

Дэниел П. Фридман, Анураг Мендхекар

The Little Learner:

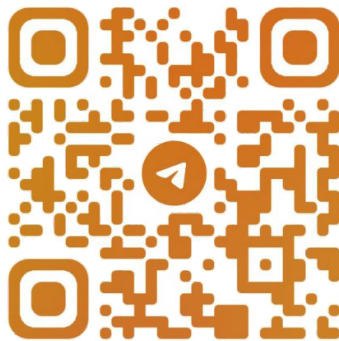
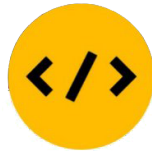
Чудесное машинное обучение



ФМК
ИЗДАТЕЛЬСТВО

Дэниел П. Фридман
Анураг Мендхекар

The Little Learner: чудесное машинное обучение



@CODELIBRARY_IT

The Little Learner

A Straight Line to Deep Learning

Daniel P. Friedman

Anurag Mendhekar

Drawings by Qingqing Su

Forewords by Guy L. Steele Jr. and Peter Norvig

The MIT Press
Cambridge, Massachusetts
London, England

The Little Learner:

чудесное машинное обучение

Дэниел П. Фридман
Анураг Мендхекар

Рисунки Цинцин Сю



Москва, 2024

УДК 004.04
ББК 32.372
Ф82

Дэниел П. Фридман, Анураг Мендхекар
Ф82 The Little Learner: чудесное машинное обучение. / пер. с англ.
В. С. Яценкова. – М.: ДМК Пресс, 2024. – 428 с.: ил.

ISBN 978-5-93700-233-4

В книге охвачены все концепции, необходимые для интуитивного понимания работы глубоких нейронных сетей, включая тензоры, расширенные операторы, алгоритмы градиентного спуска, искусственные нейроны, полносвязные, сверточные сети и остаточные сети, а также автоматическое дифференцирование. Читатель начнет с азов глубокого обучения и познакомится с полной реализацией полезного приложения: распознавателя зашумленных сигналов азбуки Морзе.

Разговорный стиль, постепенное движение от простого к сложному, забавные иллюстрации, формат вопросов и ответов делают обучение доступным и увлекательным.

Для читателей, знающих математику на уровне средней школы и имеющих некоторый опыт программирования.

@MIT Press 2023. Права на русское издание получены через агентство Александра Корженевского.

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.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-262-54637-9
ISBN 978-5-93700-233-4

© Daniel P. Friedman and Anurag Mendhekar, 2023
© Оформление, перевод, издание, ДМК Пресс, 2024

*Дорогая Мэри, как говорится в песне,
«...посвящаю себя тому, кого я люблю».
В какое чудесное приключение ты превратила мою жизнь!
С глубочайшей любовью и восхищением.*

– Дэнни

*Сияющим звездам моей жизни: Аруне, Ришме и Арье Нине –
и моим постоянным четвероногим спутникам Чикки и Хиру.
Без вас вокруг была бы только тьма.*

– Анураг

Оглавление

Предисловие Гая Стила-младшего	XI
Предисловие Питера Норвига	XIV
Благодарности	XVIII
Введение	XX
Как читать эту книгу.....	XXI
Как написаны наши программы	XXI
Как запустить код	XXII
Как есть десерты	XXII
Перевод сокращенной записи в язык Scheme	XXIII
0. Ну что, заинтригованы?.....	1
1. Строки сегодня спят.....	17
Правило параметров (начальный вариант).....	25
Сундук с полезными вещами	27
2. Чем крепче ум, тем толще книги.....	29
Правило ранга	34
Правило о членах и элементах	37
Правило об однородной форме	38
Закон ранга и формы.....	39
Закон простой передачи аккумулятора	40
Тензорные побрякушки	41
Интерлюдия I. Расширяем функции, углубляем тензоры	43
Закон суммы	50
Растягивающиеся побрякушки.....	51
3. Куда нас линия ведет?	53
Закон обновления (начальная версия).....	67
Потерянные вещи.....	68

4. Скользим, скользим неудержимо!	69
Закон обновления (итоговая версия).....	82
Градиентные побрякушки	88
Интерлюдия II. Чем больше игрушек, тем больше шума.....	89
Правило гиперпараметров.....	91
Гиперактивные побрякушки.....	93
5. Точно в цель!.....	95
Правило наборов данных.....	104
Правило параметров (итоговая версия)	106
Правило θ	106
Побрякушки для стрельбы по мишеням.....	107
Интерлюдия III. Облик грядущего.....	109
6. Одно яблоко в день.....	113
Правило пакетов.....	118
Закон о размере пакета.....	126
Случайные побрякушки	127
7. Суета вокруг хвоста.....	129
Закон об обновлениях	132
Закон об <i>ate</i> -функциях.....	141
Поехавшие побрякушки	142
8. Чем ближе цель, тем меньше шаг.....	143
Скоростные побрякушки	152
Интерлюдия IV. Оператор Smooth.....	153
Мягкие игрушки	159
9. Будь непреклонен!	161
Закон градиентного спуска	173
Быстрые побрякушки.....	173
Интерлюдия V. Экстензио магнифико!	175
Больше разных побрякушек	192
10. Причудливый танец нейронов	193
Правило искусственных нейронов.....	201
Нейронные побрякушки.....	211

11. Прекрасные формы Relu.....	213
Закон о плотных слоях (начальная версия).....	218
Закон о плотных слоях (финальная версия)	219
Форменные побрякушки.....	234
12. Построим блок, сыграем рок.....	235
Правило о блоках.....	246
Блочные побрякушки.....	247
13. Взгляните на ирис.....	249
Правило инициализации слоя (начальная версия).....	261
Классные побрякушки.....	268
Интерлюдия VI. Как обучаются модели.....	269
Обучающие побрякушки.....	280
Интерлюдия VII. Вы готовы принять сигнал?.....	281
Закон упакованных сигналов.....	292
Побрякушки на молнии	294
14. Все очень сильно перепутано... ..	295
Закон о корреляции (версия с одним фильтром)	304
Правило фильтров	305
Закон о корреляции (версия с банком фильтров)	313
Скользющие побрякушки.....	314
15.....Но при этом взаимосвязано!	315
Закон о пропусковых соединениях.....	334
Корреляционные побрякушки	336
Эпилог. Все только начинается	337
1. Математическая база	338
2. Распределения генерации данных.....	338
3. Задачи.....	339
4. Другие функции потерь.....	340
5. Другие решатели.....	341
6. Многомерные сигналы	343
7. Системы для обработки естественного языка.....	343
8. Генеративные сети.....	344

9. Прикладные соображения	345
10. Вперед, отважные ученики!	347
Приложение А. Призрак в машине.....	348
Приложение В. Я так хочу, чтобы гонка не кончалась.....	374
Литература	400
Предметный указатель	401

Мы случайно наткнулись друг на друга в пятницу, 13 апреля 2018 года, на многолюдном официальном открытии Школы информатики, вычислительной техники и инженерии Luddy, и решили написать книгу по машинному обучению после очень глубокого и обстоятельного разговора, последовавшего сразу за окончанием мероприятия:

Анураг: Я хотел бы написать с тобой книгу из серии *The Little Book*.

Дэн: Так давай напишем!

⋮

Несколько секунд спустя

⋮

Дэн: На какую тему?

Анураг: Машинное обучение.

Дэн: Ну ладно, интересная тема!

А потом мы долго болтали о всякой всячине...

Предисловие

Гая Стила-младшего

Эта книга – именно то, что нужно.

Дэн Фридман уже более четырех десятилетий пишет книги в своем уникальном стиле *The Little Book*¹ в компании талантливых и опытных соавторов. Каждая из них – истинная жемчужина, объясняющая глубокие и важные идеи в области компьютерных наук небольшими порциями. Дэн и его соавторы возвели формат вопрос–ответ в ранг искусства, до уровня беседы, кажущейся почти беззаботной. Кроме того, в этой книге представлены два новшества – каркасный код функций и логические столбцы, – которые еще больше упрощают представление фрагментов программного кода и их поведения.

Что же касается фундаментальных идей машинного обучения, то в этой книге они представлены именно в том виде и именно в том порядке, который вам необходим.

Поэтому читайте книгу последовательно (не забегая вперед!). Сами авторы отмечают:

Наши «книжки-малышки» предназначены для аккуратной упаковки небольших идей в маленькие коробочки.

Какие идеи изложены в этой книге? Конечно же, математика и вычислительные методы машинного обучения: вы узнаете о последовательной аппроксимации, стохастическом градиентном спуске, нейронных сетях и автоматическом методе обратного распространения ошибки. Однако как специалисту по языкам программирования мне также интересно, как авторы используют язык программирования для изложения математики. На мой взгляд, ключевая и всеобъемлющая идея книги заключается в том, как авторы используют функции высшего порядка, то есть функции, которые принимают другие функции в качестве аргументов и/или возвращают другие

¹ С разговорного английского *The Little Book* переводится приблизительно как «книжка-малышка», а еще в Америке так называют адаптированное карманное издание Евангелия. И хотя книги этой серии вполне обычные по количеству страниц, информация в них разбита на маленькие пронумерованные «фреймы», а изложение следует шаг за шагом и сопровождается забавными рисунками и отступлениями, не давая читателю заскучать. – Прим. перев.

функции в качестве результатов, для объяснения структур данных и вычислений в машинном обучении.

Фундаментальной структурой данных является тензор, который на первый взгляд выглядит как обычный массив или матрица; но авторы поясняют, что тензорные операции обладают дополнительным свойством автоматического использования функции отображения более высокого порядка, когда это необходимо, и это – почти волшебным образом, как мне кажется, – позволяет использовать функцию, явно написанную для скаляров (одиночных чисел) или тензора определенного ранга, которую без дополнительных усилий можно применять ко всем типам и размерам массивов, векторов и матриц.

Другим применением функций высшего порядка является каррирование¹, которое позволяет вам передать функции некоторые из ее аргументов сейчас, а другие позже – когда вы передаете ей некоторые из ее аргументов, она возвращает другую функцию, которую можно применить к другим аргументам позже, чтобы получить окончательный ответ. Авторы в этой книге используют каррирование в качестве понятного – и, как по мне, приятно неожиданно – способа объяснить разницу между аргументами и параметрами в машинном обучении и почему они должны быть представлены в строго определенном порядке, некоторые – сейчас, некоторые – позже. (Третья разновидность аргумента, гиперпараметры, также объясняется с помощью еще одного механизма языка программирования. Если вы знакомы с модным понятием «динамическая область видимости», то вам будет интересно; если нет, не беспокойтесь – гиперпараметры и их поведение доходчиво объясняются на примере.)

Третье назначение функций высшего порядка в этой книге состоит в том, чтобы разбить большие нейронные сети на мелкие составляющие блоки и объяснить поведение и обучение этих сетей.

Четвертое назначение функций высшего порядка – обеспечение абстракции данных. Поначалу параметры всегда представляют собой простые числа, но код их обработки настолько искусно определен с помощью всего нескольких интерфейсных функций – именно тех, которые нужны, – что код более высокого порядка не нужно менять при расширении представления параметров. (Ключевой идеей является проекция: у нас есть две функции – одна проецирует данные в альтернативное представление, на котором легче вычислять, а

¹ Преобразование функции от нескольких аргументов в набор вложенных функций, каждая из которых является функцией от одного аргумента. – *Прим. перев.*

другая извлекает вычисленный результат обратно в исходное представление. Тогда функция, которая принимает две такие функции в качестве аргументов, может использоваться в очень общем виде.) Аналогично, скаляры – это простые числа на протяжении большей части книги, но когда возникает необходимость обобщить их на двойственные числа в приложении А, функции более высокого порядка упрощают задачу.

Поэтому если вас интересуют общие идеи языков программирования, помните об этих применениях функций высшего порядка.

Возможно, вам понравится замечать их по ходу чтения.

Но если вам не важны функции высшего порядка, то пожалуйста – игнорируйте все, что я только что сказал.

Погрузите себя в мир машинного обучения.

Вообще-то, эта книга не нуждается в представлении; она идеально подходит для своей цели.

Я прочитал всю серию *The Little Book* и с каждой новой книгой говорил себе: «Эта лучшая из всех!» Но Дэн и Анураг вновь превзошли себя – эта книга лучше прежних. На книжной полке она должна стоять особняком; вам не нужно читать предыдущие «книжки-малышки», чтобы понять эту, и вам не нужно заранее разбираться в Scheme или любом другом языке программирования. Около дюжины нужных вам идей по языку программирования объясняются по пути, каждая именно тогда, когда она вам нужна, и с использованием многочисленных примеров. Не спешите и наслаждайтесь повествованием.

*Гай Стил-мл.
Лексингтон, Массачусетс,
август 2022 г.*

Предисловие

Питера Норвига

Привет, я Питер Норvig, опытный исследователь и практик в области машинного обучения.

Мне повезло стать одним из первых читателей этой книги, и меня попросили оставить несколько комментариев к ней. Я собираюсь сделать это в форме диалога с моим уважаемым коллегой, типичным читателем.

Добро пожаловать в это предисловие, уважаемый читатель, как дела?

Хорошо, Типи.

Я могу сказать, что мне очень понравилась книга и то, как авторы постепенно и тщательно раскрывают ключевые понятия.

А вы что думаете о книге?

Вы верно подметили.

Эта книга более интерактивна, чем большинство других книг, и требует постоянного участия читателя, задавая ему вопросы.

Я не могу сказать наперед, подходит ли вам книга, но я могу сказать, что лучший способ получить опыт в любой области – это осмысленная практика, а не просто пассивное слушание или чтение.

¹ Спасибо, я счастлив быть здесь, даже если я всего лишь плод воображения.

И кстати, зовите меня просто Типи.

² Честно говоря, я еще не вчитывался. Пока я только привыкаю к необычному стилю. Книга выглядит увлекательно, но я еще не решил, стоит ли она моего времени и усилий.

³ Несомненно, она требует больше усилий при чтении, чем обычная книга, потому что вам придется старательно проработать все примеры.

Работа с примерами сильно помогает закрепить материал!

Припоминаю, как однажды я обсуждал технику машинного обучения с коллегой, и он сказал, что не понимает, как работает одна особенно сложная часть. Я беззаботно ответил, что это кажется сложным, но на самом деле все просто.

Я вспомнил, что у Эндрю Ына было отличное видео, которое совершенно элегантно объясняло, как это работает. Я попытался повторить то, что узнал из видео, и... не смог этого сделать!

Объяснение Эндрю было настолько понятным, когда я смотрел его, что я не стал утруждать себя его изучением. Мой коллега быстро нашел это видео, и сказал: «Это, должно быть, то самое видео, давайте посмотрим его еще раз», но я отказался. На этот раз я намеревался получить ответ самостоятельно, а затем запомнить его.

И я помню его по сей день.

Эта книга не о языках программирования, и речь идет не о наборе инструментов машинного обучения.

Авторы объясняют фундаментальные концепции машинного обучения и их реализации простейшим образом.

Я уверен в этом.

Это один из самых простых языков программирования. Вам достаточно придерживаться основных математических нотаций, которые превращаются в исполняемый код, поэтому нет никакой двусмысленности в отношении того, как все это работает.

⁴ Хорошо, вы убедили меня в том, что практика приносит пользу.

Но я привык видеть инструменты машинного обучения в основном на языке Python, или, может быть, Java, или R, а Scheme встречается не очень часто.

⁵ Вы полагаете, язык Scheme – удачный выбор для этой цели?

⁶ Но научит ли меня эта книга делать правильные вещи?

Должны ли вы взять код Scheme из этой книги и применить его в своем следующем проекте машинного обучения?

Не обязательно. Но даже если вы используете готовый инструментарий для машинного обучения, такой как Tensorflow или Pytorch, из этой книги вы вынесете понимание того, как работают основы.

Это понимание позволит вам намного проще решить, что делать дальше, когда Tensorflow или Pytorch ведет себя не так, как вы задумали.

Эта книга проведет вас через ключевые идеи машинного обучения от теории к практике.

В ней минимальное количество математических формул и большинство тем излагается с помощью кода и разговорной речи, а не уравнений.

Я из тех людей, которым всегда удобнее читать код, чем формулы, поэтому книга наполнила меня уютным чувством комфорта, а не страхом.

Думаю, темы представлены в правильном темпе. Книга начинается с основ языка Scheme и линейных уравнений прямой. На стр. 61 дано определение функции потерь L2, скорости обучения и градиентного спуска.

Часто темы освещаются сначала упрощенно, а потом детально; например, мы возвращаемся к градиентному спуску на стр. 140 и переходим к алгоритму оптимизации Adam (современная версия градиентного спуска) на стр. 171.

⁷ Полностью согласен. Но я опасаюсь, что в этой книге сплошная теория и математика... Я стремлюсь к глубокому пониманию основ, но хочу, чтобы это было полезно на практике.

⁸ Звучит привлекательно. О каких ключевых темах я узнаю? И сколько времени уходит на то, чтобы добраться до них?

⁹ Вы меня убедили, что недостаточно лишь пролистать книгу, и я намерен ее хорошенько проработать.

Кстати, в книге было что-то, с чем вы не согласны?

Когда мы перейдем к стр. 201, вы увидите, что определение «Искусственный нейрон – это параметризованная линейная функция, совмещенная с нелинейной функцией принятия решения» идеально объединяет знания, полученные на предыдущих страницах.

Вы узнаете о наиболее часто используемых функциях решателей, таких как *relu*, и на стр. 209 найдете простое, но исчерпывающее описание того, как любая функция может быть аппроксимирована инструментами, которые у нас уже есть.

Книга постоянно опирается на ранее заложенный фундамент, чтобы исследовать проблемы, которые возникают на практике, например при обсуждении исчезающих и взрывающихся градиентов на стр. 259.

Я всецело согласен с педагогическим стилем изложения, когда мы вместе с авторами прорабатываем проблему, достигаем осязаемого результата, а потом наслаждаемся перерывом.

Я использовал технику Помодоро, чтобы сосредоточиться на рабочих интервалах; эта книга прекрасно подходит для такого метода организации работы.

Но я боюсь, что если вы будете строго следовать советам книги по употреблению десертов в перерывах, это будет явное излишество.

Побольше функций, поменьше сладкого!

*Питер Норvig
Пало Альто, Калифорния,
август 2022 г.*

¹⁰ Как типичный читатель я потребляю 77 г сахара в день.

Но я прислушаюсь к вашему предупреждению и постараюсь приблизиться к суточному потреблению в 24 г в день.

Благодарности

Гай Стил и Питер Норвиг стали идеальными авторами предисловий. Они точно поняли, что мы пытались сказать, и каждый из них прекрасно уловил дух повествования. Мы сожалеем лишь о том, что наши слова не могут в полной мере передать, насколько мы были взволнованы, когда включали в книгу их изящно сформулированные соображения.

Рисункам Цинцин Сю присущ совершенно особый стиль, вдохновленный иллюстрациями из предыдущих изданий *The Little Book*, но ее творения обладают собственной уникальной индивидуальностью. Мы благодарим ее за мастерство, с которым она создала эти рисунки.

Мы благодарны Сюзанне Мензел и Митчу Ванду, которые долгое время работали над серией *The Little Book*. Они старательно и мягко поясняли нам, что было неправильным, сбивающим с толку или что можно было бы улучшить, и благодаря им мы внесли много существенных улучшений. Мы признательны и благодарны Джули Сассман, которая вмешалась буквально в предпоследнюю минуту, чтобы избавить эту книгу от большего количества недостатков в написании, чем мы ожидали.

Наши первые читатели, Стив Бецольд, Рон Гарсиа, Ник Фаро и Ричард Оттен, вселили в нас уверенность в результатах собственного труда. Мы хотим особенно поблагодарить Джейсона Хеманна, чей отзыв было приятно читать. Спасибо также Паркеру Моресу и Заку Вилкерсону, которые поддерживали нас в использовании необычного стиля нумерации страниц.

Мы хотели бы поблагодарить Ника Дрозда, Ширама Кришнамурти, Вейцзи Ма, Зака Сейлигера, Адитхия Селвапритхвирая и Майкла Ваньера за их терпеливое прочтение и отзывы. Цинъи Цзи помог нам проявить сочувствие к нашим читателям, особенно в первых главах. Джон Росси был ярким энтузиастом обучения, который показал нам, как помочь некоторым читателям, которых часто упускают из виду.

Мы особенно благодарны Даршалу Шетти и Чаникье Вманагари не только за отзывы, но и за их работу по улучшению примеров кода. Мы также благодарим Юфея Янга, чей острый глаз уловил несколько очень тонких ошибок уже на этапе завершения книги.

Мы благодарим Маттиаса Феллейзена и всю команду Racket, которые привнесли дух языка Scheme в Racket. Мы восхищаемся

этой чрезвычайно успешной, продолжающейся много лет работой. И особая благодарность Дораи Ситараму за его создание SLAT_{EX}.

Наша редакционная команда в MIT Press, состоящая из Элизабет Суэйзи, Мэтью Валадеса и Джея Марси, заслужила нашу благодарность и признательность за то, что сделала процесс подготовки издания максимально беспроblemным. Спасибо также Алекс Хупс за ее первоначальное участие в процессе.

Мы благодарим Южень Е, Линн Миколон, Бениту Браун и Чарльза Поупа, которые сделали нашу жизнь в Университете Индианы намного проще, чем кто-либо мог ожидать. Мы благодарим Кристину Лао, Рунака Виджана и Джонатана Ву, а также Спенсера Балло и Амога Батвала, которые были первыми читателями, за вдохновение, полученное в результате обсуждений и технических диалогов.

И наконец, мы благодарим наши семьи за их поддержку, терпение и стойкость на протяжении бесконечных видеозвонков между Дэном и Анурагом, а также за все выходные дни и будние ночи, поглощенные нашей писательской страстью.

Дэн благодарит свою замечательную жену Мэри; Роберту (читательницу рукописи) и Шеннона и их детей, Саманту и Чейза; Рэйчел и Джозефа и их дочь Уиллоу; Сару и Трэвиса и их детей, Бруклин, Арию и Вьятт.

Анураг благодарит свою замечательную жену Аруну и двух дочерей, Ришму и Арию Нину (также одну из первых читательниц).

Введение

Нельзя пропускать главы, в жизни так не бывает. Надо читать каждую строчку, знакомиться с каждым персонажем. Никто не обещал вам сплошное удовольствие. Некоторые главы повергнут вас в ад и заставят рыдать над ними целыми неделями. Вам придется читать вещи, которые мучительно неприятны; иногда вы, напротив, будете мечтать, чтобы страницы не заканчивались. Но надо идти вперед. Истории движут мир. Живите полной жизнью, не пропуская главы.

– Кортни Пеппернелл, цитата из книги *Pillow Thoughts II*, издание Andrews McMeel Publishing, © 2018

Глубокое обучение не только стало новой областью искусственного интеллекта, оно произвело революцию в способах решения задач, будь то победа в го, распознавание кошек на картинках или просьба к умной колонке заказать пиццу. Самое прекрасное в глубоком обучении – это то, как изящно простые части объединяются для решения больших и сложных задач. Как лучше всего понять, что заставляет эти инструменты глубокого обучения работать? Наш подход заключается в том, чтобы создавать базовые функции – составные части глубокого обучения – и наблюдать за их работой сначала по отдельности, а потом вместе.

Способность справляться с зашумленными данными – вот что делает глубокое обучение, являющееся разновидностью машинного обучения, таким заманчивым. Теперь стало ясно, что получать на 100 % однозначные ответы на зашумленных данных не только невозможно, но и не нужно. Наше ощущение обязательной точности решений, характерное для многих задач, для которых мы пишем программы, постепенно исчезает. Тем не менее мы можем и должны поддерживать чувство уверенности в функциях, которые мы определяем, чтобы не противоречить собственной интуиции, чтобы мы могли быть уверены, что эти функции соответствуют нашим ожиданиям.

Для освоения инструментов глубокого обучения вам потребуются только базовые знания математики старшей средней школы, а также некоторый опыт программирования. Функции, которые мы определяем, предназначены для запуска и экспериментов с кодом. Конечно, эту книгу можно читать, не запуская их, но каждое определение необходимо хорошо понимать.

Как читать эту книгу

Если тема этой книги для вас абсолютно нова или вам кажется непривычным способ изложения, вы должны читать каждую главу, пока полностью не поймете ее. Возможно, некоторые главы придется прочитать несколько раз. Не продвигайтесь вперед, пока у вас не появится чувство полного понимания. Обменивайтесь знаниями с другими и обсуждайте с ними непонятное, пока каждая мелочь не встанет на свои места.

В этой книге бесполезно забегать вперед, потому что изложение строго упорядочено в виде последовательности пронумерованных фреймов, разделенных горизонтальными линиями. Каждая глава довольно короткая и содержит много пробелов, поэтому вам не составит труда внимательно перечитать всю главу, чтобы поймать потерянную нить.

В конце книги есть два приложения – тоже строго упорядоченных. Они объясняют, как создавать базовые инструменты, помогающие в глубоком обучении. Эти приложения также предполагают наличие лишь минимальных базовых знаний, но предъявляют к читателю чуть более высокие требования, чем остальная часть книги.

Как написаны наши программы

Мы оформляем свои идеи в виде небольших программ Scheme, которые позволяют выражать мысли наиболее ясно и прямо. Это очень лаконичный и интуитивно понятный язык, так что код говорит сам за себя.

Мы используем очень небольшое подмножество Scheme: `define` (или `let`) позволяет присвоить значению глобальное (или локальное) имя, `lambda` создает функцию как значение, а `cond` выполняет диспетчеризацию по последовательности пар (тестовых значений). Существуют также примитивные функции, такие как `+`, `-`, и `*`, которые работают с числами. О них будет рассказано в главе 0, которая служит своего рода развернутым введением.

Мы используем маленькие рамки для выделения кода в тексте и объясняем, как работают программы в этих рамках. Если вы не пропускаете главы, понять объяснения не составит труда. Эта книга представляет собой набор понятий, почти все из которых являются функциями. Если мы просим дать определение функции (в левой части фрейма), потратьте некоторое время и создайте правдоподобное определение, прежде чем смотреть на наш ответ (в правой части фрейма). Вот простой пример фрейма с определениями функций в рамках:

Это рамка с кодом

```
(define функция
... которая хорошо сюда помещается
...)
```

¹ Это красная рамка.

Она встречается не очень часто.

Впервые мы используем ее во фрейме 26:25,

и

эта запись означает фрейм 25 на стр. 26

```
(define другая_функция
... запомните эту функцию! ...)
```

Мы также выделяем ключевые свойства как правила или законы. Правила касаются структуры сущностей, например размеров. Законы описывают поведение сущностей, такое как их равенства и инварианты.

Как запустить код

Мы собрали функции и синтаксические расширения, необходимые для кода из этой книги, в пакет Machine Learning Toolkit под названием Malt. Malt – это пакет в Racket, который является надмножеством нашего небольшого подмножества Scheme. Пакет включает наш код и примеры, а также инструменты, необходимые для экспериментов с ними. Рекомендации по его использованию доступны на сайте www.thelittlelearner.com.

Как есть десерты

Те, кто знаком с предыдущими «книжками-малышками», уже знают, что мы обожаем игру слов и стараемся добавить немного юмора при каждом удобном случае. Очень трудно найти юмор в цифрах, поэтому мы ищем его в других местах. Мы составили список десертов для употребления в ближайшем кафе. Не откажите себе в удовольствии! Но и не слишком налегайте на сладкое и не забывайте сначала съесть отварные овощи.

Мы надеемся, что этот небольшой экскурс в глубокое обучение доставит вам удовольствие и что читать его так же интересно, как и писать.

Приятного аппетита!

Дэниел Фридман, Блумингтон, Индиана

Анураг Мендхекар Лос-Альтос, Калифорния

Перевод сокращенной записи в язык Scheme

Некоторые из наших функций мы пишем с помощью более компактной нотации, чтобы они были легче для чтения и плотно помещались в «маленьких коробочках». Перед запуском программы обязательно переведите нашу запись в код Scheme. В таблице ниже показано, как это сделать.

Первый столбец в таблице ниже указывает на место самого первого применения в книге обозначений, показанных во втором столбце. Третий столбец показывает, как переводить запись программ (например, $[5 (+ 10 2) 28]$ переводится в `(tensor 5 (+ 10 2) 28)`).

Страница:фрейм	Обозначение	Расшифровка
23:17	$[t\ ts\ \dots]$	<code>(tensor t ts . . .)</code>
25:24	l_i	<code>(ref l i)</code>
26:27	$(\text{list } m\ \dots)$	<code>(list m . . .)</code>
32:17	$\{t\}$	<code>(tlen t)</code>
34:24	$t _i$	<code>(tref t i)</code>
39:42	$ l $	<code>(len l)</code>
48:22	$\langle\langle op \rangle^{(rank)} t \rangle$	<code>(⟨op⟩ – ⟨rank⟩ t)</code>
74:15	$(\nabla f \theta)$	<code>(gradient-of f θ)</code>
102:25	$(\bullet \cdot t\ u)$	<code>(dot-product t u)</code>
103:26	$\langle\langle op \rangle^{(rank_1), (rank_2)} t \rangle$	<code>(⟨op⟩ – ⟨rank₁⟩ – ⟨rank₂⟩ t)</code>
121:27	$t _b$	<code>(trefs t b)</code>
226:49	l_{i_l}	<code>(refr l i)</code>

Например, на стр. 48, фрейм 22, мы вводим дефис между *sum* и 1 для перевода записи sum^1 в `sum - 1`, а на стр. 103, фрейм 26, вводим второй дефис для перевода записи $\bullet^{1,1}$ в `dot-product-1-1`.

Греческие буквы и письменные варианты имен переменной, такие как $\hat{a}$, α , $\hat{\alpha}$, $an-\alpha$, β , $\hat{c}$, ϵ , θ , Θ , λ , μ и π , в коде программы пишут-

ся соответственно как $\hat{a}$, α , $\alpha\text{-hat}$, $\text{an-}\alpha$, β , $\hat{c}$, ϵ , θ , $\text{big-}\theta$, λ , μ и π . В именах формальных аргументов, функций и в ключевых словах можно использовать символы Unicode.

Для успешного выполнения кода требуется установка пакета Malt для Racket v8.0 или более поздней версии. Подробную информацию о загрузке и запуске кода можно найти на www.thelittlelearner.com.

О. НУ ЧТО, ЗАИНТРИГОВАНЫ?



Эй. Эй. Ээээй!

1 Сдается мне, Тотошка, мы вовсе не в Канзасе[†].

[†] По мотивам произведений Лаймена Баума (1856–1919) и Эдгара Вульфа (1881–1943).

Как насчет краткого обзора?

2 Обзора чего?

Языка программирования, который мы будем использовать в этой книге.

3 Да, было бы здорово узнать про новый язык.

Вы прочитали краткое введение?

4 Конечно, ведь никто не пропускает страницы введения. Они такие содержательные и полезные...
 Ой, ладно, сейчас прочитаю.

Те, кто уже знаком с языком Scheme[†], могут перейти к следующей главе «Строки сегодня спят», быстро посмотрев таблицу на стр. **xxiii**. Возможно, вы найдете там несколько знакомых обозначений.

5 Хорошо.

[†] Спасибо Джону Маккарти (1927–2011), Джеральду Сассману (1947–) и Гаю Стилу-мл. (1954–).

Первым делом научимся давать имена величинам

6 Получается, мы дали имя числу 3.14?

```
(define pie† 3.14)
```

[†] Это не самый вкусный торт (*pie*), ему постоянно немного не хватает, чтобы получить ровно 4 балла.

Да, так и есть.

Вот еще несколько определений:

```
(define a-radius 8.4)
```

```
(define an-area  
  (* pie  
    (* a-radius a-radius)))
```

7 Если символ * обозначает умножение, то значение *an-area* будет равно
 221.5584?

Верно.

Вызов функции может содержать ноль или больше аргументов:

$\langle \text{функция} \rangle \langle \text{аргумент} \rangle \dots$

Поскольку все математические операции (такие как $*$, $+$ и т. д.) являются функциями, их записывают таким же образом.

8 Как можно создать новую функцию?

Давайте создадим функцию с одним формальным аргументом r .

Формальные аргументы – это имена, присвоенные аргументам, передаваемым в функцию при вызове

$(\lambda (r)$
 $(* \text{pie}$
 $(* r r)))$

В этом выражении λ (или `lambda`) обозначает начало новой функции. Далее идут формальные аргументы. В данном случае это r . За ним следует тело функции – выражение, которое возвращает значение функции.

Что вернет эта функция?

9 Эта функция возводит в квадрат значение r и перемножает результат с `pie`. Судя по всему, функция возвращает площадь окружности с радиусом r .

У этой функции есть имя?

Пока нет.

Но мы можем его задать при помощи оператора `define`

```
(define area-of-circle
  (\lambda (r)
    (* pie
      (* r r))))
```

10 Ясно.

Оператор λ используют для создания функции, а `define` – чтобы дать ей имя.

Получается, что функции можно рассматривать как величины?

Да.

Функции тоже являются величинами, и с ними можно обращаться как с величинами.

11 Означает ли это, что функции могут порождать другие функции?

Совершенно верно.

Вот вам пример:

```
define area-of-rectangle
  (λ (width)
    (λ (height)
      (* width height))))
```

Как думаете, что представляет собой

(area-of-rectangle 3.0)?

12 Функция *area-of-rectangle* – это функция с одним формальным аргументом. Но она порождает функцию

```
(λ (height)
  (* width height))
```

Правильно?

Почти.

Важный нюанс, о котором нельзя забывать, заключается в том, что *width* внутри этой функции всегда имеет значение 3.0.

Вот как это можно записать по-другому:

```
(λ (height)
  (* 3.0 height))
```

13 Ловкий трюк!

Внутренняя функция запоминает аргумент, переданный в качестве формального аргумента внешней функции.

Можем ли мы также передавать функции в качестве аргументов другим функциям?

Конечно можем.

Вот пример:

```
(define double-result-of-f
  (λ (f)
    (λ (z)
      (* 2 (f z)))))
```

Поясните, какую роль здесь играет *f*?

14 Здесь *f* является формальным аргументом *double-result-of-f*, а затем *f* вызывается для *z*. Следовательно, *f* – это функция.

Правильно.

Рассмотрим пример функции, которая прибавляет 3 к своему формальному аргументу:

```
(define add3
  (λ (x)
    (+ 3 x)))
```

Что произойдет, если мы вызовем *double-result-of-f* с *add3* в качестве аргумента?

(*double-result-of-f add3*)

Очень хорошо.

Эти соображения можно записать так:

```
(λ (z)
  (* 2 (add3 z)))
```

Что произойдет, когда мы вызовем эту функцию с аргументом 4?

Отлично.

Теперь закончите свою мысль.

15 Вызывая *double-result-of-f* для *add3*, мы получаем внутреннюю функцию (функцию-аргумент).

Мы помним, что формальный аргумент *f* передается из внешней функции, следовательно, мы знаем, что *f* – это *add3*.

16 Значение *z* внутри этой функции теперь равно 4, поэтому мы получаем

```
(* 2 (add3 4))
```

17 Выражение

```
(add3 4)
```

эквивалентно

```
((λ (x)
  (+ 3 x))
  4)
```

а результат равен 7.

Подставляя его в предыдущее выражение, мы получаем

```
(* 2 7)
```

что дает нам 14.

Правильно.

Такой способ запоминания аргументов, передаваемых из внешних функций, внутри вложенных функций называется β -редукцией[†].

Это полезный инструмент для понимания вложенности функций. Язык Scheme идеально подходит для выполнения β -редукции.

[†] Предложил Алонзо Чёрч (1903–1995). При выполнении β -редукции необходимо соблюдать осторожность: все имена в определении должны быть уникальными на каждом шаге. Формальным аргументам можно дать новые имена, чтобы гарантировать, что они всегда уникальны.

18 Является ли `define` функцией?

Вовсе нет.

Это ключевое слово.

Выражения, которые начинаются с ключевого слова, называются также особыми формами. Это не то же самое, что вызов функции.

19 Понятно, а λ является ключевым словом?

Да.

