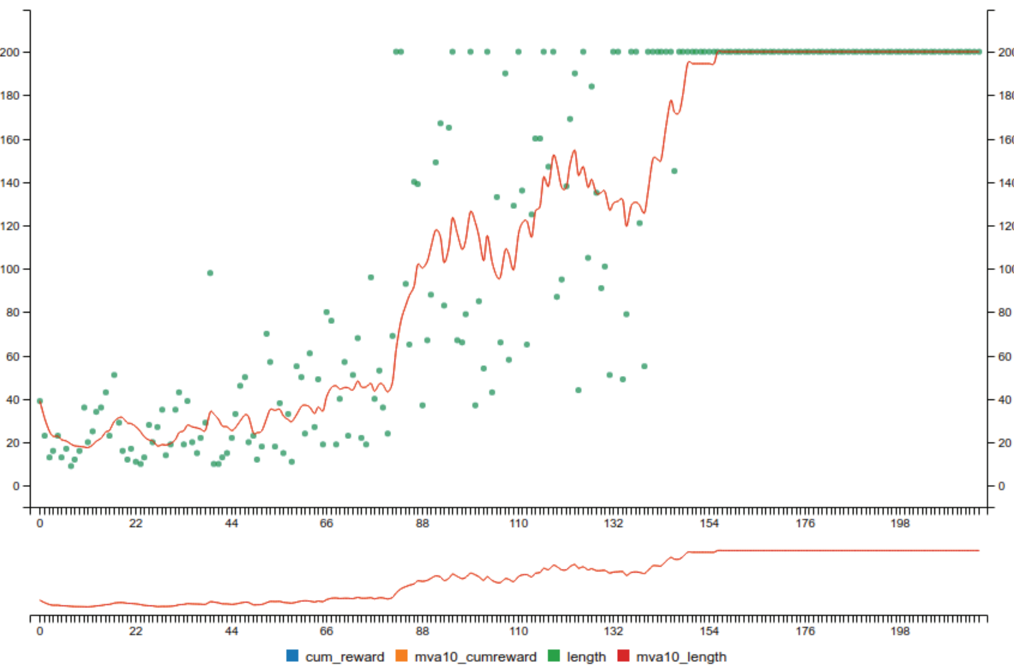


Рис. B.9 ❖ График накопительного вознаграждения

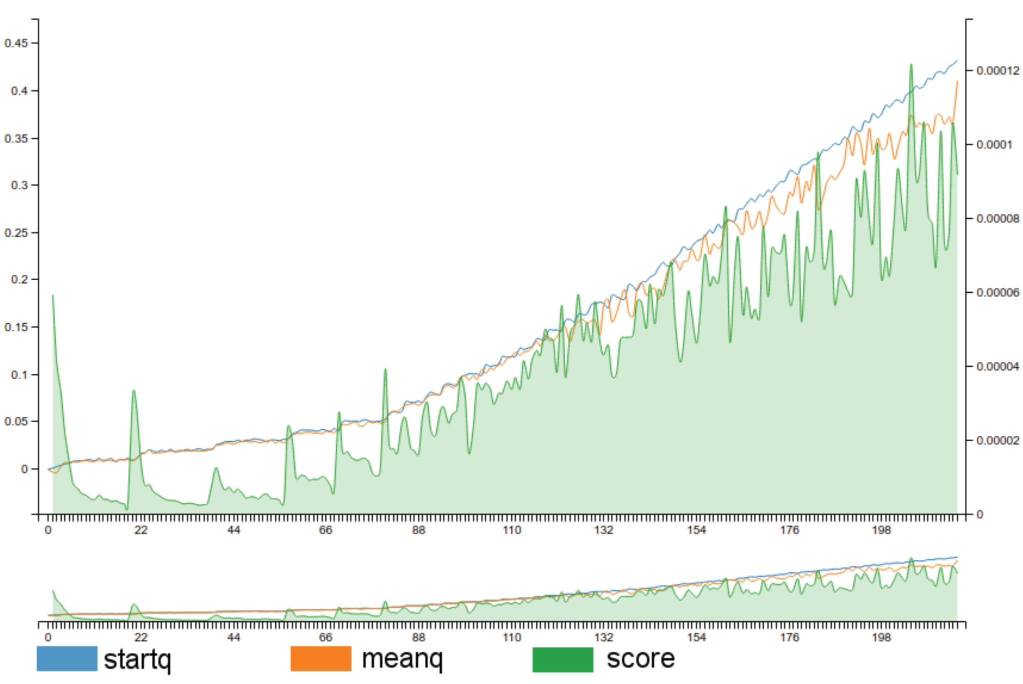


Рис. B.10 ❖ Графики оценки сети и среднего значения Q

RL4J

Библиотека RL4J) доступна на GitHub по адресу https://github.com/deeplearning4j/rl4j. В настоящее время реализованы глубокие Q-сети (DQN) с воспроизведением опыта, двойное Q-обучение и ограничение[11]. Включены также асинхронное обучение с подкреплением с помощью алгоритма A3C (Asynchronous Advantage Actor Critic) и асинхронное N-шаговое Q-обучение. Можно играть в пиксельные игры, а также решать задачи более низкой размерности (например, балансирование шеста на тележке – Cartpole). Реализация асинхронного обучения с подкреплением считается экспериментальной. Хочется надеяться, что пользователи обогатят библиотеку собственным кодом.

В примере B.3 приведен работающий код простой DQN для игры в Cartpole. Можно играть и в Doom. Дополнительные примеры можно найти по адресу https://github.com/rubenfiszel/rl4j-examples. В качестве аргумента любого метода обучения можно указать собственную модель нейронной сети.

Пример B.3 ❖ Пример использования RL4J на Scala

```
public static QLearning.QLConfiguration CARTPOLE_QL =
        new QLearning.QLConfiguration(
                123,    // Случайное начальное значение
                200,    // Максимальный шаг по периодам
                150000, // Максимальный шаг
                150000, // Максимальный размер окна воспроизведения опыта
                32,     // Размер пакета
                500,    // Обновление целевого значения (жесткое)
                10,     // Число шагов поор-прогрева
                0.01,   // Коэффициент масштабирования вознаграждения
                0.99,   // Гамма
                1.0,    // Ограничение ошибки TD
                0.1f,   // Минимальное значение эпсилон
                1000,   // Число шагов для отжига епсилон
                true    // Двойное Q-обучение
        );

public static DQNFactoryStdDense.Configuration CARTPOLE_NET =
        new DQNFactoryStdDense.Configuration(
                3,      // Число слоев
                16,     // Число скрытых блоков
                0.001,  // Скорость обучения
                0.00    // L2-регуляризация
        );

public static void main( String[] args )
{

    // Записать обучающие данные из rl4j-data в новый каталог
    DataManager manager = new DataManager(true);

    // Определить MDP в виде gym (name, render)
```

[11] Дополнительные сведения см. в оригинальной статье по адресу https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html.

```
    GymEnv<Box, Integer, DiscreteSpace> mdp = new GymEnv("CartPole-v0", false,
        false);

    // Определить процесс обучения
    QLearningDiscreteDense<Box> dql = new QLearningDiscreteDense(mdp,
        CARTPOLE_NET, CARTPOLE_QL, manager);

    // Обучить
    dql.train();

    // Получить окончательную политику
    DQNPolicy<Box> pol = dql.getPolicy();

    // Сериализовать и сохранить (демонстрация сериализации, шаг не обязателен)
    pol.save("/tmp/pol1");

    // Закрыть mdp (закрыть http)
    mdp.close();

}
```

Заключение

Это было увлекательное путешествие в мир глубокого обучения с подкреплением. С точки зрения уравнений и кода, Q-обучение является мощным, но при этом довольно простым алгоритмом. В области обучения с подкреплением ведутся активные и многообещающие исследования. На самом деле обучение с учителем можно рассматривать как частный случай обучения с подкреплением (если считать метки вознаграждением). Быть может, не далек тот день, когда обучение с подкреплением станет лекарством от всех болезней искусственного интеллекта. Ну а пока мы можем с восхищением наблюдать, как оно успешно применяется для решения все более головоломных задач. Пользуясь случаем, хочу поблагодарить компанию Skymind и весь ее замечательный коллектив за чрезвычайно полезную дипломную практику.

Приложение C

Числа, которые должен знать каждый

В табл. C.1 приведены «12 чисел Джеффа Дина», которые должен знать каждый.

***Таблица C.1.** 12 чисел Джеффа Дина*

Компьютерная операция	Длительность (в наносекундах)
Обращение к кэшу L1	0.5
Неправильное предсказание ветвления	5
Обращение к кэшу L2	7
Захват/освобождение мьютекса	100
Обращение к основной памяти	100
Сжатие 1 КБ с помощью библиотеки Zippy	10 000
Передача 2 КБ по сети со скоростью 1 Гб/с	20 000
Последовательное чтение 1 МБ из памяти	250 000
Время кругового обращения пакета внутри ЦОДа	500 000
Позиционирование головки диска	10 000 000
Последовательное чтение 1 МБ из сети	10 000 000
Последовательное чтение 1 МБ с диска	30 000 000
Передача пакета Канада->Нидерланды->Канада	150 000 000

Приложение D

Нейронные сети и обратное распространение: математическое описание

Алекс Блэк

Введение

В этом приложении мы рассмотрим математический аппарат, лежащий в основе обучения нейронных сетей: *алгоритм обратного распространения*. Но прежде зададимся вопросом: что мы пытаемся сделать в процессе обучения нейронной сети?

По существу (оставляя в стороне такие проблемы, как переобучение), мы хотим в процессе обучения подобрать (на основе обучающих данных) такие параметры нейронной сети, чтобы сеть выдавала верные предсказания. Два ключевых момента здесь: *верные предсказания* и *подобрать параметры*.

Рассмотрим задачу классификации, когда от сети требуется предсказать класс предъявленного примера. Существует много способов количественно оценить качество классификации: верность, F-мера, отрицательное логарифмическое правдоподобие и т. д. Все эти показатели годятся, но одни гораздо труднее оптимизировать, чем другие. Так, верность сети не обязательно изменится, если немного изменить любой параметр, а это значит, что прямая оптимизация верности методами, основанными на вычислении градиента, невозможна (функция верности не дифференцируема).

Напротив, другие показатели, например отрицательное логарифмическое правдоподобие, возрастают или убывают вместе с изменением значений параметров, даже небольшим. Если для количественного выражения качества предсказаний сети ограничиться только дифференцируемыми функциями потерь (такими как отрицательное логарифмическое правдоподобие) и применить методы математического анализа, то мы получим простой и элегантный алгоритм обучения нейронных сетей.

Основные идеи алгоритма обратного распространения просты:

- количественно выразить качество текущих предсказаний сети с помощью дифференцируемой функции потерь;

- вычислить градиенты по каждому параметру сети, применив к функции потерь и структуре сети формулы анализа функций многих переменных;
- с помощью вычисленных градиентов итеративно корректировать параметры сети в направлении, минимизирующем функцию потерь.

Таким образом, алгоритм стохастического градиентного спуска (СГС) формулируется следующим образом:

Вход: параметры сети $\mathbf{w}$, функция потерь L, обучающие данные $\boldsymbol{D}$, скорость обучения $\alpha > 0$
while *условия остановки не выполнены* **do**
  $(features, labels) \leftarrow \boldsymbol{D}.getRandomMiniBatch()$
  $out \leftarrow getNetworkOutput(\mathbf{w}, features)$
  $\partial L/\partial\mathbf{w} \leftarrow calculateParameterGradients(\mathbf{w}, L, out, labels)$
  $\mathbf{w} \leftarrow \mathbf{w} - \alpha(\partial L/\partial\mathbf{w})$
end

Самое главное здесь – вычисление частных производных $\partial L/\partial\mathbf{w}$. Если вы не знаете, что такое частная производная, то можете считать, что она описывает изменение интересующей нас величины (значения функции потерь L) в зависимости от изменения какой-то другой величины (значения какого-то параметра w_i из числа представленных вектором весов $\mathbf{w}$) в предположении, что значения всех остальных параметров остаются постоянными.

Размер шага

В большинстве случаев $\partial L/\partial w_i$ также является функцией от w_i. Именно поэтому мы а) можем выполнять только небольшие шаги (т. е. задавать малую скорость обучения α) и б) вынуждены вычислять градиенты после изменения параметров на каждой итерации.

Если $\partial L/\partial w_i$ положительна, то при небольшом увеличении w_i функция потерь L для текущего примера увеличивается, а при уменьшении w_i – уменьшается.

Собственно, в этом и состоит алгоритм обратного распространения: определить функцию потерь, вычислить ее частные производные по всем параметрам и сделать небольшой шаг в направлении, которое минимизирует функцию потерь. Эта простая, но исключительно продуктивная идея лежит в основе всего глубокого обучения.

Обратное распространение в многослойном перцептроне

Но хватит слов, перейдем к математике. Мы будем рассматривать простой многослойный перцептрон, т. е. стандартный полносвязный слой нейронной сети прямого распространения, или `DenseLayer` в терминологии DL4J, взяв в качестве функции потерь сумму квадратов ошибок (L2 в терминологии DL4J). Обучение будем производить на одном примере:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2,$$

где N – количество выходов, y_i – i-я метка и $\hat{y}_i = \text{output}(\mathbf{w},\mathbf{f})$ – предсказанное сетью значение y_i при заданном векторе признаков $\mathbf{f}$ и текущих параметрах $\mathbf{w}$.