Вот пример еще одной особой формы:

```
(cond
  ((= pie 4) 28)
  (< pie 4) 33)
(else 17))
```

Результатом этого выражения будет 33.

Поясните, почему.

20 Здесь мы видим ключевое слово `cond`[†].

Слово `cond` является сокращением от *conditional* (условный), верно?

[†] Также называется оператором Маккарти.

Да.

21 Значит, мы проверяем два условия:

(= *pie* 4)

и

(< *pie* 4)

Верно.

22 Поскольку *pie* не равно 4, первый тест не проходит[†], но второй тест пройдёт успешно[‡].

[†] Результат *false* (ложь) записывается в Scheme как *#f*.

[‡] Результат *true* (истина) записывается в Scheme как *#t*.

Да, вы правы.

23 Выражение *cond* возвращает 33, потому что это значение соответствует истинному условию.

Каждая комбинация теста и значения называется *клаузой*[†] (*clause*, утверждение), где *else* рассматривается как истина. Значение выражения *cond* равно значению первой клаузы, прошедшей тест.

[†] В оригинальном тексте автор использует термин *clause* намеренно. Клауза – это не просто любое утверждение, а «синтаксически и лексически завершённое краткое предложение», являющееся частью более длинного текста. Это позволяет передать точный смысл комбинации теста и значения. Термин *клауза* широко известен в профессиональных кругах лингвистики, филологии и философии, и его использование в переводе помогает сохранить точность и адекватность оригинала. – *Прим. ред.*

Очень хорошо.

Теперь объединим три особые формы, которые мы только что изучили:

```
(define abs
  (λ (x)
    (cond
      ((< x 0) (- 0 x))
      (else x))))
```

Как вы думаете, что делает *abs*?

24 Эта функция находит модуль числа!

Она принимает единственное число *x*, и если оно меньше 0, то вычитает его из 0.

В результате получается положительное число.

В остальных случаях функция просто возвращает неотрицательное число *x*.

Нам также нужен способ определять локальные имена внутри наших функций, причем эти имена не должны быть видны снаружи функций.

25 Да, это должно быть полезно для объявления хорошо читаемых функций.

Это делается с помощью особой формы *let*, например

```
(define silly-abs
  (λ (x)
    (let ((x-is-negative (< x 0)))
      (cond
        (x-is-negative (- 0 x))
        (else x)))))
```

26 Думаю, что мы присвоили условию

(< x 0)

локальное имя

x-is-negative

Да, вы правы.

Последнее выражение в *let* (в данном примере это *cond*) называется телом *let*. Разумеется, вместо *cond* может быть любое другое выражение. Имя *x-is-negative* можно использовать где угодно внутри тела *let*, но только не снаружи функции.

27 Это удобно.

Прекрасно. Теперь у нас есть все, что нужно.	28	Постойте, мы еще не рассмотрели циклы.
В нашем языке нет циклов.	29	Что? Как можно определить сложные функции без циклов?
Мы используем рекурсивные функции.	30	Это когда в теле функции мы ссылаемся на имя, данное самой функции?
Точно. Давайте определим простую функцию нахождения остатка от целочисленного деления [†] . Для начала опишем порядок ее работы. Допустим, нам нужно найти остаток от деления 13 на 4. Остаток – это то, что останется от числа 13 после удаления из него максимально возможного количества четверок. Что мы получим в результате?	31	Остаток будет равен 1, потому что число 13 можно составить из трех четверок и единицы.
----- [†] Это просто, потому что мы предполагаем, что первый аргумент неотрицательный, а второй положительный.		
Верно. Пройдем этот процесс по шагам. Поскольку 13 больше, чем 4, мы вычитаем 4 и получаем 9. Каким будет следующий шаг?	32	Число 9 также больше, чем 4, поэтому мы можем вычесть его еще раз и получить 5.

Хорошо.

Число 5 больше, чем 4, поэтому можно выполнить еще одно вычитание и получить 1.

Прекрасно.

Рассмотрим каркасный код функции *remainder*, которая находит остаток от деления:

```
(define remainder
  (λ (x y)
    (cond
      ((< x y) X)
      (else R))))
```

Какие выражения должны находиться внутри рамок, обозначенных буквами *X* и *R*?

В таком случае мы неоднократно вычтем *y* из *x* и продолжим находить остаток, используя результат вычитания.

Правильно.

Мы делаем это рекурсивно, вызывая функцию *remainder*.

Отлично.

Вот окончательный вид функции *remainder*:

```
(define remainder
  (λ (x y)
    (cond
      ((< x y) x)
      (else (remainder (- x y) y)))))
```

33 Теперь результат вычитания меньше, чем 4, и мы не можем продолжать вычитать. Следовательно, остаток равен 1.

34 Чтобы найти остаток, сначала нужно проверить, можем ли мы вычесть *y* из *x*. Это возможно только в том случае, когда *x* больше или равен *y*.

Если же *x* меньше, чем *y*, то *x* должен быть остатком.

Поэтому когда

$(< x y)$

функция возвращает *x*. Значит, *X* – это *x*.

Что произойдет, если *x* не меньше, чем *y*?

35 Ага, значит, мы должны найти остаток от деления

$(- x y)$ и *y*.

36 Значит, вместо *R* нужно подставить

$(remainder (- x y) y)$

37 Этот пример можно использовать в качестве шаблона для определения рекурсивных функций?

Да, конечно.

Каждая рекурсивная функция проверяет свои аргументы на соответствие требованиям теста. Мы будем называть его основным тестом, а результирующее значение – основным значением.

Каковы основные тест и значение в нашем примере?

38 Первая клауза в `cond` – это

$(< x y)$ – основной тест

и

x – основное значение.

А у второй клаузы есть название?

Да.

Она называется рекурсивной.

39 Не могли бы вы привести еще один пример?

Охотно.

Рассмотрим каркасный код еще одной рекурсивной функции *add*, которая складывает два натуральных числа[†], не используя «+» непосредственно в функции:

40 Основной тест должен проверить, является ли одно из чисел нулем, так как мы ожидаем только натуральные числа.

Означает ли это, что нам нужны два основных теста?

```
(define add
  (λ (n m)
    (cond
      ([ T ] [ V ])
      (else [ R ]))))
```

Аргумент n – это первое число, а m – второе.

Нам нужно найти T , V и R , но сперва разберемся, каким должен быть основной тест в этой функции.

[†] Другое название неотрицательных целых чисел, по определению включающих 0; для них определена функция *add1*.

Это возможно, но не обязательно.

Поскольку при сложении порядок аргументов не имеет значения, мы можем ограничиться проверкой только второго числа.

Каким должно быть основное значение?

Это хорошая отправная точка.

Теперь мы можем заполнить поля T и V .

Запишите обновленный каркасный код. Используйте функцию `zero?`, которая проверяет, не является ли аргумент нулем.

41 Основным значением должно быть первое число, так как прибавление любого числа к 0 возвращает это число.

42 Вот обновленный код:

```
(define add
  (λ (n m)
    (cond
      ((zero? m) n)
      (else R))))
```

Правильно.

Теперь займемся полем R . Если m не является нулем, мы можем сказать, что это некоторое число

$$(k + 1)$$

где k – натуральное число.

Что мы получим, если сложим $k + 1$ и n ?

43 Мы получим

$$(k + 1) + n$$

Хорошо.

Мы можем переписать это выражение так:

$$(k + n) + 1$$

44 Поясните, чем это нам поможет?

В случае натуральных чисел мы всегда можем полагать, что существует такая функция *add1*, которая возвращает $k + 1$ для заданного k .

Аналогично, для заданного $k + 1$ всегда существует функция *sub1*, которая возвращает k .

45 К чему мы приходим?

В соответствии с фреймом 44 с предыдущей страницы поле *R* должно содержать выражение

$$(k + n) + 1$$

У нас уже есть n , как нам получить k из m ?

46 Так как m равно $k + 1$, мы можем найти k с помощью $(\text{sub1 } m)$

Отлично.

В поле *R* также нужно прибавить 1 к сложению k (которое равно $(\text{sub1 } m)$) и n .

47 Получается, поле *R* должно выглядеть так:
 $(\text{add1 } \dots \text{ сложение с } n \text{ and } (\text{sub1 } m) \dots)$

Точно.

Теперь у нас есть понимание важного момента.

Так как n и $(\text{sub1 } m)$ представляют собой натуральные числа, мы можем сложить их при помощи вызова *add*!

48 Потрясающе!
Значит, теперь *R* можно записать проще:
 $(\text{add1 } (\text{add } n (\text{sub1 } m)))$

Правильно!

Теперь запишите окончательный вариант функции *add*.

49 Вот он:

```
(define add
  (λ (n m)
    (cond
      ((zero? m) n)
      (else (add1 (add n (sub1 m)))))))
```

Великолепно.	50	Так как 2 не удовлетворяет условию <i>zero?</i> , мы попадаем в рекурсию. Поэтому результат должен быть
Давайте для примера возьмем числа 7 и 2		$(add1 (add1 (add1 (add1 (add1 (add1 (sub1 2))))))$
$(add1 7 2)$		что аналогично результату
Что произойдет?		$(add1 (add1 (add1 7 1)))$

Верно.	51	Теперь второй аргумент равен нулю, значит, выражение
Поскольку второй аргумент для сложения в этом рекурсивном вызове равен 1 (т. е. не ноль), мы можем переписать выражение как		$(add1 7 0)$
$(add1 (add1 (add1 (add1 (add1 (add1 (sub1 1))))))$		дает нам
что аналогично		7
$(add1 (add1 (add1 7 0)))$		

Правильно.	52	Это и есть наш результат!
Теперь мы можем выполнить два вложенных вызова <i>add1</i> для 7		
$(add1 (add1 7))$		
и получим		
$(add1 8)$		
что равно		
9		

Более кратко мы можем описать это поведение в виде логического столбца[†]:

1. $(add\ 7\ 2)$
2. $(add1\ (add\ 7\ (sub1\ 2)))$
3. $(add1\ (add\ 7\ 1))$
4. $(add1\ (add1\ (add\ 7\ (sub1\ 1))))$
5. $(add1\ (add1\ (add\ 7\ 0)))$
6. $(add1\ (add1\ 7))$
7. $(add1\ 8)$
8. 9

Итак, есть еще семь шагов, помимо исходной задачи:

$(add\ 7\ 2)$

и, осторожно переходя от одного выражения к другому, которое всегда имеет одно и то же значение, мы упрощаем результат до

9

Теперь попробуйте сделать это для

$(remainder\ 13\ 4)$

[†] Этот способ записи был представлен в книге *The Little Typer* (2018) и показывает последовательность выражений, которые логически равнозначны между собой.

Спорное утверждение! В обоих наших примерах есть как минимум один уменьшающийся аргумент, передаваемый в основной тест.

⁵³ Это увлекательно, но не так сильно, как банановый сплит с шоколадной помадкой, верно?

1. $(remainder\ 13\ 4)$
2. $(remainder\ (-\ 13\ 4)\ 4)$
3. $(remainder\ 9\ 4)$
4. $(remainder\ (-\ 9\ 4)\ 4)$
5. $(remainder\ 5\ 4)$
6. $(remainder\ (-\ 5\ 4)\ 4)$
7. $(remainder\ 1\ 4)$
8. 1

⁵⁴ Это справедливо для всех рекурсивных функций?

Да, если вы рассчитываете получить от рекурсивной функции значение.

В общем случае, определяя рекурсивную функцию, необходимо выполнить следующие шаги:

- понять, каким будет основной тест;
- понять, каким будет основное значение;
- решить, каким образом меняются аргументы рекурсивного вызова, особенно убывающие;
- использовать рекурсивный вызов как часть более крупного выражения для получения искомого результата функции.

Часть выражения, включающую в себя рекурсивный вызов, называют его оберткой (*wrapper*)[†].

[†] В определении *add* это часть между «(*add1*» и соответствующей правой скобкой.

Вы внимательно прочитали каждую букву пояснений в этой главе?

Тогда все в порядке.

У нас есть все, что нужно для начала.

Остальное вы узнаете по ходу дела.

55 Все равно немножко непонятно.

56 Да, и примечания тоже.

57 Замечательно!

1. СТРОКИ СЕГОДНЯ СПЯТ¹



¹ В оригинале глава называется «The Lines Sleep Tonight» по мотивам названия песни Соломона Нтселе (1909–1962) «Львы сегодня спят» (The Lions Sleep Tonight). Эта песня звучала в фильме «Эйс Вентура: розыск домашних животных».

С возвращением!

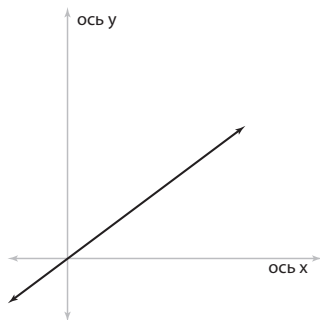
1 Как хорошо здесь!

Еще бы.

2 Да.

Помните этот график?

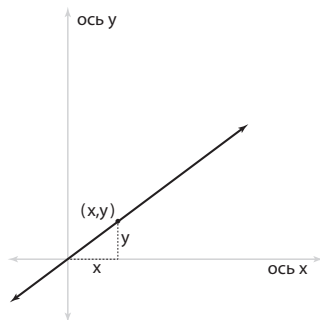
Это линия на плоскости (двумерной), проходящая через начало системы координат, образованный осями x и y [†].



[†] Изобретена Рене Декартом (1596–1650).

Существует уравнение, соотносящее значения x и y для каждой точки (x, y) на линии.

3 Означает ли использование стрелок на обоих концах линии, что она простирается бесконечно в обоих направлениях, и следует ли из этого, что каждому x соответствует y ?



Да.

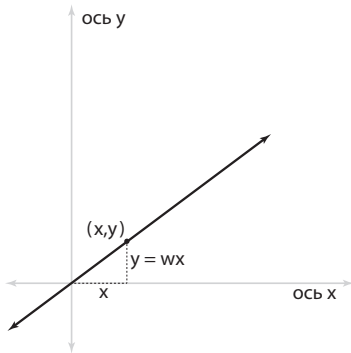
4 Что это такое?

Наша линия проходит через начало координат.

Точка $(0, 0)$, в которой пересекаются ось x и ось y , называется началом координат. Черная линия на рисунке проходит через начало координат.

5 Так что за уравнение описывает эту линию?

Поскольку эта линия проходит через начало координат, значение y получается умножением x на постоянный множитель w , который называется наклоном линии.

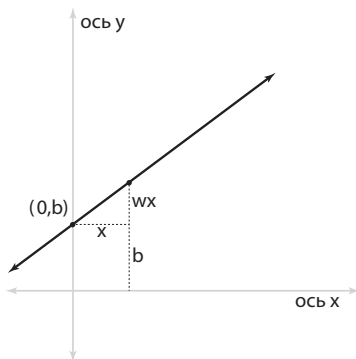


⁶ Получается, это уравнение $y = wx$?

Да, вы правы.

⁷ А что, если линия не проходит через начало координат?

Хороший вопрос.
В этом случае график выглядит так:



⁸ Мне кажется, в этом случае линия поднята на величину b . Теперь y можно определить через x при помощи уравнения

$$y = wx + b^\dagger$$

[†] Возможно, вам больше знакомо уравнение $y = mx + b$.

Как в этом случае найти y ?

Так выглядит первая попытка написать функцию *line*[†]

```
| (define line
|   (λ (x)
|     (λ (w b)
|       (let ((y (+ (* w x) b)))
|         y))))
|
```

[†] Пунктирная линия слева означает, что это определение не является окончательным. Несмотря на все попытки, пунктирная линия может никогда не исчезнуть и перейти в финальную рамку черного либо красного цвета.

⁹ Это функция от одного аргумента, которая возвращает функцию от двух аргументов[‡]?

Определение функции *line* нуждается в пояснении. Почему мы не получаем **w** и **b** перед *x*?

[‡] Если вам непонятно, о чем речь, перечитайте предыдущую главу.

Хороший вопрос!

Такое определение предполагает, что **w** и **b** известны раньше, чем аргумент *x*.

Однако здесь мы имеем дело с задачей другого рода, где *x* известен, но **w** и **b** должны быть вычислены из множества заданных значений *x* и *y*.

¹⁰ Почему это определение функции *line* не является окончательным?

Дело в том, что мы можем сделать функцию *line* еще чуть проще. Поскольку тело *let*, т. е. *y*, представляет собой всего лишь имя, присвоенное выражению

$(+ (* w x) b)$

¹¹ Получается, следующее определение с пунктирной линией должно быть нашей следующей попыткой получить окончательную версию функции *line*

```
| (define line
|   (λ (x)
|     (λ (w b)
|       (+ (* w x) b))))
|
```

при помощи `let`, мы получаем следующий список аналогий†:

1. `(let ((y (+ (* w x) b)))`
`y)`
2. `(+ (* w x) b)`

Несмотря на то что здесь больше нет `y`, мы иногда ссылаемся на это значение как на `y`, связанное с заданным `x`.

На первый взгляд все правильно, почему осталась пунктирная линия?

† Поясняется во фрейме 15:53.

Хороший вопрос.

Функция *line* сохранила пунктир потому, что скоро мы попробуем еще один подход. Но пока этого достаточно.

Поскольку **w** и **b** используются для нахождения `y`, соответствующего входному `x`, они считаются особым видом формальных аргументов. Мы называем их параметрами функции *line* и записываем жирным шрифтом, тогда как `x` – это аргумент *line*.

12 Как используется функция *line*?

Рассмотрим пример.

Каким будет результат вызова
 (*line* 8)?

13 (*line* 8) – это функция, которая запоминает, что `x` равен 8, и ожидает получить аргументы для своих параметров **w** и **b**

`(λ (w b)`
`(+ (* w 8) b))`

Правильно.

Если (*line* 8) вызвать с указанием **w** и **b**, мы можем определить у.

Скажите теперь, каким будет результат вызова

((*line* 8) 4 6)?

14 При **w** = 4 и **b** = 6 будут выполнены следующие шаги:

1. | ((*line* 8) 4 6)
2. | ((λ (w b)
| (+ (* w 8) b))
| 4 6)
3. | (+ (* 4 8) 6)
4. | (+ 32 6)
5. | 38

Это означает, что если $x = 8$, то $y = 38$.

Замечательно.

Функция, которая получает параметры после аргументов, называется параметризованной функцией.

Является ли *line* параметризованной функцией?

15 Да.

Она получает **w** и **b** как свои параметры после того, как получила аргумент x .

Но зачем нужны параметризованные функции?

Хороший вопрос.

Параметризованные функции используются там, где мы должны определить правильные значения параметров (здесь **w** и **b**) из заданных значений x и соответствующих значений y .

16 Можете привести пример?

Конечно.

Пусть *line-xs* представляет координаты *x*, а *line-ys* – координаты *y*

```
(define line-xs  
  [2.0 1.0 4.0 3.0])
```

```
(define line-ys  
  [1.8 1.2 4.2 3.3])
```

Каждой координате *x* в *line-xs* соответствует координата *y* в *line-ys*, и наоборот.

17 Я правильно понимаю, что *line-xs* и *line-ys* вместе образуют четыре точки?

(2.0, 1.8)

(1.0, 1.2)

(4.0, 4.2)

(3.0, 3.3)

Да.

18 Хорошо.

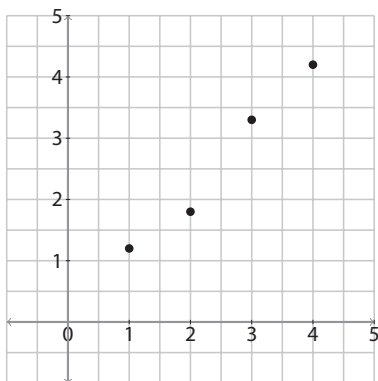
Вместе они образуют набор данных.

Так набор данных

(*line-xs*, *line-ys*)

выглядит в виде точек на графике:

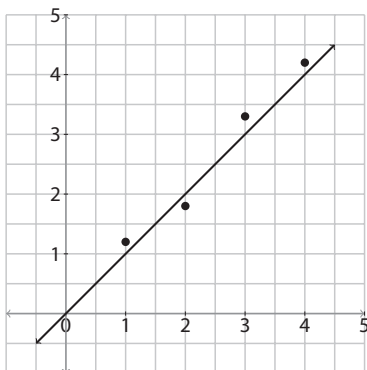
19 Эти точки выглядят так, будто они расположены почти на одной линии.



Это не случайно.

Проведите линию как можно ближе к этим четырем точкам.

20 Как вам такая линия?



Да, достаточно близкое приближение.

Линии, как было сказано во фрейме 11, имеют два параметра – w и b .

Что вы можете сказать про w и b в этом случае?

21 Так как линия проходит через начало координат, значение b равно 0.0.

И еще, значения координат x и y для этой линии всегда равны. Например, (0.0, 0.0), (1.2, 1.2) и т. д. Координаты точки имеют вид (a, a) для любой координаты a по оси x .

Значит, параметр w должен быть равен 1.0.

Вы правы.

Эта функция представляет собой прямую с параметрами

$$w = 1.0$$

и

$$b = 0.0$$

Как нам пригодится эта информация?

22 Если нам станет известна новая координата по оси x , мы можем предсказать соответствующую координату y , используя уравнение этой прямой.

Например, если $x = 3.79$, то

$$y = 1.0 \times 3.79 + 0.0 = 3.79.$$

Правило параметров (начальный вариант)

Каждый параметр является числом.

Верно.

Мы используем параметры, чтобы предсказать y при заданном x .

Процесс нахождения параметров функции по известному набору данных называется обучением.

²³ Это означает, что мы обучили функцию, которая приблизительно соответствует точкам из набора данных
(*line-xs, line-ys*)

Параметры w и b вместе образуют набор (или множество) параметров, который мы будем обозначать буквой θ .

В данном случае θ состоит из двух параметров. Их называют элементами набора.

Кроме того, w является первым элементом θ и обозначается как $\theta_0^\dagger$.

²⁴ Значит, b – это θ_1 ?

[†] Произносится «тэта малая».

[‡] Элементы индексируются начиная с 0.

Да, верно.
Давайте перепишем *line* с
учетом новых обозначений

```
(define line
  (λ (x)
    (λ (θ)
      (+ (* θ0 x) θ1))))
```

25 Почему вы поместили *line*
в рамку красного цвета с
красным кружком в углу?

Хороший вопрос.
Мы выбрали красную
рамку с кружком для
обозначения итоговых
определений, которые важны
и иногда служат базовыми
компонентами[†].

[†] Кружок предназначен читателям,
которым сложно отличить красные
линии от черных.

26 Красный – выразительный
цвет для разных штучек.
Как мы составляем θ из w и b ?

Хороший вопрос.
Вот пример.
Если
 $w = 1.0$ и $b = 0.0$,
мы составляем набор так:
(list 1.0 0.0)

27 Значит,
 θ – это список элементов,
таких, что
 θ_0 соответствует $w = 1.0$
и
 θ_1 соответствует $b = 0.0$.

Рассмотрим пример вызова
функции *line* с набором
параметров θ :

```
1. | ((line 7.3) (list 1.0 0.0))
2. | ((λ (θ)
      | (+ (* θ0 7.3) θ1))
      | (list 1.0 0.0))
```

28 Легко:

```
3. | (+ (* (list 1.0 0.0)0 7.3)
      | (list 1.0 0.0)1)
4. | (+ (* 1.0 7.3) 0.0)
5. | 7.3
```

Продолжите
последовательность действий
в виде логического столбца.

Правильно. До сих пор мы изучали наш набор данных на графике и находили значение θ .	29	А потом мы использовали его для предсказания координаты y по заданной координате x , которой может и не быть в нашем наборе данных.
Другими словами, мы обучились на наборе данных тому, как получать координату y по заданной координате x .	30	Ага, мы чему-то научились!
Тем не менее мы умеем делать это только визуально – рассматривая график с точками данных, что не всегда удобно.	31	Можем ли мы создать функцию, которая определяет θ для любого набора данных?
Блестящий вопрос. Да, можем, и такая функция – пример того, что нам известно как машинное обучение [†] . Именно этому посвящена оставшаяся часть книги!	32	Сейчас самое время сделать перерыв.

[†] Существует много разных форм машинного обучения. Здесь мы рассматриваем только одну из них.

Сундук с полезными вещами

(*line-xs*, *line-ys*) стр. 24

line, стр. 26

l_i (где l – список, а i – индекс с отсчетом от нуля) стр. 26

(*list m ...*) (где $m ...$ – элементы списка) стр. 26

Сделайте перерыв!

**КАК НАСЧЕТ МОРОЖЕНОГО
С КАРАМЕЛЬЮ?**

2. ЧЕМ КРЕПЧЕ УМ, ТЕМ ТОЛЩЕ КНИГИ



У нас до сих пор послевкусие мороженого с карамелью.	1	Ммм... фантастика!
Является ли 5 натуральным числом?	2	Да. Это неотрицательное целое число.
А что насчет 0?	3	Да. Это натуральное число.
Является ли натуральным число -5 ?	4	Нет. Натуральные числа не бывают отрицательными.
А число 7.18?	5	Это вещественное число.
Число -13.713 тоже вещественное?	6	Да. Можем ли мы сказать, что все натуральные числа также являются вещественными?
Да, мы считаем их вещественными. Но в этой книге мы будем использовать другое название для всех вещественных чисел. Мы будем называть их скалярами.	7	То есть число 7.18 – это скаляр?
Да. А что вы думаете про число π ? ----- † Это π намного вкуснее, чем πe из фрейма 2:6, потому что π содержит обильную посыпку из десятичных знаков!	8	Вы имеете в виду число 3.141592653589793...? Да, π – тоже скаляр.

Правильно.

9 Понятно.

Предикат *scalar?* проверяет, является ли что-то скаляром, и, как и все предикаты, относится к функциям, которые всегда возвращают логическое значение – либо ложь $\#f$, либо истину $\#t$.

[†] Алгебру двоичной логики разработал Джордж Буль (1815–1864).

А вот перед вами *tensor*^{1†}. Он содержит только скаляры:

[5.0 7.18 π]

10 А над словом тензор[‡] стоит надстрочный индекс, верно?

[‡] Термин «тензор» ввели Вольдемар Фогт (1850–1919), Григорио Риччи-Курбастро (1853–1925), Туллио Леви-Чивита (1873–1941), а вклад в популяризацию внес Альберт Эйнштейн (1879–1955).

[†] Произносится «тензор один».

Да, так и есть.

11 Эти квадратные скобки кажутся мне знакомыми!

Можно сказать, что *tensor*¹ представляет собой группу[†] скаляров

[2.0 1.0 4.0 3.0]

[†] *tensor*¹ можно рассматривать как вектор или одномерный массив. Пустой тензор не существует.

Наверное, потому что вы их уже видели в *line-xs* во фрейме 23:17.

12 Этот *tensor*¹ содержит скаляр 8.

А существует ли *tensor*^{2†}?

Вот еще один пример:

[8]

[†] Произносится «тензор два».

Да, причем *tensor*² состоит из элементов *tensor*^{1†}.

13 Что такое элемент?

[†] *tensor*² можно рассматривать как «матрицу» или двумерный массив.

Например, тензор [[7 6 2 5] [3 8 6 9] [9 4 8 5]] содержит 3 элемента: [7 6 2 5] [3 8 6 9] и [9 4 8 5]	14	То есть если у нас есть тензор, который состоит из элементов tensor^m , мы будем обозначать его tensor^{m+1} ?
Да, но при одном условии. Все tensor^m должны иметь одинаковое количество элементов.	15	Звучит разумно.
Что вы можете сказать о трех tensor^1 во фрейме 14?	16	Каждый из них содержит 4 элемента, и каждый элемент – скаляр.
Количество элементов в тензоре t можно получить вызовом функции (расшифровка на стр. xxiii) $\dagger t \dagger$ Например: 1. $\dagger [17\ 12\ 91\ 67] \dagger$ 2. 4	17	Этот вызов вернет 2: 1. $\dagger [3\ 2\ 8] [7\ 1\ 9] \dagger$ 2. 2
Что вы скажете про вызов $\dagger [[3\ 2\ 8] [7\ 1\ 9]] \dagger$		
Да, правильно.	18	Является ли тензором [[[[8]]]]?
Да, тензор [[[[8]]]] – это tensor^4 с элементом [[[8]]] – это tensor^3 с элементом [[8]] – это tensor^2 с элементом [8]	19	Он выглядит как tensor^2 , состоящий из трех элементов tensor^1 . Может ли существовать тензор [[[5] [6] [7]] [[8] [9] [0]]]?

– это tensor^1 с элементом

8

представляющим собой скаляр.

Что вы скажете про

$[[[5\ 6\ 4]\ [9\ 1\ 1]\ [0\ 6\ 2]]]$?

Да, тензор

$[[[5]\ [6]\ [7]]\ [[8]\ [9]\ [0]]]$

– это tensor^3 , состоящий из двух tensor^2 , каждый из которых содержит три элемента tensor^1 , представляющих собой одиночный скаляр.

20 Если tensor^{m+1} должен иметь только элементы tensor^m , то tensor^1 должен состоять из элементов tensor^0 .

Означает ли это, что скаляр представляет собой tensor^0 ?

Верно подмечено.

Действительно, скаляр (например, 9) – это tensor^0 †.

† Но массивы «нулевой размерности» встречаются очень редко, если вообще встречаются.

21 Имеет ли верхний индекс тензора отдельное название?

Да, имеет.

Верхний индекс называется рангом тензора.

22 Что означает ранг тензора?

Ранг тензора говорит нам о том, насколько глубоко вложены его элементы.

Ниже мы видим tensor^3 , потому что он содержит 1 элемент tensor^2 , который состоит из 2 элементов tensor^1 , содержащих по два скаляра в каждом:

$[[[8\ 9]\ [4\ 7]]]$ †

† Ранг тензора можно определить, просто сосчитав количество левых квадратных скобок перед крайним левым скаляром (здесь 8).

23 Пожалуй, мы можем написать функцию для определения ранга тензора.

Правило ранга

Ранг тензора – это количество квадратных скобок слева от крайнего левого скаляра.

Конечно можем.

Но перед этим давайте посмотрим, как забраться внутрь тензора, чтобы посмотреть на его элементы. Операция[†]

$$t|_i$$

выбирает i -й элемент тензора.

[†] Произносится «тензор t i -й».

Вот функция, которая находит ранг тензора:

```
(define rank
  (λ (t)
    (cond
      ((scalar? t) 0)
      (else (add1 (rank t|0))))))
```

Здесь $t|_0$ обозначает 0-й элемент t , который существует всегда, потому что мы определили, что t не скаляр, и кроме того, 0-й элемент сам по себе является тензором.

Теперь поговорим о ранге.

[†] Здесь (*add1* и соответствующая закрывающая скобка *)* образуют обертку рекурсивного вызова (*rank t|₀*). Эта форма записи рекурсивной функции, где мы следуем структуре аргументов, называется естественно-рекурсивной функцией. См. *The Little LISP* (1974), стр. 21, или *The Little Schemer* (1996), стр. 45. См. также *zero?* во фрейме 12:42 и *add1* и *sub1* во фрейме 13:45.

²⁴ Как это поможет нам узнать ранг тензора?

²⁵ Если основным тест[‡] (*scalar? t*) пройден успешно, из этого следует, что ранг тензора равен 0. В общем случае если элементы тензора представляют собой *tensor^m*, то тензор имеет ранг

$$m + 1$$

Следовательно, в случае рекурсии мы находим значение

$$m$$

которое является рангом 0-го элемента t

$$(rank\ t|_0)$$

а затем используем *add1*, чтобы получить ранг t .

Можем ли мы увидеть с помощью логического столбца, как *rank* возвращает 3 для тензора из фрейма 23?

[‡] Этот термин представлен в главе 0.

Конечно.

Возьмем тот же тензор

$[[[8] [9]] [[4] [7]]]$,

что и во фрейме 23, и покажем, как *rank* возвращает значение 3:

1. $(rank [[8] [9]] [[4] [7]])$
2. $(add1 (rank [[8] [9]]))$
3. $(add1 (add1 (rank [8])))$
4. $(add1 (add1 (add1 (rank 8))))$
5. $(add1 (add1 (add1 0)))$
6. $(add1 (add1 1))$
7. $(add1 2)$
8. 3

26 Это элегантный естественный способ.

Есть и другой взгляд на эту задачу. Нам достаточно взглянуть на

$t|_0$

т. е. только на первый элемент тензора.

27 Да, но почему?

Дело в том, что в любом заданном тензоре вложенные тензоры имеют одинаковое количество элементов. Например, все вложенные тензоры $tensor^2$ являются $tensor^1$, и каждый из этих $tensor^1$ имеет одинаковое количество $tensor^0$.

28 Я помню это условие из фрейма 15.

Верно.

Это означает, что $tensor^m$, который является элементом $tensor^{m+1}$, имеет одинаковую с ним форму (*shape*).

29 Что такое форма тензора?

Форма тензора $[[5.2\ 6.3\ 8.0]\ [6.9\ 7.1\ 0.5]]$ представляет собой список положительных натуральных чисел $(\text{list } \underline{2}\ 3)^\dagger$ Откуда взялись эти натуральные числа?	³⁰ Этот тензор представляет собой tensor^2, состоящий из <u>2</u> tensor^1, каждый из которых имеет по 3 элемента tensor^0.
----- [†] Иногда мы подчеркиваем часть выражения, чтобы обратить на нее внимание.	
Конечно. Какова форма тензора $[[[5]\ [6]\ [8]]\ [[7]\ [9]\ [5]]]$?	³¹ Форма тензора $[[[5]\ [6]\ [8]]\ [[7]\ [9]\ [5]]]$? такая: $(\text{list } \underline{2}\ 3\ 1)$
И снова правильно. $[[[5]\ [6]\ [8]]\ [[7]\ [9]\ [5]]]$ – это tensor^3 из <u>2</u> элементов tensor^2. Каждый из них содержит 3 элемента tensor^1. Каждый из них содержит 1 элемент tensor^0, являющийся скаляром. Какова форма тензора $[9\ 4\ 7\ 8\ 0\ 1]$?	³² $[9\ 4\ 7\ 8\ 0\ 1]$ – это tensor^1 из <u>6</u> скаляров. Следовательно, $(\text{list } 6)$ является формой тензора $[9\ 4\ 7\ 8\ 0\ 1]$, верно?
Да, верно.	³³ Имеет ли скаляр форму?
Да, имеет. Но так как у скаляра нет составных частей, его форма представляет собой пустой список.	³⁴ Как нам записать пустой список?

Вот так:
(list)

35 То есть форма скаляра 9
имеет вид
(list)?

Правильно.
Чтобы найти форму тензора,
нам необходимо знать
количество элементов, из
которых он состоит.

36 Поэтому мы используем
† t †?

Именно!
Так выглядит функция *shape*,
которая находит форму тензо-
ра *t*:

37 Но постойте, что такое
cons? ‡

```
(define shape
  (λ (t)
    (cond
      ((scalar? t) (list))
      (else (cons † t †
                  (shape t|₀)†))))
```

‡ Некоторые читатели могут
вспомнить «CONS The
Magnificent» на стр. 25 книги *The
Little LISPer*.

Поясните, как работает эта
функция.

† *cons † t †* и соответствующая
закрывающая скобка *)* представляют
собой обертку для *(shape t|₀)*. Сравните
с оберткой *rank* во фрейме 25.

Правило о членах и элементах

У непустых списков есть члены списков, а у нескаляр-
ных тензоров есть элементы тензоров.

Функция *cons* получает два аргумента – значение *v* и список *l* – и возвращает новый список путем добавления

v
в начало списка
l

Например:

1. `(cons 3 (list 7 9))`
2. `(list 3 7 9)`

Теперь вы можете объяснить, как работает *shape*.

Правильно.

Что представляет собой
(shape 9)?

Что представляет собой
(shape [9 4 7 8 0 1])?

38 Если аргумент *t* функции *shape* является скаляром, то форма – просто пустой список.

В ином случае это не скалярный тензор, поэтому мы находим форму 0-го элемента, которая является списком, путем рекурсивного вызова *shape*

`(shape t0)`

Фактически форма – это результат операции `† t †`, помещенный в начало списка при помощи *cons*.

39 *(shape 9)* представляет собой пустой лист, потому что 9 – это скаляр.

40 *(shape [9 4 7 8 0 1])* представляет собой

`(cons 6 (shape 9))`

то есть

`(list 6)`

Правило об однородной форме

Все элементы тензора должны иметь одинаковую форму.

Используя последовательность аналогий, распишите

(shape t), где *t*

`[[[5] [6] [8]] [[7] [9] [5]]]`

из фрейма 31.

41 Последовательность выглядит так:

1. `(shape [[[5] [6] [8]] [[7] [9] [5]]])`
2. `(cons 2 (shape [[5] [6] [8]]))`
3. `(cons 2 (cons 3 (shape [5])))`
4. `(cons 2 (cons 3 (cons 1 (shape 5))))`
5. `(cons 2 (cons 3 (cons 1 (list))))`
6. `(cons 2 (cons 3 (list 1)))`
7. `(cons 2 (list 3 1))`
8. `(list 2 3 1)`

Количество членов списка *ls* 42
записывается как

$|ls|$

Как связаны *rank* и *shape*?

Так-так!

Количество элементов формы
тензора также является рангом
тензора[†]

$(= |(shape\ t)|\ (rank\ t))$

[†] Мы тоже можем использовать
форму тензора, чтобы определить
общее количество его скаляров, взяв
произведение членов его формы.

Закон ранга и формы

Ранг тензора равен длине его формы.

Осталось еще одно важное
определение, которое нужно
рассмотреть.

43 Жду с нетерпением.

Так выглядит наш итоговый
вариант определения *rank*, но
он не содержит обертку, т. е. в
нем нет обернутого рекурсив-
ного вызова!

```
(define rank
  (λ (t)
    (ranked t 0)))

(define ranked
  (λ (t a)
    (cond
      ((scalar? t) a)
      (else (ranked t|0
                    (add1 a))))))
```

Это определение *rank* содер-
жит вспомогательную функ-
цию *ranked*, которая включает
дополнительный формаль-
ный аргумент *a*, играющий
роль аккумулятора.

44 Вот что у меня получилось:

1. $(rank\ [[[8]\ [9]]\ [[4]\ [7]]])$
2. $(ranked\ [[[8]\ [9]]\ [[4]\ [7]]]\ 0)$
3. $(ranked\ [[8]\ [9]]\ (add1\ 0))$
4. $(ranked\ [[8]\ [9]]\ 1)$
5. $(ranked\ [8]\ (add1\ 1))$
6. $(ranked\ [8]\ 2)$
7. $(ranked\ 8\ (add1\ 2))$
8. $(ranked\ 8\ 3)$
9. 3

Вместо того чтобы помещать
в обертку рекурсивный вызов
rank

мы применяем
add1

к аккумулятору
a

переходя к
 $t|_0$

Чтобы понять, как это работает, повторите пример из фрейма 26, но используя итоговый вариант *rank*.

Все правильно.

45 Спасибо!

Такой способ использования tl_0 в качестве аргумента в развернутом рекурсивном вызове действительно удобен.

В отличие от варианта *rank* с пунктиром во фрейме 25, это новое определение рекурсивной функции не использует обертку.

46 Хотелось бы взглянуть на него.

На самом деле это настолько важно, что у нас есть специальный закон.

Закон простой передачи аккумулятора

В простом определении функции передачи аккумулятора каждый вызов рекурсивной функции разворачивается, и определение имеет не более одного аргумента, который не изменяется, аргумент, который меняется в случае истинного базового теста, и еще один, который аккумулирует результат.

Этот закон простой передачи аккумулятора важен.

47 Почему?

Этот закон позволяет нам работать с очень большими тензорами и списками.

48 Как?

При объединении простых определений функций передачи аккумулятора их можно рассматривать как один очень большой цикл.

49 Как это возможно?

Каждая система Scheme должна поддерживать оптимизацию хвостовых вызовов, благодаря которой каждый развернутый рекурсивный вызов ведет себя так же, как цикл.

50 Чем еще помогает этот закон?

Вытекающие из закона требования гарантируют, что мы используем единый шаблон для наших определений функций, чтобы их было легко читать и понимать.

51 Эти определения функции простой передачи аккумулятора, судя по всему, сулят замечательные возможности!

Настало время сделать перерыв!

52 Да, это именно то, что нужно. Мы очень интенсивно продвигались вперед.

Тензорные побрякушки

scalar? стр. 31

$[e \dots]$ (где $e \dots$ – элементы тензора) стр. 32

$\dagger t \dagger$ стр. 32

$t|_i$ стр. 34

shape стр. 37

cons стр. 38

$|ls|$ стр. 39

rank стр. 40

**Предлагаем постоять
в очереди за чем-нибудь
ВКУСНЫМ.**

**КАК НАСЧЕТ БАБЛ-ЧАЯ
С ПИРОЖНЫМИ?**

ИНТЕРЛЮДИЯ I. РАСШИРЯЕМ ФУНКЦИИ, УГЛУБЛЯЕМ ТЕНЗОРЫ



Как вам чай?	1	Пирожные с манговым джемом были великолепны!
А как насчет интерлюдии?	2	Что такое интерлюдия?
Это способ сместить фокус нашего внимания.	3	На что именно?
Мы немного поиграем со знаком «+».	4	Это операция сложения, верно?
Да. Сколько будет $(+ 1 1)$?	5	Это просто: $(+ 1 1)$ будет $2^\dagger$ ----- † Этот способ записи придумали Альфред Уайтхед (1861–1947) и Бертран Рассел (1872–1970).
Великолепно. А каким будет результат операции $(+ [2] [7])$?	6	Эта задача немного сложнее. Здесь аргументы типа tensor^1 . Наверное, $(+ [2] [7])$ – то самое, что $[9]$?
Да, но почему? Так выглядит логический столбец, из которого видно, почему получается $[9]$: 1. $(+ [2] [7])$ 2. $[(+ 2 7)]$ 3. $[9]$	7	Зачем здесь голубые скобки?
Когда встречается вызов функции, такой как «+» для тензоров, мы используем голубые скобки, чтобы подчеркнуть, что мы собираемся заглянуть внутрь этих тензоров и там найти возвращаемое значение.	8	Операция $(+ [5 6 7] [2 0 1])$ вернет нам $[7 6 8]$, верно?

Каков результат операции

(+ [5 6 7] [2 0 1])?

Вот логический столбец, который показывает, как получился этот результат:

1. | (+ [5 6 7] [2 0 1])
2. | [(+ 5 2) (+ 6 0) (+ 7 1)]
3. | [7 6 8]

⁹ Похоже, что «+» переходит к аргументам tensor^1 , чтобы получить другой tensor^1 .

Последний шаг приводит нас к tensor^1 , состоящему из значений трех сумм.

Правильно.

А вот задача посложнее:

(+ [[4 6 7] [2 0 1]]
[[1 2 2] [6 3 1]])?

¹⁰ Мы складываем два tensor^2 одинаковой формы следующим образом:

1. | (+ [[4 6 7] [2 0 1]]
[[1 2 2] [6 3 1]])
2. | [(+ 4 1) (+ 6 2) (+ 7 2)]
[(+ 2 6) (+ 0 3) (+ 1 1)]
3. | [[5 8 9] [8 3 2]]

Должны ли тензоры перед сложением иметь одинаковую форму?

Да, должны.

Использование «+» для работы с тензорами произвольного ранга называется расширением[†] «+».

Функции, построенные с использованием расширения, называются расширенными функциями. Можно ли аналогичным образом расширить другие функции, работающие со скалярами?

¹¹ Думаю, что можно.

В операции «+» нет ничего особенного. Другие скалярные функции должны расширяться аналогично.

[†] Также называется точечным расширением (pointwise extension).

Рассмотрим еще один способ применить расширение. Каким будет результат операции

(+ 4 [3 6 5])?

Верно.

Когда мы такое встречаем, то поступаем следующим образом:

1. $\left| \begin{array}{l} (+\ 4\ [3\ 6\ 5]) \end{array} \right|$
2. $\left| \begin{array}{l} [(+\ 4\ 3)\ (+\ 4\ 6)\ (+\ 4\ 5)] \end{array} \right|$

Дозаполните логический столбец.

Замечательно.

Сможете продолжить логический столбец?

1. $\left| \begin{array}{l} (+\ [6\ 9\ 1]\ [[4\ 3\ 8]\ [7\ 4\ 7]]) \end{array} \right|$

Превосходно.

Рассмотрим теперь расширенную версию умножения «*».

Это так называемое произведение Адамара[†].

¹² Но у этих тензоров разная форма!

¹³ Ого, оказывается, мы проникаем внутрь аргумента `tensor`¹ и прибавляем 4 к каждому элементу.

Окончательный ответ:

3. $\left| \begin{array}{l} [7\ 10\ 9] \end{array} \right|$

¹⁴ Мы можем заглянуть внутрь аргумента `tensor`² и прибавить аргумент `tensor`¹ к каждому элементу, как мы только что делали:

2. $\left| \begin{array}{l} [(+\ [6\ 9\ 1]\ [4\ 3\ 8]) \\ \quad (+\ [6\ 9\ 1]\ [7\ 4\ 7])] \end{array} \right|$
3. $\left| \begin{array}{l} [[(+\ 6\ 4)\ (+\ 9\ 3)\ (+\ 1\ 8)] \\ \quad [(+\ 6\ 7)\ (+\ 9\ 4)\ (+\ 1\ 7)]] \end{array} \right|$
4. $\left| \begin{array}{l} [[10\ 12\ 9] \\ \quad [13\ 13\ 8]] \end{array} \right|$

¹⁵ Вот необходимые шаги:

2. $\left| \begin{array}{l} [(*\ [4\ 6\ 5]\ 3)\ (*\ [6\ 9\ 7]\ 3)] \end{array} \right|$
3. $\left| \begin{array}{l} [[(*\ 4\ 3)\ (*\ 6\ 3)\ (*\ 5\ 3)] \\ \quad [(*\ 6\ 3)\ (*\ 9\ 3)\ (*\ 7\ 3)]] \end{array} \right|$
4. $\left| \begin{array}{l} [[12\ 18\ 15] \\ \quad [18\ 27\ 21]] \end{array} \right|$

Продолжите логический столбец:

1. $\left| \begin{array}{l} (* \text{ } [[4 \text{ } 6 \text{ } 5] \text{ } [6 \text{ } 9 \text{ } 7]] \text{ } 3) \end{array} \right.$

[†] Спасибо, Жак-Саломон Адамар (1865–1963).

В первых двух шагах используются голубые скобки, потому что мы спускаемся к этим `tensor`¹ам. Последние два шага не имеют голубых скобок, потому что у нас остались только скаляры, поэтому мы просто умножаем их.

Рассмотрим еще одну расширенную функцию

$\text{sqrt}^{\dagger}$

1. $\left| \begin{array}{l} (\text{sqrt } 9) \\ 3 \end{array} \right.$

[†] Функция `sqrt` всегда возвращает неотрицательное значение квадратного корня числа.

¹⁶ Применяв функцию `sqrt` к скаляру 9, мы получаем его квадратный корень, т. е. 3.

Теперь вызовем функцию `sqrt` для тензора

`[9 16 25]`

На этот раз ранг аргумента равен 1. Поэтому `sqrt` просто спускается внутрь `tensor`¹:

1. $\left| \begin{array}{l} (\text{sqrt } [9 \text{ } 16 \text{ } 25]) \\ [(\text{sqrt } 9) (\text{sqrt } 16) (\text{sqrt } 25)] \\ [3 \text{ } 4 \text{ } 5] \end{array} \right.$

¹⁷ Функция `sqrt` была вызвана для каждого скаляра в `tensor`¹.

В этом логическом столбце тензор снова спускается внутрь тензора, обозначенного голубыми скобками.

¹⁸ Можете привести еще один пример?

Конечно.

Вот пример того, как *sqrt* ведет себя с *tensor*²:

1. $\left| \begin{array}{l} (\text{sqrt} \text{ [[49 81 16] [64 25 36]])} \end{array} \right.$
2. $\left| \begin{array}{l} [(\text{sqrt} \text{ [49 81 16]}) \\ (\text{sqrt} \text{ [64 25 36]})] \end{array} \right.$
3. $\left| \begin{array}{l} [(\text{sqrt} \text{ 49}) (\text{sqrt} \text{ 81}) (\text{sqrt} \text{ 16})] \\ [(\text{sqrt} \text{ 64}) (\text{sqrt} \text{ 25}) (\text{sqrt} \text{ 36})] \end{array} \right.$
4. $\left| \begin{array}{l} [[7 \text{ 9} \text{ 4}] \\ [8 \text{ 5} \text{ 6}]] \end{array} \right.$

¹⁹ Видно, что *sqrt* спускается внутрь каждого *tensor*¹ в аргументе, пока не встретит *tensor*⁰, и извлекает из него квадратный корень.

Да, мы будем часто использовать этот прием спуска внутрь тензора.

В случае тензоров высокого ранга процесс повторяется до тех пор, пока не встретятся скаляры, затем из них извлекают квадратный корень.

²⁰ Звучит логично.
Можем ли мы аналогичным образом применить функцию двух аргументов?

Да, можем.

Теперь мы знаем, как можно заставить работать расширенные функции двух аргументов, когда два аргумента имеют разные ранги, – путем спуска в тензор с более высоким рангом.

²¹ Расширенные функции всегда спускаются вглубь, пока не найдут скаляры?

Некоторые функции так не делают.

Возьмем, к примеру, новую функцию

*sum*¹

Поведение, которое мы от нее ожидаем:

1. $\left| \begin{array}{l} (\text{sum}^1 \text{ [10.0 12.0 14.0]}) \end{array} \right.$
2. $\left| \begin{array}{l} 36.0 \end{array} \right.$

²² Очевидно, эта функция суммирует скаляры в *tensor*¹.
Но что означает верхний индекс 1 после *sum*?

Верхний индекс служит напоминанием, что функция ожидает $\text{tensor}^{\dagger}$.

Теперь, когда мы знаем, что sum^1 всегда получает tensor^1 , давайте определим ее.

 $\dagger$ Мы используем аналогичное соглашение о верхнем индексе и для других функций.

Так выглядит определение sum^1 :

```
(define sum1
  (λ (t)
    (summed t (sub1 ‡ t ‡) 0.0)))

(define summed
  (λ ((t i a)†
    (cond
      ((zero? i) (+ t0 a))
      (else
       (summed t (sub1 i)
                (+ ti a))))))
```

Объясните, как она работает.

 $\dagger$ Это способ размещения формальных аргументов для простого определения функции передачи аккумулятора. Если есть аргумент, который не изменяется, как t , он будет первым формальным аргументом; тот, что накапливается, как a , будет последним; а тот, который изменяется в сторону истинного значения основного теста, например i , находится слева от аккумулятора. См., например, функцию *rank* на стр. 40.

23 Звучит любопытно.

24 Мы вызываем вспомогательную функцию *summed* для тензора

t

последнего индекса в t

$(\text{sub1 } \ddagger t \ddagger)$

и аккумулятора с начальным значением

0.0

Функция *summed* считает индекс от i до нуля. На каждом шаге она прибавляет i -й элемент тензора t к аккумулятору и рекурсивно вызывает *summed*.

Наконец, когда функция достигает 0-го элемента тензора, она прибавляет этот элемент к аккумулятору и возвращает значение суммы всех элементов.

Мы обозначили расширенную версию sum^1 как sum , которая спускается по своему аргументу до тех пор, пока не найдет $tensor^1$ вместо $tensor^0$.

25 Как работает функция sum ?

Вот пример работы sum с аргументом $tensor^3$:

26 Это $tensor^2$, поэтому его ранг равен 2, то есть на 1 меньше, чем ранг входных данных, что также верно для sum^1 .

1. $(sum$
 $[[[1\ 2]\ [3\ 4]]\ [[5\ 6]\ [7\ 8]]])$
2. $[(sum\ [[1\ 2]\ [3\ 4]])$
 $(sum\ [[5\ 6]\ [7\ 8]])]$
3. $[[(sum^1\ [1\ 2])\ (sum^1\ [3\ 4])]$
 $[(sum^1\ [5\ 6])\ (sum^1\ [7\ 8])]]$
4. $[[3\ 7]$
 $[11\ 15]]$

Что мы можем сказать о ранге этого результата?

Закон суммы

Для тензора t с рангом $r > 0$ ранг результата $(sum\ t)$ равен $r - 1$.

Так и есть.

Это полезное свойство sum , которым мы воспользуемся позже.

27 Что можно сказать о функциях, которые составлены из этих расширенных функций?

Отличный вопрос.

Если мы используем расширенные функции для создания новых функций, эти новые функции автоматически становятся расширенными и работают соответствующим образом.

28 Можете привести пример такой функции?

Конечно.

Вот пример, который вы просили.

Каким будет результат выполнения функции

$((line\ [2\ 7\ 5\ 11])\ (list\ 4\ 6))$?

Да, являются.

Мы хотим найти значение y для нескольких (здесь четыре) значений x , где

$\theta_0 = 4$ и $\theta_1 = 6$.

Теперь вы можете завершить логический столбец для определения четырех предполагаемых значений y :

1. $| ((line\ [2\ 7\ 5\ 11])\ (list\ 4\ 6))$

29 Мы вызываем *line* с аргументом *tensor*¹.

Означает ли это, что функции $+$ и $*$ в составе *line* являются расширенными?

30 Великолепно!

31 Вот остальная часть:

2. $| ((\lambda\ (\theta)$
 $| \quad (+\ (*\ \theta_0\ [2\ 7\ 5\ 11])\ \theta_1))$
 $| \quad (list\ 4\ 6))$
3. $| (+\ (*\ 4\ [2\ 7\ 5\ 11])\ 6)$
4. $| (+\ [(^*\ 4\ 2)\ (^*\ 4\ 7)\ (^*\ 4\ 5)\ (^*\ 4\ 11)]\ 6)$
5. $| (+\ [8\ 28\ 20\ 44]\ 6)$
6. $| [(+ 8\ 6)\ (+ 28\ 6)\ (+ 20\ 6)\ (+ 44\ 6)]$
7. $| [14\ 34\ 26\ 50]$

Эта интерлюдия посвящена тому, как можно расширить функции, работающие с тензорами фиксированного ранга, до функций, которые также принимают тензоры разных рангов. Для этого мы используем концепцию спуска внутрь тензоров более высокого ранга.

32 И мы узнали, что, делая это, мы спускаемся в тензор высокого ранга, переходя в тензоры с более низким рангом, пока не сможем выполнить операцию с одним или двумя аргументами.

Растягивающиеся побрякушки

sum стр. 50

Как насчет чашечки капучино?

С ФРАНЦУЗСКИМИ ПОНЧИКАМИ
БЕНЬЕ?

3. КУДА НАС ЛИНИЯ ВЕДЕТ?



Как вам капучино?

1. Великолепно сочетается с бенье[†].

[†] Рекомендуем посетить *Café du Monde* в Новом Орлеане.

Давайте вернемся к определению функции автоматического нахождения набора параметров θ (из фрейма 26:29), наиболее подходящего к набору данных из фрейма 23:17. Пока мы будем называть такой θ подогнанным.

2. Так же, как мы визуальное подогнали линию к точкам во фрейме 24:20?

Совершенно верно.

Далее мы обсудим метод последовательных приближений[†].

[†] Предложил Джозеф Рафсон (1648–1715).

3. Метод последовательных приближений – это способ получить подобранный θ ?

Да.

Мы определяем θ_0 и θ_1 , произвольно начиная их с 0.0. Затем мы неоднократно корректируем θ , чтобы максимально приблизить его к тому, каким он должен быть «на самом деле»[†].

[†] Последовательное приближение – это семейство математических методов. Возможно, вы знакомы с методом определения корней многочлена, такого как квадратный корень. Авторами этого метода являются Исаак Ньютон (1643–1727) и Джозеф Рафсон.

4. Покажите это на примере, пожалуйста.

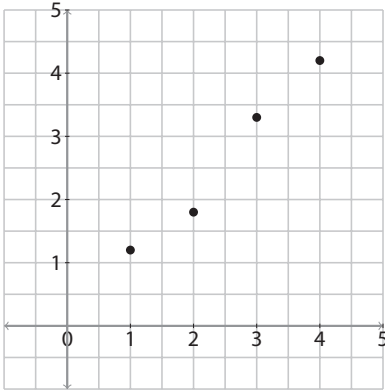
Охотно.

Возьмем набор данных из фрейма 23:17

`line-xs = [2.0 1.0 4.0 3.0]`

и

`line-ys = [1.8 1.2 4.2 3.3]`



5. То есть мы начинаем с

$$\theta_0 = 0.0$$

и

$$\theta_1 = 0.0?$$

Да, это хорошая отправная точка.

Какая линия у нас получается при таком наборе параметров θ ?

6. Это можно выяснить, используя `(list 0.0 0.0)` как θ

1. `((line line-xs)  $\theta$ )`
2. `((line [2.0 1.0 4.0 3.0]) (list 0.0 0.0))`
3. `(+ (* 0.0 [2.0 1.0 4.0 3.0]) 0.0)`
4. `[0.0 0.0 0.0 0.0]`

Все координаты y , полученные из координат x в `line-xs`, равны 0.0. Значит, эти параметры θ представляют ось x !

Так и есть.

7. Что это значит?

Эти значения y , которые во фрейме 25:23 мы называли предсказанными, очень сильно отличаются от реальных значений в `line-ys`!

Это значит, что параметры θ_0 и θ_1 пока не подходят для нашего набора данных.

8 Будем ли мы изменять значения θ_0 и θ_1 , чтобы привести предсказанные значения y в лучшее соответствие с реальными значениями *line-ys*?

Да, будем.

9 Почему скаляр, а не *tensor*¹?

Но для этого нам нужно знать, насколько мы далеки от качественно подобранных параметров θ .

Нам нужен единственный скаляр, который говорит нам, как близко или как далеко мы находимся от идеала. Этот скаляр называется потерей (*loss*)[†]. Хорошо подобранным набором параметров будет тот, при котором эта потеря максимально близка к 0.0.

Если потеря равна 0.0 – это идеальное соответствие данным.

[†] Также называется стоимостью (*cost*).

Используя скаляр, нам проще решить, как изменить θ .

10 Но как использовать этот скаляр?

Мы находим этот скаляр (потерю) каждый раз, когда меняем θ . Так как потеря показывает нам, насколько мы далеки от идеально выбранного θ , мы используем ее как руководство для изменения θ .

11 Как нам найти потерю для текущего θ ?

Разумеется, при помощи функции, которую мы определим!

Но мы сделаем это за два шага. Сначала мы определим тензор, показывающий, насколько мы далеки от идеала, а затем понизим его ранг до скаляра.

Простейший способ найти потерю – определить разницу между заданными *line-us* и предсказанными *us*, как показано в каркасном коде:

```
(- line-us P )
```

Найдите *P*.

Правильно.

Мы получаем

```
(- line-us ((line line-xs)
            (list  $\theta_0$   $\theta_1$ )))
```

Но это пока не скаляр, а *tensor*¹, поэтому мы должны превратить его в скаляр.

Какая функция из тех, что мы уже определили ранее, превращает *tensor*¹ в скаляр?

¹² Думаю, что *P* – это прогнозируемые *us*. Значит, вместо *P* нужно подставить

```
((line line-xs) (list  $\theta_0$   $\theta_1$ ))
```

¹³ Это функция *sum* из фрейма 50:25.

Достаточно ли следующего кода

```
(sum
  (- line-us ((line line-xs)
              (list  $\theta_0$   $\theta_1$ ))))
```

чтобы получить скаляр?

Почти достаточно, но это еще не все.

И вот почему. Допустим, у нас есть другой набор данных и другие параметры θ , которые дают нам разности

```
[4.0 -3.0 0.0 -4.0 3.0]
```

Что получится в результате вызова функции

```
(sum
 [4.0 -3.0 0.0 -4.0 3.0])?
```

Мы решим эту проблему возведением в квадрат каждого элемента[†]. Эта операция превратит все отрицательные значения в положительные, а положительные сохранят свой знак:

```
(sum
 (sqr
  (- line-ys ((line line-xs)
               (list  $\theta_0$   $\theta_1$ ))))))
```

Здесь расширенная функция «-» спускается в два `tensor1` разностей, затем `sqr` возводит в квадрат результирующий `tensor1` разностей, передавая `tensor1` в `sum` для получения скаляра.

Теперь сумма квадратов положительная, если хотя бы один элемент разности не равен нулю.

14 Функция вернет 0.0 – идеальную потерю, – хотя очевидно, что это не так!

Выходит, даже если существуют отдельные большие расхождения, значение `sum`, равное 0.0, говорит нам, что параметры θ подобраны идеально.

Так-так! Эта проблема возникла из-за того, что аргумент `sum` содержит отрицательные значения.

Что нам делать?

15 Многообещающая стратегия. Но все же как будет выглядеть наша функция потерь?

[†] Другие функции, такие как `abs` (абсолютное значение) из фрейма 9:24, тоже сработают, но мы используем только возведение в квадрат, потому что оно работает лучше в большинстве случаев.

Так выглядит функция, кото-
рая нам нужна:

```
(define l2-loss
  (λ (xs ys)
    (λ (θ)
      (let ((pred-ys† ((line xs) θ)))
        (sum
          (sqr
            (− ys pred-ys))))))))
```

[†] Предсказанные значения *ys* обозна-
чены как *pred-ys*.

Она относится к семейству
функций, использующих воз-
ведение в степень для нахож-
дения потери. Поскольку мы
используем *sqr* (возведение
в степень 2), эта функция
потерь (*loss*) содержит обозна-
чение *l2[†]*.

[†] В математике квадратный корень
из суммы квадратов *tensor*¹ назы-
вается нормой L2, или евклидо-
вым расстоянием в честь Евклида
Александрийского (325–270 гг.
до н. э.).

¹⁶ Почему эта функция называ-
ется *l2-loss*?

¹⁷ А почему определение с
пунктирной линией?

Но при вычислении функции потерь
квадратный корень обычно не при-
меняют. Другой вариант значения
потерь получается делением суммы
квадратов на количество элементов
тензора и называется среднеквад-
ратичной ошибкой (mean squared error,
MSE).

Потому что дальше мы
получим обобщенную функ-
цию *l2-loss*.

Эта *l2-loss* зависит от функции
line. Но с точки зрения *l2-loss*,
line – это просто некоторая
функция, которая возвращает
pred-ys, когда заданы *xs* и *θ*.

¹⁸ Как можно использовать это
наблюдение?

Мы можем сделать *l2-loss* менее зависимой от *line*, реализовав возможность подставлять любую другую функцию. Давайте переопределим *l2-loss* так, чтобы она получала *line* как аргумент:

```
(define l2-loss
  (λ (line)
    (λ (xs ys)
      (λ (θ)
        (let ((pred-ys ((line xs) θ)))
          (sum
            (sqr
              (- ys pred-ys))))))))))
```

¹⁹ Но почему новая версия *l2-loss* помечена пунктиром?

Мы оставили пунктир, потому что имя функции *line* слишком конкретное.

Поскольку мы вынесли *line* в формальный аргумент, это имя больше не представляет конкретную функцию *line*. Поэтому мы должны заменить имя *line* на более осмысленное.

²⁰ На каком имени остановимся?

В наших предыдущих вычислениях *line* играла роль обучающей функции (*target*), потому что именно для этой функции мы стремимся найти наилучшие параметры θ .

Значит, вместо имени *line* мы будем использовать имя *target*[†].

²¹ Да, так понятнее.

[†] Этот процесс, вскользь упомянутый во фрейме 7:18, предложил Алонзо Чёрч. Он называется α -подстановкой и указывает, как могут быть правильно переименованы аргументы λ -выражения.

Так выглядит итоговый вариант функции *l2-loss* с использованием имени *target* вместо *line*:

22 Ура!

```
(define l2-loss
  (λ (target)
    (λ (xs ys)
      (λ (θ)
        (let ((pred-ys ((target xs) θ)))
          (sum
            (sqr
              (- ys pred-ys))))))))
```

Мы записали в более общем виде функцию *l2-loss*, показанную ранее во фрейме 19. Если мы снова захотим использовать функцию *line*, как передать ее в обобщенную *l2-loss*?

23 Мы должны вызвать *l2-loss* с аргументом *line*:

(l2-loss line)

Правильно.

Ниже показан логический столбец для такого вызова:

24 Почему мы называем

(λ (xs ys) ...)

ожидающей функцией?

```
1. | (l2-loss line)
2. | (λ (xs ys)
    |   (λ (θ)
    |     (let ((pred-ys ((line xs) θ)))
    |       (sum
    |         (sqr
    |           (- ys pred-ys))))))
```

Эта функция, которая создается, когда *l2-loss* вызывается с целевой функцией, называется *ожидающей функцией* (*expectant function*).

Потому что она ожидает набор данных в качестве аргумента.

25 Разумно.

Что делает ожидающая функция, когда получает набор данных?

Рассмотрим следующую функцию:

26 Что делает целевая функция?

```
1. ((l2-loss line) line-xs line-ys)
2. (λ (θ)
   (let ((pred-ys ((line line-xs) θ)))
     (sum
      (sqr
       (- line-ys pred-ys))))))
```

Функция, которая ожидает θ в качестве аргумента, называется *целевой функцией*[†] (*objective function*).

[†] Здесь может возникнуть путаница, потому что «objective function» и «target function» очень часто переводят одинаково – «целевая функция». Но в данной книге *target function* – это функция, которую мы ищем при обучении модели (т. е. обучаемая функция), а назначение *objective function* – оценить, насколько мы близки к этой цели при текущих параметрах, и показать, в какую сторону двигаться дальше. – Прим. перев.

Получив θ , целевая функция возвращает скаляр, представляющий потери, которые служат мерой того, насколько мы далеки от качественно подобранных параметров θ .

27 Тем самым она помогает нам найти правильно подобранные параметры θ !

Давайте попробуем эти функции в деле с нашими текущими θ_1 и θ_2 , которые по-прежнему равны 0.0.

28 В этом случае значением аргумента *target* будет *line*, поэтому *pred-ys* можно найти так:

```
((l2-loss line) line-xs line-ys)
(list 0.0 0.0))
```

Согласно финальной версии *l2-loss* во фрейме 22, мы должны сначала определить *pred-ys*

```
((target xs) θ)
```

Воспользуйтесь идеей из логического столбца во фрейме 55:31, чтобы найти *pred-ys* для текущих *line-xs* и θ .

```
1. ((target xs) θ)
2. ((line line-xs) θ)
3. ((line [2.0 1.0 4.0 3.0])
   (list 0.0 0.0))
4. (+ (* 0.0 [2.0 1.0 4.0 3.0]) 0.0)
5. [0.0 0.0 0.0 0.0]
```

Отлично.

Располагая значением *pred-ys*, найдем потерю для текущих параметров θ :

1. `((l2-loss target) xs ys)  $\theta$ )`
2. `((l2-loss line) line-xs line-ys) (list 0.0 0.0))`
3. `((l2-loss line) [2.0 1.0 4.0 3.0] [1.8 1.2 4.2 3.3]) (list 0.0 0.0))`
4. `(sum (sqr (- [1.8 1.2 4.2 3.3] pred-ys)))`

Завершите эту логическую цепочку.

²⁹ Вот завершение:

5. `(sum (sqr (- [1.8 1.2 4.2 3.3] [0.0 0.0 0.0 0.0])))`
6. `(sum (sqr [1.8 1.2 4.2 3.3]))`
7. `(sum [(sqr 1.8) (sqr 1.2) (sqr 4.2) (sqr 3.3)])`
8. `(sum [3.24 1.44 17.64 10.89])`
9. `33.21`

Потеря говорит нам, как далеко мы находимся, а значение 33.21 говорит о том, что мы довольно далеко.

Как нам изменить θ , чтобы приблизить скалярную потерю 33.21 к идеальной потере 0.0?

Мы начнем с изменения θ_0 , чтобы посмотреть, как это повлияет на потерю; параметром θ_1 мы займемся позже.

Изменим θ_0 на небольшую произвольную величину, например

$$\theta_0 = 0.0099$$

Найдите значения *pred-ys* для нового θ .

³⁰ Хорошо.

1. `((target xs)  $\theta$ )`
2. `((line line-xs)  $\theta$ )`
3. `((line [2.0 1.0 4.0 3.0]) (list 0.0099 0.0))`
4. `(+ (* 0.0099 [2.0 1.0 4.0 3.0]) 0.0)`
5. `[0.0198 0.0099 0.0396 0.0297†]`

[†] Язык Scheme иногда возвращает значения наподобие 0.029700000000000004 по причине особенностей работы с числами с плавающей запятой. В логических столбцах мы для наглядности округляем такие значения.

Отлично.

Ниже показан логический столбец нахождения новой потери для обновленного значения θ_0 :

1. $((((12 - \text{loss line}) \text{ line-xs line-ys})$
 $(\text{list } 0.0099 \ 0.0)))$
2. $((((12 - \text{loss line})$
 $[2.0 \ 1.0 \ 4.0 \ 3.0]$
 $[1.8 \ 1.2 \ 4.2 \ 3.3])$
 $(\text{list } 0.0099 \ 0.0)))$

Завершите логическую цепочку.

Другими словами, мы стали немного ближе к идеальной потере!

Потеря изменилась на

$$(32.59 - 33.21) = -0.62$$

за счет изменения θ_0 с 0.0 до 0.0099.

Наша попытка оказалась успешной.

Возможно.

Но изменение θ_0 каждый раз на 0.0099 может потребовать слишком много шагов, пока мы доберемся до правильных параметров.

Однако есть способ, который требует меньше правок.

Напомним, что увеличение θ_0 на 0.0099 привело к изменению потерь на -0.62 . Значит, скорость изменения[†] равна

$$\frac{0.62}{0.0099} = -62.63$$

31 Вот завершение:

3. $(\text{sum}$
 $(\text{sqr}$
 $(- [1.8 \ 1.2 \ 4.2 \ 3.3] \text{pred-ys})))$
4. $(\text{sum}$
 $(\text{sqr}$
 $(- [1.8 \ 1.2 \ 4.2 \ 3.3]$
 $[0.0198 \ 0.0099 \ 0.0396$
 $0.0297])))$
5. 32.59

Потеря уменьшилась.

32 Означает ли это, что мы должны продолжить изменять θ_0 с шагом 0.0099, пока не приблизимся к идеальной потере, насколько возможно?

33 Покажите же его!

34 Как нам пригодится скорость изменения?

[†] Это понятие ввели Готтфрид Лейбниц (1646–1716) и Исаак Ньютон. Скорость изменения также называется производной.

Скорость изменения функции (здесь – целевой функции) определяет, как изменится ее результат при изменении ее аргумента (т. е. θ).

Осторожно используя скорость изменений, мы можем получить хороший результат с меньшим количеством исправлений.

35 Что в данном случае означает «осторожно»?

Эта скорость изменения имеет большое абсолютное значение[†]. Это означает, что небольшое увеличение θ_0 вызывает относительно большое уменьшение потерь.

Мы можем воспользоваться этим фактом, чтобы определить, насколько еще нужно изменить θ_0 , чтобы еще больше сократить потери. Но мы должны быть осторожны.

36 Чего нам следует опасаться?

[†] Абсолютное значение скаляра, определенное как функция во фрейме 7:24, представляет собой величину без знака. Например, абсолютным значением чисел -62.63 и 62.63 будет 62.63 .

Мы должны следить за тем, чтобы изменения θ_0 всегда приближали нас к идеальной потере, но не позволяли проскочить ее.

Один из вариантов, например, состоит в том, чтобы изменить θ_0 на 62.63 , что является абсолютным значением всей скорости изменения. Но если мы это сделаем, то получим потерю в размере $113\,763.027$.

Это намного больше, чем предыдущее значение потери 32.59 , и намного дальше от идеальной потери.

37 Этого действительно следует опасаться.
Как решить эту проблему?