На рис. D.1 показана однослойная нейронная сеть. Продираясь через последующие математические формулы, обращайтесь к этому рисунку.

$$L(\mathbf{y}, \hat{\mathbf{y}}) = (y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 + (y_3 - \hat{y}_3)^2$$

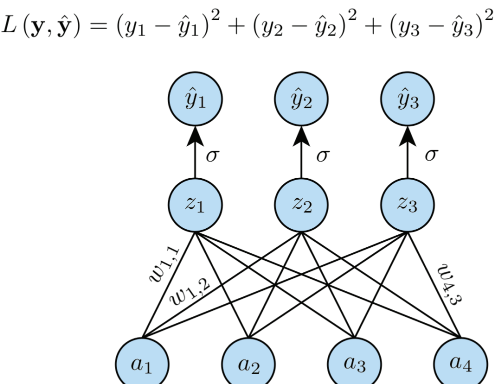


Рис. D.1 ❖ Однослойная нейронная сеть

Обозначим **a** входной вектор для текущего слоя (длины 4), σ – поэлементную нелинейность (функцию активации, например tanh или блок линейной ректификации – ReLU). Тогда уравнения прямого распространения в сети имеют вид:

$$z_i = b_i + \sum_{j=1}^{4} w_{j,i} a_i;$$
$$\hat{y}_i = \sigma(z_i),$$

где b_i – смещение, а $w_{j,i}$ – вес связи между входом j и нейроном i.

Прежде всего мы хотим вычислить первую частную производную функции потерь по выходу сети $\hat{y}_i$:

$$\frac{\partial L}{\partial \hat{y}_j} = \frac{\partial}{\partial \hat{y}_j}\left(\sum_{i=1}^{N} (y_i - \hat{y}_i)^2\right)$$
$$= \frac{\partial}{\partial \hat{y}_j}(y_j - \hat{y}_j)^2$$
$$= -2(y_j - \hat{y}_j).$$

Далее, двигаясь по сети в обратном направлении, мы хотим вычислить $\partial L/\partial z_i$ как функции от $\partial L/\partial \hat{y}_i$. Здесь всё зависит от вида функции активации σ(*z*). Для простоты будем предполагать, что это простая функция типа сигмоиды, tanh или ReLU (их производные являются простыми функциями одной переменной; некоторые функции, например softmax, не обладают таким свойством).

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial \hat{y}_j}\frac{\partial \hat{y}_i}{\partial z_i}$$
$$= \sigma'(z_i)\frac{\partial L}{\partial \hat{y}_i}.$$

Например, в случае сигмоиды $\sigma(z) = 1/(1 + e^{-z})$ имеем $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

Далее применим правило дифференцирования сложной функции для вычисления частных производных по весам $w_{j,i}$, зная уже вычисленные производные $\partial L/\partial z_i$:

$$\begin{aligned}\frac{\partial L}{\partial w_{j,i}} &= \sum\nolimits_{k=1}^{3} \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial w_{j,k}} \\ &= \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial w_{j,i}} \\ &= \frac{\partial L}{\partial z_i} \frac{\partial}{\partial w_{j,i}} \left(b_i + \sum\nolimits_{k=1}^{4} w_{k,i} a_i \right) \\ &= a_i \frac{\partial L}{\partial z_i}.\end{aligned}$$

Тот же подход можно применить для доказательства того, что производные по смещениям равны $\partial L/\partial b_i = \partial L/\partial z_i$.

Наконец, мы хотим вычислить производные функции потерь по входным активациям a_i, опять-таки зная $\partial L/\partial z_i$:

$$\begin{aligned}\frac{\partial L}{\partial a_i} &= \sum\nolimits_{j=1}^{3} \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial a_i} \\ &= \sum\nolimits_{j=1}^{3} \frac{\partial L}{\partial z_j} \frac{\partial}{\partial a_i} \left(b_j + \sum\nolimits_{k=1}^{4} w_{k,j} a_j \right) \\ &= \sum\nolimits_{j=1}^{3} \frac{\partial L}{\partial z_j} w_{i,j}.\end{aligned}$$

Вот и все. Если бы ниже был еще один слой, то мы могли применить точно такой же подход, только в уравнениях нужно было бы заменить y_i на a_i. Мы просто следуем по имеющимся в сети связям в обратном направлении, применяя по ходу дела правило дифференцирования сложной функции.

Обобщить этот анализа на мини-пакет (т. е. на несколько примеров) элементарно: просто нужно усреднить градиенты по каждому параметру по всему мини-пакету. На практике эти уравнения реализуются с помощью умножения матриц и векторных операций над мини-пакетами. Тот же подход можно использовать для вычисления производных в более сложных моделях, например рекуррентных и сверточных нейронных сетях.

Приложение E

ND4J API

ND4J – библиотека научных расчетов для виртуальной машины Java (JVM). При ее проектировании ставилась задача добиться высокого быстродействия в производственной среде. К числу основных характеристик относятся:

- гибкий *n*-мерный массив;
- многоплатформенность, в т. ч. возможность работы с графическими процессорами (GPU);
- функции для выполнения операций линейной алгебры и обработки сигналов.

ND4S – версия ND4J для Scala.

Неудобство использования легло пропастью между программистами, пишущими на Java, Scala и Clojure, и самыми мощными средствами для анализа данных типа NumPy или Matlab. Библиотеки типа Breeze не поддерживают *n*-мерных массивов, или тензоров, необходимых для глубокого обучения и других задач.

Библиотека ND4J с открытым исходным кодом, поддерживающая распределенные вычисления и GPU, делает доступными на JVM интуитивно понятные инструменты научных расчетов, привычные сообществу пишущих на Python. По своей структуре она похожа на SLF4J. ND4J предлагает инженерам простой способ перенести алгоритмы и интерфейсы с другими библиотеками в экосистемы Java и Scala. ND4J и ND4S используются в национальных лабораториях, например для моделирования климата, где необходим огромный объем вычислений.

Полное онлайновое руководство пользователя по ND4J
ND4J поддерживает гораздо больше операций, чем упомянуто в этом приложении. Полное руководство доступно по адресу http://nd4j.org/userguide.

Полная документация по ND4J API в формате Javadoc
Полная документация по ND4J API размещена по адресу http://nd4j.org/doc/.

Дизайн и основы использования

ND4J проектировалась для работы во многих средах и интегрирована с современными системами обработки данных. Перечислим некоторые ее возможности:

- работа с GPU, поддерживающими архитектуру CUDA;
- интеграция с Hadoop и Spark;
- API спроектирован по образцу Numpy.

Большая часть операций ND4J относится к манипулированию числовыми массивами. Основная структура данных называется *NDArray*.

Что такое NDArray?

По существу, NDArray – это *n*-мерный массив чисел. Объект NDArray характеризуется следующими свойствами:

- ранг;
- форма;
- длина;
- шаг;
- тип данных.

Далее мы объясним, для чего нужно каждое свойство.

NDArray и INDArray

Термином NDArray мы обозначаем общую концепцию многомерного массива. А INDArray относится к конкретному интерфейсу Java, определенному в библиотеке ND4J. На практике оба термина употребляются как синонимы.

Ранг

Рангом NDArray называется число его измерений. У двумерных массивов NDArray ранг равен 2, у трехмерных – 3 и т. д. Разрешается создавать объекты NDArray любого ранга.

Форма

Форма NDArray определяет размер по каждому измерению. Двумерный NDArray с 3 строками и 5 столбцами имеет форму `[3,5]`.

Длина

Длина NDArray определяет общее число элементов в массиве. Длина равна произведению размеров по каждому измерению, составляющих форму.

Шаг

Шагом (stride) NDArray называется расстояние (во внутреннем буфере данных) между соседними элементами в каждом измерении. Шаг определяется отдельно для каждого измерения, т. е. у массива ранга *N* имеется *N* шагов. Отметим, что обычно шаг нам неизвестен (и неинтересен) – просто нужно иметь в виду, что так работает ND4J. В следующем разделе будет приведен пример шагов.

Тип данных

В массиве NDArray хранятся данные определенного типа (например, с плавающей точкой одинарной или двойной точности). Отметим, что тип задается глобально на уровне ND4J, так что для всех объектов NDArray тип данных одинаков. Как задавать тип данных, описано ниже.

Что нужно знать об индексации и размерностях NDArray

Строки считаются нулевой размерностью, столбцы – первой. Таким образом, `INDArray.size(0)` – это число строк, а `INDArray.size(1)` – число столбцов. Как и для обычных массивов в большинстве языков программирования, индексация начинается с 0. Поэтому индекс строки изменяется от 0 до `INDArray.size(0)-1` и точно так же для остальных размерностей.

NDArray и размещения в памяти

Объекты NDArray хранятся в памяти в виде плоского массива чисел (или, если угодно, в виде одного непрерывного блока памяти) и этим сильно отличаются от типичных многомерных массивов Java, таких как `float[][]` или `double[][][]`.

Физически область, в которой хранится INDArray, находится вне кучи, т. е. не является частью JVM. У такого решения много достоинств, включая производительность, интероперабельность с высокопроизводительными библиотеками BLAS и отсутствие некоторых недостатков, препятствующих использованию JVM для высокопроизводительных вычислений (в частности, длина массива Java ограничена $2^{31} - 1$ [2.14 миллиарда] элементами, поскольку индекс – целое число).

Общий синтаксис ND4J

В ND4J есть три типа операций:

- скалярные;
- преобразования;
- аккумуляторы.

Большинство операций принимает перечисление, или список дискретных значений, допускающий автозавершение.

- **Скалярные операции.** Скалярная операция принимает два аргумента: вход и скаляр, который следует применить ко входу. Например, `ScalarAdd()` принимает объект `INDArray x` и скаляр `Number num`, т. е. ее сигнатура имеет вид `ScalarAdd(INDArray x, Number num)`. Все остальные скалярные операции имеют такой же формат.
- **Преобразования** – самая простая операция, потому что она принимает один аргумент и что-то делает с ним. Преобразование `abs(IComplexNDArray ndarray)` вычисляет абсолютную величину своего аргумента. Аналогично преобразование `sigmoid()` возвращает «сигмоиду x».
- **Аккумуляторы.** Наконец, аккумуляторы (в контексте GPU их называют также редукторами) складывают массивы и векторы и могут понизить размерность массивов в результате замены строки суммой ее элементов. Например, применив аккумулятор к массиву

```
[1 2
 3 4]
```

получим вектор:

```
[3
 7]
```

Количество столбцов (т. е. измерений) уменьшилось с двух до одного.

Аккумуляторы могут быть попарными или скалярными. В случае попарной редукции имеются два массива, x и y, одинаковой формы. Тогда можно, например, вычислить косинусное расстояние между x и y, применив некоторую операцию к парам соответственных элементов:

```
cosineSim(x[i], y[i])
```

Или вычислить евклидово расстояние `EuclideanDistance(arr, arr2)` между массивами `arr` и `arr2`.

Основы работы с массивами NDArray

Класс ND4J

В этот класс включены вспомогательные статические методы для создания объектов NDArray. Далее мы обсудим наиболее употребительные.

Nd4j.zeros(int ...). Целые аргументы задают форму массива. Например, чтобы создать заполненный нулями массив с 3 строками и 5 столбцами, мы пишем `Nd4j.zeros(3,5)`.

Nd4j.ones(int ...). Аналогичен `.zeros(int...)`, но создает массив, заполненный не нулями, а единицами.

Инициализация другими значениями. Для создания массивов, содержащих другие значения, можно использовать комбинацию методов класса ND4J с другими операциями. Например, чтобы создать массив, все элементы которого равны 10, можно написать:

```
INDArray tens = Nd4j.zeros(3,5).addi(10)
```

Здесь мы сначала создаем массив 3 × 5, заполненный нулями, а затем прибавляем к каждому элементу 10.

Инициализация случайными числами. ND4J предоставляет несколько методов для создания объектов INDArray, заполненных псевдослучайными числами.

Для генерации случайных чисел с равномерным распределением в диапазоне от 0 до 1 служит метод `Nd4j.rand(int nRows, int nCols)` (для двумерных массивов) или `Nd4j.rand(int[])` (для массивов с 3 и более измерениями).

Аналогично для генерации случайных чисел, имеющих нормальное распределение со средним 0 и стандартным отклонением 1, служат методы `Nd4j.randn(int nRows, int nCols)` и `Nd4j.randn(int[])`.

Для воспроизводимости результатов можно задать начальное значение генератора случайных чисел ND4J методом `Nd4j.getRandom().setSeed(long)`.

Управление формой NDArray

Далее мы покажем, как управлять формой массивов NDArray при выполнении основных операций.

Создание массивов

Ниже создается одномерный массив с двумя столбцами, в которых находятся значения { 1, 2 }.

```
INDArray nd = Nd4j.create(new float[]{1,2},new int[]{2}); // Вектор-строка
```

Пример: создание NDArray 2 × 2. В следующем примере создается двумерный массив NDArray с двумя строками и двумя столбцами. В первой строке находятся значения {1, 2}, а во второй – {3, 4}.

```
INDArray arr1 = Nd4j.create(new float[]{1,2,3,4},new int[]{2,2});
System.out.println(arr1);
```

В результате будет напечатано:

```
[[1.0 ,3.0]
[2.0 ,4.0]
]
```

Помните об организации строк и столбцов в ND4J

Массив NDArray может быть организован, как в C (по строкам) или как в Fortran (по столбцам). О различиях между двумя организациями можно прочитать в статье по адресу https://en.wikipedia.org/wiki/Row-major_order.

В ND4J можно использовать организацию массива по строкам и по столбцам одновременно. Обычно пользователи предпочитают организацию по умолчанию, но имейте в виду, что можно точно задать организацию конкретного массива, если понадобится.

Пример: сложить два массива NDArray размера 2 × 2. Создаем второй массив (arr2) и прибавляем его к первому (arr1):

```
INDArray arr2 = ND4j.create(new float[]{5,6,7,8},new int[]{2,2});
arr1.addi(arr2);
System.out.println(arr1);
```

В результате будет напечатано:

```
[[7.0 ,11.0]
[9.0 ,13.0]]
```

Создание объектов NDArray из массивов Java

ND4J предоставляет удобные методы для создания массивов из массивов Java типа float и double.