Мы возьмем небольшой скаляр (около 0.01), умножим на него скорость изменения и изменим θ_0 на полученную величину.

38 Следовательно, мы получим
 $0.01 \times -62.63 = -0.6263$
 Будет ли это достаточно небольшим изменением, чтобы не проскочить идеальную потерю?

Да, скорее всего.

39 Хорошо.

Эта небольшая скалярная величина называется *скоростью обучения* (*learning rate*).

Мы будем часто использовать этот скаляр, поэтому присвоим ему специальное обозначение $\alpha^\dagger$.

Перепишем наш предыдущий пример:

$$\alpha \times -62.63 = -0.6263$$

 $\dagger$ Скорость обучения (обычно небольшое число между 0.0 и 0.01) также называют шагом обучения. В следующих главах мы обсудим выбор подходящей скорости обучения.

 Шаг обучения иногда обозначают символом λ , но мы уже используем этот символ для иных целей, поэтому придется взять другую греческую букву.

Так как нам нужно увеличивать θ_0 для уменьшения потери, мы будем вычитать отрицательное значение из текущего θ_0 . Значит, новое значение параметра θ_0 будет равно

40 Но почему мы не оставили значение 0.0099?

$$0.0 - -0.6263 = \underline{0.6263}.$$

Просто потому, что нашли скорость изменения.

41 Ладно.

После того как она стала известна, можно забыть про 0.0099.

Итак, мы обновляем θ_0 , перемножая скорость обучения и скорость изменения, а затем вычитая результат из нашего текущего θ_0 ?

Закон обновления (начальная версия)

Новый $\theta_0 = \theta_0 - (\alpha \times \text{скорость изменения потери с учетом } \theta_0)$

Конечно!

Поступим, как прежде, и найдем новые *pred-ys* для нового значения θ_0 :

1. `((line line-xs)  $\theta$ )`
2. `((line [2.0 1.0 4.0 3.0])
(list 0.6263 0.0))`
3. `(+ (* 0.6263 [2.0 1.0 4.0 3.0]) 0.0)`
4. `[1.2526 0.6263 2.5052 1.879]`

Теперь найдите потерю.

42 Чтобы найти потерю, будем использовать новые *pred-ys*:

1. `((l2-loss line) line-xs line-ys)
(list 0.6263 0.0))`
2. `(sum
(sqr
(- line-ys pred-ys)))`
3. `(sum
(sqr
(- [1.8 1.2 4.2 3.3]
[1.2526 0.6263 2.5052
1.879])))`
4. `5.52`

Мы уменьшили потерю с 33.21 до 5.52. Это намного лучше, чем когда мы просто увеличили θ_0 на 0.0099 во фрейме 30!

Невероятно. Мы сделали огромный шаг к идеальной потере.

Повторим этот процесс, чтобы получить следующее обновление.

43 Можем ли мы снова вычесть -0.6263 из θ_0 ?

44 Это значение отличается от скорости, которую мы нашли во фрейме 34.

Означает ли это, что скорость изменения зависит от текущего θ_0 ?

щего θ_0 ?

щего θ_0 ?

щего θ_0 ?

1. $(\div (- (obj (list (+ 0.6263 0.0099) 0.0)) (obj (list 0.6263 0.0))) 0.0099)$
2. -25.12

45 Значит, нам нужно определить скорость изменения заново, используя $\theta_0 + 0.0099$?

⁴⁶ Звучит інтригующе.

Что это за способ?

47 Какие вкусняшки посоветуете, чтобы лучше подготовиться к следующей главе?

l2-loss стр. 59

НЕ ЗАБУДЬТЕ ТЩАТЕЛЬНО УБРАТЬ
КРОШКИ!

4. СКОЛЬЗИМ, СКОЛЬЗИМ НЕУДЕРЖИМО!¹



¹ В оригинале глава названа по мотивам песни Поля Фредерика Симона *Slip Slidin' Away*.

Как вам печенье?

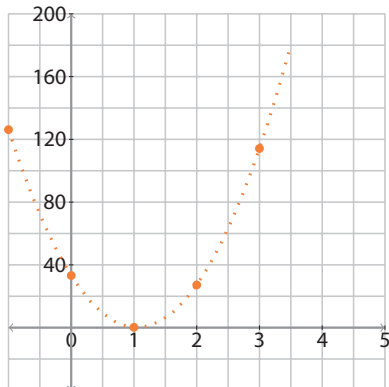
¹ Просто тает во рту.

Готовы продолжить?

Но давайте продолжим!

Мы начнем с такого графика:

² Он заметно отличается от предыдущего. Что это?



Это график потерь в нашем примере относительно θ_0 при фиксированном значении θ_1 , равном 0.0. По оси y здесь представлены потери, а по оси x – значения θ_0 , которые мы также называем весом.

³ Понятно. Для любого возможного значения веса этот график показывает соответствующее значение потери. Что означают большие оранжевые точки?

Для построения этого графика мы выбрали пять весов: -1.0 , 0.0 , 1.0 , 2.0 и 3.0 .

⁴ Как мы можем найти эти пять потерь?

Для каждого веса мы нашли соответствующую потерю, сохраняя фиксированное значение $\theta_1 = 0.0$ и используя набор данных

(line-xs, line-ys)

Оранжевые точки показывают потери при каждом из этих весов.

Для начала присмотримся более внимательно к нашей целевой функции

$((l2-loss\ line)\ line\text{-}xs\ line\text{-}ys)$

Составьте для нее логический столбец.

⁵ Вот он:

```
1. ((l2-loss line) line-xs line-ys)
2. ((l2-loss line) [2.0 1.0 4.0 3.0]
   [1.8 1.2 4.2 3.3])
3. (λ (θ)
   (let ((pred-ys
         (line [2.0 1.0 4.0 3.0]) θ)))
     (sum
      (sqr
       (- [1.8 1.2 4.2 3.3]
          pred-ys))))))
```

Отлично.

Присвоим этому λ -выражению имя *obj* (от objective function) из фрейма 62:26.

Поясните, как мы можем использовать *obj* для определения потерь.

⁶ Данное λ -выражение получает параметры линии как аргументы и возвращает скаляр, показывающий, насколько близко эта линия соответствует точкам данных (т. е. какова потеря).

Значит, для каждого из пяти весов мы должны составить набор параметров θ , состоящий из θ_0 и $\theta_1 = 0.0$.

Соответствующие потери будут такими:

```
(obj (list -1.0 0.0)), → 126.21
(obj (list 0.0 0.0)), → 33.21
(obj (list 1.0 0.0)), → 0.21
(obj (list 2.0 0.0)), → 27.21
(obj (list 3.0 0.0)), → 114.21
```

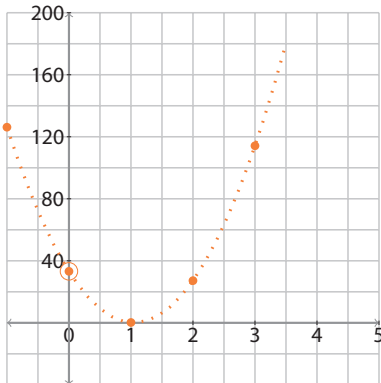
Верно.

Мы используем оранжевые точки, чтобы показать значения, которые не являются частью нашего набора данных.

Какое у нас начальное предпологаемое значение θ_0 ?

⁷ Это 0.0.

Из фрейма 63:29 мы знаем, что когда θ_0 равен 0.0 (и θ_1 сохраняет начальное значение 0.0), потеря равна 33.21. Точка (0.0, 33.21) обведена кружком на следующем графике:



8 Точно!

Судя по всему, потеря наименьшая в нижней точке этой кривой[†].

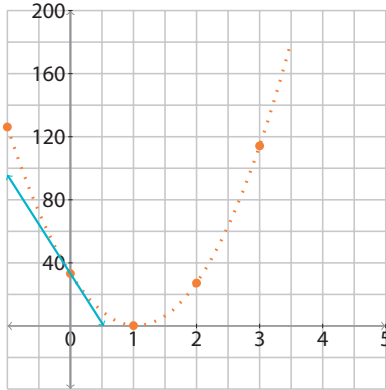
[†] Нижняя часть этой кривой визуально соответствует значению 0.0 по оси y , но на практике так бывает редко. В зависимости от набора данных нижняя точка обычно больше, чем 0.0.

Правильно.

Нам нужно «скатиться» вниз по этому склону, чтобы добраться до его основания, и во фрейме 65:37 мы установили, что можем использовать скорость изменения, чтобы скатываться вниз быстрее, чем раньше, но не проскакивая точку минимума.

9 Как можно отобразить скорость изменения на этом графике?

На следующем графике показано, как мы это делаем при начальной оценке $\theta_0 = 0.0$:



10 Что представляет собой бирюзовая линия?

Бирюзовая линия отличается от линии, для которой мы пытаемся подобрать параметры θ_0 и θ_1 . Эта линия называется касательной.

11 Это точка с координатами (0.0, 33.21)?

Касательная соприкасается с кривой потерь только в одной точке.

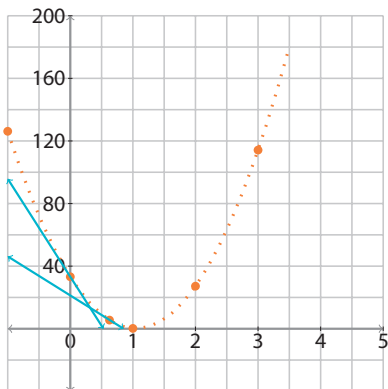
Да, так и есть.

12 Можем ли мы построить касательную для нашего следующего значения θ_0 , равного 0.6623?

Скорость изменения, которую мы определили, представляет собой наклон касательной.

Этот наклон имеет собственное имя, поэтому мы не спутаем его с θ_0 . Он называется градиент.

Ниже показаны касательные для первых двух оценок θ_0 :



Как вы думаете, на что здесь нужно обратить внимание?

Хорошая работа.

Градиент является универсальным способом описания скорости изменения параметризованной функции для всех ее параметров.

Чтобы найти градиент функции при заданных значениях ее аргументов, нам нужно использовать новую функцию ∇ . Первый аргумент ∇ – это функция f , принимающая список тензоров, включая θ_s . Второй аргумент ∇ – это список, для которого нам нужны градиенты f .

13 Мы видим, что касательная ко второму θ_0 более пологая, чем к первому. Следовательно, градиент во второй точке меньше, чем в первой.

Мы рассмотрели взаимосвязь θ_0 и градиента, но у нас еще есть θ_1 .

Относится ли все вышесказанное к θ_1 ?

14 Как мы можем найти этот градиент?

15 Эта функция возвращает результат в виде градиента?

Да[†].

Результатом ∇ является список градиентов f по каждому параметру в θ ; он так и называется – списком градиентов.

Итак, например, если нам нужно найти градиент f для параметров u и v , мы записываем это так:

$$(\nabla f (\text{list } u \ v))$$

¹⁶ Приведите пример, пожалуйста.

[†] Те, кто знаком с автоматическим дифференцированием, могут вспомнить, что (частные) производные функции по заданному набору аргументов определяются автоматически.

Уильям Клиффорд (1845–1879),
 Роберт Венгер (1922–2001),
 Л. Н. Королев (1926–2016), Л. М. Беда,
 Т. С. Фролова, Н. В. Сухих.

Это отличная идея.

Найдем градиент sqr для аргумента, равного 27.0

1. $(\nabla (\lambda (\theta) (sqr \ \theta_0)) (\text{list } 27.0))$
2. $(\text{list } 54.0)$

Первый аргумент ∇ – это функция, которая ожидает, что θ содержит один параметр, и возводит в квадрат этот единственный тензор, используя sqr . Вторым аргументом – это θ , содержащий параметр 27.0. Выражение возвращает градиент sqr при скаляре 27.0[†].

¹⁷ Почему бы нам вместо списка не использовать здесь непосредственно скаляр?

[†] Те, кто знаком с градиентами, наверняка знают, что градиент x^2 для любых x равен $2x$.

Описание внутреннего механизма функции ∇ можно найти в приложении «Призрак из машины» на стр. 349.

Можно заметить, что если бы ∇ принимала аргументы по одному, то нахождение (∇f) было бы эквивалентно математическому определению градиента f . Однако здесь мы определяем его с помощью двух аргументов для простоты.

Мозес Шонфинкель (1889–1942) и Хаскел Карри (1900–1982).

Хороший вопрос.

В то время как использование скаляров напрямую без упаковки их в список имеет смысл для простых примеров, более сложные задачи обучения сопровождаются очень большими наборами параметров.

На самом деле проще упаковать их в список, поэтому мы разработали функцию ∇ , работающую только со списками.

Да, верно.

∇ всегда возвращает список градиентов – по одному на каждый параметр в θ .

При параметрах θ_1 и θ_2 , равных 0.0, градиент функции *obj*

((*l2-loss line*) *line-xs line-ys*)

будет следующим:

1. $(\nabla \text{ obj } (\text{list } 0.0 \ 0.0))$
2. $(\nabla ((\text{l2-loss line}) \text{ line-xs line-ys}) (\text{list } 0.0 \ 0.0))$
3. $(\text{list } -63.0 \ -21.0)$

18 В общем, ∇ получает функцию и список параметров, а возвращает список градиентов, по одному на каждый параметр?

19 Как нам использовать ∇ , чтобы определить градиент функции *l2-loss* относительно

θ_1 и θ_2 ,

где начальное значение каждого параметра равно 0.0?

20 Значит, градиент *obj* относительно θ_0 равен -63.0 , и градиент *obj* относительно θ_1 равен -21.0 .

Но почему значение

-63.0

отличается от скорости изменения

-62.63

для

θ_0 ,

найденной во фрейме 64:34?

Прекрасный вопрос.

Во фрейме 64:34 мы нашли скорость изменения путем аппроксимации (приближения).

Функция ∇ возвращает более точный результат и одновременно вычисляет градиент для θ_1 .

21 Теперь, когда у нас есть способ найти скорость изменения, должны ли мы использовать его циклически, как делали во фрейме 67:43, чтобы найти оптимально подобранный набор параметров θ для *obj*?

Конечно! Мы изменяем θ с помощью ∇ понемногу за раз, на протяжении многих циклов. Для этого мы определим функцию итерации, которая сделает за нас всю рутинную работу.

22 То есть функция итерации поможет нам выполнять вызов других функций снова и снова?

Да.

Ниже показан каркасный код функции *revise*, получающей функцию обновления f , которая не изменяется, натуральное число *revs* и аккумулятор θ . Она изменяет θ *revs* раз и каждый раз возвращает новый набор θ путем вызова f для текущих значений θ :

```
(define revise
  ( $\lambda$  (f revs  $\theta$ )
    (cond
      ((zero? revs)  $\theta$ )
      (else
       (

R

))))
```

Здесь *revs* – количество оставшихся обновлений параметров, а θ – значения параметров, полученные к этому моменту.

23 Выражение R должно рекурсивно вызывать *revise* с новыми значениями *revs* и θ . Мы получаем новое значение θ путем вызова функции обновления f для θ :

($f \theta$)

Кроме того, чтобы наше обновление когда-то завершилось, необходимо находить новое значение *revs*:

(*sub1 revs*)

Следовательно, R будет выглядеть так:

(*revise f (sub1 revs) (f θ)*)

Когда *revs* обнуляется, функция возвращает итоговый набор θ . Если же *revs* не равен нулю, выполняется выражение *R*, обновляющее θ .

Найдите *R*.

Так выглядит *revise*:

```
(define revise
  (λ (f revs θ)
    (cond
      ((zero? revs) θ)
      (else
       (revise f (sub1 revs) (f θ))))))
```

24 Можно испытать эту функцию в деле?

Конечно.

25 Что такое *map*?

Используя *revise*, мы запишем логический столбец с функцией обновления *f*:

```
(λ (θ)
  (map (λ (p)
        (- p 3))
       θ))
```

При начальном значении

```
revs = 5
```

и начальном значении θ

```
(list 1 2 3)
```

окончательное значение θ

```
(list -14 -13 -12)
```

Своевременный вопрос!

В своей наиболее частной форме *map* принимает функцию и список и вызывает функцию для каждого члена этого списка, чтобы создать новый список, который является результатом *map*. Например:

1. $(map (\lambda (x) (add1 x)) (list 5 7 3))$
2. $(list (add1 5) (add1 7) (add1 3))$
3. $(list 6 8 4)$

²⁶ Если это очень частная форма, то как выглядит более общая форма *map*?

В более общей форме *map* принимает более одного списка и вызывает свою функцию для соответствующих членов каждого списка:

1. $(map (\lambda (x y) (+ x y)) (list 12 17 32) (list 8 3 11))$
2. $(list (+ 12 8) (+ 17 3) (+ 32 11))$
3. $(list 20 20 43)$

²⁷ Итак, во фрейме 25 вызов *map* для функции обновления f и θ порождает новый список, в котором каждый элемент на 3 меньше, чем соответствующий элемент θ .

Так выглядит начало логического столбца:

1. `(revise f 5 (list 1 2 3))`
2. `(revise f 4 (f (list 1 2 3)))`
3. `(revise f 4 (list -2 -1 0))`
4. `(revise f 3 (f (list -2 -1 0)))`
5. `(revise f 3 (list -5 -4 -3))`

Завершите его.

28 Спасибо за подсказку. Вот завершение:

6. `(revise f 2 (f (list -5 -4 -3)))`
7. `(revise f 2 (list -8 -7 -6))`
8. `(revise f 1 (f (list -8 -7 -6)))`
9. `(revise f 1 (list -11 -10 -9))`
10. `(revise f 0 (list -14 -13 -12))`
11. `(list -14 -13 -12)`

Теперь вернемся к определению функции, которая ведет нас от одного θ к другому.

Третьим аргументом для `revise` должен быть список, содержащий начальные значения θ_0 и θ_1 в указанном порядке. Как будет выглядеть этот начальный θ ?

29 Мы инициализируем θ_0 и θ_1 значениями 0.0, поэтому начальный набор параметров θ следует записать так:

`(list 0.0 0.0)`

Теперь хотелось бы узнать, как выглядит функция обновления f .

Охотно сделаем это.

Вот каркасный код этой функции, где `revs = 1000`, а скорость обучения $\alpha = 0.01$:

```
(let (( $\alpha$  0.01)
      (obj ((l2-loss line) line-xs line-ys)))
  (let ((f ( $\lambda$  ( $\theta$ )
              (let ((gs ( $\nabla$  obj  $\theta$ )))
                (list
                  

W



B


                  )))))
    (revise f 1000 (list 0.0 0.0))))
```

Объясните, как работает вызов `revise` в этом примере.

30 Мы должны выполнить 1000 обновлений, начиная с `(list 0.0 0.0)` и вызывая f для каждого обновления.

Функция обновления получает начальный набор θ и возвращает другой θ , повторяя это действие 1000 раз.

Очень хорошо.

В функции обновления мы определяем градиент по параметрам θ_0 и θ_1 , которые упакованы в набор θ .

Мы используем ∇ для получения списка градиентов gs – фактически списка из двух скалярных значений.

Первым в списке идет

градиент потерь относительно θ_0 ;

вторым идет

градиент потерь относительно θ_1 .

Теперь найдите выражения для W и B .

31 Функция обновления должна возвращать новое значение θ , поэтому

W – это новое значение θ_0

и

B – это новое значение θ_1 .

Правильно.

Но как нам определить эти новые элементы набора θ , т. е. новые параметры?

32 Мы умножаем градиенты, соответствующие текущим параметрам, на скорость обучения θ и вычитаем результат из значения параметра.

Градиент относительно θ_0 содержится в 0-м элементе списка градиентов, т. е. gs_0 . Значит, выражение W будет следующим:

$$(-\theta_0 (* \alpha gs_0))$$

Аналогично, градиент относительно θ_1 – это gs_1 , и выражение B

$$(-\theta_1 (* \alpha gs_1))$$

Да, неплохо.

Запишем теперь итоговую версию функции f с заполненными полями W и B . Мы показываем вызов $revise$, используя новую f в коде $revision$.

Посмотрим на результат после большого количества обновлений

```

▶ (let ((f (λ (θ)
              (let ((gs (∇ obj θ)))
                (list
                 (- θ0 (* α gs0))
                 (- θ1 (* α gs1)))))))
    (revise f 1000 (list 0.0 0.0)))
▶ (list 1.05 1.87e-06†)

```

[†] В языке Scheme $1.87e-06$ (1.87×10^{-6}) – настолько малое число, что его можно рассматривать как 0.0.

33 Итак, это означает, что после 1000 обновлений мы получим достаточно хорошо подобранный набор θ .

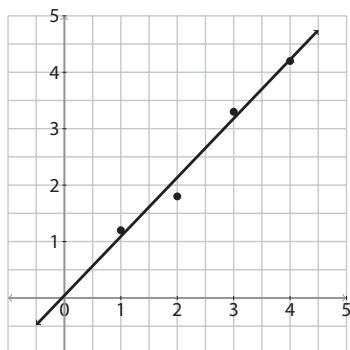
Что представляет собой строка

(list 1.05 1.87e-06)?

Закон обновления (итоговая версия)

Новый $\theta_i = \theta_i - (\alpha \times \text{скорость изменения потери с учетом } \theta_i)$

Это линия, подогнанная к исходным точкам:



34 Это невероятно! Она почти совпадает с линией, построенной визуально во фрейме 24:20.

Но возникает другой вопрос. Почему мы выбрали для $revs$ значение 1000? Откуда мы знаем, что этого достаточно для качественного подбора θ ?

Очень хороший вопрос.
Мы выбираем достаточно большое число *revs*, которое хорошо приближает нас к точному θ .

В зависимости от типа функции, которую мы оптимизируем, мы можем выбрать подходящее значение *revs* (обычно за счет комбинации размера набора данных и экспериментов).

35 Это немного похоже на поиск наобум. Существует ли подход получше?

Да, но для ограниченного перечня задач.

Поскольку нам встречаются задачи с θ , включающими гораздо большие тензоры и целевые функции, которые делают гораздо более интересные вещи, фиксированное количество *revs* обычно является хорошим подходом.

36 Есть ли название у этого способа подбора параметров θ_0 и θ_1 ?

Да.
Это так называемый алгоритм *оптимизации градиентным спуском*[†].

37 Кто бы мог подумать, что наклонные линии настолько увлекательные!

[†] Предложил Огюстен Луи Коши (1789–1857).

Это точно!

Мы можем дополнительно упростить функцию обновления f .

Взгляните на выражение в функции обновления, которое возвращает новый набор θ :

```
(list
  (-  $\theta_0$  (*  $\alpha$   $gs_0$ ))
  (-  $\theta_1$  (*  $\alpha$   $gs_1$ )))
```

38 Что с ним не так?

Это выражение предназначено только для θ и содержит ровно два параметра.

39 Ох, и правда, оно работает лишь с функциями, которым нужны ровно два параметра в θ (такими как *line*).

Вот именно.

В общем случае мы можем вырабатывать новый θ независимо от его длины и в соответствии с длиной gs .

40 Можем ли мы использовать здесь *map*?

Да, можем.

Мы можем вычесть θ раз этот элемент gs из элемента θ . Это действие можно оформить как функцию двух аргументов, где p – элемент θ , а g – соответствующий элемент gs :

```
( $\lambda$  ( $p$   $g$ )
  (-  $p$  (*  $\alpha$   $g$ )))
```

41 Действительно, теперь мы распространяем эту функцию на
 θ
и
 gs .

Верно.

Теперь перепишем вызов *revise*:

```
(let ((f (λ (θ)
  (let ((gs (∇ obj θ)))
    (map (λ (p g)
      (- p (* α g)))
      θ
      gs))))))
  (revise f 1000 (list 0.0 0.0)))
```

42 То есть мы должны заменить строки

```
(list
  (- θ0 (* α gs0))
  (- θ1 (* α gs1)))
```

на аналогичный вызов с использованием *map* из фрейма 25:

```
(map (λ (p g)
  (- p (* α g)))
  θ
  gs)
```

Да, правильно.

Параметризованная функция в выражении

```
((l2-loss line) line-xs line-ys)
```

представляет собой целевую функцию из фрейма 62:26, потому что наша цель – найти θ , при котором эта функция минимальна, т. е. приводит нас к нижней точке функции потерь, где потеря минимальна.

Теперь мы можем записать полный градиентный спуск для нашего набора данных:

```
(let ((α 0.01)
  (obj ((l2-loss line)
    line-xs line-ys)))
  (let ((f (λ (θ)
    (map (λ (p g)
      (- p (* α g)))
      θ
      (∇ obj θ))))))
    (revise f 1000 (list 0.0 0.0))))
```

43 Отлично.

Мы собираемся сделать временную корректировку. За исключением начального значения θ , в этом выражении есть два «постоянных скаляра»:

количество обновлений $revs$

и

скорость обучения α .

А пока давайте определим эти имена:

```
| (define revs 1000)
|
| (define  $\alpha$  0.01)
```

Как теперь мы должны записать градиентный спуск?

44 Наверное, так?

```
(let ((obj ((l2-loss line)
              line-xs line-ys)))
  (let ((f ( $\lambda$  ( $\theta$ )
                (map ( $\lambda$  (p g)
                        (- p (*  $\alpha$  g)))
                      $\theta$ 
                     ( $\nabla$  obj  $\theta$ ))))))
    (revise f revs (list 0.0 0.0))))
```

Великолепно!

45 Понятно.

В этом выражении значение obj

```
((l2-loss line) line-xs line-ys)
```

и начальное значение θ

```
(list 0.0 0.0)
```

становятся аргументами функции

gradient-descent

Итак, мы определили этот алгоритм как функцию *gradient-descent*. Символ $\Theta^\dagger$ – это просто переименованный формальный аргумент функции f со стр. [xxiii](#).

46 Это именно то, что мы начали искать!

При чем здесь Θ ?

```
| (define gradient-descent
|   ( $\lambda$  (obj  $\theta$ )
|     (let ((f ( $\lambda$  ( $\Theta$ )
|                   (map ( $\lambda$  (p g)
|                           (- p (*  $\alpha$  g)))
|                         $\Theta$ 
|                        ( $\nabla$  obj  $\theta$ ))))))
|       (revise f revs (list 0.0 0.0))))
```

```

      (∇ obj Θ))))
(revise f revs θ)))

```

Функция *gradient-descent* дает нам возможность находить хорошо подобранные θ для множества разных целевых функций с их собственными θ .

[†] Произносится как «тэта большая».

Θ – это формальный аргумент функции обновления $f^\dagger$.	⁴⁷ Но все же, зачем мы ввели обозначение Θ ?
---	--

[†] Здесь мы тоже переименовали θ в Θ , как во фрейме 60:21.

Просто мы возлагаем на Θ большие надежды [†] .	⁴⁸ Хорошо, пусть пока так и будет.
--	---

А пока Θ – просто имя, которое мы дали формальному аргументу функции обновления.

[†] «Большие надежды» – роман Чарльза Диккенса (1812–1870).

Где располагаются начальные значения θ и Θ ?	⁴⁹ θ фигурирует только в вызове <i>revise</i> .
--	---

Θ фигурирует лишь в функции обновления.

Смотрите, как мы используем обновление в виде <i>gradient-descent</i> для обучения θ_0 и θ_1 начиная со значений 0.0:	⁵⁰ Мы получили тот же результат, что и во фрейме 33.
---	---

Теперь у нас есть еще один способ обучения!

```

▷ (gradient-descent
  ((l2-loss line) line-xs line-ys)
  (list 0.0 0.0))
▶ (list 1.05 1.87e-06)

```

Да, точно!

Мы найдем ему применение в
следующей интерлюдии.

51 Фухххх! А теперь самое время
отдохнуть.

Градиентные побрякушки

∇ стр. 75

revise стр. 77

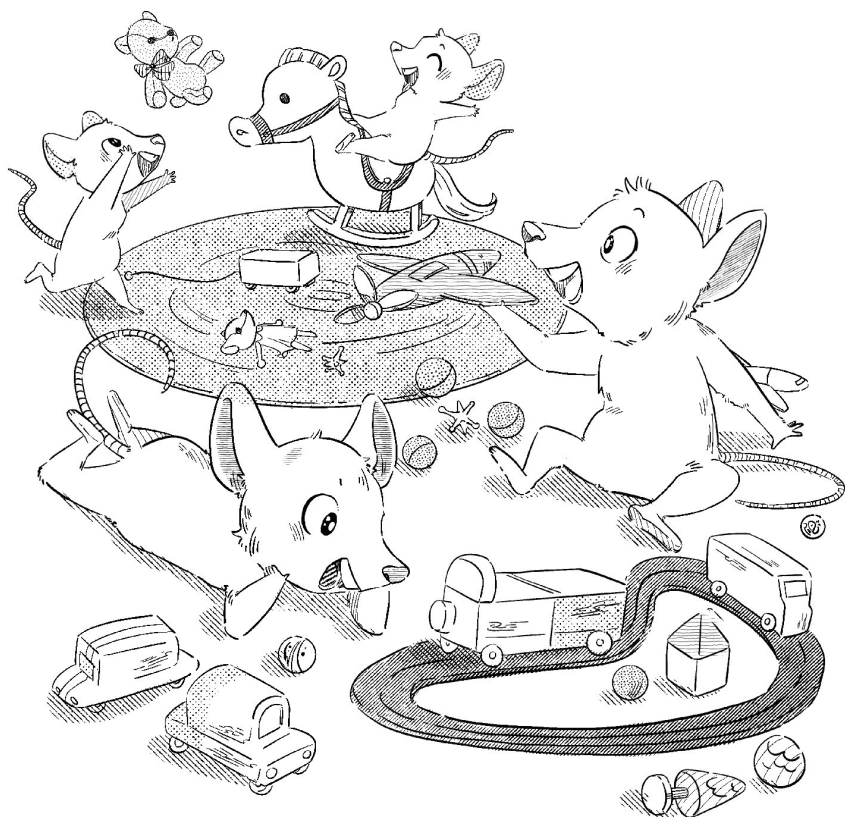
map стр. 78

gradient-descent (пунктирная версия) стр. 86

Это был довольно крутой
и скользкий спуск.

ВОЗНАГРАДИТЕ СЕБЯ ПОРЦИЕЙ
ТИРАМИСУ!

ИНТЕРЛЮДИЯ II. ЧЕМ БОЛЬШЕ ИГРУШЕК, ТЕМ БОЛЬШЕ ШУМА



Как вам тирамису?	¹ Ел бы без остановки.
Эта интерлюдия посвящена гиперпараметрам.	² Что такое гиперпараметры?
Временные рабочие параметры с именами наподобие <i>revs</i> и α , определяющие поведение функции <i>gradient-descent</i> , называются гиперпараметрами. Гиперпараметры всегда ассоциируются со скалярными значениями.	³ Почему они важны?
Эти скаляры зависят от конкретной задачи и могут быть выбраны только после некоторых размышлений и экспериментов.	⁴ Иными словами, скалярные значения, которые мы выбрали для α и <i>revs</i> в функции <i>gradient-descent</i> , не обязательно будут работать в других ситуациях?
Верно. Существуют и другие гиперпараметры, о которых мы поговорим позже. Чтобы облегчить себе жизнь, мы введем новую конструкцию.	⁵ Интересно. Можем ли мы собрать гиперпараметры в список, как мы делали для θ ?
Нет, лучше рассматривать их как имя особого типа.	⁶ В чем их особенность?
Если такие имена объявлены как гиперпараметры, их можно использовать в любой функции.	⁷ Отлично.

Вот условный пример:

(declare-hyper *smaller*)

(declare-hyper *larger*)

Чему будет равен результат операции

(+ *smaller larger*)?

⁸ Поскольку ни *smaller*, ни *larger* не ассоциированы со скалярной величиной, результату операции не будет присвоено какое-либо значение.

Правило гиперпараметров

Каждый гиперпараметр является скаляром или не содержит значения.

Да, правильно.

А каким будет результат следующего выражения?

(with-hypers
((*smaller* 1)
(*larger* 2000))
(+ *smaller larger*))

⁹ Это выражение должно вернуть 2001[†].

Но будут ли *smaller* и *larger* оставаться скалярными величинами снаружи with-hypers?

[†] Напоминание об Артуре Кларке (1917–2008) и его «Одиссее 2001».

Нет, не будут.

Эти гиперпараметры имеют скалярные значения только внутри выражения with-hypers.

¹⁰ Означает ли это, что with-hypers аналогичен let?

Вовсе нет.

Выражение с оператором let создает новое локальное имя для данного значения, а with-hypers предоставляет новое значение для уже существующего имени, которое было объявлено как гиперпараметр.

¹¹ Теперь понятно!

Как только эти гиперпараметры получают скаляры, связанные с ними с помощью `with-hyper`, эти скаляры становятся доступными для всех функций, использующих их. После тела выражения

(+ *smaller larger*)

если `with-hyper`s вернул результат, гиперпараметры больше не содержат значения[†].

[†] Так бывает не всегда. На самом деле `with-hyper` могут быть вложены друг в друга, так что выход из внутреннего `with-hyper` по-прежнему поддерживает скаляры, связанные с гиперпараметрами во внешнем `with-hyper`.

Да, верно.
 Теперь кое-что поинтереснее. Допустим, мы определили демонстрационную функцию *nonsense?* следующим образом:

```
| (define nonsense?
|   (λ (x)
|     (= (sub1 x) smaller)))
```

Чему будет равно
 (*nonsense?* 6)?

¹² Получается, гиперпараметры похожи на имена, назначенные оператором `define`, но могут быть ассоциированы со скалярными значениями только при помощи `with-hyper`s?

¹³ Здесь не будет значения по той же причине, что и прежде. Гиперпараметр *smaller* не представляет скалярное значение.

Мы продолжим, предоставив скалярное значение используемому гиперпараметру:

(with-hypers
((smaller 5))
(nonsense? 6))

Результатом будет #t (истина). Чтобы понять, как это работает, представьте, что определение *nonsense?* тоже находится внутри with-hypers, и руководствуйтесь этим.

Да!
Поясните, как мы получили результат.

¹⁴ Является ли (*nonsense?* 6) телом этого with-hypers?

¹⁵ При вызове *nonsense?* внутри with-hypers скаляр *smaller* равен 5, поэтому (*nonsense?* 6) возвращает #t. Но почему определение *nonsense?* дано с пунктиром?

Потому что было бы нелепо использовать в настоящей программе имя
nonsense? (бессмыслица?).

¹⁶ Будем ли мы использовать гиперпараметры в новой версии *gradient-descent*?

Обязательно, но займемся этим в следующей главе.

¹⁷ Жду с нетерпением!

Гиперактивные побрякушки

declare-hyper стр. 91

with-hypers стр. 91

**Эта интерлюдия слишком
коротка для перерыва!**

**ВОЗЬМИТЕ ЕЩЕ КУСОЧЕК ТИРАМИСУ –
И ПРОДОЛЖИМ!**

5. ТОЧНО В ЦЕЛЬ!



Вы готовы продолжить?	1 Жду не дождусь!
Пришло время обобщить то, что вы уже узнали.	2 Сейчас вы расскажете, почему мы возлагаем на Θ большие надежды?
Пока нет, но мы близки к этому.	3 Хорошо, с чего начнем?
Итак, мы определили два важных гиперпараметра:	4 Как мы можем применить эти гиперпараметры в <i>gradient-descent</i> ?

```
(declare-hyper revs)
```

```
(declare-hyper  $\alpha$ )
```

Мы уже встречали *revs* и α раньше, но не в качестве гиперпараметров.

Вспомним определение *gradient-descent* с пунктиром из фрейма 86:46. Там обновление повторяется *revs* раз, и при каждом обновлении уточняется результат предыдущего прохода:

```
(define gradient-descent
  (λ (obj θ)
    (let ((f (λ (Θ)
               (map (λ (p g)
                     (- p (* α g)))
                    Θ
                    (∇ obj Θ))))))
      (revise f revs θ))))
```

А помните, что такое *l2-loss*?

5 Да, вот *l2-loss* из фрейма 61:22:

```
(define l2-loss
  (λ (target)
    (λ (xs ys)
      (λ (θ)
        (let ((pred-ys ((target xs) θ)))
          (sum
            (sqr
              (- ys pred-ys))))))))
```

В этом определении *gradient-descent* применяются два гиперпараметра:

revs

и

α .