Чтобы создать одномерный массив NDArray из одномерного массива Java, пользуйтесь следующими методами:

- вектор-строка: `Nd4j.create(float[])` or `Nd4j.create(double[])`;
- вектор-столбец: `Nd4j.create(float[],new int[]{length,1})` или `Nd4j.create(double[],new int[]{length,1})`.

Для создания двумерных массивов служит метод `Nd4j.create(float[][])` или `Nd4j.create(double[][])`.

Для создания NDArray из стандартных массивов Java с тремя и более измерениями (`double[][][]` и т. д.) можно поступить следующим образом:

```
double[] flat = ArrayUtil.flattenDoubleArray(myDoubleArray);
int[] shape = ...;    // Здесь задается форма массива
INDArray myArr = Nd4j.create(flat,shape,'c');
```

Получение и установка отдельных значений NDArray

Получить или установить значение элемента INDArray можно, задав его индексы. Для массива ранга N (с N измерениями) необходимо N индексов.

Оптимизация производительности операций с массивами NDArray

С точки зрения производительности, получение или установка значений по одному (например, в цикле) – операция крайне неэффективная. По возможности следует использовать другие методы INDArray, воздействующие сразу на много элементов.

Для получения значений двумерного массива служит метод `INDArray.getDouble(int row, int column)`.

Для массивов произвольной размерности используется метод `INDArray.getDouble(int...)`.

Например, чтобы получить значение с индексами `i`, `j`, `k`, следует написать:

```
INDArray.getDouble(i,j,k)
```

Для установки значения служат варианты метода `putScalar`:

- `INDArray.putScalar(int[],double);`
- `INDArray.putScalar(int[],float);`
- `INDArray.putScalar(int[],int).`

Здесь `int[]` – индекс, а `double/float/int` – значение, записываемое в элемент с таким индексом.

Работа со строками NDArray

Для работы с частями NDArray существует много методов.

Получить одну строку. Для получения одной строки служит метод

```
INDArray.getRow(int)
```

Понятно, что этот метод возвращает вектор-строку. Но следует отметить, что это на самом деле представление: любое изменение возвращенной строки отражается на исходном массиве. Иногда это очень полезно (например, вызов `myArr.getRow(3).addi(1.0)` прибавляет 1.0 ко всем элементам третьей строки большего массива). Чтобы скопировать строку, пользуйтесь методом

```
getRow(int).dup()
```

Получить несколько строк. Для получения нескольких строк служит метод

```
INDArray.getRows(int...)
```

Он возвращает массив с составленными в стопку строками. Однако это будет копия исходных строк, а не представление, поскольку получить в этом случае представление невозможно из-за способа размещения NDArray в памяти.

Установка одной строки. Для установки одной строки служит метод

```
myArray.putRow(int rowIdx,INDArray row)
```

Он присваивает элементам строки массива `myArray` с индексом `rowIdx` значения, содержащиеся в массиве `row`.

Краткий перечень методов получения размеров и измерений NDArray

В интерфейсе INDArray определены следующие методы:

- получить количество измерений: `rank()`;
- только для двумерных NDArray: `rows()`, `columns()`;
- размер по *i*-му измерению: `size(i)`;
- получить размеры по всем измерениям: `int[]: shape()`;
- получить общее число элементов массива: `arr.length()`;
- см. также: `isMatrix()`, `isVector()`, `isRowVector()` и `isColumnVector()`.

Класс Dataset

Класс `org.nd4j.linalg.dataset.DataSet` представляет преобразование данных и содержит входные и выходные данные[1].

В контексте нейронных сетей набор данных описывает отображение входных признаков на выходной вектор (исходы). Исходы кодируются так, что все метки, считающиеся истинными, равны 1, а остальные 0.

---

[1] https://nd4j.org/doc/org/nd4j/linalg/dataset/DataSet.html.

Связь с NDArray

Объект DataSet содержит пару векторов NDArray, представляющих входной и выходной векторы. Именно так представляется вход и выход функции в процессе обучения.

Типичные применения

В примере ниже показано, как использовать ND4J API для получения признаков и меток из объекта DataSet. Мы видим, как эти структуры передаются обученной модели, которая порождает выходной NDArray.

```
DataSet t = ...
INDArray features = t.getFeatureMatrix();
INDArray lables = t.getLabels();
INDArray predicted = model.output(features,false);
```

Этот простой пример иллюстрирует связь наборами данных, объектами NDArray и моделями DL4J.

Создание входных векторов

Мы хотим специально выделить тему создания векторов, потому что эта техника используется в любой задаче, связанной с моделированием. Мы должны понимать как стратегию векторизации, так и применение ND4J API для ее практической реализации. Как уже было сказано, входными данными для модели является набор признаков, с которыми ассоциирован вектор меток. В этом разделе мы покажем, как создать то и другое и связать их вместе с помощью объекта DataSet.

Простые примеры векторизации

Приведенные ниже примеры упрощены, чтобы продемонстрировать основные приемы задания признаков и меток. Существуют и другие методы ND4J, которые могут оказаться более эффективными. Нередко читатели записей, включенные в DL4J, сами выполняют большую часть работы. В этом разделе мы только хотим познакомить вас с практическими способами манипуляции данными.

Основы создания векторов

Начнем с создания вектора с двумя столбцами:

```
INDArray myFeatures = ND4j.create(new float[]{0.5, 0.5},new int[]{1,2});
```

Задание размера вектора

Для задания размера вектора служит второй параметр метода ND4j.create():

```
new int[]{1,2}
```

В данном случае мы говорим, что хотим создать NDArray, содержащий одну строку с двумя столбцами признаков.

Задание значений признаков

Есть много способов задать значения признаков. В примере ниже показан простейший из них – задание элементов массива вручную:

```
for ( int row = 0; row < myFeatures.rows(); row++ ) {
    for ( int col = 0; col < myFeatures.getRow( row ).columns(); col++ ) {
        myFeatures.getRow(row).putScalar(col, 0.9);
    }
}
```

Здесь всем элементам присваивается значение 0.9.

Задание меток

Метки задаются так же, как признаки, поскольку в обоих случаях используется одна и та же структура данных: NDArray. При выборе значений нужно учитывать, какую модель мы строим и как хотим представлять выходные данные.

Одна метка на выходе. В этом случае выходной массив NDArray будет содержать один столбец, а метка может принимать значение 0.0 или 1.0:

```
myFeatures.getRow(0).putScalar(0, 1.0);
```

Несколько меток на выходе. В этом случае выходной массив NDArray будет содержать по одному столбцу для каждой метки. Каждому классу (метке) в обучающем наборе данных будет соответствовать определенный столбец в выходном NDArray. В примере ниже в столбец 1 записано значение метки 1.0, а в остальные два – значение 0.0:

```
myFeatures.getRow(0).putScalar(0, 0.0);
myFeatures.getRow(0).putScalar(1, 1.0);
myFeatures.getRow(0).putScalar(2, 0.0);
```

Регрессия. В регрессионной модели выходной массив NDArray должен содержать один столбец, в который записывается значение с плавающей точкой, ассоциированное с входным вектором.

```
myFeatures.getRow(0).putScalar(0, 55.25);
```

Класс MLLibUtil

В библиотеке DL4J имеются средства для поддержки интероперабельности с другими библиотеками машинного обучения. В этом разделе мы продемонстрируем несколько методов класса `MLLibUtil` для работы с объектами векторов Spark.

Преобразование INDArray в вектор MLLib

Для преобразования объекта NDArray в вектор MLLib служит метод `MLLibUtil.toVector( INDArray )`:

```
Vector prediction = MLLibUtil.toVector( myNDArray );
```

Преобразование вектора MLLib в INDArray

Для преобразования вектора MLLib в объект NDArray служит метод `MLLibUtil.toVector(Vector)`:

```
INDArray ndArray = MLLibUtil.toVector( labeledPoint.features() );
```

Получение предсказаний от модели в DL4J

В этом разделе мы на уровне API рассмотрим, как взаимодействуют классы из DL4J и ND4J для получения предсказаний от модели.

Совместное использование DL4J и ND4J

Обычно для получения предсказаний от модели мы используем класс `MultiLayerNetwork` для представления модели и передаем объект INDArray методу `output()`. Этот метод возвращает список вероятностей каждой метки, и метку с наибольшей вероятностью мы считаем предсказанием модели:

```
MultiLayerNetwork.output(INDArray input, boolean train)
```

Ниже мы увидим ряд примеров использования этого метода обученной модели DL4J для порождения выхода, представленного массивом NDArray.

Откуда берутся выходные значения?
Это значения активации в последнем (выходном) слое сети, полученные в результате прямого распространения входного вектора.

Массив NDArray, возвращенный методом `output()`, обычно имеет вид:

```
[0.5, 0.5]
```

Как видим, массив содержит одну строку с двумя столбцами. Значение в каждом столбце – оценка соответствующей метки. Интерпретация значений зависит от типа выходного слоя, а как именно, описано ниже.

Интерпретация выходного вектора в зависимости от типа выходного слоя

Напомним, что в случае функции активации softmax сумма выходов (вероятностей) равна 1.0, тогда как в случае сигмоиды ограничение налагается на каждый выход в отдельности, например для каждого выходного блока активация может получиться равной 0.9. У сигмоидных слоев нет «поперечных» ограничений.

Логистический выходной слой для бинарной классификации. Выход слоя такого типа представляет вероятность того, что бинарная метка равна true.

```
INDArray predictions = trainedNetwork.output( ndArrayFeatures );
```

Каждый элемент выходного вектора представляет метку одного типа и не зависит от других меток (в теоретико-вероятностном смысле).

Выходной слой softmax для многометочной классификации. Выходной слой содержит список вероятностей, сумма которых равна 1.0:

```
INDArray predictions = trainedNetwork.output( ndArrayFeatures );
for ( int row = 0; row < output.rows(); row++ ) {
    System.out.println( "Input Row: " + row );
    for ( int col = 0; col < output.getRow( row ).columns(); col++ ) {
        System.out.println( "\tColumn: " + col + ":" + output.getRow( row )
            .getDouble( col ) );
    }
}
```

В результате будет напечатано:

```
Input Row: 0
    Column: 0:0.996791422367096
    Column: 1:0.0032307980582118034
Input Row: 1
    Column: 0:0.0016306628240272403
    Column: 1:0.9983481764793396
Input Row: 2
    Column: 0:0.0016311598010361195
    Column: 1:0.9983481168746948
Input Row: 3
    Column: 0:0.9988488554954529
    Column: 1:0.0011729325633496046
```

Каждый элемент выходного вектора представляет вероятность ассоциированной с ним метки. С каждым столбцом ассоциирована своя метка.

Линейный выходной слой в регрессионной модели. В регрессионной модели единственное выходное значение, возвращаемое «тождественной» (линейной) функцией активации, представляет значение искомой величины.

Получение предсказанной метки из возвращенного INDArray

Находим максимум в массиве значений, возвращенных методом `output()`:

```
INDArray predictions = trainedNetwork.output( ndArray );
int maxLabelIndex = Nd4j.getBlasWrapper().iamax( predictions );
```

Преимущества метода .output() по сравнению с .predict()

Метод `output()` можно использовать и для регрессии, и для классификации, так что он оказывается несколько более гибким.

Библиотека DataVec

Алекс Блэк

DataVec – библиотека для работы с данными в машинном обучении. Она отвечает за компонент извлечения, преобразования и загрузки (ETL), или *векторизации*. Цель DataVec – упростить подготовку и загрузку исходных данных в формате, пригодном для машинного обучения. DataVec умеет загружать табличные (файлы в формате CSV) данные, изображения и временные ряды как для обработки на одной машине, так и для распределенной обработки (с помощью Apache Spark).

DataVec и создание вектора ND4J

DataVec берет на себя многие рутинные обязанности по созданию векторов признаков и меток, упомянутые ранее в этой книге. В технологических конвейерах на одной машине и в кластере Spark рекомендуется применять DataVec.

Возможности DataVec можно отнести к двум категориям:

- загрузка данных в различных форматах;
- выполнение типичных операций преобразования данных (часто называется *подготовкой данных* [data wrangling или data munging]).

Загрузка данных для машинного обучения

Данные для машинного обучения поступают в различных форматах, и для каждого нужны свои средства загрузки. На практике нередко приходится писать одноразовый код для загрузки данных, что отнимает много времени и чревато ошибками. DataVec оказывает помощь в двух отношениях: во-первых, предоставляет общую функциональность загрузки данных в типичных случаях (например, чтение изображений и файлов в формате CSV), а во-вторых, содержит простой набор абстракций для реализации новых форматов и источников данных.

ETL, предварительная обработка и векторизация

Рекомендуется использовать DataVec или другой подобный инструмент для предварительного преобразования данных в формат, который DL4J умеет читать и автоматически векторизовать. Векторизация исходных данных вручную – трудоемкая работа, без которой лучше бы обойтись.

Набор абстракций, предоставляемых DataVec, довольно прост.

Интерфейс Writable представляет элемент данных. Например, можно использовать класс DoubleWritable для представления чисел с двойной точностью, а класс Text – для текстовых данных[1].

Интерфейс RecordReader предлагает механизм перехода от формата исходных данных к формату примера. Точнее, объект типа RecordReader принимает исходные данные и преобразует их в список List<Writable>, который может быть прочитан объектом класса RecordReaderDataSetIterator, входящего в состав DL4J; этот же класс отвечает за формирование мини-пакетов (объектов класса DataSet) из примеров. Весь процесс показан на рис. F.1.

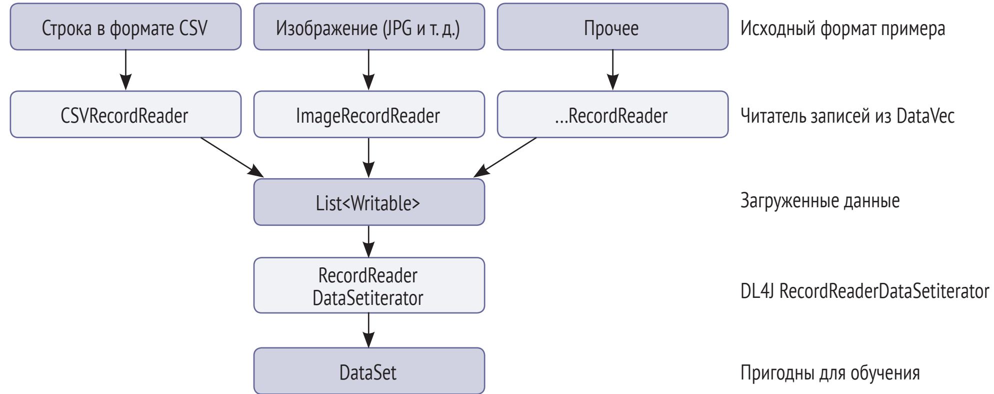


Рис. F.1 ❖ Обработка данных с помощью DataVec

Аналогичный интерфейс SequenceRecordReader предоставляет механизм загрузки последовательных данных (временных рядов). Если единичный пример в DataVec представляется объектом типа List<Writable>, то последовательность – объектом типа List<List<Writable>>. Здесь внешний список – это список временных шагов, а внутренний – список значений на каждом временном шаге. Иначе говоря, выражение mySequence.get(i).get(j) возвращает j-ое значение на i-м временном шаге. Читатель записей-последовательностей используется практически так же, как читатель обычных записей (рис. F.2).

Важно, что благодаря паттерну Итератор данные загружаются только тогда, когда в них возникает необходимость. Это означает, что мы можем загружать обучающие данные в память постепенно (с асинхронной предвыборкой, если необходимо), а не сразу все (что в случае больших наборов было бы невозможно). Для этой цели в интерфейсах RecordReader, SequenceRecordReader и DataSetIterator имеются методы next(), hasNext() и reset().

[1] Имеются также типы IntWritable, LongWritable, FloatWritable и NullWritable. Кроме того, DataVec предоставляет класс NDArrayWritable для эффективной работы с числовыми массивами, представленными средствами библиотеки ND4J, например изображениями.

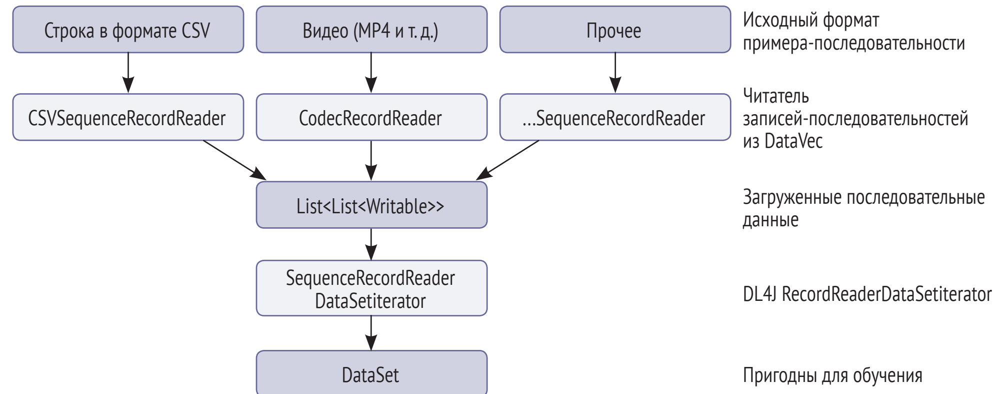


Рис. F.2 ❖ Обработка данных с помощью DataVec

Загрузка CSV-данных для многослойного перцептрона

Для загрузки данных в формате CSV достаточно всего нескольких строчек кода. Описанная ниже процедура годится и для других форматов с разделителями, например когда разделителем является знак табуляции.

В примере F.1 показано, как загрузить файл данных в формате CSV и создать объект `DataSetIterator`, пригодный для обучения сети в DL4J.

Пример F.1 ❖ Загрузка данных в формате CSV

```
// Определим местонахождение файла и некоторые свойства:
File file = new File( "/path/to/my/file.csv" );
int numLinesToSkip = 0;     // Факультативно, позволяет пропустить строки
String delimiter = ",";     // Запятая в качестве разделителя

// Создать и инициализировать читатель записей:
RecordReader reader = new CSVRecordReader( numLinesToSkip, delimiter );
InputSplit inputSplit = new FileInputSplit( file );
reader.initialize( inputSplit );

// Создать DataSetIterator. Предполагается, что мы решаем задачу классификации:
int minibatchSize = 10;     // Число примеров в одном мини-пакете
int labelIndex = 7;         // Индекс столбца, содержащего метку
int numClasses = 5;         // Число классов (различных меток)
DataSetIterator iterator =
    new RecordReaderDataSetIterator( reader, minibatchSize, labelIndex, numClasses );

// Обучить сеть, применяя DataSetIterator:

myNetwork.fit( iterator );
```

Отметим наиболее важные моменты.

- конструктор без аргументов (`new CSVRecordReader()`) предполагает, что разделителем является запятая и пропускается 0 строк;
- при создании объекта `RecordReaderDataSetIterator` размер мини-пакета задается пользователем;

- в классе `RecordReaderDataSetIterator` существует ряд других конструкторов, например для многометочной регрессии и случая, когда метки отсутствуют, как при обучении без учителя;
- для задачи классификации класс `RecordReaderDataSetIterator` предполагает, что в одном из столбцов находятся целочисленные метки, равные индексу класса, т. е. значения от 0 до `numClasses – 1` включительно;
- допустимы другие разделители, например `\t` (знак табуляции) или статическое поле `CSVRecordReader.QUOTE_HANDLING_DELIMITER` на случай, если поля записей могут быть заключены в кавычки и содержать внутри запятые;
- `CSVRecordReader` выводит примеры в том порядке, в котором они встречаются в файле. Если порядок данных не случайный (например, сначала идут все примеры из класса 0, затем все примеры из класса 1), то данные следует предварительно перемешать.

Загрузка изображений для сверточной нейронной сети

Класс `ImageRecordReader` предназначен для чтения изображений и выполнения типичных операций над ними, например кадрирования. Поддерживаются различные графические форматы (JPG, PNG, TIFF, BMP и другие), а эффективность загрузки и преобразований обеспечивается библиотеками JavaCV и OpenCV.

Класс `ImageRecordReader` обладает большой гибкостью в части загрузки файлов данных. В примере F.2 приведена простая демонстрация. Предполагается, что файлы изображений находятся в каталогах, имена которых совпадают с метками (класс изображения): *.../root_directory/label_0*, *.../root_directory/label_1/*. Отметим, что метки могут быть произвольными строками.

Пример F.2 ❖ Загрузка изображений

```
// Сначала зададим глубину входных изображений (число каналов)
int inputNumChannels = 3;     // 3 - цветное RGB, 1 - полутоновое
int outputHeight = 32;        // Масштабировать на высоту 32 пикселя
int outputWidth = 32;         // Масштабировать на ширину 32 пикселя

// Зададим корневой каталог и допустимые форматы
// Объект класса Random используется для рандомизации порядка файлов
File rootDir = new File( "/path/to/my/root_directory/" );
String[] allowedExtensions = BaseImageLoader.ALLOWED_FORMATS;
Random rng = new Random();
FileSplit inputSplit = new FileSplit( rootDir, allowedExtensions, rng );

// С каждым изображением должна быть ассоциирована метка. Для ее получения используем
// путь к файлу с помощью класса ParentPathLabelGenerator
ParentPathLabelGenerator labelMaker = new ParentPathLabelGenerator();

// Создать и инициализировать ImageRecordReader
ImageRecordReader reader =
    new ImageRecordReader( outputHeight, outputWidth, inputNumChannels, labelMaker );
reader.initialize( inputSplit );

// Создать объект DataSetIterator:
int minibatchSize = 10;      // Число примеров в одном мини-пакете
int labelIndex = 1;          // Для ImageRecordReader всегда 1
int numClasses = 3;          // Число классов (различных меток)
```

```
DataSetIterator iterator =
    new RecordReaderDataSetIterator( reader, minibatchSize, labelIndex, numClasses );

// Обучить сеть, применяя DataSetIterator:

myNetwork.fit( iterator );
```

У класса `ImageRecordReader` есть много других возможностей. Например, чтобы разбить набор изображений на обучающий и тестовый, можно добавить в пример F.2 такой код.

Пример F.3 ❖ Разбиение набора данных на обучающий и тестовый

```
InputSplit[] trainTest = inputSplit.sample( null, 80, 20 );
InputSplit trainData = trainTest[0];    // 80% под обучающий
InputSplit testData = trainTest[1];     // 20% под тестовый

(прочий код опущен)

reader.initialize( trainData );
```

Чтобы добавить в процесс дополнительные шаги преобразования, можно передать конструктору `ImageRecordReader` объект класса `ImageTransform`, как показано в примере F.4, где мы случайным образом переворачиваем и кадрируем изображения.

Пример F.4 ❖ Шаги преобразований

```
int maxCropPixels = 20;

ImageTransform transform =
    new MultiImageTransform( new Random(),
        new FlipImageTransform(), new CropImageTransform( maxCropPixels ) );


reader.initialize( trainData, transform );
```

Загрузка последовательных данных для рекуррентных нейронных сетей

Для последовательных данных (временных рядов) есть много представлений и форматов. В этом разделе мы рассмотрим простой, но полезный формат:

- данные хранятся в формате CSV, по одному временному ряду в файле (длина временного ряда в разных файлах может быть различной);
- строка CSV-файла представляет один временной шаг;
- для всех временных шагов в файле представлены признаки и метки (т. е. один столбец содержит метку для классификации, а все остальные содержат признаки).

Как и в примере CSV-файла выше (для табличных данных, а не последовательностей), в примере F.5 предполагается задача классификации, когда метка класса – целое число от 0 до `numClasses – 1` включительно.

Пример F.5 ❖ Загрузка последовательных данных

```
// Зададим базовый каталог, содержащий CSV-файлы
File baseDir = new File("/path/to/base_directory/");
```

```
// Создать и инициализировать читатель записей-последовательностей
// Для рандомизации порядка файлов используется генератор случайных чисел
InputSplit inputSplit = new FileSplit(baseDir, new Random());
int numLinesToSkip = 0;    // Факультативно, позволяет пропустить строки заголовка
String delimiter = ",";    // Запятая в качестве разделителя

SequenceRecordReader reader = new CSVSequenceRecordReader(numLinesToSkip, delimiter);
reader.initialize(inputSplit);

// Создать объект DataSetIterator:
int minibatchSize = 10;    // Число примеров в одном мини-пакете
int labelIndex = 7;        // Индекс столбца, содержащего метку
int numClasses = 5;        // Число классов (различных меток)
boolean regression = false;

DataSetIterator iterator =
    new SequenceRecordReaderDataSetIterator( reader, minibatchSize, numClasses,
        labelIndex, regression );

DataSetIterator iterator =
    new SequenceRecordReaderDataSetIterator( reader, minibatchSize, labelIndex,
        numClasses );

// Обучить сеть, применяя DataSetIterator
myNetwork.fit(iterator);
```

Класс `SequenceRecordReader` поддерживает и другие сценарии, например регрессию и загрузку признаков и меток из разных файлов с помощью разных объектов `SequenceRecordReaders`.

Подготовка данных средствами DataVec

Зачастую данные для машинного обучения представлены в виде, нуждающемся в дополнительной предварительной обработке. Это может быть совсем простая операция, например удаление ненужных столбцов, или весьма сложная, например соединение и очистка данных из нескольких независимых источников. DataVec предоставляет средства для этого этапа конвейера, включая развитые преобразования стандартных данных и последовательностей.

Типичный технологический процесс подготовки данных выглядит следующим образом:

1) определить схему исходных данных (об этом чуть ниже);
2) определить набор операций над данными (удаление столбцов, обработка отсутствия значений и т. д.);
3) загрузить данные;
4) выполнить операции;
5) сохранить обработанные данные.

Если данные нуждаются в предварительной обработке, то мы рекомендуем отделить ее от обучения сети, т. е. сначала выполнить предварительную обработку, сохранить ее результаты на диске, а затем загрузить их с диска, когда понадобится обучить сеть.

Для выполнения набора операций над набором данных используется Apache Spark; это дает возможность работать как в кластере, так и на одной машине (в ло-

кальном режиме Spark). Кроме того, благодаря Spark мы можем масштабировать обработку на большие наборы данных (сотни миллионов и более записей). Хотя в настоящее время реализовано только выполнение на платформе Spark, следует иметь в виду, что сам API DataVec не зависит от платформы, в будущем могут быть добавлены и другие системы кластерных вычислений.

Преобразования DataVec: основные понятия

Для поддержки преобразований в DataVec используется несколько важных идей и классов.

Во-первых, примеры в DataVec представляются так, как описано в предыдущем разделе: для стандартных данных каждый пример – это список `List<Writable>`, а для последовательностей – список списков `List<List<Writable>>`. Результаты каждой операции будут представлены в одном из этих форматов. Это также означает, что для загрузки данных, подлежащих преобразованию, можно использовать классы `RecordReader` и `SequenceRecordReader`.

Схемой в DataVec называется класс, определяющий три вещи:

- имя каждого столбца;
- тип каждого столбца (числовой, категориальный, строковый и т. д.);
- ограничения (если имеются) на допустимые значения в каждом столбце (например, столбец может содержать только положительные значения).