Используя *l2-loss* и *with-hyper* из 91:9, присвойте скалярное значение каждому гиперпараметру и покажите, как вызвать *gradient-descent* в этом случае.

6 Вот обновленный каркасный код:

```
▶ (with-hypers
  ((revs 1000)
   ( $\alpha$  0.01))
  (gradient-descent
   ((l2-loss line) line-xs line-ys)
   (list 0.0 0.0)))
▶ (list 1.05 1.87e-06)
```

Мы получили тот же результат, что и во фрейме 87:49, при этом у нас четко выделены гиперпараметры.

Определения *gradient-descent* и *l2-loss* во фрейме 5 получают функции как аргументы и не привязаны к конкретной целевой функции (типа *line*).

7 Это означает, что мы можем использовать эти две функции для обучения параметров θ произвольной целевой функции.

У нас развязаны руки!

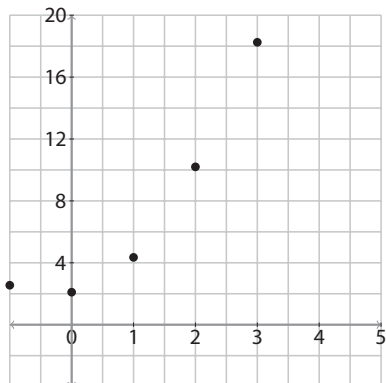
Рассмотрим новый набор данных:

```
(define quad-xs
  [-1.0 0.0 1.0 2.0 3.0])

(define quad-ys
  [2.55 2.1 4.35 10.2 18.25])
```

Отобразите данные на графике.

8 Эти точки не располагаются на прямой линии:



Да, это явно не прямая линия. Поэтому нам придется обучить θ для другой целевой функции. В этом примере мы используем простую нелинейную функцию.	9	Что такое нелинейная функция?
Нелинейная функция нелинейно зависит от своих аргументов.	10	И...
Линейная функция – это функция, которая использует только сложение и масштабирование для получения результата.	11	Что такое масштабирование?
Масштабирование – это умножение на фиксированное значение или на параметр. Например, выражение (* 5.0 x) масштабирует значение x с коэффициентом 5.0. И если у вас есть параметр θ_0 то (* $\theta_0 x$) масштабирует x с коэффициентом θ_0.	12	Так вот почему <i>line</i> является линейной функцией. Она масштабирует аргумент x параметром θ_0 и прибавляет к результату θ_1. При помощи какой нелинейной функции получен набор данных во фрейме 8?
Очень хороший вопрос. Примем как допущение, что это квадратичные данные.	13	Что означает слово квадратичные?
Это означает, что они могут быть предсказаны при помощи квадратичной функции.	14	Хорошо, но что такое квадратичная функция?

Вот обобщенное определение квадратичной функции[†]:

```
(define quad
  (λ (t)
    (λ (θ)
      (+ (* θ0 (sqr t))
         (+ (* θ1 t) θ2))))))
```

Функция *quad* имеет три скалярных параметра в наборе θ .

[†] Возможно, более узнаваемым будет квадратичное уравнение, в котором вместо t используется x :

$$ax^2 + bx + c = 0.$$

Конечно.

Пусть

t будет равен 3.0

и

θ представлен списком
(list 4.5 2.1 7.8).

Составьте логический столбец для этого примера.

15 Можно увидеть пример того, как работает *quad*?

16 Вот он:

1. | ((quad 3.0) (list 4.5 2.1 7.8))
2. | (+ (* 4.5 (sqr 3.0))
- | (+ (* 2.1 3.0) 7.8))
3. | (+ 40.5
- | (+ 6.3 7.8))
4. | 54.6

Думаю, что использование *sqr* делает функцию *quad* нелинейной.

Совершенно верно.

Здесь мы используем *gradient-descent* и *l2-loss* как функцию потерь, чтобы обучить θ на новом наборе данных с *quad* в качестве функции-цели.

Какой будет новая ожидающая функция из фрейма 61:24?

17 Ожидающая функция – это функция потерь, вызываемая для функции-цели:

(l2-loss quad)

Какой будет целевая функция?	18 Целевая функция из фрейма 62:26 – это функция, полученная путем применения ожидающей функции к новому набору данных <code>((l2-loss quad) quad-xs quad-ys)</code>
Отлично. Мы вызываем <i>gradient-descent</i> для этой целевой функции и начального набора параметров θ.	19 Можем ли мы, как прежде, использовать нулевые начальные значения параметров <code>(list 0.0 0.0 0.0)</code>?
Да, можем. Теперь нужно обсудить гиперпараметры.	20 Естественно. Ведь сейчас мы будем подбирать θ для другой целевой функции. Какие скаляры нужно присвоить гиперпараметрам <i>revs</i> и α?
Поскольку наша цель – квадратичная функция, возводящая в квадрат свой аргумент, который здесь равен <i>quad-xs</i>, вполне вероятно, что наши градиенты могут стать очень большими. Чтобы не проскочить оптимум из-за больших скачков между обновлениями параметров, имеет смысл выбрать уменьшенную скорость обучения. Опыт подсказывает нам, что подходящим значением будет 0.001.	21 Какое скалярное значение мы выберем для <i>revs</i>?

На данный момент давайте оставим прежнее значение 1000, и наш вызов *gradient-descent* будет телом для *with-hyper* в следующей каркасной функции обновления:

```
► (with-hypers
  ((revs 1000)
   ( $\alpha$  0.001))
  (gradient-descent
   ((l2-loss quad) quad-xs quad-ys)
   (list 0.0 0.0 0.0)))
► (list 1.48 0.99 2.05)
```

Конечно!

А теперь сделаем следующие допущения о том, как использовать *gradient-descent* и *l2-loss*.

Так выглядит новый набор данных:

```
(define plane-xs
  [[1.0 2.05]
   [1.0 3.0]
   [2.0 2.0]
   [2.0 3.91]
   [3.0 6.13]
   [4.0 8.09]])

(define plane-ys
  [13.99
   15.99
   18.0
   22.4
   30.2
   37.94])
```

22 Итак, результат *gradient-descent* с этими гиперпараметрами и его целью:

```
(list 1.48 0.99 2.05)
```

Мы узнали, что три значения параметров для *quad* следующие: 1.48, 0.99 и 2.05. Это похоже на то, как мы ранее узнали значения θ_0 и θ_1 для *line*!

Мы использовали *gradient-descent* с функцией-целью, отличной от *line*!

23 Этот набор данных отличается от предыдущего, потому что *plane-xs* представляет собой *tensor*², состоящий из 6 двухэлементных *tensor*¹, в то время как *plane-ys* представляет собой *tensor*¹, состоящий из 6 скаляров.

Да, верно[†].

В наборах данных
(*line-xs*, *line-ys*)

и

(*quad-xs*, *quad-ys*)

элементы x и y представляют собой `tensor`¹ одинаковой формы.

Теперь расширим наш подход. Допустим, что x и y имеют разную форму, но оба имеют одинаковое количество вложенных тензоров. Другими словами, мы должны допустить, что

$\dagger xs \dagger$

и

$\dagger ys \dagger$

одинаковые.

Если вы забыли, что означают записи $\dagger xs \dagger$ и $\dagger ys \dagger$, перечитайте фрейм 32:17.

[†] Подобно тому, как прямая представляет собой линейную связь между двумя координатами точки в двух измерениях, плоскость представляет собой линейную связь между координатами точки в трех измерениях.

Отличный вопрос.

Ниже показана обучаемая функция, которую мы подбираем (как это делали для *line*):

```
(define plane
  (λ (t)
    (λ (θ)
      (+ (• θ0 t) θ1))))
```

24 Как это обобщение повлияет на *gradient-descent* и *l2-loss*?

25 Что означает здесь символ точки $\bullet$ [†]?

[†] Читается как «произведение с точкой» или «скалярное произведение». Обозначение ввели Джозайя Гиббс (1839–1903) и Эдвин Уилсон (1879–1964).

Здесь t – аргумент *plane* – представляет собой tensor^1 .
В этом отличие от *line*, которая ожидает скаляр.

Функция $\bullet^{1,1}$ определяется так:

```
(define  $\bullet^{1,1}$ 
  ( $\lambda$  w t)
  (sum1
    (* w t))))
```

где w и t – тензоры ранга 1, которые должны иметь одинаковую форму.

Охотно.

Так функция $\bullet^{1,1}$ работает с двумя tensor^1 :

```
1. | ( $\bullet^{1,1}$  [2.0 1.0 7.0] [8.0 4.0 3.0])
2. | (sum1
    | (* [2.0 1.0 7.0] [8.0 4.0 3.0]))
```

Закончите этот логический столбец.

26 Покажите на примере, как работает эта функция.

27 Вот завершение:

```
3. | (sum1
    | [( $\bullet^{1,1}$  [2.0 1.0 7.0] [8.0 4.0 3.0])
    | (* [2.0 1.0 7.0] [8.0 4.0 3.0])])
4. | (sum1
    | [16.0 4.0 21.0])
5. | 41.0
```

Мы попарно перемножаем скаляры, получая новый tensor^1 , а потом вычисляем *sum* этого тензора.

Использование функции $\bullet^{1,1}$ – это способ перемножить два tensor^1 и получить одиночный скаляр.

Расширенная функция $\bullet$ получается расширением определения $\bullet^{1,1}$ на тензоры с рангом больше 1.

28 Понятно.

Следовательно, ранг полученного тензора меньше, чем ранг аргументов.

29 Это следствие закона на стр. 50?

Правило наборов данных

В наборе данных (x_s, y_s) как x_s , так и y_s должны иметь одинаковое количество элементов. Но элементы x_s могут иметь форму, отличную от формы y_s .

Конечно.

Теперь мы обучим набор параметров θ функции *plane* на текущем наборе данных с помощью

l2-loss

и

gradient-descent

Мы начнем с того, что попробуем найти начальные значения θ .

Какими они должны быть?

Мы определим это, исходя из формы тензоров в наборе данных

$(plane-x_s, plane-y_s)$,

зная, что они будут использоваться как аргументы в *plane*.

Мы знаем, что каждый элемент в *plane-x_s* – это *tensor*¹ и каждый элемент в *plane-y_s* – это скаляр.

³⁰ Но чтобы это сделать, мы должны знать форму θ_0 и θ_1 во фрейме 25.
Откуда мы это узнаем?

³¹ Отлично!

Таким образом, когда мы вызываем *plane* для элемента из *plane-x_s*, точно подобранный θ должен давать результат, который имеет ту же форму, что и элемент *plane-y_s*.

Верно.

Поскольку `•` ведет себя точно так же, как *sum*, и возвращает результат на один ранг ниже, чем аргументы, чтобы получить скаляр, оба его аргумента должны представлять собой `tensor1` одинаковой длины.

Итак, какую форму должен иметь θ_0 ?

32 Первым аргументом функции является

θ_0

Вторым аргументом, во фрейме 26, является `tensor1` из

plane-xs.

Следовательно, форма θ_0 должна быть такой же, как у `tensor1` из *plane-xs*, а именно (list 2)

Тоже правильно.

А что насчет формы θ_1 ?

33 Тензор

θ_1

должен иметь ту же форму, что и результат функции

plane

из фрейма 25, и такую же, как форма каждого скаляра, извлеченного из

plane-ys

а значит, θ_1 должен быть скаляром с формой

(list)

Иными словами, θ_1 является скаляром.

И снова вы правы.

Теперь определите начальный список параметров θ , используя формы, основанные на том, как они применяются внутри функции-цели, например во фреймах 25 и 26:25.

34 Как насчет

(list [0.0 0.0] 0.0)

где

θ_0 инициализирован тензором [0.0 0.0]

и

θ_1 инициализирован скаляром 0.0?

Правило параметров (итоговая версия)

Каждый параметр является тензором.

Великолепно.

Здесь θ_0 и θ_1 имеют разную форму. Поскольку θ – это список, его элементы, в отличие от элементов тензора, могут иметь разную форму.

35 Означает ли наличие разных форм, что мы должны переписать некоторые из наших функций?

Вовсе нет.

36 Какое облегчение!

Правило θ

θ – это список параметров, которые могут иметь разную форму.

Нам нужна еще одна вещь!

37 Хорошо.

Нам нужно решить, какие скаляры использовать для гиперпараметров $revs$ и α .

В качестве первого предположения давайте сохраним те же скаляры, что и для *quad* во фрейме 22.

Напишите обновленную версию кода, использующего `with-hyprs` для набора данных из фрейма 23.

38 Вот она:

```

▶ (with-hyprs
  ((revs 1000)
   ( $\alpha$  0.001))
  (gradient-descent
   ((l2-loss plane) plane-xs plane-ys)
   (list [0.0 0.0] 0.0)))
▶ (list [3.98 2.04] 5.78)

```

Прекрасно.

Мы нашли точно подобранные параметры θ .

39 Откуда мы знаем, что получили достаточно точную модель?

Мы проверим ее на точках из набора данных во фрейме 23. Возьмем, к примеру,

$plane\text{-}xs|_3$

40 Это значение расположено достаточно близко к реальному значению $plane\text{-}xs|_3$ (22.4). Но почему они не совсем совпали?

1. $((plane\ [2.0\ 3.91])$
 $(list\ [3.98\ 2.04]\ 5.78))$
2. $(+ (\bullet\ [3.98\ 2.04]\ [2.0\ 3.91])$
 $5.78)$
3. 21.71

Потому что данные часто бывают зашумлены, и функцию невозможно подогнать под них с идеальной точностью. Достаточно близкое подобие – вот что мы обычно стремимся получить на практике.

41 Теперь пора сделать перерыв!

Побрякушки для стрельбы по мишеням

radient-descent (с гиперпараметрами *revs* и α) стр. 96

(*quad-x*, *quad-y*) стр. 97

quad стр. 99

(*plane-x*, *plane-y*) стр. 101

plane стр. 102

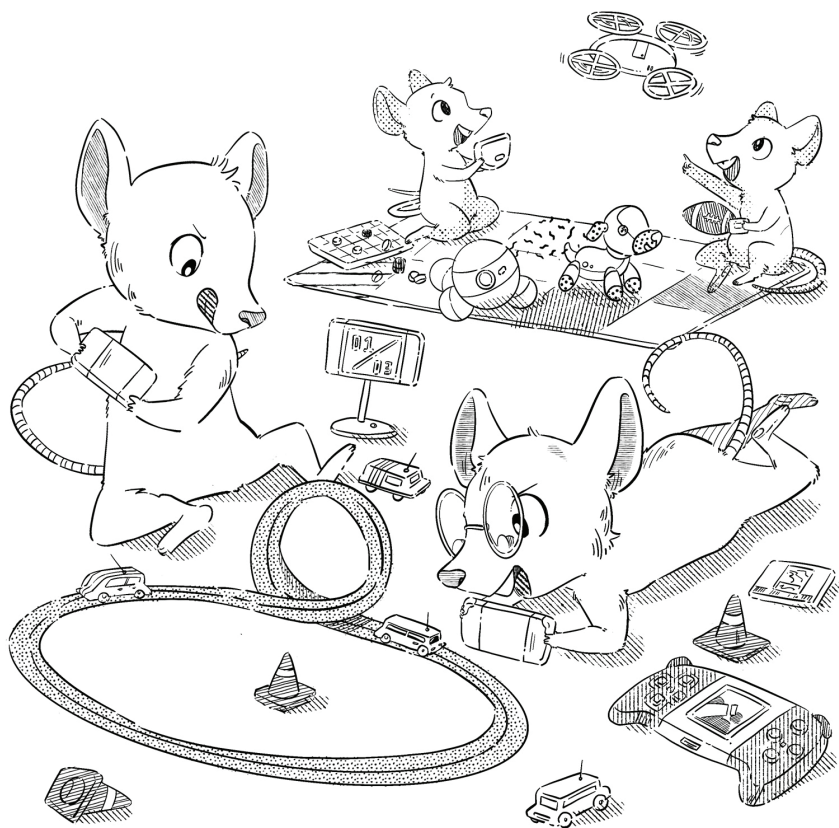
• стр. 103

Мы проделали тяжелую
работу!

САМОЕ ВРЕМЯ ДЛЯ БЕСАН ЛАДУ¹
(बेसन लड्डू)!

¹ Бесан ладу (besan laddoo) – популярная индийская сладость в виде шариков, приготовленных из нутовой муки, также известной как бесан, сахарной пудры и топленого масла. – *Прим. перев.*

ИНТЕРЛЮДИЯ III. ОБЛИК ГРЯДУЩЕГО¹



¹ Название романа Герберта Уэллса (1866–1946). В английском языке идиома *the shape of things to come* (облик грядущего) обозначает нечто, что станет очень популярным в будущем, – стиль, форма, идея. – Прим. перев.

Наша начальная подготовка завершена.	1	Наконец-то! Как минимум, мы познакомились со вкусом индийских сладостей.
Какова форма тензора tensor^2 [[2 4 5] [6 7 9]]?	2	Вот она: (list 2 3)
Давайте снабдим tensor^2 описанием его формы: [[2 4 5] [6 7 9]] _(2 3)	3	Так проще понять форму тензора.
Теперь отбросим вложенные квадратные скобки и снова запишем тензор [†] : $\begin{bmatrix} 2 & 4 & 5 \\ 6 & 7 & 9 \end{bmatrix}_{(2\ 3)}$ ----- [†] Мы записали tensor^2 способом, которым обычно записывают матрицы, и добавили обозначение формы.	4	Я вижу, что каждый вложенный тензор [†] записан как отдельная строка без окружающих скобок. ----- [†] Это менее громоздкая запись, но при определении функций мы предполагаем, что эти тензоры записываются в их вложенной форме, как во фрейме 2.
Да, вы правы. Отдельным важным случаем является tensor^2 , где каждая строка представляет собой tensor^1 , состоящий ровно из одного скаляра.	5	Можно увидеть пример?
Возьмем tensor^2 [[5] [7] [8]] и запишем его новым способом [†] : $\begin{bmatrix} 5 \\ 7 \\ 8 \end{bmatrix}_{(3\ 1)}$ ----- [†] Это так называемая столбцовая матрица.	6	Теперь мы можем узнать ранг и форму тензора без подсчета квадратных скобок, как того требовал закон на стр. 34. Как записать tensor^2 [[5 7 8]]?

Хитрый вопрос!

Этот tensor^2

$[[5\ 7\ 8]]$

имеет форму

$(\text{list } 1\ 3)$

Поэтому мы запишем его так[†]:

$[5\ 7\ 8]_{(1\ 3)}$

[†] Это так называемая матрица-строка.

Бесспорно,

$[5\ 7\ 8]$

– это тензор¹, который имеет форму

$(\text{list } 3)$

Поэтому мы должны писать

$[5\ 7\ 8]_{(3)}$

Но для простоты записи принято опускать обозначение формы у всех tensor^1 .

⁷ К сожалению, такая нотация похожа на запись

$[5\ 7\ 8]$

Но это другой тензор, потому что он имеет форму

$(\text{list } 3)$

⁸ Итак,

$[5\ 7\ 8]$

– это просто tensor^1 с 3 скалярами, в то время как

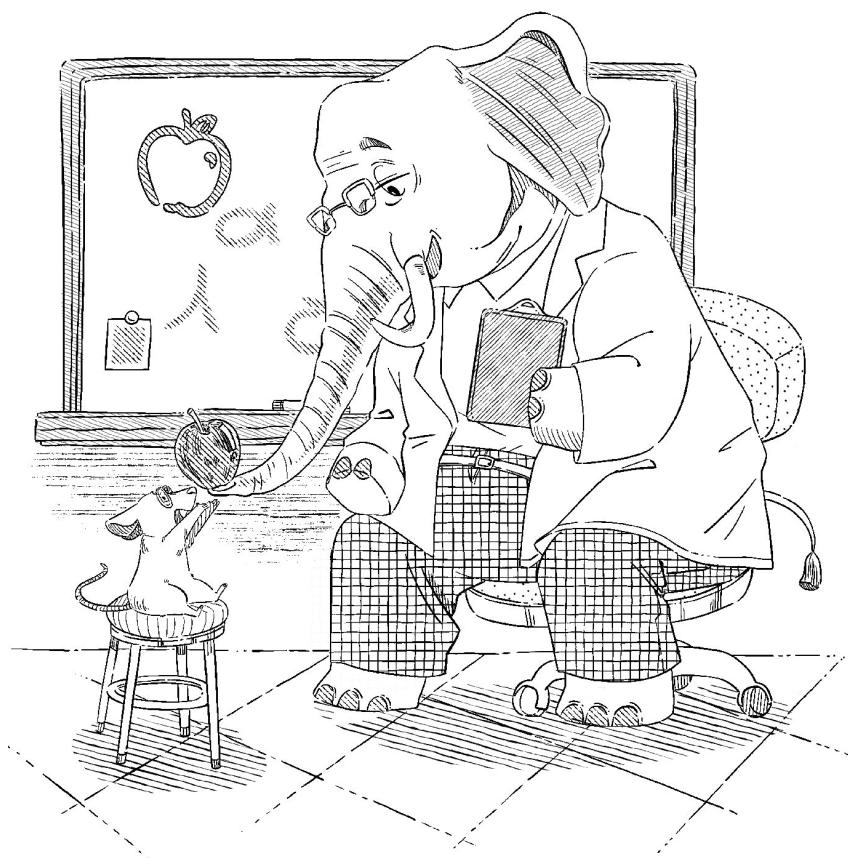
$[5\ 7\ 8]_{(1\ 3)}$

– это tensor^2 , включающий один tensor^1 .

Устроим яблочный перерыв!

ХОТИТЕ СОЧНУЮ АНТОНОВКУ?

6. ОДНО ЯБЛОКО В ДЕНЬ



Готовы похрустеть яблоками?	¹ Конечно! Но разве яблоки утолят нашу жажду узнать, какие большие надежды мы возлагаем на Θ ?
Нет, но остался буквально один момент. Просто подождите еще немного. Мы понемногу накрываем на стол.	² Ждем, затаив дыхание.
Отлично. Двигаемся дальше с яблоком в руках! Наши недавние наборы данных содержали очень мало точек. Однако реальные наборы данных состоят из тысяч, миллионов и даже миллиардов точек.	³ В чем тут проблема?
Вернемся к функции <i>l2-loss</i> из фрейма 61:22: <pre>(define l2-loss (λ (target) (λ (xs ys) (λ (θ) (let ((pred-ys ((target xs) θ))) (sum (sqr (− ys pred-ys))))))))</pre> Эта функция использует весь набор данных (т. е. тензоры <i>xs</i> и <i>ys</i>) при каждом вызове, потому что должна находить разницу между <i>ys</i> в наборе данных и соответствующими <i>pred-ys</i> , предсказанными целевой функцией.	⁴ Короче говоря, <i>gradient-descent</i> вызывает функцию <i>l2-loss</i> <i>revs</i> раз, и каждое обновление использует весь набор данных. Это проблема?

Да, но что более важно, в этом нет необходимости. ⁵ Что это за способ?

Существует более удачный способ, для которого не нужно перебирать весь набор данных тысячи раз.

Скоро узнаете. ⁶ Что такое выборка?

Но сначала нужно познакомиться с понятием выборки.

Представьте, что некая садовница по имени Мария[†] собрала 1000 яблок на продажу. ⁷ При чем тут яблоки?

[†] Мария «Гренни» Смит (1799–1870) впервые культивировала сорт яблок Гренни Смит методом случайной выборки сеянцев. Granny (англ.) – бабуля.

Они вкусные, и, кстати, нам их врачи рекомендуют. ⁸ Так как же ей поступить?

Но на самом деле не все яблоки одинаково вкусные, поэтому Мария должна убедиться, что продает хорошие яблоки.

Хороший вопрос. ⁹ Но как обойтись без проверки вкуса каждого яблока?

Мария не стала откусывать от каждого яблока! В таком случае она бы испортила весь свой товар.

Мария попробовала небольшое количество случайно выбранных яблок. ¹⁰ Ясно!

Она взяла выборку яблок.

Правильно.

Испробовав достаточное количество яблок, Мария получила очень хорошее приближенное представление о вкусе яблок и о том, каким будет спрос на рынке.

На что похожа задача Марии?

11 Она похожа на то, что происходит во фрейме 4. Если при обновлении мы проверяем все точки данных, это сродни надкусыванию каждого яблока.

Можем ли мы использовать выборку, чтобы решить эту проблему?

Конечно можем!

Использование небольшой случайной выборки нескольких точек из набора данных дает достаточно хорошее приближение потерь, которое можно использовать для обновления θ . Мы называем эту выборку пакетом, а его создание – выборкой пакета из набора данных.

12 То есть при каждом обновлении мы проверяем только небольшую часть набора данных?

Да, и мы выполняем много обновлений с новой выборкой для каждого обновления, чтобы максимально приблизиться к идеальной потере.

13 Как это реализовать на практике?

Начнем с помеченного пунктиром объявления *gradient-descent* из фрейма 96:5.

```
(define gradient-descent
  (λ (obj θ)
    (let ((f (λ (Θ)
               (map (λ (p g)
                     (- p (* α g)))
                    Θ
                    (∇ obj Θ))))))
      (revise f revs θ))))
```

14 Мы получаем целевую функцию, когда ожидающая функция вызывается с *xs* и *ys*.

Например, для ожидающей функции

(l2-loss line)

и набора данных из фрейма 23:17 целевой функцией будет

((l2-loss line) line-xs line-ys)

Что мы можем сказать про целевую функцию obj и xs и ys из набора данных?

Целевая функция вырабатывает потерю для текущего θ , и наша цель – сделать эту потерю как можно меньше, максимально приблизившись к идеалу.

Да.

15 А 2-loss – функция потерь.

В общем случае мы можем написать

$((l2\text{-loss target})\ xs\ ys)$,

где

$target$ – это функция-цель

и

(xs, ys) – текущий набор данных.

Если нам нужно вызвать ожидающую функцию

16 Это означает, что нужно создать вспомогательную функцию!

$(l2\text{-loss target})$

с пакетами, выбранными только из xs и ys , мы должны случайным образом выбрать парные тензоры из xs и ys для каждого обновления и передать эти выборочные тензоры в ожидающую функцию.

Конечно! Только нам перед этим нужны кое-какие основы.

17 Она будет состоять из i -го элемента xs

и

i -го элемента ys .

Если i – натуральное число, меньшее, чем $\text{length } xs$, которое совпадает с $\text{length } ys$, какой будет i -я точка набора данных?

Верно. Мы используем i в качестве индекса, как делали во фрейме 49:24.

Итак, для выборки пакета мы генерируем несколько случайных индексов, а затем выбираем соответствующие элементы из xs и ys . Мы называем эти случайные индексы пакетными индексами.

18 Как нам сгенерировать пакетные индексы?

Правило пакетов

Индексный пакет состоит из случайных индексов, представляющих собой натуральные числа меньше $\dagger xs \dagger$.

Воспользуемся функцией *samples*. Она получает два аргумента – n и s .

Первый – это количество точек данных в наборе.

Второй – это размер выборки.

Оба аргумента представляют собой натуральные числа больше 1, и $s \geq n$.

19 Почему мы назвали нашу функцию *samples*?

Потому что она выбирает пакет индексов тем же способом, которым Мария выбирала случайные яблоки на пробу.

20 Понятно.

Вызов функции *samples* возвращает список из *s* элементов, каждый из которых является случайно выбранным индексом меньше *n*. Ниже мы извлекаем пакет из 3 индексов из набора данных длиной 20:

(*samples* 20 3)

Так выглядит результат с учетом того, что индексы будут случайными:

(list 18 2 11)

21 Как определена функция *samples*?

Чтобы определить *samples*, нам сначала нужно научиться случайным образом генерировать натуральные числа для индексов.

Для этого мы воспользуемся функцией *random*, которая получает один аргумент *n* – положительное натуральное число и генерирует случайно выбранный индекс в интервале от 0 до *n* – 1.

Например,

(*random*[†] 45)

может сгенерировать

31

или любой другой индекс от 0 до 44.

22 Понятно.

Как использовать *random* для генерации списка индексов?

[†] При каждом вызове *random* будет сгенерировано новое натуральное число.

Так выглядит каркасный код *samples*:

```
(define samples
  (λ (n s)
    (sampled n s (list))))

(define sampled
  (λ (n i a)
    (cond
      ((zero? i) a)
      (else
       (sampled n (sub1 i)
                 (cons R a))))))
```

Здесь мы используем вспомогательную функцию *sampled*, которая ведет обратный отсчет от *s* до 0 и на каждом шаге накапливает случайно сгенерированный индекс.

Найдите *R*.

Отлично.

Так выглядит окончательная версия *samples*:

```
(define samples
  (λ (n s)
    (sampled n s (list))))

(define sampled
  (λ (n i a)
    (cond
      ((zero? i) a)
      (else
       (sampled n (sub1 i)
                 (cons (random n) a))))))
```

Удовлетворяет ли *sampled* требованиям к простой передаче аккумулятора, упомянутым на стр. 40?

²³ Поскольку мы накапливаем случайно сгенерированные индексы, вместо *R* нужно подставить

(*random n*)

²⁴ Конечно.

Вызов *sampled* не обернут, поэтому мы обращаем внимание только на формальные аргументы функции:

n не изменяется,
i изменяется при прохождении основного теста,
a накапливает результат[†].

[†] Так как каждое число в *a* выбрано случайно, порядок индексов в пакете не имеет значения.

Прекрасно.

25 Нет, не будет.

Если *samples* выбирает индексы случайным образом, будут ли вызов

(*samples* 20 3)

возвращать одинаковый результат при каждом вызове?

Это так в большинстве случаев[†]. При каждом обновлении *samples* возвращает новый пакет индексов. При достаточно большом размере набора данных n шанс случайно получить те же самые s индексов очень мал.

26 Итак, теперь, когда мы знаем, как случайным образом выбирать индексы, как мы можем использовать их для получения пакета из набора данных?

[†] Одиночный вызов *samples* тоже может вернуть повторяющиеся индексы, поскольку каждый индекс выбирается независимо от других.

Нам потребуется кое-что новое!

27 Можно увидеть пример?

Напомним, $t|_i$ выбирает i -й элемент тензора t , где i – индекс.

Теперь покажем, как можно выбрать из тензора t больше одного элемента. Пусть элементы, которые будут выбраны, заданы пакетом индексов b . Извлечем их из t при помощи операции

$t||_b$

Мы получим тензор того же ранга, что и t , но содержащий только те элементы t , которые определены пакетом индексов b .

Конечно.

Пусть t – это `tensor1` из 7 элементов:

[5.0 2.8 4.2 2.3 7.4 1.7 8.1]

Так выглядит список b из четырех индексов:

(list 6 0 3 1)

28 Что делать дальше со списком индексов?

Вот что:

1. $t|_b$
2. [5.0 2.8 4.2 2.3 7.4 1.7 8.1]_(list 6 0 3 1)
3. [8.1 5.0 2.3 2.8]

29 Может ли t иметь ранг больше 1?

Да, ранг t может быть больше 1. Вот пример, где t представляет собой `tensor2`, а b – список из предыдущего примера.

30 Это полезно! Готовы ли мы теперь определить вспомогательную функцию, о которой шла речь во фрейме 16?

1. $t|_b$
2. $\begin{bmatrix} 5.0 & 1.0 & 2.1 \\ 2.8 & 3.3 & 7.4 \\ 4.2 & 6.7 & 8.2 \\ 2.3 & 3.4 & 5.1 \\ 7.4 & 8.0 & 9.1 \\ 1.7 & 3.0 & 2.7 \\ 8.1 & 9.3 & 5.4 \end{bmatrix}_{(7\ 3)} \quad ||_{(list\ 6\ 0\ 3\ 1)}$
3. $\begin{bmatrix} 8.1 & 9.3 & 5.4 \\ 5.0 & 1.0 & 2.1 \\ 2.3 & 3.4 & 5.1 \\ 2.8 & 3.3 & 7.4 \end{bmatrix}_{(4\ 3)}$

Конечно готовы!

Так выглядит каркасный код функции *sampling-obj*, которая получает три аргумента. Первый из них – *expectant* – ожидающая функция, а остальные два – *xs* и *ys* – образуют набор данных:

```
(define sampling-obj
  (λ (expectant xs ys)
    (let ((n † xs †))
      (λ (θ)
        (let ((b B))
          ((expectant X Y) θ))))))
```

31 Мне кажется, эта функция возвращает другую функцию
(λ (θ) ...)

Правильно.

Она возвращает целевую функцию, которая делает выборку из набора данных, вместо того чтобы использовать его целиком.

Подставьте выражения вместо *B*, *X* и *Y*.

32 В этом каркасном коде *b* должен быть пакетом индексов (списком), а *X* и *Y* должны соответствовать выборкам, извлеченным из набора данных с помощью упомянутых индексов.

Поэтому вместо *B* должен быть список индексов выборки. Но нам не хватает важной информации.

Какого размера должен быть пакет индексов?

Очень хороший вопрос.

Размер пакета зависит от набора данных и обучаемой функции, с которыми мы работаем.

33 Значит, мы должны объявить этот размер как гиперпараметр?

Да.

```
(declare-hyper batch-size)
```

А что такое B во фрейме 31?

34 Выражение B генерирует пакет индексов с помощью $samples$ с количеством точек в наборе данных $\uparrow xs \downarrow$ в качестве первого аргумента и $batch-size$ в качестве второго. Но в теле выражения `let  $\uparrow xs \downarrow$` ассоциирован с n . Следовательно, вместо B нужно подставить $(samples\ n\ batch-size)$

Прекрасно.

Что насчет X и Y ?

35 Эти выражения должны использовать b для выборки соответствующего тензора из формальных аргументов xs и ys .
Итак, вместо X подставляем $xs|_b$ и вместо Y подставляем $ys|_b$

Правильно.

Вот окончательный вариант *sampling-obj*:

```
(define sampling-obj
  (λ (expectant xs ys)
    (let ((n  $\uparrow xs \downarrow$ ))
      (λ (θ)
        (let ((b (samples n batch-size)))
          ((expectant xs|b ys|b) θ))))))
```

36 Хорошо, но как использовать *sampling-obj*?

Теперь при вызове функции *gradient-descent* мы передаем ей *sampling-obj*.

Например:

```
(with-hypers
  ((revs 1000)
   ( $\alpha$  0.01)
   (batch-size 4))
 (gradient-descent
  (sampling-obj
   (l2-loss line) line-xs line-ys)
  (list 0.0 0.0)))
```

Здесь наш новый гиперпараметр *batch-size* представлен скалярным значением 4.

37 Это означает, что при каждом обновлении для измерения потерь мы используем только пакет данных размером 4.

Таким образом, мы проверяем всего лишь

($\ast$ 4 1000)

точек, вместо того чтобы просматривать весь набор данных при каждом обновлении.

И даже если набор данных содержит миллиарды точек, каждое обновление просматривает только 4, верно?

Верно на 100 %!

При каждом обновлении происходит выборка всего

batch-size точек.

Разновидность градиентного спуска, в котором целевая функция использует выборки, называется стохастический градиентный спуск.

38 Что означает слово стохастический?

Закон о размере пакета

Каждое обновление в стохастическом градиентном спуске использует только пакет размера *batch-size* из набора данных, а ранги тензоров в пакете такие же, как ранги тензоров в наборе данных.

Своевременный вопрос.

Стохастический – это еще один способ сказать, что мы используем случайные числа для определения наших результатов. Использование выборок как части нашей целевой функции делает ее стохастической.

39 Работает ли стохастический градиентный спуск при поиске разных целевых функций?

Конечно работает! Вернемся к примеру, рассмотренному во фрейме 102:25, где целевой функцией является *plane*.

Как выглядит наша ожидающая функция?

40 Так как наша целевая функция – *plane*, ожидающая функция должна быть
(*l2-loss plane*)

Правильно.

Пусть

revs будет 15000,

α будет 0.001

и

batch-size будет 4.

41 Тогда наши гиперпараметры должны выглядеть так:

```
(with-hypers
  ((revs 15000)
   ( $\alpha$  0.001)
   (batch-size 4))
...)
```

Из фрейма 105:34 позаимствуем начальное значение θ

```
(list [0.0 0.0] 0.0)
```

Как будет выглядеть вызов *gradient-descent*?

42 Итоговый вызов *gradient-descent* выглядит так:

```
► (with-hypers
  ((revs 15000)
   ( $\alpha$  0.001)
   (batch-size 4))
  (gradient-descent
   (sampling-obj
    (l2-loss plane) plane-xs
    plane-ys)
   (list [0.0 0.0] 0.0)))
► (list [3.98 1.97] 6.16)
```

Отлично.

На этом пока остановимся.

43 Ура! Предвкушаю отличный перерыв.

Случайные побрякушки

samples стр. 120

$t||_b$ стр. 121

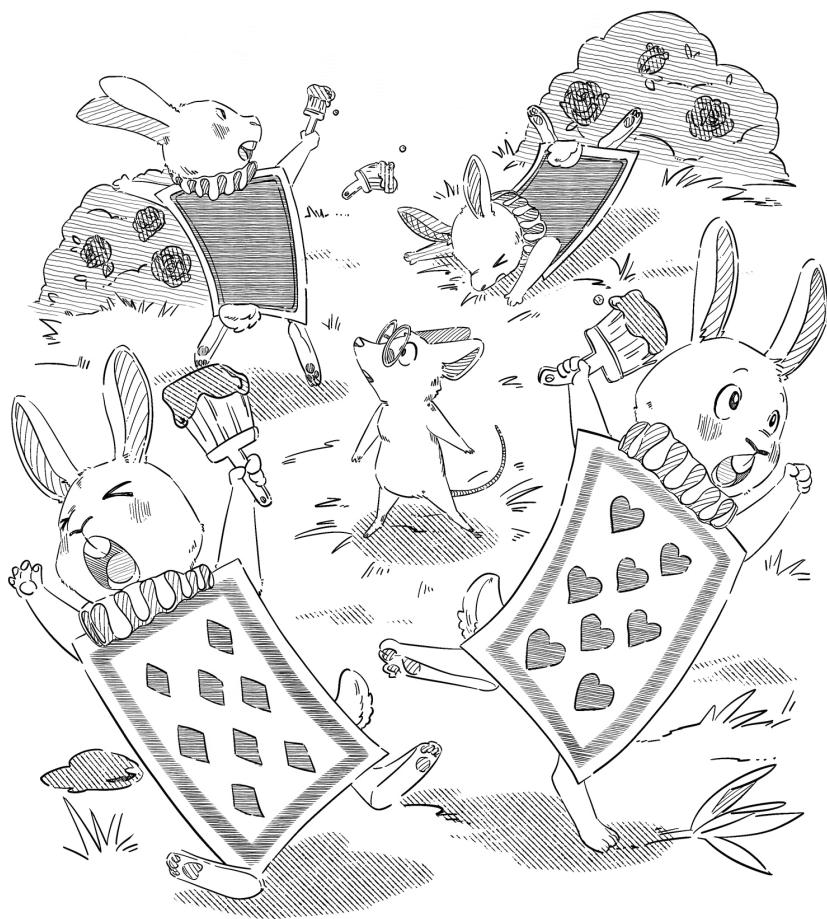
batch-size стр. 124

sampling-obj стр. 124

**Давайте возьмем по куску
яблочного пирога!**

**ЖЕЛАТЕЛЬНО С ШАРИКОМ
МОРОЖЕНОГО!**

7. СҮЕТА ВОКРУГ ХВОСТА



Как вам яблочный пирог?	¹ Теплый, мягкий и с шариком ванильного мороженого.
Ммм... Великолепно. А теперь мы выучим несколько новых приемов использования функции <i>gradient-descent</i>	² Что за приемы?
Приемы, которые делают эту функцию более гибкой и позволяют изменить ее поведение.	³ Но зачем нам менять ее поведение?
Например, мы сможем получить точно подобранные параметры θ всего за несколько обновлений (т. е. с меньшим значением <i>revs</i>).	⁴ Веский аргумент. Что нам понадобится, чтобы сделать эту функцию более гибкой?
Одна из причин, по которой <i>gradient-descent</i> из фрейма 96:5 не может добраться до точно выбранного θ с меньшим количеством обновлений, заключается в том, что ей недостаточно информации для работы.	⁵ Какая дополнительная информация нужна?
Прежде чем мы перейдем к деталям, важно отметить, что мы должны найти способ хранить эту информацию и обновлять ее по мере изменения параметра. Прежде всего нужно помнить, что эта дополнительная информация связана с каждым параметром в θ . Мы будем называть эту информацию сопровождающей, а соответствующий параметр – сопровождаемым. Мы будем продвигаться к пониманию шаг за шагом.	⁶ Хорошо.

Начнем с нашей привычной функции *gradient-descent*, у которой остается пунктирная линия:

```
(define gradient-descent
  (λ (obj θ)
    (let ((f (λ (Θ)
               (map (λ (p g)
                     (- p (* α g)))
                    Θ
                    (∇ obj Θ))))))
      (revise f revs θ))))
```

7 И мы наконец узнаем, почему во фрейме 87:48 было сказано, что Θ вселяет большие надежды?

Да, наконец настал этот момент!

Что в этой функции делает *map*?

8 Здесь *map* вызывает функцию

```
(λ (p g)
  (- p (* α g)))
```

для каждого элемента

Θ

и

```
(∇ obj Θ)
```

попарно.

Правильно.

В этой версии *gradient-descent* мы используем *revise*, чтобы повторно пересчитывать и обновлять список параметров.

Подобно *map*, функция *revise* является функцией общего назначения. Мы можем использовать ее для проверки любого значения.

Пока мы тщательно следим за тем, чтобы *gradient-descent* получала θ и в конечном итоге приводила к точно подобранному θ , не имеет значения, как мы трансформируем θ по пути.

9 Можно ли познакомиться с примером обновления другого типа значений?

Закон об обновлениях

Пока мы уверены, что градиентный спуск принимает начальный набор параметров θ и приводит к точно подобранному θ , допускается любой разумный способ обновления на всем протяжении, от первого до последнего шага.

Конечно. Вот пример, по большей части бесполезный, но он иллюстрирует прием преобразования параметров путем их обертывания и последующего развертывания[†].

Каким будет значение выражения?

```
(map (λ (p)
      (list p))
      θ)
```

[†] Такое использование терминов «обертывание» и «развертывание» для параметров отличается от их использования при описании рекурсивных определений в главе 2.

¹⁰ Оно обертывает каждый параметр p в θ в одноэлементное множество (синглтон, singleton)[‡] и формирует список этих множеств.

[‡] Список с одним элементом.

Да.
 Всякий раз, когда параметр включен в список, мы называем его сопровождаемым параметром[†].

[†] С 1964 года джазовый квинтет Майлза Дэвиса мог бы называться «Синглтоном Майлза Дэвиса», если бы не его напарники-аккомпаниаторы Тони Уильямс (1945–1997), Уэйни Шортер (1933–), Рон Картер (1937–) и Герби Хэнкок (1940–). Но ключевым параметром всегда был Майлз Дэвис (1926–1991). Спасибо им всем за прекрасный джаз.

¹¹ Но здесь ничего не сопровождает параметр в этом списке, правильно?

Да, правильно. Параметр пре-
бывает в одиночестве.

Это наше одиночное пред-
ставление[†]. Θ представляет
собой список сопровождае-
мых параметров, но в данном
одиночном представлении
сопровождения нет.

12 Как мы будем использовать
одиночное представление?

[†] Одиночное представление просто
показывает пример, в котором список
не имеет сопровождения. Мы доба-
вим информацию, сопровождающую
параметры, чтобы получить более
содержательные представления.

Мы будем использовать
его для обновления внутри
gradient-descent.

Но сначала обеспечим возмож-
ность преобразования между
 Θ и θ туда и обратно. Для этого
нам понадобятся две вспо-
могательные функции. Ниже
показана первая из них. Она
получает θ и конвертирует в Θ .

Мы назовем ее *lonely-i*, где *i* –
от слова *inflate* (раздуваться):

```
(define lonely-i
  (lambda (theta)
    (map (lambda (p)
          (list p))
         theta)))
```

Объясните, как она работает.

Правильно.

Теперь самостоятельно опре-
делите функцию *lonely-d*, где
d – от слова *deflate* (сжимать-
ся), выполняющую обратное
действие. Она должна полу-
чать Θ и возвращать θ .

13 Эта функция получает θ и
обертывает каждый параметр
в список, предоставляя нам Θ .

14 Итак, мы можем применить
map к Θ и отделить обнов-
ленный параметр от его
сопровождаемого параметра:

```
(define lonely-d
  (lambda (Theta)
    (map (lambda (P)
          P_0)
         Theta)))
```

Здесь P – сопровождаемый
параметр, верно?

Да, верно.	15	Как можно отличить P от p ?
Θ – это список P , а θ – это список p !	16	Ого! Так намного понятнее!
Так как теперь мы обновляем Θ, а не θ, объявим функцию, которая выполняет это обновление: <pre> (define lonely-u (λ (Θ gs) (map (λ (P g) (list (- P₀ (* α g))) Θ gs))) </pre> Буква u здесь означает <i>update</i> (обновление), потому что эта функция относится к функциям обновления. Объясните, как работает <i>lonely-u</i>.	17	Эта функция попарно проходит по всем Θ и градиентам gs. Так как каждый элемент Θ представляет собой одноэлементный список (синглтон), мы извлекаем параметр из сопровождаемого параметра и используем градиент и скорость обучения для получения обновленного параметра. Но зачем обертывать результат в список?
Мы делаем это, потому что <i>lonely-u</i> должна вернуть обновленный Θ , а не θ .	18	Ага! Обертывание результата в список гарантирует, что точно подобранный Θ останется списком синглтонов.
Мы используем эти три функции <pre> lonely-i lonely-d </pre> и <pre> lonely-u </pre> для обобщения функции <i>gradient-descent</i>.	19	Как это можно сделать?

Вместо того чтобы разбрасывать три *ate*-функции по всей функции *gradient-descent*, мы передаем их в более обобщенную форму *gradient-descent*:

```
(define gradient-descent
  (λ (inflate deflate update)
    (λ (obj θ)
      ...)))
```

Теперь определите функцию *lonely-gradient-descent*, используя новую форму

gradient-descent.

20 Вот она:

```
(define lonely-gradient-descent
  (gradient-descent
    lonely-i lonely-d lonely-u))
```

Выглядит как стихотворение, не правда ли?

Да, занятно.

Запишем более обобщенный каркасный код функции *gradient-descent*:

```
(define gradient-descent
  (λ (inflate deflate update)
    (λ (obj θ)
      (let ((f (λ (Θ)
                  ( ( U
                     Θ
                     (∇ obj
                      D ) ) ) ) )
        ( ( R
          (revise f revs
            I ) ) ) ) ) ) ) )
```

Найдите *I*, *D*, *U* и *R*.

21 Поскольку мы обновляем список синглтонов, это означает, что мы должны вызывать *revise* с начальным Θ . Значит, вместо *I* надо подставить

(inflate θ)

D является вторым аргументом ∇ , поэтому должен быть списком параметров. Значит, мы должны конвертировать Θ обратно в θ . Итак, вместо *D* подставляем

(deflate Θ)

U необходимо вызвать для обновления Θ , поэтому вместо *U* подставляем

update

Наконец, *gradient-descent* должна вернуть θ . Но вызов *revise* возвращает Θ . Поэтому мы должны конвертировать точно подобранный Θ в θ . Так что вместо *R* подставляем

deflate

Превосходно.

Вот предпоследний вариант функции *gradient-descent*:

```
(define gradient-descent
  (λ (inflate deflate update)
    (λ (obj θ)
      (let ((f (λ (Θ)
                  (update
                   Θ
                   (∇ obj
                    (deflate Θ))))))
        (deflate
         (revise f revs
          (inflate θ)))))))
```

22 Но пунктирная линия осталась!

Да, верно.

Ведь мы должны проверить функцию, прежде чем утверждать, что закончили. Для этого нужно работать с целой кучей разных значений *inflate*, *deflate* и *update*, поэтому было бы разумно немного облегчить себе задачу.

Определите функцию *try-plane*, получающую функцию-аргумент *a-gradient-descent*, которая может быть вызвана внутри выражения *with-hypers* во фрейме 127:42.

23 Вот новое определение с пунктиром:

```
(define try-plane
  (λ (a-gradient-descent)
    (with-hypers
     ((revs 15000)
      (α 0.001)
      (batch-size 4))
     (a-gradient-descent
      (sampling-obj
       (l2-loss plane)
       plane-xs plane-ys)
      (list [0.0 0.0] 0.0)))))
```

Мы используем *try-plane* с конкретной функцией градиентного спуска, например

lonely-gradient-descent

24 Это очень удобное определение!

А вот код обновления:

```
▷ (try-plane lonely-gradient-descent)
▶ (list [3.98 1.97] 6.16)
```

Это превосходно!

Несмотря на то что в этом процессе много обертываний и развертываний, он все равно приходит к тому же результату, что и раньше, во фрейме 127:42.

25 Это удобная функция, чтобы периодически выполнять тестирование.

Рассмотрим другое представление, где Θ идентичен θ . Мы будем называть это представление открытым (*naked*).

В этом случае сжимаемый (*deflate*) p и расширяемый (*inflate*) P идентичны.

Определите функцию расширения.

26 Даже если θ идентичен Θ , мы должны стараться использовать шаблоны, поэтому *naked-i* отображает (*map*) функцию тождественности на θ :

```
(define naked-i
  (λ (θ)
    (map (λ (p)
          (let ((P p))
            P))
         θ)))
```

Аналогично, *naked-d* получает P , который нужно расширить, и возвращает идентичный сжатый p .

Запишите определение функции сжатия для этого представления.

27 Используя тот же шаблон, получаем:

```
(define naked-d
  (λ (Θ)
    (map (λ (P)
          (let ((p P))
            p))
         Θ)))
```

Точно!

Учитывая, что θ и Θ одинаковые, определите *naked-u*.

28 θ и Θ одинаковые, поэтому *naked-u* выглядит так:

```
(define naked-u
  (λ (Θ gs)
    (map (λ (P g)
          (- P (* α g)))
         Θ
         gs)))
```

Почему бы нам не использовать P_0 вместо P ?

Нет, использование P_0 предполагало бы, что P является сопровождаемым.

29 Таким образом, P не является сопровождаемым, потому что *naked-i* просто отображает функцию тождественности.

Теперь определите *naked-gradient-descent*.

30 Вот определение:

```
(define naked-gradient-descent
  (gradient-descent
   naked-i naked-d naked-u))
```

Отлично.

Перепишите пример из фрейма 24, на этот раз используя открытое представление.

31 Новый код будет таким:

```
▷ (try-plane naked-gradient-descent)
▶ (list [3.98 1.97] 6.16)
```

Но определения для одиночного и открытого представлений по-прежнему с пунктиром!

Да, это так. Мы их упростим. Чтобы сделать это, представьте нечто особенное. Допустим, определения *inflate* и *deflate* выглядят, как показано ниже:

```
(define inflate
  (λ (θ)
    (map (λ (p)
          ...)
         θ)))
```

```
(define deflate
  (λ (Θ)
    (map (λ (P)
          ...)
         Θ)))
```

32 Вот так:

```
(define update
  (λ (Θ gs)
    (map (λ (P g)
          ...)
         Θ
         gs)))
```

Но почему это важно?

Как будет выглядеть *update*?

Отличный вопрос!	33	Как такое возможно?
Теперь о том, что не давало нам покоя в <i>ate</i> -функциях из фрейма 20. Если мы внесем последовательные изменения в <i>gradient-descent</i> с пунктиром, то сможем упростить каждое определение.		
Обратите внимание, что мы передавали аргументы в каждую <i>ate</i> -функцию и затем уже поверх них вызывали <i>map</i> .	34	Да, это так.
Мы можем использовать это наблюдение и переместить <i>map</i> таким образом, чтобы упростить <i>ate</i> -функции.	35	Понятия не имею, как это сделать.
Есть какие-либо соображения?		
Возьмем функцию <i>inflate</i> . Наша текущая функция <i>gradient-descent</i> вызывает <i>inflate</i> для θ так: $(\text{inflate } \theta)$ Поскольку функции <i>inflate</i> , которые мы видели до сих пор, полагаются на тот факт, что θ является списком, и каждая из них отображает некоторое λ -выражение в этом списке, мы можем временно переписать этот вызов <i>inflate</i> следующим образом: $(\text{map } (\lambda (p) \dots) \theta)$	36	Получается, независимо от того, с какой функцией расширения мы сталкиваемся, она всегда начинается с вызова <i>map</i> .
Точно!	37	Будет ли этот прием работать с функциями
Поэтому мы можем упростить функции <i>inflate</i> , перенеся <i>map</i> в <i>gradient-descent</i> и оставив <i>inflate</i> лишь роль λ -выражения.		<i>deflate</i> и <i>update</i> ?

Да, и вот как.

Чтобы использовать

ate-функции в *gradient-descent*:

(...*ate* θ) становится (*map* ...*ate* θ),

(...*ate* Θ) становится (*map* ...*ate* Θ)

и

(...*ate* Θ *gs*) становится (*map* ...
ate Θ *gs*)

В последний раз доработайте пунктирную версию *gradient-descent*, но на этот раз с учетом всего вышесказанного.

Перейдя к финальной версии *gradient-descent*, мы должны удалить *map* из определения каждой *ate*-функции.

Начнем с *lonely-i* во фрейме 13:

```
(define lonely-i
  (λ (θ)
    (map (λ (p)
          (list p))
         θ)))
```

Как нужно доработать эту функцию, чтобы она соответствовала финальной версии *gradient-descent*?

А как насчет *lonely-d* и *lonely-u*?

38 Вот финальная версия функции *gradient-descent*:

```
(define gradient-descent
  (λ (inflate deflate update)
    (λ (obj θ)
      (let ((f (λ (Θ)
                  (map update
                       Θ
                       (λ obj
                        (map deflate Θ))))))
        (map deflate
         (revise f revs
          (map inflate θ)))))))
```

39 Теперь *gradient-descent* отвечает за применение функции расширения к θ , поэтому определение *lonely-i* становится таким:

```
(define lonely-i
  (λ (p)
    (list p)))
```

40 Теперь они выглядят так:

```
(define lonely-d
  (λ (P)
    P0))

(define lonely-u
  (λ (P g)
    (list (- P0 (* α g)))))
```

Закон об *ate*-функциях

В любом своем представлении три *ate*-функции связаны только с одним параметром и его сопровождением и не связаны напрямую ни с θ , ни с Θ .

Правильно.

Вот новое определение функции *lonely-gradient-descent*:

```
(define lonely-gradient-descent
  (gradient-descent
   lonely-i lonely-d lonely-u))
```

Каким будет код из фрейма 24?

Теперь у нас есть аналогичный опыт использования открытых представлений.

Доработайте

naked-i

naked-d

и

naked-u

41 Теперь он выглядит так:

▷ (*try-plane lonely-gradient-descent*)

► (list [3.98 1.97] 6.16)

42 Как вам нравятся такие *ate*-функции?

```
(define naked-i
  (λ (p)
    (let ((P p))
      P)))

(define naked-d
  (λ (P)
    (let ((p P))
      p)))

(define naked-u
  (λ (P g)
    (- P (* α g))))
```

Превосходно!

Запишем окончательный вариант определения функции *naked-gradient-descent*:

```
(define naked-gradient-descent
  (gradient-descent
   naked-i naked-d naked-u))
```

Каким будет вызов этой функции на реальном примере?

43 Вероятно, таким:

▷ (*try-plane naked-gradient-descent*)

► (list [3.98 1.97] 6.16)

Совершенно верно!

Пришло время завершить эту главу. В следующей главе мы воспользуемся нашим новым подходом к функции *gradient-descent*, чтобы сделать ее поведение еще лучше!

⁴⁴ Да, научим ее вести себя хорошо!

Поехавшие побрякушки

gradient-descent стр. [139](#)
naked-gradient-descent стр. [141](#)

Как насчет тщательно
взбитого суфле?

ХОЧЕТСЯ ПОПРОБОВАТЬ
ЭТО СЛАДКОЕ ОБЛАЧКО!

8. ЧЕМ БЛИЖЕ ЦЕЛЬ, ТЕМ МЕНЬШЕ ШАГ¹



¹ В названии главы обыграны слова из песни Пола Саймона *Slip Slidin' Away*.

Понравилось суфле?	1	О да, не заметил, как проглотил!
Сейчас мы выучим несколько новых трюков с новой версией <i>gradient-descent</i>	2	Что за трюки?
Мы научимся получать точно подобранные параметры θ всего за несколько обновлений.	3	Звучит многообещающе! Можно начинать?
Отправляемся на гонки! Точнее, к эстафете.	4	Это интригует. Почему именно эстафета?
Допустим, знаменитые «Братья Маркс» [†] решили участвовать в эстафете и выбрали тренером Граучо.	5	Это очень смешная команда.

[†] Комедийный коллектив братьев Маркс в составе: Леонард «Чико» (1887–1961), Артур «Харпо» (1888–1964), Юлиус «Граучо» (1890–1977), Милтон «Гаммо» (1892–1977), Герберт «Зеппо» (1901–1979).		
Конечно. Чико немного быстрее, чем Харпо, который немного быстрее, чем Гаммо, который немного быстрее, чем Зеппо. Они бегут в перечисленном порядке; каждый бегун пробегает один круг и передает палочку следующему.	6	Зеппо – самый медленный из них.
Да. Чтобы получить хорошо подготовленную команду, Граучо должен заставить медленных бегунов двигаться быстрее.	7	Это непростая задача.

Граучо придумал хитрую, возможно слегка нечестную, тренерскую стратегию.

Вместо того чтобы отпустить эстафетную палочку, передавая ее Харпо, Чико некоторое время продолжает бежать, когда Харпо уже схватился за палочку. Это заставляет Харпо пытаться бежать так же быстро, как Чико, пока они оба на дистанции.

Чико тащит Харпо за собой часть пути, что делает скорость Харпо намного выше.

8 Вот как!
Харпо перенимает часть скорости Чико.

Точно.
Затем Харпо проделывает тот же трюк с Гаммо.

9 Гаммо перенимает часть скорости Харпо.

И немного скорости Чико!

10 Ох, и правда.
Если Харпо получает часть скорости Чико, это можно сказать и про Зеппо?

Да, можно.

11 Теперь медленные бегуны стали немного быстрее, потому что быстрые бегуны поделились с ними своей скоростью.

Да.
Граучо удалось поднять низкую скорость у бегунов, распределив ее между ними.
Теперь все вместе они могут пробежать эстафету быстрее.

12 Но какое отношение к этому имеет *gradient-descent*?

Помните функцию *naked-u* из фрейма 141:42?

```
(define naked-u
  (λ (P g)
    (- P (* α g))))
```

13 Да.

Это часть, отвечающая за обновление в открытом представлении *gradient-descent*.

Поясните, как эта функция обновляет параметры.

14 Эта функция умножает градиент g на скорость обучения α и вычитает результат из параметра P , чтобы получить следующий P , так что в конечном итоге мы приближаемся к точно подобранному θ .

Верно.
Теперь вспомните график потерь из фрейма 74:13.

15 Да, к нему были проведены две касательные.

О нем и речь.
Что можно сказать о касательных, которые приближаются к нижней точке графика?

16 Они становятся все более пологими по мере приближения к нижней точке кривой.

Точно.
Наклон касательной (т. е. градиент) становится меньше. Фактически при движении к нижней точке кривой значение градиента становится все ближе к 0.0. В самой нижней точке значение градиента точно равно нулю.

17 Ух ты, результат будет еще меньше!

Что произойдет, если мы перемножим совсем небольшой градиент с маленьким значением скорости обучения, как мы это делаем

Значит, с каждым обновлением, приближающим нас к нижней точке, величина изменения каждого параметра становится меньше и меньше, верно?

(αg)

в функции обновления?

Верно.

Изменение, которое мы вносим в определенный параметр с каждым обновлением, называется скоростью спуска.

Итак, что мы можем сказать о скорости спуска, когда приближаемся ко дну кривой?

- 18 Скорость существенно снижается!
Постойте! Это очень похоже на нашу эстафетную команду!

Но мы можем увеличить скорость процесса в целом, используя хитрую стратегию Граучо!

- 19 Как именно?

Еще раз взгляните на функцию *naked-u* из фрейма 141:42.
(define naked-u
 (λ (P g)
 ($- P$ ($* \alpha g$))))

- 20 Так как мы вычитаем ($* \alpha g$), то изменение P (т. е. скорость) будет равно
($- (* \alpha g)$)

Что является скоростью в этом λ -выражении?

Правильно.
Стратегия Граучо подразумевает, что мы должны увеличить скорость добавлением некоторой величины μ к скорости v предыдущего обновления, чтобы получить ожидаемое изменение в текущем обновлении.

- 21 Что это означает для нашей скорости?

Новое выражение для вычисления скорости приобретает вид

- 22 Является ли μ гиперпараметром?

($+ (* \mu v) (- (* \alpha g))$)
и его можно записать более наглядно:

($- (* \mu v) (* \alpha g)$)

Да.

23 Откуда берется v ?

Объявим его:

```
(declare-hyper  $\mu$ )
```

Значение гиперпараметра μ располагается в интервале между 0.0 и 1.0 и представляет собой десятичную долю от предыдущей скорости, которую мы хотим сохранить для следующей скорости[†].

[†] Рекомендуемое скалярное значение μ – обычно около 0.9.

Здесь v представляет собой скорость самого последнего обновления (т. е. скорость бегуна с эстафетной палочкой).

24 Сопровождает ли скорость соответствующий параметр модели?

Да, так и есть.

Что вы скажете о форме v ?

25 Так как скорость – это изменение соответствующего (сопровождаемого) параметра, она должна иметь форму этого параметра.

Правильно!

Вот расширяющая функция *velocity-i*:

26 Что такое *zeroes*?

```
(define velocity-i  
  ( $\lambda$  ( $p$ )  
    (list  $p$  (zeroes  $p$ ))))
```

Эта функция добавляет начальное сопровождающее значение к параметру p .

Функция *zeroes* генерирует тензор с той же формой, что у аргумента, но все элементы этого тензора равны нулю. Вот логический столбец данной операции:

1. $\left(\text{zeroes} \begin{bmatrix} 2.1 & 9.3 & 1.5 \\ 7.2 & 3.3 & 6.6 \end{bmatrix}_{(2\ 3)} \right)$
2. $\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}_{(2\ 3)}$

Объясните, как работает функция *velocity-i*.

- 27 Она возвращает нулевой тензор той же формы, что и *p*. Почему мы решили, что скорость должна быть нулевым тензором?

Потому что при первом обновлении не было предыдущих изменений какого-либо параметра[†]. Так что использование нулевого тензора является вполне обоснованным выбором для начальной скорости.

- 28 Понятно.

[†] В соответствии с законом на стр. 141 мы можем перестать думать о θ и Θ и сосредоточиться только на параметрах.

Теперь определите соответствующую сжимающую функцию *velocity-d*.

- 29 Эта сжимающая функция должна возвращать параметр, сопоставленный с индексом 0 в расширенном представлении:

```
(define velocity-d
  ( $\lambda$  (P)
    P0))
```

Прекрасно.

Теперь мы можем определить *velocity-u*. Она ожидает сопровождаемый параметр P , где параметр P_0 сопровождается его скоростью P_1 из последнего обновления. Как и во всех функциях обновления, градиент g является вторым аргументом.

Вот каркасный код функции *velocity-u*:

```
(define velocity-u
  (λ (P g)
    (let ((v V)))
      (list (+ P0 v) v))))
```

Найдите V .

Правильно.

Так выглядит объявление *velocity-u*

```
(define velocity-u
  (λ (P g)
    (let ((v (- (* μ P1) (* α g))))
      (list (+ P0 v) v))))
```

Теперь определите *velocity-gradient-descent*.

Теперь придадим *try-plane* более общую форму (хотя пунктир останется) за счет введения дополнительного аргумента a -*revs*. Это позволит нам более легко экспериментировать с различными скалярными значениями гиперпараметра *revs*:

30 Мы передаем имя v в выражение V . Тело выражения `let` представляет собой обновленный сопровождаемый параметр, который возвращается из функции. Сопровождением в данном случае является v . Следовательно, вместо V должна быть скорость, изменение которой мы должны использовать для получения P_0 .

Выражение скорости возьмем из фрейма 22:

```
(- (* μ P1) (* α g))
```

31 Легко.

Мы передаем *velocity-i*, *velocity-d* и *velocity-u* в *gradient-descent*:

```
(define velocity-gradient-descent
  (gradient-descent
   velocity-i velocity-d velocity-u))
```

32 Ого.

Мы снова нашли тот же ответ, но с *revs*, равным 5000 вместо 15 000, благодаря тому что использовали скорость вместе с градиентом, чтобы решить, какие изменения вносить в θ каждый раз.

```
(define try-plane
  (λ (a-gradient-descent a-revs)
    (with-hypers
      ((revs a-revs)
       (α 0.001)
       (batch-size 4))
      (a-gradient-descent
       (sampling-obj
        (l2-loss plane) plane-xs
        plane-ys)
       (list [0.0 0.0] 0.0)))))
```

Наверное, эта разновидность градиентного спуска называется скоростным градиентным спуском?

Ниже показан пример кода обновления с использованием этой модификации, который показывает, как вызвать *a-gradient-descent* для примера во фрейме 136:23:

```
▷ (with-hypers
  ((μ 0.9))
  (try-plane
   velocity-gradient-descent
   5000))
▶ (list [3.98 1.97] 6.16)
```

На самом деле нет.

33 Какое странное название.
При чем тут импульс?

Его называют

импульсным градиентным спуском[†].

[†] Дэвид Румельхарт (1942–2011),
Джеффри Хинтон (1947–),
Рональд Уильямс (1944–).

Вопрос в точку.

34 Существуют ли другие
способы увеличить скорость
спуска?

Дело в том, что мы умножаем скорость v на константу μ . Это выражение аналогично формуле момента в физике[†].

[†] $\mathbf{p} = m\mathbf{v}$, где $\mathbf{p}$ – момент, m – масса
объекта, v – скорость.

Да, конечно, но это тема
следующей интерлюдии!

Так что мы перенесем
немного своей скорости из
этой главы в интерлюдию.

³⁵ Не съесть ли нам что-нибудь
вкусное?

Скоростные побрякушки

μ стр. [144](#)

velocity-gradient-descent стр. [151](#)

Как насчет кусочка пирога
с фиником и пеканом?

ЭТО ЛЮБИМОЕ ЛАКОМСТВО ГРАУЧО!

ИНТЕРЛЮДИЯ IV. ОПЕРАТОР SMOOTH¹



¹ Спасибо Лили Томлин (1939–) за ее роли Эрнестины и Эдит Энн; а также Шаде Аду (1959–), Дэйву Эрли (1957–1996), Полу Спенсеру Данману (1957–), Стюарту Колину Мэтьюману (1960–) и Эндрю Хейлу (1962–).

Пирог с финиками и орехами
пекан от Граучо?†

1 Теперь мы знаем, почему это
любимый десерт Граучо!

† Спасибо Розмари Вильсон (1910-
1986) и Рут Пул (1934-).

Хорошо.

2 Так что же такое сглажива-
ние?

Прежде чем мы рассмотрим
другие алгоритмы обновле-
ния, мы должны понять, что
такое сглаживание (*smoothing*)

Взгляните на функцию *smooth*

3 Поскольку *decay-rate* является
скаляром между 0.0 до 1.0,
похоже, что функция смешива-
ет два тензора, используя
decay-rate
и

```
(define smooth)
  (λ (decay-rate average g)
    (+ (* decay-rate average)
      (* (- 1.0 decay-rate) g))))
```

где *decay-rate* всегда должен
быть скаляром между 0.0
и 1.0, *average* – историческое
накопительное среднее† и
g – градиент. Оба последних
аргумента должны быть тен-
зорами одинаковой формы.
Что делает *smooth*?

(*- 1.0 decay-rate*)

в качестве весов.

† Сокращается до «среднеисторичес-
кого».

Совершенно верно.

4 Этот вызов функции *smooth*
вернет скаляр 5.03

Рассмотрим расширенный
пример использования этой
функции. Вот семь скалярных
значений:

50.3 22.7 4.3 2.7 1.8 2.2 0.6

Найдите

(*smooth* 0.9 0.0 50.3)

- | | | |
|----|--|--------------------------------------|
| 1. | | (<i>smooth</i> 0.9 <u>0.0</u> 50.3) |
| 2. | | (+ (* 0.9 <u>0.0</u>) (* 0.1 50.3)) |
| 3. | | (+ 0.0 5.03) |
| 4. | | <u>5.03</u> |

Правильно.

А как насчет такого вызова?

(*smooth* 0.9 5.03 22.7)

⁵ Мы смешиваем скаляр 5.03 со следующим скаляром 22.7 в указанной последовательности.

Вот как это можно записать:

1. | (*smooth* 0.9 5.03 22.7)
2. | (+ (* 0.9 5.03) (* 0.1 22.7))
3. | (+4.53 2.27)
4. | 6.8

Да, это верно.

Выполним вызов еще раз.

Найдите

(*smooth* 0.9 6.8 4.3)

⁶ Теперь получится

1. | (*smooth* 0.9 6.8 4.3)
2. | (+ (* 0.9 6.8) (* 0.1 4.3))
3. | 6.55

Великолепно.

Вот семь исходных скаляров из фрейма 4

50.3 22.7 4.3 2.7 1.8 2.2 0.6

и среднеисторические значения после сглаживания

5.03 6.8 6.55 6.16 6.07 5.64 5.14[†]

Сравните их с исходными скалярами.

⁷ Скаляры после функции *smooth* намного более «гладкие», чем исходные скаляры. Другими словами, сглаженные скаляры изменяются не так сильно, как исходные скаляры, и различия между ними гораздо мягче.

[†] Подчеркнутые скаляры – это результаты первых трех вызовов *smooth*, исключая начальное значение 0.0.

Да, семь скаляров в сглаженных среднеисторических значениях гораздо ближе друг к другу, чем соответствующие семь исходных скаляров.

Многократный вызов *smooth* для этих скаляров «усредняет» вариации путем смешивания исторических скаляров с новыми.

Чем важно значение 0.9, которое мы передаем как *decay-rate*?

⁸ Оно означает, что для нахождения нового среднеисторического значения мы используем только 90 % предыдущего среднеисторического значения, и каждый новый встреченный скаляр уменьшается до 10 % его значения. Таким образом, новые скаляры, которые сильно отклоняются от среднеисторического, вносят лишь малую долю в новое значение.

Всегда ли *decay-rate* остается равным 0.9?

Конечно нет, это просто пример.

Как правило, *decay-rate* принимает значения между 0.0 и 1.0.

⁹ Понятно.

Теперь рассмотрим первый скаляр 50.3. При первом вызове *smooth* мы умножаем его на 0.1 и получаем 5.03.

В следующий раз, когда мы вызовем *smooth*, этот скаляр умножится на 0.9. Это означает, что вклад первого скаляра в следующий сглаженный результат (округленный до двух десятичных знаков) составит

¹⁰ Ага!