Для последовательных данных применяется класс `SequenceSchema`, который для каждого столбца содержит ту же информацию, что обычная схема. DataVec отслеживает изменения схемы после каждой операции. Мы можем запросить схему после выполнения всех операций или в любой точке процесса.

Класс `TransformProcess` определяет последовательность операций с данными. Для его конструирования нужно знать:

- схему исходных данных;
- множество подлежащих выполнению операций.

Все это задается с помощью паттерна Построитель (см. примеры ниже) или с применением конфигурационного файла в формате JSON или YAML.

DataVec предлагает несколько типов операций над данными.

Таблица F.1. Типы операций DataVec

Название	Описание	Примеры применения
Преобразование	Общая операция, применимая к одному примеру или одной последовательности	Удаление столбцов, математические операции, разбор даты и времени
Фильтрация	Удаление примеров, удовлетворяющих условию	Отбрасывание примеров, в которых некоторые значения отсутствуют или некорректны
Редукция	Группировка примеров по ключу и редукция	Вычисление минимума, максимума или суммы для каждого заказчика
Преобразование в последовательность	Группировка отдельных примеров в последовательность по одному или нескольким ключевым столбцам	Обработка журнала: группировка в последовательность записей с одинаковым IP-адресом или идентификатором заказчика
Преобразование из последовательности	Разбиение каждого временного шага последовательности данных на отдельные примеры	Разложение последовательности транзакций на отдельные записи

Преобразования DataVec: пример

В этом разделе приведен простой пример выполнения типичных операций над небольшим набором данных.

Будем предполагать, что у нас имеются данные о транзакциях, для которых мы хотим предсказать метку (мошенническая или нормальная). Предположим также, что имеется простой набор данных со столбцами [customerID, dateTime, amount, label]:

```
3420348,2016-01-01 06:55:07,150.00,legitimate
9087434,2016-01-01 15:16:18,78.10,legitimate
4530843,2016-01-02 11:39:24,780.83,fraud
```

Первым делом определим схему данных:

```
Schema schema = new Schema.Builder()
.addColumnLong("customerID")
.addColumnString("dateTime")
.addColumnDouble("amount")
.addColumnCategorical("label", Arrays.asList("legitimate","fraud"))
.build();
```

Столбцы здесь перечислены в том же порядке, что и в файле.

Однако в таком виде набор данных использовать невозможно. Для обучения нейронной сети все входные данные должны быть числовыми. Чтобы подготовить данные для обучения, выполним следующие действия:

- удалим столбец с идентификатором заказчика;
- преобразуем категориальную (строковую) метку в целое число 0 или 1;
- разберем столбец dateTime, выделив из него час суток, который будем использовать в качестве нового признака.

Все эти операции можно описать с помощью класса `TransformProcess`:

```
TransformProcess process = new TransformProcess.Builder(schema)
.removeColumns("customerID")
.categoricalToInteger("label")
.stringToTimeTransform("dateTime","YYYY-MM-dd HH:mm:ss", DateTimeZone.UTC)
.transform(new DeriveColumnsFromTimeTransform.Builder("dateTime")
    .addIntegerDerivedColumn("hourOfDay", DateTimeFieldType.hourOfDay()).build())
.removeColumns("dateTime")
.build();
```

Операции выполняются в том порядке, в каком определены. По большей части назначение операции понятно из названия. Информацию об остальных (например, `stringToTimeTransform`) можно найти в документации по DataVec.

Наконец, следует загрузить данные, выполнить только что определенные операции и сохранить результат. Делается это так:

```
// Инициализация Apache Spark:
SparkConf conf = new SparkConf();
conf.setMaster("local[*]");
conf.setAppName("DataVec Example");
JavaSparkContext sc = new JavaSparkContext(conf);
```

```
// Загрузить данные с помощью Spark
String path = "/path/to/my/file.csv";
JavaRDD<String> lines = sc.textFile(path);
JavaRDD<List<Writable>> examples =
    lines.map(new StringToWritablesFunction(new CSVRecordReader()));

// Выполнить операции
SparkTransformExecutor executor = new SparkTransformExecutor();
JavaRDD<List<Writable>> processed = executor.execute(examples, process);

// Сохранить обработанные данные:
JavaRDD<String> toSave = processed.map(new WritablesToStringFunction(","));
toSave.saveAsTextFile("/path/to/save/to/");
```

По завершении обработки данные примут такой вид:

```
6,150.00,0
15,78.10,0
11,780.83,1
```

В этих данных имеются столбцы «hourOfDay», «amount» и «label». Мы можем получить от схемы имена и типы выходных столбцов, вызвав метод `process.getFinalSchema()`.

Вот, собственно, и все. Всего несколько строчек – и мы загрузили данные, выполнили ряд операций предварительной обработки (способом, который легко масштабируется на очень большие наборы данных) и сохранили данные в виде, пригодном для обучения. Пример очень простой, но мы надеемся, что он демонстрирует полезность и гибкость библиотеки DataVec для подготовки данных к машинному обучению.

Приложение G

Работа с DL4J на уровне исходного кода

Некоторые разработчики, возможно, захотят написать свои расширения, модифицировать ядро DL4J или работать с последней доступной версией. Для них мы рассказываем, как работать с DL4J на уровне исходного кода.

GitHub – размещенная в Интернете система управления версиями, которая де-факто является пристанищем для большинства проектов с открытым исходным кодом. Если вы планируете внести свой вклад в проекты ND4J или DL4J в виде исправления ошибок или написания нового кода, то вам понадобятся Git и GitHub.

А так ли вам нужно работать с исходным кодом?
Если вы собираетесь просто использовать библиотеку, то устанавливать Git не нужно, и исходный код скачивать ни к чему.

Проверка, установлен ли Git

Чтобы проверить, что Git установлен и работает, выполните следующую команду:

```
git --version
```

Если команда напечатает сообщение об ошибке, то установите Git. Если у вас еще нет учетной записи в GitHub, рекомендуем зарегистрироваться, это несложно и бесплатно.

Клонирование основных проектов, связанных с DL4J

Чтобы работать с исходным кодом, нужно клонировать ряд проектов. Для этого выполните следующие команды:

```
git clone https://github.com/deeplearning4j/nd4j
git clone https://github.com/deeplearning4j/datavec
git clone https://github.com/deeplearning4j/deeplearning4j
```

Если хотите, можете также клонировать примеры использования, работающие с откомпилированными версиями ND4J или DL4J (номер версии может изменяться):

```
git clone https://github.com/deeplearning4j/dl4j-0.4-examples
```

Дополнительная информация о примерах
Обзор установки примеров из Git и работы с IntelliJ и Maven см. на странице Quickstart по адресу https://deeplearning4j.org/quickstart.html#walk.

Скачивание исходного кода в виде ZIP-файла

Исходный код можно получить также, нажав кнопку Download ZIP на странице ND4J на GitHub (https://github.com/deeplearning4j/nd4j/archive/master.zip).

Сборка библиотеки из исходного кода с помощью Maven

Для корректной сборки библиотек ND4J, DataVec и DL4J используйте Maven в сочетании с Git. Чтобы гарантированно получить последнюю работающую версию, перейдите в корневой каталог соответствующего проекта и выполните команду

```
mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
```

Выполнение целей clean и install для библиотек ND4J, DataVec и DL4J в указанном порядке – правильный способ получить последние исправления ошибок и самые свежие возможности.

Приложение H

Подготовка проектов на базе DL4J

DL4J – набор инструментов, в совокупности составляющих полноценную платформу для глубокого обучения. Для выполнения различных функций, необходимых для обучения глубоких моделей, требуется много дополнительных библиотек. Для управления зависимостями при сборке проекта DL4J использует программу Maven. В этом разделе мы поговорим о нескольких зависимостях, которые бывают нужны для построения собственных глубоких моделей, инструментов и интеграции с другими системами.

Создание нового проекта на базе DL4J

DL4J – проект с открытым исходным кодом, ориентированный на профессиональных Java-разработчиков, знакомых с технологией производственного развертывания, такими интегрированными средами, как IntelliJ, и с автоматизированными средствами сборки типа Maven. Все это должно быть уже установлено, чтобы работа с нашим инструментом была эффективной. ND4J и DataVec, наша библиотека векторизации, будут автоматически установлены, если выполнить приведенные ниже инструкции.

К системе разработки предъявляются следующие требования:

1) Java 7 или более поздней версии;
2) Maven 3.2.5 или более поздней версии (средство управления зависимостями и автоматизации сборки);
3) Git.

Дополнительно может понадобиться установить:

- Cuda 7 для GPU;
- Scala 2.10.x;
- Windows;
- GitHub.

Приступим к подготовке среды и начнем с Java.

Java

Java – основной язык ND4J, поскольку используется для всего на свете: от распределенных облачных систем с тысячами узлов до устройств интернета вещей с небольшим объемом памяти. Это язык, на котором «один раз написал – и всюду работает».

Чтобы проверить установленную версию Java (да и установлена ли она вообще), выполните команду

```
java -version
```

Если Java 7 (или более поздняя версия) не установлена, скачайте Java Development Kit (JDK) по адресу https://docs.oracle.com/javase/7/docs/webnotes/install/mac/mac-jdk.html. Для более новых версий Mac в сообщении о версии должна упоминаться система Mac OS X (число после jdk-7u увеличивается на 1 при каждом обновлении версии):

Mac OS X x64 185.94 MB – jdk-7u79-macosx-x75.dmg

Работа с Maven

Программа Maven – средство автоматической сборки проектов на Java (среди прочего). Она находит последние версии библиотек ND4J и DL4J (*jar*-файлов) и автоматически скачивает их. Соответствующие файлы находятся на сайте Maven Central по адресу https://search.maven.org/. Maven хорошо интегрируется с такими средами разработки (IDE), как IntelliJ.

Чтобы проверить, установлена ли на компьютере Maven и какой версии, выполните команду

```
mvn --version
```

Если установлена не последняя версия Maven (на момент написания книги 3.3.x), обновите ее. Инструкции по установке Maven имеются на сайте https://maven.apache.org/.

Скачайте сжатый файл, содержащий последнюю стабильную версию Maven, и следуйте инструкциям для своей операционной системы, изложенной в соответствующем разделе, например «*Unix-based Operating Systems (Linux, Solaris, and Mac OS X)*».

Минимальный POM-файл

Для включения DL4J в свои проекты мы рекомендуем использовать Apache Maven пишущим на Java или SBT – пишущим на Scala. Ниже перечислены основные зависимости и их версии:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
           http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>MyGroupID</groupId>
    <artifactId>MyArtifactId</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <nd4j.version>0.5.0</nd4j.version>
        <dl4j.version>0.5.0</dl4j.version>
```

```
        <datavec.version>0.5.0</datavec.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.deeplearning4j</groupId>
            <artifactId>deeplearning4j-core</artifactId>
            <version>${dl4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-x86</artifactId>
            <version>${nd4j.version}</version>
        </dependency>

    </dependencies>
</project>
```

Что такое объектная модель проекта. Файлы, описывающие объектную модель проекта (Project Object Model – POM), могут быть довольно сложными, поэтому мы остановимся на основных моментах конфигурации, чтобы вы лучше понимали, что происходит. Вот две важнейшие зависимости, которые должны присутствовать в POM-файле любого проекта на базе DL4J:

- `deeplearning4j-core` – содержит реализации основных алгоритмов нейронных сетей;
- `nd4j-x86` – версия вспомогательной библиотеки ND4J для CPU.

Интегрированные среды разработки (IDE)

IDE упрощает работу с API библиотеки и создание конфигурации сети. Мы рекомендуем IntelliJ или Eclipse, обе работают с установленной версией Java и умеют взаимодействовать с Maven для управления зависимостями.

Краткое руководство по использованию IntelliJ в проекте на базе DL4J

Для бесплатной версии IntelliJ имеются инструкции по установке.

После установки выполните следующие действия (пользователи Windows могут ознакомиться с разделом «Walkthrough» на странице по адресу http://deeplearning4j.org/quickstart.html#walk):

1) выполните команду:

   ```
   git clone https://github.com/deeplearning4j/dl4j-0.4-examples.git
   ```

   Предполагается, что мы работаем с примерами к версии 0.0.4.x;
2) в меню IntelliJ выберите команду **File** ⇒ **New** ⇒ **Project from Existing Sources** и создайте новый проект с использованием Maven;
3) укажите на корневой каталог примеров. В результате он откроется в IDE;
4) скопируйте файл pom.xml, описанный в следующем разделе. Это описание проекта, управляемого Maven;
5) дождитесь, когда IntelliJ загрузит все зависимости (к правом нижнем углу имеется индикатор хода выполнения);
6) выберите пример из дерева файлов слева и нажмите «run».

Настройка других POM-файлов Maven

В этом разделе мы рассмотрим, как настроить файл *pom.xml* для ND4J.

ND4J и Maven

В библиотеке ND4J реализованы линейно-алгебраические операции, без которых никакие нейронные сети работать не могут. Оптимальная версия ND4J зависит от процессора: для CPU быстрее всего работает версия x86, а для GPU – версия Jcublas. Чтобы найти версии ND4J на сайте Maven Central, выполните следующие действия:

1) щелкните по номеру версии ниже Latest Version;
2) скопируйте код зависимости в левой части следующего экрана;
3) в IntelliJ вставьте этот код в файл pom.xml в корне проекта.

Код зависимости для nd4j-x86 выглядит так:

```
<dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-x86</artifactId>
    <version>${nd4j.version}</version>
</dependency>
```

Версия nd4j-x86 работает со всеми примерами. Инструкции по установке дополнительной зависимости OpenBlas на платформах Windows и Linux имеются на странице «Getting Started» проекта DL4J.

Приложение I

Подготовка проектов на базе DL4J к работе с GPU

Вячеслав Кокорин и Сьюзан Эрали

При обучении нейронных сетей приходится выполнять много вычислений, связанных с линейной алгеброй. Графические процессоры (GPU), оснащенные тысячами ядер, проектировались специально для подобных задач. Поэтому они часто применяются для ускорения обучения и позволяют значительно повысить рентабельность инвестиций.

Переключение библиотек в режим работы с GPU

DL4J может работать с графическими процессорами компании nVIDIA, поддерживающими архитектуру Cuda 7.5. Поддержка Cuda 8.0 будет добавлена, как только эта версия появится. Архитектура DL4J спроектирована в расчете на взаимозаменяемость модулей. Это значит, что для переключения вычислений из режима CPU на режим GPU достаточно изменить строчку `artifactId` в зависимости `nd4j` в файле *pom.xml*.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-cuda-7.5-platform</artifactId>
            <version>${nd4j.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Выбор GPU

Вообще говоря, рекомендуется высокопроизводительная модель потребительского класса или устройство Tesla профессионального уровня. На момент написания этой книги двух GPU NVidia GeForce GTX 1070 было бы вполне достаточно для начала.

При покупке GPU нужно обращать внимание на следующие характеристики:

- количество мультипроцессоров (ядер) на плате – тут все просто – чем больше, тем лучше. От числа ядер зависит, сколько потоков GPU может исполнять одновременно;

- объем доступной устройству памяти – от этого зависит, сколько данных можно передать устройству для обработки. Важен и тип памяти, поскольку он определяет пропускную способность шины, а значит, и скорость передачи. Тип GDDR5 считается абсолютным минимумом. GDDR5X лучше, а HBM/HBM2 – предел желаний.

Отметим также факультативную встроенную поддержку половинной точности, имеющуюся в устройствах Tesla P100 высшего класса и специализированных устройствах Tegra. При наличии этой возможности скорость глубокого обучения можно повысить на 200–300% в зависимости от размерности данных и размера модели.

Важно также учитывать, как соединены устройства. На момент написания книги на рынке были представлены только две технологии межсоединения: PCIe и NVLink, причем NVLink очевидно лидировала, обеспечивая полосу пропускания шириной 160 ГБ/с. Поэтому наличие NVLink – крайне желательная характеристика любой системы с несколькими GPU для всех моделей параллельного глубокого обучения. NVIDIA даже предлагает готовый сервер DGX-1 с 8 устройствами Tesla P100 и межсоединениями NVLink. Он стоит внушительных денег и умно рекламируется как «суперкомпьютер в коробке». Опровергнуть этот тезис нелегко.

Обучение на системе с несколькими GPU

DL4J поддерживает обучение на системе с несколькими GPU в режиме распараллеливания по данным. Класс `ParallelWrapper` может преобразовать существующую модель в обучаемую параллельно.

Приведем простой пример:

```
ParallelWrapper wrapper = new ParallelWrapper.Builder(YourExistingModel)
        .prefetchBuffer(24)
        .workers(4)
        .averagingFrequency(1)
        .reportScoreAfterAveraging(true)
        .build();
```

Объект `ParallelWrapper` принимает существующую модель в качестве аргумента и производит ее обучение параллельно. При наличии нескольких GPU рекомендуется, чтобы число исполнителей было не меньше числа GPU. Точное значение надо настраивать, поскольку оно зависит от характера задачи и от оборудования.

Внутри `ParallelWrapper` исходная модель дублируется, и каждый исполнитель обучает свою копию модели. После каждых *X* итераций, число которых задается методом `averagingFrequency(X)`, параметры всех моделей усредняются и раздаются всем исполнителям. Затем обучение продолжается.

CUDA на разных платформах

Ниже показано, где найти подробные инструкции по CUDA для разных платформ:

- CUDA для Linux – http://docs.nvidia.com/cuda/#axzz4C8adWDKM;
- CUDA для Windows – http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-microsoft-windows/#axzz4C8adWDKM;
- CUDA для OS X – http://docs.nvidia.com/cuda/#axzz4C8adWDKM.

Мониторинг производительности GPU

Начав обучать нейронную сеть на GPU, вы, конечно, захотите узнать, насколько хорошо GPU работают. Для этого рекомендуется установить интерфейс управления системой NVIDIA (SMI) – см. https://developer.nvidia.com/nvidia-system-management-interface.

Использование NVIDIA SMI

В журналах очень много информации. Чтобы разобраться в том, что делает ND4J, ищите слово «Java».

Приложение J

Отладка проблем с установкой DL4J

Если при запуске примеров возникают ошибки, придется произвести небольшое расследование. Далее мы обсудим типичные проблемы, с которыми сталкиваются начинающие пользователи DL4J.

Предыдущая установка

Если на компьютере уже была установлена DL4J, а после установки новой версии посыпались ошибки, обновите библиотеки. Если вы используете Maven, то нужно только изменить номера версий в файле *pom.xml* в соответствии с последними версиями на сайте Maven Central. При работе с исходным кодом выполните команду `git clone` для ND4J, Canova и DL4J, а затем команду `mvn clean install -Dskiptests=true -Dmaven.javadoc.skip=true` в каждом из трех каталогов, именно в таком порядке.

Ошибки нехватки памяти при сборке из исходного кода

С ростом кодовой базы для сборки библиотеки из исходного кода требуется больше памяти. Если в ходе сборки DL4J возникнет ошибка `Permgen error`, то увеличьте размер кучи. Для этого следует изменить скрытый файл *.bash_profile* в своем домашнем каталоге. В этом файле задаются значения переменных среды. Чтобы увеличить размер кучи до 512 МБ, выполните такую команду:

```
echo "export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=512m"" > ~/.bash_profile
```

Старые версии Maven

При работе со старыми версиями Maven, например 3.0.4, могут возникать исключения `NoSuchMethodError`. Чтобы исправить ошибку, перейдите на последнюю версию, на момент написания книги это была версия 3.3.x. Чтобы узнать, какая версия Maven установлена, выполните команду `mvn -v`.

Maven и переменная среды PATH

После установки Maven может появиться сообщение

```
mvn is not recognised as an internal or external command,
operable program or batch file.
```

Это означает, что путь к Maven отсутствует в переменной среды PATH, которую можно изменить, как и любую другую переменную среды.

Недопустимые версии JDK

Если вы видите сообщение:

```
Invalid JDK version in profile 'java8-and-higher': Unbounded range:
    [1.8, for project com.github.jai-imageio:jai-imageio-core
    com.github.jai-imageio:jai-imageio-core:jar:1.3.0
```

значит, что-то не так с Maven. Перейдите на версию 3.3.x.

C++ и другие средства разработки

Для компиляции некоторых зависимостей ND4J необходимо установить средства разработки для C и C++.

Инструкции по компиляции ND4J
Точные инструкции имеются в руководстве по ND4J на странице http://nd4j.org/getstarted.html#devtools.

Windows и путь к включаемым файлам

Путь к включаемым файлам для Java CPP не всегда работает в Windows. Одно из решений – скопировать файлы-заголовки из каталога include Microsoft Visual Studio в подкаталог include установочного каталога Java Run-Time Environment (JRE).

standardio.h
Это относится к таким файлам, как standardio.h. Дополнительную информацию можно найти по адресу http://nd4j.org/getstarted.html#windows.

Мониторинг GPU

Как уже отмечалось в приложении I, для мониторинга работы GPU можно установить NVIDIA System Management Interface (SMI).

Дополнительные сведения о мониторинге GPU см. по адресу http://nd4j.org/getstarted.html#gpu.

Использование JVisualVM

Одна из главных причин использовать Java – диагностика, зашитая в JVisualVM. Наберите в командной строке `jvisualvm` – и система покажет сведения о CPU, куче, PermGen, классах и потоках.

Представление Sampler
Одним из самых полезных является представление Sampler. Щелкните по крайней справа вкладке **Sampler**, а затем нажмите кнопку **CPU** или **Memory button** для получения выборочных данных.

Работа с Clojure

При использовании deeplearning4j-nlp из приложения, написанного на Clojure, и сборке uberjar-файла с помощью утилиты Leiningen необходимо прописать следующую строку в файле *project.clj*, чтобы ресурсные файлы akka *reference.conf* были правильно объединены:

```
:uberjar-merge-with {#"\.properties$" [slurp str spit] "reference.conf"
[slurp str spit]}
```

(Отметим, что первая запись в отображении для файла *properties* – обычное умолчание.) Если этого не сделать, то при попытке выполнить код из получившегося uberjar-файла возникнет исключение:

```
Exception in thread "main" com.typesafe.config.ConfigException$Missing:
No configuration setting found for key 'akka.version'
```

Поддержка чисел с плавающей точкой в OS X

Тип float в OS X поддерживается некорректно. Если при выполнении примеров вы встречаете NAN там, где должно быть число, перейдите на тип double.

Ошибка разветвления-соединения в Java 7

В Java 7 имеется ошибка в реализации разветвления-соединения. При переходе на Java 8 она пропадает. Если возникает ошибка OutofMemory, сопровождаемая приведенным ниже сообщением, то виновата проблема разветвления-соединения:

```
java.util.concurrent.ExecutionException: java.lang.OutOfMemoryError ….java.util
    .concurrent.ForkJoinTask.getThrowableException(ForkJoinTask.java:536)
```

Онлайновая поддержка через Gitter

Не стесняйтесь задавать вопросы, касающиеся сообщений об ошибках, на нашем живом чате Gitter по адресу https://gitter.im/deeplearning4j/deeplearning4j.
Задавая вопрос, включите следующую информацию (тогда можно рассчитывать на более быстрый ответ):

- операционная система (Windows, OS X, Linux) и ее версия;
- версия Java;
- версия Maven;
- трасса стека.

Предостережения

Ниже приведены рекомендации о том, что следует проверить, если примеры, прилагаемые к DL4J, не собираются или работают неправильно.

Клонирование других репозиториев

Убедитесь, что вы создали локальный клон правильного репозитория. Основной репозиторий DL4J постоянно совершенствуется, и в самой последней версии примеры, возможно, еще не были протестированы.

Проверьте зависимости Maven

Убедитесь, что все необходимые для работы примеров зависимости скачаны из репозитория Maven, а не найдены на локальной машине. Для удаления старых зависимостей выполните команду

```
rm -rf ls ~/.m2/repository/org/deeplearning4j
```

Переустановка зависимостей

Чтобы пересобрать примеры из исходного кода и правильно установить их, перейдите в каталог *dl4j-0.4-examples* и выполните команду

```
mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
```

Если ничего не помогает

При возникновении проблемы первым делом проверьте файл *pom.xml*.

Различные платформы

Для компиляции некоторых зависимостей ND4J необходимо установить средства разработки на C, в т. ч. компилятор gcc. Чтобы проверить, установлен ли gcc, выполните команду `gcc -v`.

OS X

В некоторых версиях инструмента разработки Xcode от компании Apple gcc уже установлен. Если это не так, выполните команду

```
brew install gcc
```

Windows

Пользователям Windows придется установить бесплатную версию Visual Studio Community 2017. Скачать ее можно по адресу https://www.visualstudio.com/ru/thank-you-downloading-visual-studio/?sku=Community&rel=15.

Задание переменной среды PATH

Путь к Visual Studio нужно добавить в переменную среды PATH вручную. Выглядит он как-то так: *C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin.*

Чтобы проверить, правильно ли установлен путь к Visual Studio, выполните команду:

```
cl
```

Может появиться сообщение об отсутствии некоторых DLL-файлов. Проверьте, что путь к папке Visual Studio включен в PATH. Если в ответ выводится информация о порядке запуска `cl`, значит, вы на верном пути.

Настройка Visual Studio

После установки Visual Studio 2017 и настройки переменной PATH необходимо выполнить файл *vcvars32.bat*, чтобы правильно задать значения переменных среды INCLUDE, LIB, LIBPATH и не копировать файлы-заголовки. Но если этот файл выполняется из Explorer, то заданные значения будут временными. Поэтому запускайте *vcvars32.bat* из того же окна CMD, что `mvn install`, – тогда все будет правильно.

Не забудьте установить C++

При установке Visual Studio нужно явно отметить C++. Он больше не устанавливается по умолчанию.

Java CPP и Windows

Ко всему прочему, путь к каталогу include для Java CPP не всегда работает в Windows. Одно из решений – скопировать файлы-заголовки из каталога include Microsoft Visual Studio в подкаталог include установочного каталога Java Run-Time Environment (JRE). Это относится, в частности, к файлу *standardio.h*.

Работа в Windows на 64-разрядной платформе

Получить DLL-файлы для Windows на 64-разрядной платформе можно по адресу http://avulanov.blogspot.cz/2014/09/howto-to-run-netlib-javabreeze-in.html.

Скачайте DLL-файлы в подкаталог bin установочного каталога Java (например, *C:\prg\Java\jdk1.7.0_45\bin*).

Другие зависимости

Библиотека *netlib-native_system-win-x86_64.dll* зависит от: *libgcc_s_seh-1.dll*, *libgfortran-3.dll*, *libquadmath-0.dll*, *libwinpthread-1.dll*, *libblas3.dll* и *liblapack3.dll* (*liblapack3.dll* и *libblas3.dll* – просто переименованные копии *libopeblas.dll*).

Скачать откомпилированные библиотеки можно с сайта http://www.openblas.net/.

Linux

Инструкции для дистрибутивов Ubuntu и Centos различны.

Ubuntu

Сначала выполните команду

```
sudo apt-get update
```

Затем нужно будет выполнить команду вида

```
sudo apt-get install linux-headers-$(uname -r) build-essential
```

`$(uname -r)` зависит от версии Linux. Чтобы узнать версию Linux, откройте окно терминала и введите команду:

```
uname -r
```

Будет напечатана строка вида `3.2.0-23-generic`. Скопируйте ее и вставьте в команду вместо `$(uname -r)`.

Centos

В окне терминала (или в сеансе ssh) выполните от имени пользователя root команду:

```
yum groupinstall 'Development Tools'
```

После этого по экрану терминалу побегут сообщения об установке различных пакетов. Когда все закончится, проверьте, установлен ли gcc:

```
gcc --version
```

Более подробные инструкции имеются по адресу http://www.cyberciti.biz/faq/centos-linuxinstall-gcc-c-c-compiler/.

Предметный указатель

A

AdaDelta, 100, 227
AdaGrad, 100, 227
Adam, 100, 227
AlexNet, 86, 130
AlphaGo, 350
Apache Hadoop, 233, 277, 301. *См.* также Spark и Hadoop
Apache Nutch проект, 303
ApplicationMaster, 307

C

CIFAR-10 набор данных, 119
Clojure, 402
CUDA, 236, 398

D

DataSetIterator класс, 154
DataSet класс, 154, 377
DataVec
  ETL и векторизация, 285
  векторизация последовательных данных, 290
  загрузка CSV-данных для многослойного перцептрона, 384
  загрузка изображений для сверточной нейронной сети, 385
  загрузка последовательных данных для рекуррентных нейронных сетей, 386
  нормировка и векторизация изображений, 288
  преимущества, 152
  преобразование данных, 388
  цель, 382
Deeplearning4j библиотека
  архитектура распараллеливания, 234
  вычислительная эффективность, 238
  использование совместно с ND4J, 380
  корректоры, 100, 225
  модель многослойного перцептрона, 156
  настройка задач для Spark, 311
  основные концепции, 153
    загрузка и сохранение моделей, 153
    задание архитектуры модели, 154
    обучение и оценивание, 155
    получение входных данных для модели, 154
  отладка, 400
  память и точность, 219
  подготовка проектов
    IDE, 395
    настройка на работу с GPU, 397
    создание нового проекта, 393
    файлы Maven, 396
  поддержка Hadoop, 303
  получение статистики сети, 246
  представление изображений, 286
  работа на уровне исходного кода, 391
  рекомендации по работе со Spark, 322
  функции активации в слоях, 127
  эталонные тесты производительности, 153
DeepMind, 359
DeepWalk, 300
DistBelief, 235
Doc2Vec, 201
Downpour, 234

E

ETL (извлечение, преобразование, загрузка), 277, 279, 285, 324, 382, 387

F

FaceNet, 208
fit() метод, 155
F-мера, 51, 243

G
GloVe, 200
Google File System, 232
GoogLeNet, 130
Gov2Vec, 208
GPU (графические процессоры)
  Mesos, 309
  мониторинг, 401
  распараллеливание вычислений, 236

H
Hadoop Distributed File System (HDFS), 151, 302, 323
Hadoop YARN (Yet Another Resource Negotiator), 302

I
ImageRecordReader класс, 288
INDArray, 373, 379
IntelliJ, 395
Iris, набор данных, 29
Item2Vec, 208

J
Java Development Kit (JDK), 394, 401
JVM настройка, 311

K
Kerberos, 304, 323
Kryo зависимость, 317, 318
K средних, метод, 40

L
L1 и L2, штраф по норме, 101, 213, 239
L-BFGS алгоритм оптимизации, 98, 229
LeNet, СНС, 130, 162, 335
LSTM-сети (сети с долгой краткосрочной памятью)
  архитектура, 137
  блок LSTM, 138
  важнейшие элементы, 136
  вычислительная сложность обучения, 137
  методы настройки, 267
  моделирование последовательных данных, содержащих показания датчика, 177
  на платформе Spark, 328
  новые типы нейронов, 88
  нормировка, 284
  обучение, 141
  ортогональная инициализация весов, 221
  порождение текста в стиле Шекспира, 169
  примеры приложений, 137
  свойства, 137
  слои, 141
  сравнение с GRU, 141
LU-разложение, 31

M
MapReduce, 232, 301, 303, 308
Mark I Perceptron, 57
max-пулинг, 129
MCXENT функция потерь, 214
Mesos, 306, 309
MLLibUtil класс, 379
MNIST набор данных, 85
  моделирование рукописных цифр, 162
  модель в кластере Spark, 332
  обнаружение аномалий, 183
  реконструкция в ОМБ, 106
  реконструкция цифр, 189

N
ND4J библиотека
  дизайн и основы использования, 372
  и Maven, 396
  интероперабельность, 379
  использование совместно с DL4J, 380
  класс DataSet, 377
  основные характеристики, 372
  отладка, 318
  память и точность, 219
  руководство пользователя, 372
  создание входных векторов, 378
  среды выполнения, 152
  ускорение обучения, 152
  эталонные тесты, 153
ND4S, 372
NDArray структура данных, 154, 373
NeuralNetConfiguration класс, 155
Node2Vec, 208, 300
NodeManager, 307
nVIDIA, 397

NVIDIA SMI, интерфейс управления системой, 399
N-граммы, 299

P
ParallelWrapper класс, 398
PhysioNet Challenge, 51, 242

Q
Q-обучение
  Беллмана уравнение, 356
  воспроизведение опыта, 359
  выборка начальных состояний, 357
  двойное, 361
  исследование и использование, 355
  масштабирование вознаграждений, 362
  моделирование Q(s,a), 359
  назначение, 350
  обработка истории, 361
  ограничение, 362
  перебор политик, 352
  предварительная обработка изображений, 360
  приоритетное воспроизведение, 362
  реализация, 358

R
RDD (надежные распределенные наборы данных), 302
RecordReader класс, 154
ReLU, 74, 127, 213, 220, 222
ReLU с утечкой, 75, 220, 222
ResNet, 130
RL4J (Reinforcement Learning for the JVM), 365
RMSProp, 100, 176, 226, 227

S
SerDe (сериализация и десериализация), 319
Spark и Hadoop
  LSTM-сети, 328
  безопасность в Hadoop и Kerberos, 304
  запуск Spark из командной строки, 303
  конфигурирование и настройка Spark, 305
  минимальный пример обучения, 320
  многослойный перцептрон, 323
  моделирование набора MNIST, 332
  обучение на одной машине, 320
  основные компоненты Spark, 302
  отладка, 318
  параллельное выполнение DL4J, 319
  подготовка проекта Maven, 312
  рекомендации, 322
Symbolic Aggregated approXimation (SAX), 291

T
TD-gammon алгоритм, 349
TF-IDF, 296

V
VGGNet, 130

W
webapp-rl4j контрольная панель, 362
Word2Vec
  альтернативы, 200
  векторная арифметика и погружение слов, 198
  другие применения, 200
  моделирование контекста, 197
  модель и алгоритм, 196
  обучение сходной семантике и семантическим связям, 197
  пример, 198
  сравнение с векторной моделью, 299

Y
YARN, 307

Z
ZF Net, 130

А
Автокодировщики, 108, 111, 183
Автоматическое выделение признаков, 24, 89, 112, 274
Анализ тональности высказываний, 118
Антисверточная сеть, 115
Апостериорная вероятность, 34
Апостериорное масштабирование, 244
Априорные функции, 239

Асинхронное N-шаговое Q-обучение, 365
Асинхронное обучение с подкреплением, 365

Б

Байеса теорема, 34
Безгессианная оптимизация, 98, 229
Беллмана уравнение, 356
Бернулли распределение, 284
Бинаризация, 284
Бинаризация признаков, 284
Бинарная классификация
 выходной слой, 95
 кусочно-линейная функция потерь, 224
 одним и двумя выходами, 214
 примеры, 40
Блок линейной ректификации, 74, 213
Блок пороговой логики (БПЛ), 57
Большие данные, 233
Бройдена-Флетчера-Гольдфарба-Шанно (BFGS) алгоритм, 98, 229
Бутстрэппинг, 37

В

Вариационные автокодировщики (VAE), 109, 117
Векторизация
 N-граммы, 299
 атрибуты табличных данных, 277
 в ND4J, 378
 графовых структур, 300
 изображений, 286
 конструирование признаков, 279
 методы нормировки, 281
 обработка отсутствия значений, 280
 последовательных данных, 289
 проблемы, 274
 средствами DataVec, 152, 285, 382
 текста, 294
 ядерное хэширование, 299
Векторная модель (VSM), 295
Вектор частот термов, 296
Векторы абзацев, 201
Вентиль забывания, 140
Вентильные рекуррентные блоки (GRU), 88, 141, 221
Верность, 50
Вероятностная выборка, 244
Вероятность
 апостериорная, 34
 и шанс, 33
 распределения, 35
 совместная, 47
 условная, 34, 47
 частотный и байесовский подходы, 33
Веса связей, 62, 220
Взрывные градиенты, 267
Внешнее произведение, 28
Вращательная инвариантность, 127
Временные ряды, 132, 134, 150, 177
Входные слои
 в сверточных сетях, 121
 в сетях прямого распространения, 63
 отсутствие функций активации, 94
Выборочное среднее, 281
Выделение признаков, 24, 89, 112
Выпуклая оптимизация, 42
Выходные слои, 95
 в многослойных сетях прямого распространения, 64
 для бинарной классификации, 95, 214
 для многоклассовой классификации, 95, 214
 для регрессии, 95, 213
 общее назначение, 114
 соотнесение с назначением модели, 213
Вычислительная эффективность, 238

Г

Гаусса метод исключения, 31
Гессиан, 46, 96
Гибридные архитектуры, 151, 269
Гибридные сети, 143
Гиперпараметры
 в сверточных слоях, 127
 импульс, 82, 225
 категории, 99
 метод Нестерова, 227
 мини-пакеты, 102
 назначение, 42
 нормировка, 281
 определение, 98

размер слоя, 99
разреженность, 82, 228, 273
регуляризация, 82, 101
рецептивное поле, 126
скорость обучения, 81, 99, 225
чрезмерно большое число, 99
Гиперплоскость, 28
Глобальные векторы, 200
Глобальный минимум, 45
Глубокая сверточная порождающая состязательная сеть (ГСПСС), 116
Глубокие Q-сети (DQN), 347
Глубокие сети
архитектурные принципы
алгоритмы оптимизации, 96
гиперпараметры, 98
параметры, 93
слои, 93
функции активации, 94
функции потерь, 95
выбор архитектуры, 148, 211
основные архитектуры
выбор, 148, 211
обзор, 111
рекуррентные нейронные сети, 131
рекурсивные нейронные сети, 144
сверточные нейронные сети, 117
сети, предобученные без учителя, 111
строительные блоки, 102
автокодировщики, 108
вариационные автокодировщики, 109
послойное предобучение без учителя, 103
Глубокие сети доверия (ГСД)
автоматическое обучение, 112
архитектура, 111
выделение признаков, 112
и сверточные нейронные сети, 114
классификация, 89
методы настройки
два этапа обучения, 272
применение импульса, 272
применение регуляризации, 273
число скрытых блоков, 273
роль в возрождении глубокого обучения, 114
слои, 88
точная настройка, 113
Глубокое обучение
достижения, 87
и искусственный интеллект, 338
и машинное обучение, 22, 24, 146
когда использовать, 147
определение, 24, 83
Глубокое обучение с подкреплением, 85, 352
Градиентный спуск, 43, 97
Граф вычислений, 215
Графы, моделирование, 300

Д

Двунаправленные рекуррентные нейронные сети (BRNN), 137
Детекторы признаков, 122
Децентрализованные системы, 24
Дисбаланс классов, 51, 242
Дискриминантные модели, 46
Добыча данных, 21
Дополнение нулями, 128, 257, 267

Е

Единичная ковариационная матрица, 283

З

Закон больших чисел, 352

И

Избыточная выборка, 244
Изображения
векторизация, 286
моделирование, 148, 162
обнаружение аномалий, 183
распознавание лиц, 208
Импульс, 82, 225
Инцепционизм, 90
Искусственный интеллект, 337
зимы, 343
и глубокое обучение, 338
изучение, 339
и машинное обучение, 340
определение, 338
современные определения, 339
Исполнители (Spark), 302, 307

Исчезающие градиенты, 136, 267
Итеративные методы, 31

К

Карта активации, 123
Карта признаков, 123
Классификация
  выходной слой модели, 213
  многоклассовая, 214
  многометочная, 214
  определение, 40
  с одной меткой, 214
  функции потерь, 78
Кластеризация, 40
Коллаборативная фильтрация, 40
Компьютерное зрение, 86, 143
Коннекционистские модели, 135
Конструирование признаков
  копирование признаков, 280
  определение, 279
  переход к автоматизированному обучению, 274
  приемы, 280
Косинусный коэффициент, 28
Кусочно-линейная функция потерь, 78, 224

Л

Латентное семантическое индексирование, 196
Лемматизация, 295
Линейная алгебра
  векторы, 26
  гиперплоскости, 28
  и ND4J, 152
  математические операции, 28
  матрицы, 27
  решение систем уравнений, 30
  скаляры, 26
  тензоры, 27
Линейная регрессия, 38
Логистическая регрессия, 47
Локальный минимум, 45, 54, 217

М

Манера художника, моделирование, 91
Марковские модели, 135
Марковский процесс принятия решений (MDP), 347
Маскирование, 134, 267, 294
Масштабирование, 281, 283
Матрица неточностей, 49
Матрицы, 27
  обращение, 31
  разложение, 31
Матрицы смежности, 300
Машинное обучение
  гиперпараметры, 81
  и глубокое обучение, 24
  и добыча данных, 21
  и искусственный интеллект, 340
  история, 20
  логистическая регрессия, 47
  математические основания
    линейная алгебра, 26
    статистика, 32
  методы оптимизации, 38
    выпуклая оптимизация, 42
    градиентный спуск, 43
    квазиньютоновские методы оптимизации, 46
    классификация, 40
    кластеризация, 40
    недообучение и переобучение, 41, 82
    параметрическая оптимизация, 41
    порождающие и дискриминантные модели, 46
    регрессия, 38
    стохастический градиентный спуск, 45, 71
  определение, 21, 96
  оценивание модели, 49
Метод главных компонент (PCA), 283
Метод обратного распространения, 54
  доля ответственности за ошибку, 70
  интуитивное описание, 65