По мере появления новых данных они вносят все меньший и меньший вклад в среднеисторическое значение.

$$0.9 \times 0.1 \times 50.3 = 4.53$$

-
- Да, так и есть.
- Мы называем *decay-rate* долей вклада. Вклад более ранних элементов уменьшается по мере появления большего количества элементов.
- Найдите формулу вклада скаляра 50.3 после n вызовов *smooth*.
- 11 После n вызовов *smooth* вклад будет равен
- $$0.9^{n-1} \times 0.1 \times 50.3$$
-
- Правильно.
- В общем, это справедливо для любого поступившего скаляра. Вклад этого скаляра убывает со временем согласно формуле.
- 12 Очевидно, сглаживание является способом включения исторической информации, которая становится менее актуальной по мере нашего продвижения вперед.
-
- Да, но это верно также для тензоров любого ранга!
- Поскольку *smooth* определена с использованием расширенных функций, аргументы *average* и *g* могут быть тензорами с совместимыми формами. В большинстве случаев они имеют одинаковую форму.
- И если это так, то сглаженный результат имеет ту же форму, что и аргументы.
- 13 Можно увидеть пример?
-
- Допустим, спустя некоторое время мы имеем среднеисторический тензор
- $$[0.8 \ 3.1 \ 2.2]$$
- и у нас появились три новых тензора
- $$[1.0 \ 1.1 \ 3.0]$$
- $$[13.4 \ 18.2 \ 41.4]$$
- $$[1.1 \ 0.3 \ 67.3]$$
- 14 Разумеется, при помощи *smooth*!
- Мы продолжим использовать значение *decay-rate*, равное 0.9?

Как мы должны смешать эти новые тензоры со среднеисторическим тензором?

Хорошая идея.

¹⁵ Вот результат:

Начнем с первого тензора:

1. $(smooth\ 0.9\ [0.8\ 3.1\ 2.2]\ [1.0\ 1.1\ 3.0])$
2. $(+ (*\ 0.9\ [0.8\ 3.1\ 2.2]) (*\ 0.1\ [1.0\ 1.1\ 3.0]))$

3. $(+ [0.72\ 2.79\ 1.98]\ [0.1\ 0.11\ 0.3])$
4. $[0.82\ 2.9\ 2.28]$

Завершите последовательность.

Правильно.

¹⁶ Пожалуйста:

Смешивая среднеисторический тензор

$[0.82\ 2.9\ 2.28]$

со вторым $tensor^1$

$[13.4\ 18.2\ 41.4]$

мы получаем

1. $(smooth\ 0.9\ [0.82\ 2.9\ 2.28]\ [13.4\ 18.2\ 41.4])$
2. $(+ (*\ 0.9\ [0.82\ 2.9\ 2.28]) (*\ 0.1\ [13.4\ 18.2\ 41.4]))$
3. $(+ [0.74\ 2.61\ 2.05]\ [1.34\ 1.82\ 4.14])$
4. $[2.08\ 4.43\ 6.19]$

1. $(smooth\ 0.9\ [2.08\ 4.43\ 6.19]\ [1.1\ 0.3\ 67.3])$
2. $(+ (*\ 0.9\ [2.08\ 4.43\ 6.19]) (*\ 0.1\ [1.1\ 0.3\ 67.3]))$
3. $(+ [1.87\ 3.99\ 5.57]\ [0.11\ 0.03\ 6.73])$
4. $[1.98\ 4.02\ 12.3]$

Теперь добавьте третий тензор

$[1.1\ 0.3\ 67.3]$

Очень хорошо.

Сравните три исходных `tensor`¹

[1.0 1.1 3.0]

[13.4 18.2 41.4]

[1.1 0.3 67.3]

с тремя сглаженными `tensor`¹

[0.82 2.9 2.28]

[2.08 4.43 6.19]

[1.98 4.02 12.3]

¹⁷ Это выглядит так, будто мы сгладили отдельные скаляры, соответствующие элементам этих тензоров.

Как мы будем использовать *smooth* с *gradient-descent*?

Это тема следующей главы!

¹⁸ Жду с нетерпением.

Мягкие игрушки

smooth стр. [154](#)

Пить чай или не пить?

ВОТ В ЧЕМ ВОПРОС.¹

¹ Спасибо Уильяму Шекспиру (1564–1616).

9. БУДЬ НЕПРЕКЛОНЕН!



Двигаемся дальше! Алгоритм на основе скорости из фрейма 147:21 улучшает скорость обновления, заимствуя некоторую скорость у предыдущего обновления.	1 Это выглядит как полезный прием.
Так и есть. Но существуют и другие способы увеличить скорость обновления. Эти алгоритмы используют частичное изменение градиента при каждом обновлении.	2 Интересный подход.
Из фрейма 146:17 мы знаем, что градиент стремится к значению 0.0, когда мы скатываемся к нижней точке кривой потерь. Что можно сказать о скорости градиентного спуска?	3 Поскольку значение α до сих пор было константой, очевидно, что аналогичным образом происходит замедление скорости градиентного спуска.
Конечно. Поскольку α представляет долю градиента, которую мы собираемся использовать в качестве скорости, еще один подход к решению этой проблемы состоит в том, чтобы сделать эту долю адаптивной.	4 Что такое адаптивность?
Адаптивность означает, что величина доли зависит от градиента и предыдущих значений.	5 Означает ли это, что при каждом обновлении мы заново вычисляем α ?
Хороший вопрос. Не напрямую. Мы умножаем α на коэффициент D , который отражает текущий градиент и его предыдущие значения.	6 Как ведет себя D при снижении скорости градиентного спуска?

Доля градиента, которую мы используем в качестве скорости при каждом обновлении, должна меняться более медленно, чем скорость обучения при каждом уменьшении градиента.

7 Это означает, что коэффициент D должен становиться больше, когда градиент уменьшается, так как α является константой.

Правильно.

В таких случаях говорят, что D изменяется обратно (или инверсно) по отношению к градиенту.

8 Как найти этот таинственный D ?

Простой способ найти обратную величину – разделить 1.0 на исходную величину. Поэтому D находят так:

$$\frac{1}{G}$$

Здесь G называется модификатором.

9 Теперь нам нужно найти значение G , которое, как мы знаем, зависит от градиента и его истории.

Точно.

Кроме того, нужно помнить, что при умножении α на D мы на самом деле делим его на G :

$$\alpha \times D = \alpha \times \frac{1}{G} = \frac{\alpha}{G}$$

10 Значит, мы должны разделить α на модификатор, чтобы изменить долю градиента, которую мы будем использовать.

Верно.

Как мы это сделаем?

11 Можем ли мы считать, что G – это просто сам градиент? Например, как в этой функции обновления, где g представляет градиент:

```
(define naked-u-with-divide
  (lambda (P g)
    (- P (* (div alpha g) g))))
```


Нет, не можем.

Если мы упростим арифметические вычисления, то получим

$$\frac{\alpha}{g} \times g = \alpha$$

Другими словами, эффект g был бы сведен на нет.

Нам нужно что-то, что учитывает историю g , но не восприимчиво ко всем вариациям g , чтобы влияние g не сводилось на нет.

Точно!

Решение заключается в применении *smooth*, чтобы накапливать модификатор, основанный на g .

Конечно.

Нам требуется представление с сопровождением на основе сглаженного градиента.

Взгляните на каркасный код функции *rms-u*, который определяет новый алгоритм обновления для использования с *gradient-descent*:

(define rms-u

(λ (P g)

(let ((r R))

(let ((α[‡] (÷ α G))))

(list (- P₀ (* α[‡] g)) r))))

Как и другие функции обновления, *rms-u* получает сопровождаемый параметр и градиент и обновляет сопровождаемый параметр.

¹² Ох, и правда.

Тогда наша скорость спуска стала бы постоянной α и не зависела бы от градиента, а это совсем не то, чего мы хотели.

¹³ Похоже, нам нужно использовать здесь новую штуку под названием *smooth*!

¹⁴ Можно увидеть, как выглядит эта новая функция обновления?

¹⁵ В *R* должен располагаться вызов *smooth*, так как нам нужны сглаженные исторические значения.

Для этого нам понадобится коэффициент уменьшения.

[‡] Мы используем обозначение с «шапочкой», например $\hat{\alpha}$, чтобы показать, что эта величина была получена из другой величины с таким же именем, имеющей такое же назначение (в данном случае скорость обучения).

Читатели, которым понятен контекст использования величины, могут, образно говоря, «снять шляпы».

Сопровождение P_1 – это сглаженное значение, полученное из градиента g .

Здесь r – новое сопровождающее значение. Оно является частью сопровождаемого параметра, который возвращает функция. При этом оно используется внутри G .

Найдите R .

И он у нас будет!

Мы предоставим его в виде гиперпараметра

`(declare-hyper  $\beta$ )`

16 Отлично, значит, теперь R выглядит примерно так:

$(smooth\ \beta\ P_1\ \boxed{S})$

потому что P_1 является историческим округленным значением.

Верно.

Теперь мы должны найти S .

17 Разве S – это не g ?

Не совсем.

Градиент g может быть отрицательным, и если мы получим слишком много отрицательных значений градиента, то историческое среднее значение тоже может стать отрицательным.

18 В чем здесь проблема?

Это проблема, потому что r используется в G , и его отрицательное значение может сделать отрицательным $\hat{\alpha}$. Когда это происходит, у нас получается градиентный подъем вместо спуска.

19 Означает ли это, что мы будем смещать наш θ в направлении увеличения потерь, вместо того чтобы двигаться в направлении уменьшения?

Так и есть.

Поэтому мы должны сделать так, чтобы значение r никогда не было отрицательным.

Мы столкнулись с этой проблемой во фрейме 58:14, где у нас могло получиться отрицательное значение.

Правильно.

В итоге вместо R нужно подставить такое выражение:

$$(\text{smooth } P_1 (\text{sqr } g))$$

Теперь осталось найти G .

20 Да.

Мы решили эту проблему возведением в квадрат.

Значит, вместо S нужно подставить $(\text{sqr } g)$?

21 А почему бы вместо G просто не взять r ?

Хороший вопрос.

Проблема возведения в квадрат в том, что результат возрастает намного быстрее, чем исходное скалярное значение.

Если использовать r вместо G , то модифицированная скорость обучения будет расти быстрее, чем снижается градиент.

22 Да, но все равно непонятно, в чем проблема.

Если вернуться к фрейму 65:37, там было сказано, что это может привести к «проскакиванию» нижней точки на кривой потерь. Возникает вопрос: как нам модифицировать r таким образом, чтобы точнее отслеживать градиент, избегая возведения в квадрат?

23 Ясно!

Мы можем извлечь квадратный корень с помощью функции sqr .

Получается, можно найти G с помощью выражения

$$(\text{sqr } r)?$$

Да, в основном вы правы.

Но нужно учитывать тот факт, что r может принять значение ровно 0.0.

24 Ох, в таком случае произойдет деление a на ноль, и результат будет неопределенным.

Верно.

Но эта проблема решается удивительно просто – прибавлением небольшой константы ϵ , называемой стабилизатором к выражению $(\text{sqrt } r)$.

25 Чему обычно равна константа ϵ ?

Мы определим ϵ так:

```
(define  $\epsilon$  1e-08)
```

Теперь найдите G .

26 G определяется выражением $(+ (\text{sqrt } r) \epsilon)$

Как выглядит окончательная версия rms-u ?

Вот так:

```
(define  $\text{rms-u}$ 
  ( $\lambda$  ( $P$   $g$ )
    (let (( $r$  ( $\text{smooth } \beta P_1 (\text{sqrt } g)$ )))
      (let (( $\alpha$  ( $\div \alpha (+ (\text{sqrt } r) \epsilon)$ )))
        (list ( $- P_0 (* \alpha g)$ )  $r$ ))))))
```

27 Но сперва нужно определить соответствующие функции расширения и сжатия, не так ли?

Теперь определите $\text{rms-gradient-descent}$.

Ох, и правда, спасибо!
Конечно, нужно.

Они выглядят аналогично velocity-i из фрейма 148:26 и velocity-d из фрейма 149:29, так как вначале мы устанавливаем для каждого r сопровождение в виде нулевого тензора той же формы, что и параметр.

Запишите определения rms-i и rms-d .

28 Начнем с определения rms-i . Сопровождающим p является нулевой вектор r :

```
(define  $\text{rms-i}$ 
  ( $\lambda$  ( $p$ )
    (list  $p$  ( $\text{zeroes } p$ ))))
```

Аналогично, rms-d просто извлекает параметр p из сопровождаемого параметра:

```
(define  $\text{rms-d}$ 
  ( $\lambda$  ( $P$ )
     $P_0$ ))
```

А вот теперь определите
rms-gradient-descent

29 Вот это определение:

```
(define rms-gradient-descent
  (gradient-descent
   rms-i rms-d rms-u))
```

Вы покажете пример использования?

Мы воспользуемся примером из фрейма 150:32. Но сначала мы должны определить каждый гиперпараметр. Как вы думаете, какой должна быть скорость обучения?

30 Так как *rms-u* использует непрерывно модифицируемую скорость обучения, наверное, мы должны начать с высокой скорости обучения в надежде, что *rms-gradient-descent* ее адаптирует?

Уместное предположение. Давайте начнем с $\alpha = 0.01$.

31 А что насчет *revs*?

В целом *rms-u* позволяет уменьшить значение *revs*, необходимое для достижения точно подобранного θ . В данном случае примем

32 Хорошо.

revs = 3000

вместо 5000 во фрейме 150:32.

Обратимся к немного расширенной версии *try-plane*, которая теперь получает *an- α* – начальное значение α :

33 Мы должны присвоить скалярное значение β , а затем вызвать *try-plane*:

```
(define try-plane
  (lambda (a-gradient-descent a-revs
           an-alpha)
    (with-hypers
     ((revs a-revs)
      (alpha an-alpha)
      (batch-size 4))
     (a-gradient-descent
      (sampling-obj
       (l2-loss plane) plane-xs
       plane-ys)
      (list [0.0 0.0] 0.0)))))
```

```
> (with-hypers
   ((beta 0.9))
   (try-plane
    rms-gradient-descent
    3000 0.01))
> (list [3.98 1.97] 6.16)
```

Мы получим точно подобранный θ , затратив на 2000 обновлений меньше, чем раньше!

Как вызвать эту функцию в нашем примере?

Да, так и есть.

Эта версия функции *gradient-descent* имеет немного таинственное имя *RMSProp*[†].

Сокращение *RMS* образовано от *root mean square* (среднеквадратичное значение) и отражает тот факт, что мы используем среднее историческое значение (*mean*) квадратов (*square*), а затем извлекаем из него квадратный корень (*root*). Суффикс *Prop* связан с термином *back propagation* (обратное распространение).

[†] Предложил Джеффри Хинтон.

Превосходное замечание!

Последний из алгоритмов обновления (*adam-u*) использует *smooth* для двух значений – для градиента и для квадратов.

Что касается градиента, мы используем гиперпараметр μ из *velocity-u* во фрейме 150:32, применяя, таким образом, хитрую стратегию Граучо.

В случае квадрата градиента мы продолжаем использовать β , как мы делали в *rms-u*, благодаря чему можем модифицировать скорость обучения.

34 У нас есть два разных алгоритма для ускорения градиентного спуска – *velocity-u* из фрейма 150:31 и *rms-u* из фрейма 27.

Будет ли полезно объединить их?

35 Эти гиперпараметры очень удобны.

Да, конечно.

Вот объявление функции

adam-u[†]:

```
(define adam-u
  (λ (P g)
    (let ((r (smooth β P2 (sqr g))))
      (let ((α (÷ α (+ (sqr r) ε)))
            (v (smooth μ P1 g)))
        (list (- P0 (* α v)) v r)))))
```

Ее параметр имеет два сопровождения.

Это *r* и $\hat{\alpha}$, используемые аналогично *rms-u* во фрейме 27.

Усредненное значение *v* немного отличается от фрейма 150:31.

Первое отличие в том, что мы используем *smooth* для накопления. Это приводит к тому, что градиент умножается на (-1.0μ) . Поскольку μ обычно составляет около 0.9, для следующего θ используется лишь небольшая часть градиента *g*. Остальная часть состоит из исторического среднего значения *v*.

Поэтому усредненное значение *v* сглажено намного сильнее, чем в *velocity-u* во фрейме 150:31.

Это тоже просто.

Чтобы найти точный параметр P_0 , мы не используем *g* напрямую. Вместо этого мы применяем историческое среднее *v*.

Как вы думаете, почему?

36 В чем состоит отличие?

[†] Этот алгоритм под названием Adam улучшает градиентный спуск за счет стохастической целевой функции. Авторы алгоритма – Дьедрик Кингма (1983–) и Джимми Ба.

Для тех, кто уже встречался с этим алгоритмом обновления в другом месте, мы упростили его, убрав поправку смещения для *v* и *g*.

37 Да, понятно. В функции *velocity-v* градиент *g* умножается на скорость обучения и прибавляется к *v*.

В чем заключается второе отличие?

38 Это другой способ использовать информацию о скорости предыдущего обновления в текущем обновлении.

Правильно.

Мы не должны забывать про объявление *adam-i* и *adam-d*.

Каждый параметр p в θ теперь сопровождается двумя дополнительными тензорами v и r , оба той же формы, что и p .

Запишите определение *adam-i*.

39 Вот оно:

```
(define adam-i
  (λ (p)
    (let ((v (zeroes p)))
      (let ((r v))
        (list p v r)))))
```

Запишите определение *adam-d*.

40 Функция сжатия снова извлекает 0-й элемент из сопровождаемого параметра P для получения следующего p :

```
(define adam-d
  (λ (P)
    P0))
```

Все наши функции сжатия (кроме *naked-d*) выглядят одинаково. Почему бы нам не использовать для них одинаковое имя?

Хороший вопрос.

Мы можем так поступить. Мы создадим шаблон и будем использовать его всякий раз для представлений сопровождаемых параметров, отличных от *list*.

41 Мне нравится эта идея.

Теперь напишите определение *adam-gradient-descent*.

42 Определение будет таким:

```
(define adam-gradient-descent
  (gradient-descent
   adam-i adam-d adam-u))
```


Отлично!

Название алгоритма Adam образовано как сокращение от *adaptive moment estimation* (адаптивное оценивание импульса). Он является адаптивным, так как использует переменную скорость обучения[†].

[†] Сопровождение v называется первым моментом градиента, а r – его второй момент.

Ниже показано, как мы еще раз получаем прежний результат с помощью *try-plane* из фрейма 150:32, но с некоторыми изменениями:

```

▶ (with-hypers
  ((μ 0.85)
   (β 0.9))
  (try-plane
   adam-gradient-descent
   1500 0.001))
▶ (list [3.98 1.97] 6.16)
```

Эти значения определены опытным путем, в поиске комбинации, которая обеспечивает наименьшее значение *revs*.

Теперь в нашем распоряжении несколько вариантов градиентного спуска, которые можно использовать при обучении θ для различных функций.

43 Покажите на примере, как нужно вызывать *adam-gradient-descent*.

44 Итак, теперь нам достаточно всего 1500 обновлений для достижения точной подгонки θ . Это намного меньше, чем 15000, с которых мы начали во фрейме 127:42. Но как мы выбрали скалярные значения μ и β ?

45 Понятно.

46 Что дальше?

Закон градиентного спуска

Набор параметров θ искомой функции обучается путем использования одной из функций градиентного спуска.

Мы займемся изучением того, ⁴⁷ Да, перерыв был бы очень
как работают расширенные кстати.
функции.

Но сначала сделаем перерыв
на смузи!

Быстрые побрякушки

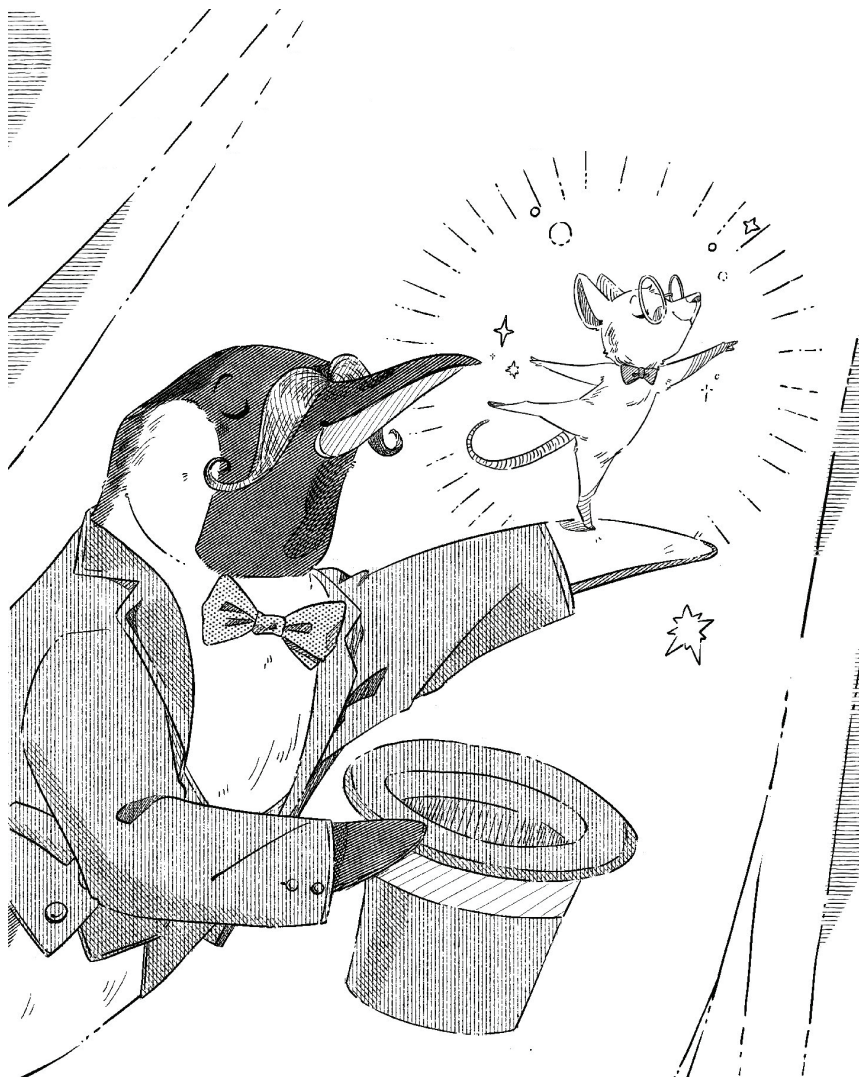
β стр. 166

rms-gradient-descent стр. 168

adam-gradient-descent стр. 171

Не хотите попробовать
смузи из шоколада
и миндального масла?

ИНТЕРЛЮДИЯ V. ЭКСТЕНСИО МАГНИФИКО!



Как вам смузи?	¹ У него очень нежная текстура.
Мы сделали допущение, что расширения применимы только к функциям наподобие + и *.	² Ого, значит, эта интерлюдия будет расширением интерлюдии I?
В отличие от Бергласа [†] , мы раскроем секрет нашего трюка с расширением функций.	
----- [†] Дэвид Берглас (1926–), британский фокусник, автор секретной техники поиска карты в колоде.	
Так и есть!	³ Это похоже на функцию <i>tmap</i> из фрейма 78:25.
Мы начнем с часто встречающихся функций от тензоров.	
При вызове этих функций аргументом является каждый элемент тензора.	
Допустим, мы применяем функцию <i>add1</i> к каждому элементу	
[3 5 4]	
и получаем	
[4 6 5]	
Да, эти две функции одинаковые.	⁴ Это выглядит так, будто функция <i>add1</i> размножается и вытягивается в тензор.
Функция <i>tmap</i> получает как аргументы функцию <i>f</i> и тензор <i>t</i> и вызывает <i>f</i> для каждого элемента <i>t</i> , а затем собирает результат в новый тензор, как показано в следующей логической цепочке:	Это аналогично спуску в тензор во фрейме 48:19.
1. $(tmap\ add1\ [3\ 5\ 4])$	
2. $[(add1\ 3)\ (add1\ 5)\ (add1\ 4)]$	
3. $[4\ 6\ 5]$	

Так и есть!

Спуск в тензор – это просто другой способ рассматривать работу *tmap*. Она спускается в тензор, чтобы вызвать рабочую функцию для каждого элемента.

⁵ Но как быть, если у нас больше одного тензора и нам нужно вызвать рабочую функцию для каждого элемента всех тензоров?

Функция *tmap* создана для работы с несколькими тензорами, при условии что все они одинаковой длины.

Например, здесь мы используем два тензора:

1. $(tmap + [3\ 4\ 6\ 1] [1\ 3\ 5\ 5])$
2. $[(+ 3\ 1) (+ 4\ 3) (+ 6\ 5) (+ 1\ 5)]$
3. $[4\ 7\ 11\ 6]$

⁶ Понятно.

Теперь давайте рассмотрим использование *tmap* с расширенными функциями.

Мы начнем с расширенной функции *sqrt*. Допустим, что *sqrt*⁰ представляет собой нерасширенную примитивную функцию, которая находит положительный квадратный корень скалярного значения.

Так выглядит каркасный код расширенной функции *sqrt*:

```
(define sqrt
  (λ (t)
    (cond
      ((scalar? t) (sqrt0 t))
      (else A))))
```

Если *t* – скаляр, мы находим квадратный корень с помощью *sqrt*⁰.

⁷ Для расширения поведения *A* должен спускаться в тензор до тех пор, пока не найдет скалярное значение, и вызывать для него функцию *sqrt*⁰.

Каким должно быть выражение A ?

Отлично.

8 Это *tmap*!

Какую функцию, позволяющую спуститься в тензор, мы знаем?

Значит, A будет выглядеть так:

$(tmap \boxed{S} t)$

Верно.

9 Что-то знакомое.

Теперь найдем S . Эта функция вызывается для каждого элемента t и должна возвращать квадратный корень. Если каждый элемент t является скаляром, необходимо вызвать sqrt^0 , но если элемент является тензором ранга 1 или выше, необходимо спуститься в него.

Мне кажется, это функция *sqrt*.

Прекрасно.

10 То есть должно получиться так, верно?

В определении во фрейме 7 наш основной тест использует условную функцию *scalar?*. Заменим ее другой функцией

```
(define of-rank?
  (λ (n t)
    (= (rank t) n)))
```

of-rank?

которая получает ранг и тензор и проверяет, имеет ли тензор указанный ранг. В данном случае мы проверяем ранг 0:

```
(define sqrt
  (λ (t)
    (cond
      ((of-rank? 0 t) (sqrt0 t))
      (else (tmap sqrt t)))))
```

Это правильное определение, но есть еще кое-что интересное. Начнем с такого каркасного кода:

```
(define of-rank?
  (λ (n t)
    (cond
      ((zero? n) S)
      ((scalar? t) B)
      (else R))))
```

Найдите S , B и R .

- 11 Начнем с выражения S , которое является истинным, только если истинно выражение $(\text{scalar? } t)$. Значит, вместо S подставляем

$(\text{scalar? } t)$

Перейдем к B . Так как первый тест `cond` не пройден, мы знаем, что n больше 0, и если t — скаляр, то B возвращает

$\#f$

Наконец, R . Тензор t имеет ранг n , только если $t|_0$ имеет ранг $n - 1$. Мы можем выполнить рекурсивную проверку. Итак, вместо R подставим

$(\text{of-rank? } (\text{sub1 } n) t|_0)$

изменив как n , так и t , чтобы в конечном итоге пройти либо первый, либо второй базовый тест.

Прекрасно.

Так выглядит финальная версия *of-rank?*:

```
(define of-rank?
  (λ (n t)
    (cond
      ((zero? n) (scalar? t))
      ((scalar? t) #f)
      (else (of-rank? (sub1 n) t|_0)))))
```

Используя тот же тензор t , что и во фрейме 39:44, запишите логический столбец для

$(\text{of-rank? } 3 \ t)$

- 12 Вот этот столбец:

1. $(\text{of-rank? } 3 \ [[[8] \ 9] \ [[4] \ 7]]])$
2. $(\text{of-rank? } 2 \ [[8] \ 9])$
3. $(\text{of-rank? } 1 \ [8])$
4. $(\text{of-rank? } 0 \ 8)$
5. $\#t$

Отлично.

Что, если заменить 3 любым другим натуральным числом?

13 Результатом будет

#f

Если вернуться к *sqrt*, почему мы оставили пунктир во фрейме 10?

Как обычно, мы собираемся внести в него корректировки. Эта функция применяется для вычисления квадратных корней, потому что мы вызываем *sqrt*⁰ только для скаляров. Будем называть ее базовой функцией.

Вместо того чтобы получать фиксированное значение базовой функции одного аргумента (здесь *sqrt*⁰), мы получаем его как дополнительный аргумент.

14 Можно ли таким способом расширить любую базовую функцию одного аргумента?

Да.

Этой обобщенной функции мы дадим имя *ext1*, потому что она расширяет (extends) функцию одного аргумента. Так выглядит каркасный код:

```
(define ext1
  (λ (f)
    (λ (t)
      (cond
        ((of-rank? 0 t) (f t))
        (else (tmap E t))))))
```

Вызов *ext1* с базовой функцией *f* возвращает функцию, которая либо вызывает *f* для *t*, если его ранг равен 0, либо спускается в *t* в противном случае.

Найдите *E*.

15 На месте *E* должна быть функция, которая вызывает базовую функцию *f* для скаляра или спускается в тензор более высокого ранга. Отсюда следует, что *E* – это

(*ext1 f*)

Превосходно.

Рассмотрим объявление *ext1* с пунктиром:

```
(define ext1
  (λ (f)
    (λ (t)
      (cond
        ((of-rank? 0 t) (f t))
        (else (tmap (ext1 f) t))))))
```

Используя *ext1*, определите *sqr*.

¹⁶ Вот это определение:

```
(define sqr
  (ext1 sqr0))
```

Как насчет других примеров?

Хорошая идея.

Как при помощи *ext1* определить функцию *zeroes* из фрейма 149:27?

¹⁷ Мы можем рассматривать *zeroes* как расширение функции, которая возвращает 0.0 для каждого скаляра!

Вот как мы определим *zeroes*:

```
(define zeroes
  (ext1 (λ (x) 0.0)))
```

Отлично.

¹⁸ Но почему определения *ext1*, *sqr* и *zeroes* с пунктирной линией?

Функция *ext1* до сих пор работала только с функциями, которые оперируют скалярами.

Но иногда нам нужно расширить функции типа *sum*¹ во фрейме 49:24, которая работает только с *tensor*¹.

¹⁹ Значит, вместо того чтобы проверять

```
(of-rank? 0 t)
```

как мы видим в *ext1*, мы должны смотреть на ранг 1, когда расширяем *sum*¹.

Правильно.

Мы называем его базовым рангом.

²⁰ То есть базовый ранг

```
sum1
```

– это 1?

В точку!

Какой базовый ранг у *sqr*⁰?

²¹ Это 0.

Правильно.

Значит, расширяя функцию f , мы должны получить ее базовый ранг как аргумент.

Вот каркасный код нашей обновленной *ext1*:

```
(define ext1
  (λ (f n)
    (λ (t)
      (cond
        ( ( N ) (f t) )
        (else (tmap (ext1 f n) t)))))))
```

Найдите N .

Верно.

Теперь запишем финальную версию *ext1*

```
(define ext1
  (λ (f n)
    (λ (t)
      (cond
        ((of-rank? n t) (f t))
        (else (tmap (ext1 f n) t))))))
```

Что вы можете сказать про *sqrt* и *zeroes*, использующие эту функцию?

22 Здесь мы должны проверить ранг n . Значит, вместо N поставим

(*of-rank?* n t)

23 Теперь при вызове *ext1* мы должны передать базовый ранг для *sqrt*⁰.

Здесь базовый ранг *sqrt* равен 0.

```
(define sqrt
  (ext1 sqrt0 0))
```

Аналогично, в случае *zeroes* базовая функция также имеет ранг 0:

```
(define zeroes
  (ext1 (λ (x) 0.0) 0))
```

Прекрасно.

Теперь определите *sum*, которая расширяет *sum*¹.

24 Мы вызываем *ext1* с базовым рангом 1:

```
(define sum
  (ext1 sum1 1))
```

Можно взглянуть на пример, где базовый ранг равен 2?

Конечно.

Рассмотрим функцию *flatten*², которая преобразует *tensor*² в *tensor*¹. Это достигается путем объединения вместе каждого из вложенных *tensor*¹.

25 Покажите это на примере, если можно.

В соответствии с обозначениями со стр. 110–111:

26 Это функция, которую мы собираемся расширить?

1. $(\text{flatten}^2 \left(\begin{bmatrix} 1.0 & 0.5 \\ 3.1 & 2.2 \\ 7.3 & 2.1 \end{bmatrix}_{(3 \ 2)} \right))$
2. $(\text{flatten}^2 \left([[1.0 \ 0.5] \ [3.1 \ 2.2] \ [7.3 \ 2.1]] \right))$
3. $[1.0 \ 0.5 \ 3.1 \ 2.2 \ 7.3 \ 2.1]$

Да!

27 Поясните, как она работает.

```
(define flatten
  (ext1 flatten2 2))
```

Охотно.

Это расширенная версия *flatten*², которая работает с тензорами произвольного ранга, но преобразует только самые глубоко вложенные *tensor*².

Например:

1. $(\text{flatten} \left([[[1.0 \ 0.5] \ [3.1 \ 2.2] \ [7.3 \ 2.1]] \ [2.9 \ 3.5] \ [0.7 \ 1.5] \ [2.5 \ 6.4]] \right))$

Завершите логический столбец.

28 Здесь мы спускаемся в тензоры:

2. $(\text{flatten} \left([[[1.0 \ 0.5] \ [3.1 \ 2.2] \ [7.3 \ 2.1]] \ (\text{flatten} \left([[2.9 \ 3.5] \ [0.7 \ 1.5] \ [2.5 \ 6.4]] \right)) \right))$
3. $(\text{flatten}^2 \left([[[1.0 \ 0.5] \ [3.1 \ 2.2] \ [7.3 \ 2.1]] \ (\text{flatten}^2 \left([[2.9 \ 3.5] \ [0.7 \ 1.5] \ [2.5 \ 6.4]] \right)) \right))$
4. $[1.0 \ 0.5 \ 3.1 \ 2.2 \ 7.3 \ 2.1 \ 2.9 \ 3.5 \ 0.7 \ 1.5 \ 2.5 \ 6.4]$

Верно.

Функция *ext1* полезна для расширения функций одного аргумента.

Далее мы рассмотрим расширение функций двух аргументов.

29 То есть теперь мы определим функцию *ext2*?

Точно!

Но прежде чем мы это сделаем, нам понадобится пара других функций сравнения рангов.

30 Наподобие
of-rank?
только более интересные?

Да, конечно!

Иногда нам нужно знать, выше ли ранг одного тензора, чем другого. Возьмем, к примеру, функцию *rank*>, которая истинна, если ранг ее первого тензорного аргумента выше, чем ранг ее второго тензорного аргумента.

Вот первая версия этой функции:

```
(define rank>
  (λ (t u)
    (> (rank t) (rank u))))
```

31 Можем ли мы тут, как обычно, написать рекурсивную версию?

Даже должны!

```
(define rank>
  (λ (t u)
    (cond
      ((scalar? t) #f)
      ((scalar? u) #t)
      (else (rank> t|0 u|0))))
```

Объясните, как работает эта функция.

32 Если *t* – скаляр, его ранг равен 0. Поскольку 0 не выше любого другого ранга, то ответ таков:

#f

С другой стороны, если *u* – скаляр, а мы уже знаем, что *t* – не скаляр, то ранг *t* выше, чем ранг *u*, поэтому ответ таков:

#t

И наконец, если ни один из них не скаляр, мы рекурсивно сравниваем ранги 0-го элемента как t , так и u .

Что еще нужно сделать?

Теперь определим функцию *of-ranks?*, которая получает аргументы n , t , m и u , где n – ранг, который мы проверяем для тензора u .

33 Вот эта функция:

```
(define of-ranks?
  (λ (n t m u)
    (cond
      ((of-rank? n t) (of-rank? m u))
      (else #f))))
```

Это было весело!

Замечательно.

Рассмотрим пример:

1. $(\text{of-ranks? } 3 \text{ } [[8] \text{ } 9]] \text{ } [[2] \text{ } 1]]) \text{ } 2 \text{ } [5])$
2. $(\text{of-rank? } 2 \text{ } [5])$
3. $\#f$

Объясните этот логический столбец.

34 Так как

$(\text{of-rank? } 3 \text{ } [[8] \text{ } 9]] \text{ } [[2] \text{ } 1]])$

дает $\#t$, основной тест клаузы *cond* успешен, и мы должны провести проверку

$(\text{rank } [5])$ равно 2

что является ложью. Значит, мы возвращаем $\#f$.

Очень хорошо.