  и обучение на мини-пакетах, 102, 121, 133
  истоки, 65
  математическое описание, 368
  мягкий, 113
  псевдокод, 67
  функции потерь, 68

Метод сопряженных
градиентов, 31, 98, 229
Метод чередующихся наименьших
квадратов, 31
Методы настройки
выбор функции активации, 221
для ГСД, 272
для ограниченных машин
Больцмана, 269
для РНС, 263
для СНС, 253
количество слоев, количество
параметров и объем памяти, 216
методы оптимизации, 229
основные концепции, 209
переобучение, 245
периоды и размер мини-пакета, 237
получение статистики сети, 246
разреженность, 228
распараллеливание, 231
регуляризация, 239
скорость обучения, 225
стратегии инициализации весов, 220
функции потерь, 223
Мешок слов, 295
Минимаксное масштабирование, 283
Мини-пакеты, обучение
на, 46, 102, 121, 133, 237, 287
Многомерное распределение
Бернулли, 284
Многослойная сеть прямого
распространения
архитектура, 63
методы настройки, 216
моделирование CSV-данных, 156
на платформе Spark, 323
обратное распространение, 369
определение, 60
сравнение с глубокими сетями, 83
топология, 54
эволюция искусственного нейрона
веса связей, 62, 220
входные данные, 61
диаграмма, 60
смещения, 62
функции активации, 62
Монте-Карло метод, 352

Н

Недообучение, 41
Неперемешанные данные, 249
Нестерова метод импульса, 100, 227
Нормальные уравнения, 31
Нормировка, 28, 281
Нэша равновесие, 350

О

Обнаружение аномалий
с помощью автокодировщиков, 183
с помощью вариационных
автокодировщиков, 189
Обработка естественного языка (ОЕЯ)
анализ тональности
высказываний, 118
классификация документов, 204
модель мешка слов, 295
обучение погружениям слов, 196
размеры фильтров, 123
распределенные представления
предложений, 201
Обратное распространение
во времени (BPTT), 141, 266
Обратное распространение сквозь
структуру (BPTS), 144
Обучение по плану, 242
Обучение с подкреплением, 84, 347
безмодельное, 349
визуальный мониторинг, 362
марковский процесс принятия
решений (MDP), 347
наблюдаемое состояние, 349
однопользовательские и
состязательные игры, 349
офлайновое и онлайновое, 357
терминология, 348
Ограниченные машины Больцмана
(ОМБ), 88, 103, 111, 209
методы настройки, 269
Онлайновые алгоритмы обучения, 232
Оптимизация
безгессианная, 98, 229
выпуклая, 42
градиентный спуск, 43, 97
другие алгоритмы, 97
квазиньютоновские методы, 46, 98

метод сопряженных градиентов, 98, 229
методы второго порядка, 96, 229
методы первого порядка, 96
определение, 38
параметрическая, 41
стохастический градиентный спуск, 45, 71, 97
эффективность, 238
якобиан, 96
Оптическое распознавание символов, 85
Ортогональная инициализация весов, 221
Отношение обновлений к параметрам, 226
Отрицательное логарифмическое правдоподобие, 80, 160, 225
Отсутствие значений, обработка, 280
Оценка максимального правдоподобия, 43
Очистка данных, 279
Ошибка TD, 356, 362

П

Пакетная нормировка, 128
Параллелизм на уровне данных, 232
Параллелизм на уровне задач, 232, 312
Параметры
в глубоких сетях, 93
векторы, 21
и структуры NDArray, 93
оптимизация, 41
разделение, 126
усреднение, 232, 235
Перебор политик, 352, 355
Перекрестная проверка, 37
Перекрестная энтропия, 81, 96, 225
Перенос обучения, 262
Переобучение, 41, 82, 99, 245
Периоды, 237
Перцептрон
алгоритм обучения, 59
история, 57
многослойный, 60, 148, 156, 369
на платформе Spark, 323
ограничения однослойного варианта, 59
однослойный, 58
определение, 57
предшественник, 57
связь перцептрона с биологическим нейроном, 58
Площадь под кривой (AUC), 243
Погружения слов, 196
Полносвязные слои, 129
Полнота, 51
Полнота по Тьюрингу, 131
Положительная прогностическая значимость, 51
Понижение размерности, 284
Порождающие модели
инцепционизм, 90
моделирование манеры конкретного художника, 91
недостатки, 117
построение, 116
сравнение с дискриминантными, 46
Порождающие состязательные сети (ПСС)
глубокие сверточные (ГСПСС), 116
недостатки, 117
обучение порождающих моделей, 114
порождающая сеть, 115
сравнение ПСС с вариационными автокодировщиками, 117
условные, 116
Поэлементное произведение, 28
Правдоподобие, 38
Предобучение, 103, 105
Произведение Адамара, 28
Прореживание (DropOut), 101, 240, 273
Прореживание связей (DropConnect), 101, 241
Пулинговые слои, 128, 261

Р

Разбиение на лексемы, 295
Разреженная инициализация, 221
Разреженность, 82, 228, 273
Распараллеливание, 231, 242, 312
Распределения вероятности, 35
непрерывные и дискретные, 35
нормальное, 35

ранжирование по частоте, 36
с длинным хвостом, 36
центральная предельная теорема, 36
Распределенные файловые системы, 323
Регрессия
выходной слой, 95, 213
линейная, 38
логистическая, 47
определение, 38
функции потерь, 77
Регуляризация, 82, 239, 245, 251, 266, 271, 273
по максимальной норме, 240
Рекомендование, 40
Рекуррентная модель зрительного внимания, 143
Рекуррентные нейронные сети (РНС)
LSTM-сети, 136
архитектура сети, 136
временные ряды, 150
и марковские модели, 135
история, 86
методы настройки
входные данные и входной слой, 264
выходные слои и RnnOutputLayer, 264
гибридные архитектуры, 269
дополнение и маскирование, 267
обучение сети, 265
отладка, 267
проблемы, 263
моделирование времени, 131
моделирование последовательных данных, 169
нормировка, 284
обратное распространение во времени, 141
приложения, 143
проблема исчезающего градиента, 136
трехмерный вход, 133
Рекурсивные нейронные сети
архитектура сети, 144
применения, 145
разновидности, 145
Рекурсивные нейронные тензорные сети (RNTN), 145
Рецептивное поле, 125
Решающая граница, 42

С

Сборка мусора, 311
Свертка, 122
Сверточные нейронные сети (СНС)
биологические корни, 118
в кластере Spark, 332
конфигурирование задачи и загрузка данных, 334
обучение сети LeNet, 335
входной слой, 121
другие применения, 129
и глубокие сети доверия, 114
известные СНС, 130
интуитивное описание, 119
методы настройки
конфигурирование пулинговых слоев, 261
конфигурирование сверточных слоев, 257
общие архитектурные паттерны, 254
перенос обучения, 262
моделирование рукописных цифр, 162
нормировка, 284
общий взгляд на архитектуру, 120
полносвязные слои, 129
представление изображений, 56, 286
пулинговые слои, 128
сверточные слои, 122
цель, 117
эволюция, 85
эффективность, 117
Сверточные слои
ReLU-слои, 127
визуализация обученных фильтров, 126
гиперпараметры, 127
карта активации, 123
методы настройки, 257
назначение, 122
операция свертки, 122

пакетная нормировка, 128
разделение параметров, 126
рецептивное поле, 118, 125
фильтры, 123, 258
Сериализация Java, 317
Сети, предобученные без учителя (UPN)
глубокие сети доверия, 111
достоинства и недостатки, 103
обсуждение архитектуры, 111
порождающие состязательные сети, 114
Скалярное произведение, 28
Скорость обучения
и корректировка параметров, 81
методы настройки, 225
определение, 71, 99
Скрытые марковские цепи, 349
Скрытые слои
задание количества, 216
Слои
LSTM, 141
антисверточные, 115
в глубококих сетях, 94
в сетях прямого распространения, 53
в СНС, 120
Смещение выборки, 37
Смещения, 62
Сопоставительное расхождение, 105
Состязательное обучение, 242
Специфичность, 50
Среднее значение, 281
Среднеквадратическая логарифмическая ошибка (СКЛО), 78
Среднеквадратическая ошибка (СКО), 77, 213
Средняя абсолютная ошибка в процентах (САОП), 78
Средняя абсолютная ошибка (САО), 78
Стандартизация, 281
Стационарная точка, 44
Стемминг, 295
Стохастический градиентный спуск (СГС)
в Q-обучении, 352
достоинства, 97
и стандартизация, 282
мини-пакетный, 71
определение, 45
распараллеливание, 234
рекомендации, 230
скорость обучения, 225
Стохастический пулинг, 242
Сходимость, 42

Т

Табличные данные, моделирование, 148, 277, 285, 384
Тензорное произведение, 28
Тензоры, 27
Теорема об отсутствии бесплатных завтраков, 146
Теория информации, 81
Типы данных
атрибуты табличных данных
интервальные, 278
номинальные, 278
относительные, 278
порядковые, 278
видео, 286
временные ряды, 150, 289
графовые структуры, 300
изображения, 149, 162, 208, 286, 385
неперемешанные, 249
табличные, 148, 285, 384
текст, 149, 294
Точка минимума, 44
Точность, 51

У

Умирающего ReLU проблема, 75, 213, 222
Унитарное представление вектора, 161, 278
Усеченный BPTT, 141, 266
Условная вероятность, 34
Условные рефлексы по Павлову, 347

Ф

Функции активации
hardtanh, 73, 222
softmax, 73, 215
softplus, 75
tanh, 73, 221
в глубоких сетях, 94

в скрытых слоях, 94
выходной слой для бинарной классификации, 95
выходной слой для многоклассовой классификации, 95
выходной слой для регрессии, 95, 213
для архитектуры общего вида, 94
линейная, 71
линейная ректификация, 74, 213
методы настройки, 221
определение, 62
сводная таблица, 223
сигмоида, 72, 94, 222
эволюция на практике, 95
Функции потерь
взвешенные, 244
для классификации, 78
для регрессии, 77
для реконструкции, 80, 96
логистическая, 79, 225
методы настройки, 223
назначение, 42, 76, 95
на псевдокоде, 68
обозначения, 76
сводная таблица, 224
соответствие функции активации, 210

Ц

Центральная предельная теорема, 36
Центрирование, 281
Центр сознания, 56

Ч

Частота истинно положительных результатов, 51
Чувствительность, 50

Ш

Шаг, 128, 257

Э

Энтропия, 80

Я

Ядерное хэширование, 299
Ядра, 123
Якобиан, 46, 96
Ячейки памяти, 138

Об авторах

Джош Паттерсон руководит отделом поддержки проектов в компании Skymind. Раньше Джон оказывал консультационные услуги в области больших данных, машинного и глубокого обучения, а до того работал главным архитектором решений в компании Cloudera и инженером по машинному обучению и распределенным системам в Управлении ресурсами бассейна реки Теннесси, где внедрял Hadoop в интеллектуальную энергосистему в рамках проекта openPDC. Джош получил ученую степень магистра информатики в университете Теннесси в Чаттануге, где опубликовал результаты исследовательской работы по ячеистым сетям (tinyOS) и алгоритмам оптимизации по типу поведения социальных насекомых. Джош занимается разработкой программного обеспечения более 17 лет и проявляет большую активность в области проектов с открытым исходным кодом. Он внес вклад в проекты DL4J, Apache Mahout, Metronome, IterativeReduce, openPDC и JMotif.

Адам Гибсон – специалист по глубокому обучению из Сан-Франциско. Работает с компаниями из списка *Fortune 500*, хэджевыми фондами, компаниями в области связей с общественностью и акселераторами стартапов, помогая в разработке проектов машинного обучения. У Адама большой опыт консультирования компаний в области обработки и интерпретации больших данных. Он увлекся компьютерами в 13 лет и активно участвует в жизни сообщества приверженцев ПО с открытым исходным кодом через сайт http://deeplearning4j.org.

Об иллюстрации на обложке

На обложке изображена ремень-рыба (*Regalecus glesne*), или сельдяной король, – крупная лучепёрая рыба, обитающая в умеренных и тропических районах океана. У нее тело лентовидной формы с длинным спинным плавником, который начинается на голове и продолжается до заднего конца тела. Ремень-рыба может достигать в длину 11 метров, что делает ее самой большой костной рыбой в мире.

Ремень-рыбы ведут одиночный образ жизни и редко попадаются на глаза человеку. Большую часть времени они проводят в мезопелагической зоне (на глубине от 200 до 1000 метров) и на поверхность поднимаются, только если больны или ранены. Ремень-рыбы – плотоядные животные, питающиеся в основном планктоном, а также мелкими рыбками, медузами и кальмарами.

Мясо ремень-рыбы имеет студенистую консистенцию, поэтому для коммерческого рыболовства она не представляет интереса. Человек обычно сталкивается только с мертвыми или умирающими особями, выброшенными на берег. Из-за размера и формы ремень-рыбы считается, что она могла породить легенды о морском змее. Хотя общая численность популяции ремень-рыбы неизвестна, ни о каких угрозах ее существованию мы не знаем.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой вымирания; все они важны для нашего мира. Если хотите узнать, чем вы можете помочь, зайдите на сайт animals.oreilly.com.

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: books@alians-kniga.ru.

Джош Паттерсон, Адам Гибсон

Глубокое обучение
с точки зрения практика

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 39,1875. Тираж 200 экз.

Веб-сайт издательства: **www.дмк.рф**