Теперь мы готовы определить *ext2*.

По своей базовой структуре она похожа на *ext1*, но поскольку расширяемая базовая функция имеет два аргумента вместо одного, нам нужны два базовых ранга:

35 Мы знаем, что

of-ranks?

выполняет две проверки:

t имеет ранг n

и

u имеет ранг m

Это означает, что в первой клаузе выражения *cond* мы проверяем, достигли ли t и u своих базовых рангов. Если это так, мы вызываем для них базовую функцию f .

```
(define ext2
  (λ (f n m)
    (λ (t u)
      (cond
        ((of-ranks? n t m u) (f t u))
        (else
         (desc (ext2 f n m) n t m u))))))
```

Мы воспользуемся двумя вспомогательными функциями:

of-ranks?

и

desc

Поскольку у нас есть два базовых ранга и два тензорных аргумента, принятие решения, когда мы должны спускаться, а когда нет, требует немного больше размышлений.

В противном случае нужно спуститься в тензор, и мы делаем это с помощью *desc*. Первый аргумент *desc* очень похож на первый аргумент *tmap* в *ext1*, когда она применяется для спуска в тензор.

Итак, *desc* вызывается с

(ext2 f n m)

что совпадает с

(ext1 f n)

в определении *ext1*.

Правильно.

Рекурсивный вызов *ext2* создает функцию, которая либо вызывает *f* для двух тензоров аргументов, если каждый из них имеет соответствующий базовый ранг, либо спускается в один или оба тензора. Это первый аргумент *desc*.

36 Как мы решаем, в какой тензор или тензоры должны спуститься?

Прекрасный вопрос.

Чтобы понять это, взгляните на *desc*:

37 В случае *desc-t*, поскольку мы спускаемся только к одному тензору, мы используем *tmap* для вызова *g* для каждого элемента *t* (с именем *et*) и *u*:

```
(define desc
  (λ (g n t m u)
    (cond
      ((of-rank? n t) (desc-u g t u))
      ((of-rank? m u) (desc-t g t u))
      ((= † t † † u †) (tmap g t u))
      ((rank> t u) (desc-t g t u))
      (else (desc-u g t u)))))
```

Здесь *g* – расширенная функция

```
(ext2 f n m)
```

из-за того, как

```
desc
```

вызывается изнутри

```
ext2
```

Первые две клаузы относятся к тому, что происходит, когда один из двух аргументов тензора достигает своего базового ранга. Например, если мы достигли базового ранга для *u*, то нам нужно спуститься только в *t*, и наоборот.

Определите *desc-t*, которая спускается в *t*, и *desc-u*, которая спускается в *u*.

```
(define desc-t
  (λ (g t u)
    (tmap (λ (et) (g et u)) t)))
```

Переходя к *desc-u*, мы аналогичным образом используем *tmap* для вызова *g* от *t* и каждого элемента *u* (с именем *eu*):

```
(define desc-u
  (λ (g t u)
    (tmap (λ (eu) (g t eu)) u)))
```

Очень хорошо.

Наша следующая клауза относится к случаям, когда *t* и *u* имеют одинаковое количество элементов, что мы определяем с помощью *† t †* и *† u †* из фрейма 32:17. Это означает, что мы спускаемся в оба тензора одновременно.

38 Что произойдет, если окажется, что ни одна клауза не релевантна?

Если ранг t больше, чем ранг u , мы используем *desc-t*; в противном случае используем *desc-u*.

³⁹ И мы определим это с помощью

rank>

из фрейма 32!

Можно увидеть пример использования *ext2*?

Конечно.

Пусть $+^{0,0}$ обозначает функцию, складывающую два скалярных значения.

Определите расширенную версию $+$ этой базовой функции.

⁴⁰ Мы должны вызвать *ext2*, так как базовая функция является функцией двух аргументов. Базовый ранг обоих аргументов равен 0:

```
(define +  
  (ext2 +0,0 0 0))
```

У вас найдутся еще примеры?

Пусть $*^{0,0}$ обозначает функцию, перемножающую два скалярных значения.

Определите расширенную версию $*$ этой базовой функции.

⁴¹ Как и прежде, мы должны вызвать *ext2*, так как базовая функция является функцией двух аргументов. Базовый ранг обоих аргументов также равен 0:

```
(define *  
  (ext2 *0,0 0 0))
```

Теперь любая функция, использующая $*$, автоматически становится расширенной. Например, определите функцию *sqr*, которая возводит в квадрат тензорный аргумент t .

⁴² Вот это определение:

```
(define sqr  
  (λ (t)  
    (* t t)))
```

У нас также есть все необходимое для определения $\bullet$ из фрейма 103:26.

Определите $\bullet$.

⁴³ Это можно сделать, расширив $\bullet^{1,1}$ с базовыми рангами 1 и 1:

```
(define •  
  (ext2 •1,1 1 1))
```

Превосходно.

Мы можем использовать *ext2* другими способами. Например, можно определить функцию $*^{2,1}$:

```
(define *2,1
  (ext2 * 2 1))
```

Объясните, как работает эта функция.

Мы используем разные базовые ранги. Это означает, что $*^{2,1}$ спускается в оба своих аргумента, пока не встретит *tensor*² в своем первом аргументе и *tensor*¹ во втором, а затем вызывает *** только для них. Взгляните на пример.

Пусть *p* – это *tensor*²

```
[[3 4 5]
 [7 8 9]]
```

и *t* – *tensor*¹

```
[2 4 3]
```

Завершите логический столбец, имея в виду, что сначала мы используем ***, каждый элемент которой имеет базовый ранг 0:

1. $(* p t)$
2. $(* [[3 4 5] [7 8 9]] [2 4 3])$

44 Здесь базовой функцией является

*

которую мы определили во фрейме 41, и ее определение опирается на $*^{0,0}$.

Почему она нуждается в расширении?

45 Здесь мы сначала спускаемся в *t*, потому что он имеет ранг больше 0:

3. $[(* [3 4 5] [2 4 3])$
 $(* [7 8 9] [2 4 3])]$
4. $[((* 3 2) (* 4 4) (* 5 3))$
 $((* 7 2) (* 8 4) (* 9 3))]$
5. $[[(*^{0,0} 3 2) (*^{0,0} 4 4) (*^{0,0} 5 3)]$
 $[((*^{0,0} 7 2) (*^{0,0} 8 4) (*^{0,0} 9 3))]$
6. $[[6 16 15]$
 $[14 32 27]]$

Именно так.

Рассмотрим логический столбец, показывающий, как $*^{2,1}$ справляется с такими же двумя аргументами. Так как p имеет базовый ранг 2 и t имеет базовый ранг 1, функция $*^{2,1}$ вызывает для них $*$:

1. $\left| (*^{2,1} p t) \right.$
2. $\left| (* p t) \right.$

Что происходит дальше?

Правильно.

Теперь возьмем другую пару тензоров и посмотрим, как $*$ работает с ними.

Пусть q – tensor²

$[[8\ 1]\ [7\ 3]\ [5\ 4]]$

и пусть r – другой tensor²

$[[6\ 2]\ [4\ 9]\ [3\ 8]]$

Завершите логический столбец

1. $\left| (* q r) \right.$

⁴⁶ Мы получаем тот же результат, что и во фрейме 45.

⁴⁷ Здесь функция $*$ сразу встречает два тензора одинаковой длины 3, поэтому она спускается в оба одновременно:

2. $\left| \begin{array}{l} (* [[8\ 1]\ [7\ 3]\ [5\ 4]] \\ \quad [[6\ 2]\ [4\ 9]\ [3\ 8]]) \end{array} \right.$
3. $\left| \begin{array}{l} [((* 8\ 6) (* 1\ 2)) \\ \quad [((* 7\ 4) (* 3\ 9)) \\ \quad \quad [((* 5\ 3) (* 4\ 8))]] \end{array} \right.$
4. $\left| \begin{array}{l} [[((*^{0,0} 8\ 6) (*^{0,0} 1\ 2)) \\ \quad [((*^{0,0} 7\ 4) (*^{0,0} 3\ 9)) \\ \quad \quad [((*^{0,0} 5\ 3) (*^{0,0} 4\ 8))]] \end{array} \right.$
5. $\left| [[48\ 2]\ [28\ 27]\ [15\ 32]] \right.$

Что происходит с $*^{2,1}$?

Давайте разберемся!

Вместо того чтобы спускаться в два тензора одновременно, эта функция обнаруживает, что q уже имеет базовый ранг, и спускается только в r :

1. $(^{*2,1} q r)$
2. $(^{*2,1} [[8\ 1]\ [7\ 3]\ [5\ 4]]$
 $\quad\quad\quad [[6\ 2]\ [4\ 9]\ [3\ 8]])$
3. $[(^{*2,1} [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [6\ 2])$
 $\quad (^{*2,1} [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [4\ 9])$
 $\quad (^{*2,1} [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [3\ 8])]$

Теперь завершите логический столбец.

48 Вызовы *2,1 происходят для базовых рангов 2 и 1 соответственно, поэтому мы вызываем $*$ для тензорных аргументов *2,1 :

4. $[(^{*2,1} [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [6\ 2])$
 $\quad (^{*2,1} [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [4\ 9])$
 $\quad (^{*2,1} [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [3\ 8])]$
5. $[(^* [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [6\ 2])$
 $\quad (^* [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [4\ 9])$
 $\quad (^* [[8\ 1]\ [7\ 3]\ [5\ 4]]\ [3\ 8])]$
6. $[[(^* [8\ 1]\ [6\ 2])$
 $\quad (^* [7\ 3]\ [6\ 2])$
 $\quad (^* [5\ 4]\ [6\ 2])]$
 $\quad [(^* [8\ 1]\ [4\ 9])$
 $\quad (^* [7\ 3]\ [4\ 9])$
 $\quad (^* [5\ 4]\ [4\ 9])]$
 $\quad [(^* [8\ 1]\ [3\ 8])$
 $\quad (^* [7\ 3]\ [3\ 8])$
 $\quad (^* [5\ 4]\ [3\ 8])]$
7. $[[[(^{*0,0} 8\ 6)\ (^{*0,0} 1\ 2)]$
 $\quad [(^{*0,0} 7\ 6)\ (^{*0,0} 3\ 2)]$
 $\quad [(^{*0,0} 5\ 6)\ (^{*0,0} 4\ 2)]]]$
 $\quad [[(^{*0,0} 8\ 4)\ (^{*0,0} 1\ 9)]$
 $\quad [(^{*0,0} 7\ 4)\ (^{*0,0} 3\ 9)]$
 $\quad [(^{*0,0} 5\ 4)\ (^{*0,0} 4\ 9)]]]$
 $\quad [[(^{*0,0} 8\ 3)\ (^{*0,0} 1\ 8)]$
 $\quad [(^{*0,0} 7\ 3)\ (^{*0,0} 3\ 8)]$
 $\quad [(^{*0,0} 5\ 3)\ (^{*0,0} 4\ 8)]]]$
8. $[[[48\ 2]\ [42\ 6]\ [30\ 8]]$
 $\quad [[32\ 9]\ [28\ 27]\ [20\ 36]]$
 $\quad [[24\ 8]\ [21\ 24]\ [15\ 32]]]$

Отличается ли это от результата для $*$ во фрейме 47?

49 Конечно, поэтому мы можем использовать базовый ранг для изменения поведения расширенных функций. Этому есть полезное применение?

Еще бы†!

⁵⁰ Мы заслужили перерыв.

Но это тема следующей главы.

† Некоторым это может быть знакомо
как предшественник умножения
матрицы на вектор.

Больше разных побрякушек

ext1 стр. 182

*sqrt*⁰ стр. 182, *sqrt* стр. 182

*zeroes*⁰ стр. 182, *zeroes* стр. 182

*sum*¹ стр. 182, *sum* стр. 182

*flatten*² стр. 183, *flatten* стр. 183

ext2 стр. 186

+^{0,0} стр. 188, *+* стр. 188

^{0,0}* стр. 188, * стр. 188

sqr стр. 188

**^{2,1}* стр. 189

Как насчет кленово-орехового шифонового пирога?

ОРЕХОВОГО И НЕЖНОГО
ОДНОВРЕМЕННО!

10. ПРИЧУДЛИВЫЙ ТАНЕЦ НЕЙРОНОВ¹



¹ В оригинале глава называется Doung neuron dance в честь песни Neutron Dance в исполнении знаменитого американского трио The Pointer Sisters: Джуна (1953–2006), Рут (1946–) и Анита (1948–). Отдельная благодарность композиторам Альте Уиллис (1947–2019) и Даниэлю Сембелло (1963–2015).

Нет ли у вас чувства тяжести после кленово-орехового пирога?	¹ Нет, это был лучший кусок пирога в моей жизни!
--	---

А теперь продолжим!	² Как так?
---------------------	-----------------------

До этого времени наш выбор обучаемой функции был очень ограничен.

Такие функции, как <i>line</i> , <i>quad</i> и <i>plane</i> , используют лишь небольшое количество параметров и подходят только для определенных типов наборов данных, которые можно смоделировать с помощью этих простых функций.	³ Почему же это проблема?
--	--------------------------------------

Мы не можем различать ирисы.	⁴ <i>Ирисы</i> ? Какое отношение имеют цветы к теме наших занятий?
------------------------------	---

Скоро узнаете.

Если бы у нас были более сложные обучаемые функции, мы могли бы научить их распознавать цветы и другие объекты, используя наборы данных, состоящие из соответствующим образом помеченных изображений.

Или мы могли бы обучить их выделять объекты, лица и другие важные признаки на изображениях. Или распознавать устную и письменную речь, или делать другие действия, которые обычно ставят машину в тупик.

	⁵ Как можно научить машину этим вещам?
--	---

Мы определим обучаемую функцию, имеющую набор параметров θ , затем воспользуемся одной из функций градиентного спуска, чтобы найти точно подобранный (обученный) набор параметров, позволяющий решать нужную задачу.

⁶ Понятно, но как мы найдем эту большую и сложную обучаемую функцию?

Как и раньше, мы построим ее из более простых частей.

⁷ Ага!
Звучит знакомо. Мы всегда разрабатывали функции, разделяя большие функции на более маленькие.
Здесь будет так же?

В каком-то смысле да.
Но в данном случае маленькие функции тоже должны быть параметризованы, потому что мы продолжим использовать оптимизацию градиентным спуском для обучения параметров этих функций, работающих вместе.

⁸ Звучит захватывающе.
Как выглядят эти более простые параметризованные функции?

Давайте разберемся вместе.
Так выглядит функция *rectify*⁰:

```
(define rectify0
  (λ ((s)
    (cond
      ((< s 0.0) 0.0)
      (else s))))
```

⁹ Функция *rectify*⁰ ожидает скалярный аргумент и возвращает 0.0, если этот аргумент отрицательный. В остальном она аналогична тождественной функции.

Поясните, как она работает.

Правильно.

Является ли *rectify*⁰ линейной функцией?

10 Функция *rectify*⁰ основана на условном выражении *cond*. Как известно из фрейма 98:11, линейная функция может использовать только сложение и масштабирование, но *cond* не является ни тем, ни другим.

Хорошо.

Расширенная версия *rectify*⁰ называется *rectify*:

```
(define rectify
  (ext1 rectify0 0))
```

11 Ясно, она работает с тензорами, применяя аналогичное действие к каждому скаляру, как мы знаем из фреймов 45:11 и 182:23.

После расширения *rectify* остается нелинейной?

Да, остается.

Нелинейные функции наподобие *rectify* принято называть *решателями* (*decider*)[†]. Они принимают локальное решение о своем аргументе и возвращают это решение в качестве результата.

12 Относится ли решатель к простейшим функциям, которые мы будем использовать?

[†] Также называются функциями активации. Мы используем термин решатель, чтобы лучше передать суть.

Да.

Использование решателя в качестве одной из элементарных функций позволяет нам принимать крошечные решения, включающие всего несколько параметров, которые объединяются в окончательное решение, например какие виды ирисов мы наблюдаем.

13 Но у *rectify* нет параметров. Как мы можем научить ее чему-нибудь?

Превосходный вопрос!

Мы будем комбинировать ее с другой простой функцией, у которой есть параметры.

Вот знакомая вам функция.

Она идентична функции *plane* из фрейма 102:25:

```
(define linear1,1
  (λ (t)
    (λ (θ)
      (+ (• θ0 t) θ1))))
```

- 14 Что означает верхний индекс «1,1» в имени этой функции?

Верхний индекс «1,1» напоминает нам, что и θ_0 и t должны иметь форму `tensor1`, и она будет вести себя как обычная расширенная функция, если любой из тензоров имеет более высокий ранг. И в отличие от *rectify*, эта функция линейна.

Поясните, почему.

- 15 Эта функция использует сложение (+) и умножение с точкой (•) из фрейма 103:26, которые представляют из себя только сложение и умножение на коэффициент масштаба[†].

Это делает функцию *linear^{1,1}* линейной в полном соответствии с именем.

[†] Нужно отметить, что (•) в этом определении использует один тензор из параметров и один из аргумента, что приводит к скалярному перемножению параметра и аргумента. Следовательно, мы говорим, что (•) представляет собой масштабирование. Мы используем сложение, когда суммируются все эти скалярные произведения. Это делает (•) в данном контексте линейной операцией.

Превосходно.

Теперь объясните, как работает $linear^{1,1}$.

16 Функция $linear^{1,1}$ объединяет в скаляр свой аргумент t типа $tensor^1$ с параметром θ_0 , который тоже представляет собой $tensor^1$. Затем она прибавляет полученный скаляр к θ_1 .

Правильно.

Мы комбинируем непараметризованный нелинейный решатель $rectify$ с $linear^{1,1}$, чтобы получить параметризованную нелинейную функцию $relu^{1,1}$:

```
(define relu1,1
  (λ (t)
    (λ (θ)
      (rectify (linear1,1 t) θ))))
```

17 Довольно странное имя $relu^{1,1}$, не так ли?

Имя $relu$ – это сокращение от *rectifying linear unit* (спрямляющий линейный элемент).

Напомним, верхний индекс «1,1» указывает, что t и θ_0 оба относятся к типу $tensor^1$.

18 Понятно, потому что функция получена комбинацией $rectify$ и $linear^{1,1}$.

Теперь это стало очевидно, правда?

Поясните, как работает функция $relu^{1,1}$.

19 Она спрямляет скалярный результат, полученный функцией $linear^{1,1}$.

Совершенно верно.

Функция $relu^{1,1}$ принимает взвешенное решение относительно аргумента, которым является тензор t .

20 Что такое взвешенное решение?

Каждый элемент тензора θ_0 называется весом. От этого веса зависит, насколько сильно соответствующий элемент аргумента влияет на конечный результат.

Чем ближе вес к 0.0, тем меньше элемент аргумента влияет на результат.

21 Это очень похоже на w из фрейма 20:11.
А что такое θ_1 ?

Хороший вопрос.

22 Как это происходит?

Этот компонент называется смещением (bias). Он соответствует b во фрейме 20:11.

Параметр смещения сдвигает точку, относительно которой *rectify* принимает свое решение, чтобы на выходе получить 0.0.

Если смещение положительное, оно увеличивает результат операции $\bullet$, повышая его шанс пройти через *rectify* в неизменном виде.

23 Ясно, от смещения зависит, будет ли окончательный результат равен 0.0.

Аналогично, если смещение отрицательное, оно уменьшает результат операции $\bullet$, уменьшая его шанс пройти через *rectify*, не превратившись в 0.0.

Правильно.

24 Можно взглянуть на пример $relu^{1,1}$ в действии?

Если аргументом является нулевой тензор, только от смещения зависит, будет ли равен нулю результат $relu^{1,1}$.

Рассмотрим логический столбец

с набором весов

θ_0 , равным [7.1 4.3 -6.4],

со смещением

θ_1 , равным 0.6,

и

с тензором аргумента

t , равным [2.0 1.0 3.0]:

1. $((relu^{1,1} [2.0 \ 1.0 \ 3.0])$
 $(list [7.1 \ 4.3 \ -6.4] 0.6))$
2. $(rectify$
 $(+$
 $(\bullet [7.1 \ 4.3 \ -6.4] [2.0 \ 1.0 \ 3.0])$
 $0.6))$

Завершите этот логический столбец.

Прекрасно.

Теперь объясните, почему мы получили 0.0.

Хорошо.

Функции наподобие $relu^{1,1}$ называются искусственными нейронами. Каждый нейрон имеет линейную часть, такую как $linear^{1,1}$, и нелинейный решатель, такой как $rectify$.

25 Вот завершение:

3. $(rectify$
 $(+$
 $(sum$
 $(\bullet [7.1 \ 4.3 \ -6.4] [2.0 \ 1.0 \ 3.0]))$
 $0.6))$
4. $(rectify$
 $(+$
 $(sum [14.2 \ 4.3 \ -19.2])$
 $0.6))$
5. $(rectify$
 $(+ -0.7 0.6))$
6. $(rectify -0.1)$
7. 0.0

26 Мы получили 0.0, потому что линейная комбинация θ_0 , θ_1 и t дает -0.1, что меньше нуля.

Вот почему функция $rectify$ не пропускает этот результат на выход и возвращает 0.0.

27 Очевидно, термин «нейрон» позаимствован в биологии.

Правило искусственных нейронов

Искусственный нейрон представляет собой комбинацию параметризованной линейной функции и нелинейного решателя.

Так и есть.

Функция $relu^{1,1}$ – это упрощенная модель реального нейрона головного мозга.

Вот почему нашу составную обучаемую функцию называют нейронной сетью, или, для краткости, нейросетью[†].

[†] Термин ввели в обиход Уоррен МакКаллох (1898–1968) и Уолтер Питтс (1923–1969).

28 Функция $relu^{1,1}$ выглядит очень просто.

Как она, в самом деле, может различать ирисы?

Отличный вопрос.

Имея достаточно большое количество таких функций-«кирпичиков», мы можем строить очень сложные функции.

29 В это сложно поверить.

Тем не менее это так.

Давайте рассмотрим пример того, как многократное применение $relu^{1,1}$ позволяет делать весьма интересные вещи.

30 Звучит интригующе!

Начнем с построения графика $relu^{1,1}$ для $\theta_0 = [1.0]$ и $\theta_1 = -1.0$.

Мы предполагаем, что для любого входного x мы найдем y , вызывая

$$((relu^{1,1} [x]) \theta)$$

Для примера найдите y при $x = 0.5$.

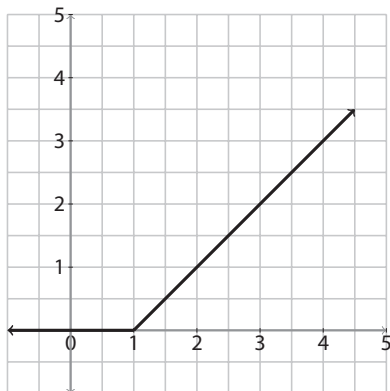
³¹ Вот логический столбец:

1. $((relu^{1,1} [0.5]) (list [1.0] -1.0))$
2. $(rectify$
 $(+$
 $(\bullet [1.0] [0.5])$
 $-1.0))$
3. $(rectify$
 $(+$
 $(sum$
 $(\bullet [1.0] [0.5]))$
 $-1.0))$
4. $(rectify -0.5)$
5. 0.0

Мы получили точку с координатами $(0.5, 0.0)$ на графике.

Верно.

Так выглядит график:



³² Он выглядит как линия с наклоном 1.0, которая отсечена ниже оси x .

Да.

Она имеет резкий изгиб в точке пересечения с осью x .

Объясните, почему.

33 Дело в том, что *rectify* не пропускает на выход значения меньше нуля.

Замечательно.

Допустим, что θ_0 – это *tensor*¹ с ровно одним элементом, и назовем этот элемент p . Другими словами, θ_0 – это

$[p]$

По аналогии с *line*, p определяет наклон верхней части графика, а θ_1 – место, где происходит отсечка по оси y (или могла бы быть, если бы *rectify* не препятствовала).

Найдите точку, в которой график *relu*^{1,1} встречается с осью x и возникает излом.

34 Уравнение линии имеет вид

$$y = px + \theta_1$$

В точке пересечения с осью x значение y равно 0.0. Решим это уравнение для x :

$$0.0 = px + \theta_1$$

$$-px = \theta_1$$

$$x = -\frac{\theta_1}{p}$$

Все правильно.

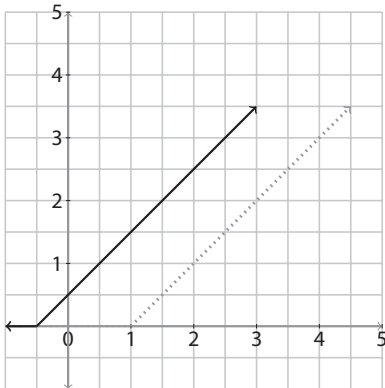
В этом уравнении, если значение p положительное, x и θ_1 движутся в противоположных направлениях из-за наличия знака «минус».

Когда θ_1 возрастает, x убывает, и

когда θ_1 убывает, x возрастает.

35 Означает ли это, что график должен смещаться вправо, когда θ_1 уменьшается, и влево – когда возрастает?

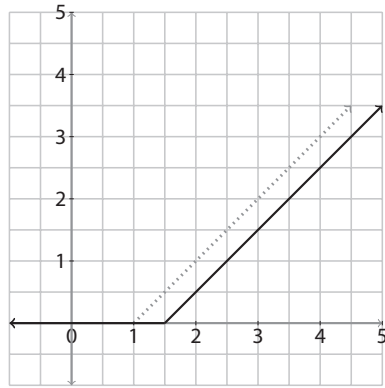
Так выглядит график, когда θ_1 увеличивается с -1.0 до $+0.5$:



Здесь сдвинутая линия (т. е. когда $\theta_1 = 0.5$) – сплошная. График сдвинулся влево, когда мы увеличили θ_1 .

Как будет выглядеть график, если мы уменьшим θ_1 с -1.0 до -1.5 ?

36 Сплошная линия окажется справа от точечной:



Но как это поможет нам построить интересные функции?

Проявите терпение! Будьте благородными, в конце концов.

Поясните работу функции *half-strip*:

```
(define half-strip
  (λ (x θ)
    (- ((relu1,1 [x]) (list θ0 θ1))
      ((relu1,1 [x]) (list θ0 θ2))))))
```

37 Для данного x эта функция определяет результаты $relu^{1,1}$ – сначала для θ_0 и θ_1 , а затем для θ_0 и θ_2 . Наконец, она вычитает второй результат из первого.

Да, правильно.

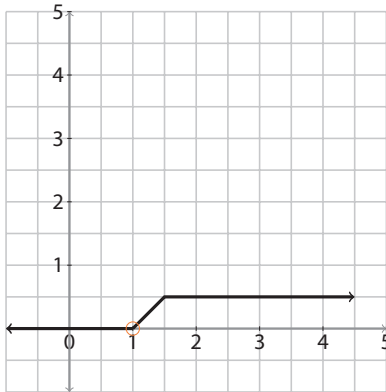
Давайте посмотрим на график *half-strip*, когда значение θ_0 такое же, как раньше (т. е. $[1.0]$), и

$$\theta_1 = -1.0$$

Пусть также

$$\theta_2 = -1.5$$

Здесь две $relu^{1,1}$ такие же, как на графике во фрейме 36.



Почему график приобрел такую форму?

Зазор между осью x и линией графика выглядит как полоса (*strip*), которая начинается там, где линия графика встречается с осью x . Мы называем это место левым концом полосы. Конец помечен оранжевым кружком.

Конечно.

Поскольку у этой полосы только один конец, а не два, ее называют полуполосой (*half-strip*).

- 38 При любом $x < 1.0$ обе $relu^{1,1}$ равны 0.0, поэтому сплошная линия лежит на оси x . Между $x = 1.0$ и $x = 1.5$ первая $relu^{1,1}$ становится положительной, но вторая по-прежнему возвращает 0.0.

Эта разница наблюдается до $x = 1.5$, но затем обе $relu^{1,1}$ начинают расти с одинаковой скоростью, сохраняя разницу в 0.5.

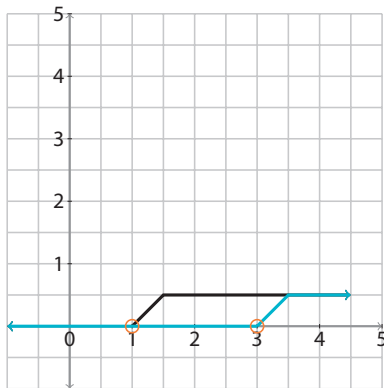
Однако почему эта функция называется *half-strip* (полуполоса)? И почему один изгиб отмечен оранжевым кружком?

- 39 Эта полоса выглядит так, будто у нее только один конец.

- 40 Значит, мы можем получить функцию *full-strip*, у которой есть оба конца полосы?

Да, можем – путем вычитания двух полуполос.

Ниже показан другой график, на котором изображены две линии функций *half-strips* с разными значениями θ_1 и θ_2 . Пусть $\theta_1 = -3.0$ и $\theta_2 = -3.5$.



Черная линия соответствует первой полуполосе из фрейма 38, а голубая – второй полуполосе.

Правильно.

Теперь объединим две полуполосы, чтобы определить полную полосу:

```
(define full-strip
  (λ (x θ)
    (- (half-strip x (list θ0 θ1 θ2))
       (half-strip x (list θ3 θ4 θ5)))))
```

⁴¹ Хорошо.

Вторая полуполоса сдвинута вправо, так как p из фрейма 34 – положительное число, меньшие значения $\theta_1 = -3.0$ и $\theta_2 = -3.5$ заставляют голубую полуполосу сместиться вправо относительно черной полуполосы с соответствующими значениями -1 и -1.5 .

⁴² Это выглядит так, будто мы вычитаем вторую полуполосу из первой.

Покажите, как будет выглядеть график.

Охотно.

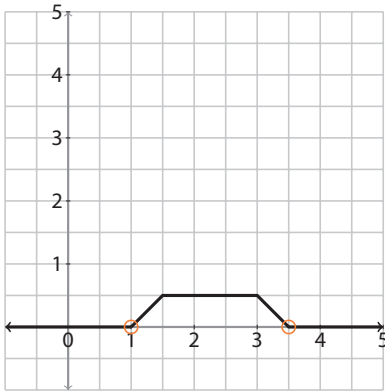
Воспользуемся параметрами двух полуполос из фрейма 41.

При вызове *full-strip* параметрам θ_0 и θ_3 присвоено одинаковое значение [1.0].

Параметру θ_1 присвоим значение -1.0 , а θ_2 — значение -1.5 .

Наконец, $\theta_4 = -3.0$, и $\theta_5 = -3.5$.

Построим график функции:



Почему график выглядит именно так?

Меткое наблюдение.

Такое количество параметров позволяет нам иметь разные наклоны на двух концах полосы, а также контролировать ширину и высоту полосы.

43 Обе полуполосы возвращают нулевое значение y для всех x меньше 1.0.

Между $x = 1.0$ и $x = 1.5$ первая полуполоса возрастает до 0.5, но вторая остается равной 0.0. Черная линия, представляющая разность, возрастает до 0.5 и сохраняет это значение до $x = 3.0$.

При $x = 3.0$ значение второй полуполосы начинает возрастать до 0.5, соответственно, разность начинает снижаться на интервале до $x = 3.5$, где возвращается к 0.0, и сохраняет это значение при всех остальных x .

Почему нам потребовались 6 разных параметров функции *full-strip*?

44 А почему функция называется *full-strip* (полная полоса)?

Она получила такое название, потому что полоса между черной линией и осью x теперь имеет левый конец (при $x = 1.0$) и правый конец (при $x = 3.5$), где черная линия встречается с осью x второй раз.

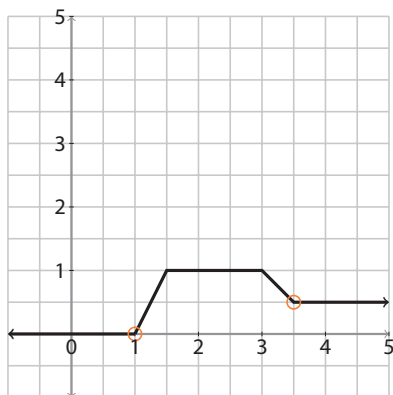
Манипулируя параметрами $\theta_0, \theta_3, \theta_1, \theta_4, \theta_2$ и θ_5 , мы можем получить полосу любого размера и с любым наклоном на двух концах.

Полные и половинные полосы можно комбинировать при помощи сложения и вычитания.

Например:

```
(+ (full-strip x
  (list [1.0] -1.0 -1.5
        [1.0] -3.0 -3.5))
  (half-strip x
    (list [1.0] -1.0 -1.5)))
```

Вот этот график:



45 Что толку от этих полу- и полных полос?

46 Сложение функций полной и половинной полос должно дать нам нелинейную функцию.

Можно увидеть ее график?

47 У нас теперь есть оранжевые кружки на концах полос и более интересный график.

Да.

То, насколько интересна функция, которую мы можем создать, обычно определяется количеством параметров, доступных для обновления, и количеством вызовов $relu^{1,1}$, которыми мы располагаем.

Сколько параметров и вызовов $relu^{1,1}$ мы имеем в данном случае?

48 Функция *half-strip* нуждается в трех параметрах и использует 2 вызова $relu^{1,1}$.

Функция *full-strip* нуждается в 6 параметрах и использует 4 вызова $relu^{1,1}$.

Итого функция во фрейме 46 имеет 9 параметров и использует 6 вызовов $relu^{1,1}$.

Поскольку $relu^{1,1}$ представляет собой искусственный нейрон, это пример того, как комбинация простых искусственных нейронов дает нам более сложные функции.

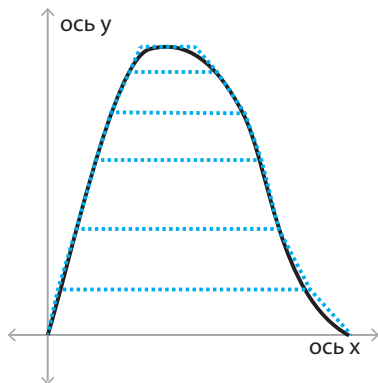
49 Как можно создать еще более интересные функции с помощью нейронов?

Чтобы получить еще больше интересных функций, мы разобьем их на полные полосы и полуполосы[†] и объединим их с помощью сложения и вычитания.

50 Можно ли при помощи *full-strip* построить полосы на этом графике, а затем сложить их?

[†] Предложил Анри Лебег (1875–1941).

Ниже показана очень интересная функция, разбитая на полосы (показаны голубыми пунктирными линиями):



Да, можно.

Принято говорить, что вместе эти полосы аппроксимируют исходный график.

⁵¹ Что в данном случае означает слово «аппроксимировать»?

Края полос представляют собой прямые линии, но график самой функции не обязательно точно совпадает с этими прямыми линиями. Например, взгляните на верхнюю полосу на графике во фрейме 50.

⁵² Понятно.

Это означает, что полосы приближают нас к исходному графику функции, но всегда остается небольшое различие с плавной кривой[†].

Вот почему мы говорим, что полосы аппроксимируют функцию[‡].

[†] Это так называемая кусочно-линейная аппроксимация.

[‡] Мы можем увеличить количество полос, уменьшив высоту каждой полосы, и это позволит лучше приблизиться к исходной кривой.

Хотя мы показали простой пример только с тензорами tensor^1 , длина которых всегда равна 1, общие принципы были доказаны для tensor^1 любой длины и для многих различных типов решателей. Результаты этого доказательства называются универсальными теоремами аппроксимации[†].

53 Ого! Такой способ построения функции-цели с помощью полос выглядит весьма трудоемким.

[†] Авторы теорем: Хэлберт Уайт (1950–2012), Джордж Цыбенко (1952–), Максвелл Стинкомб (1957–) и Курт Хорник (1963–).

Идея использования полос состоит только в том, чтобы показать, что искусственные нейроны можно использовать в качестве строительных блоков для очень сложных функций. Мы не будем использовать этот метод на практике.

54 Так как же мы будем строить нейронные сети на практике?

Мы будем делать это более элегантным способом – и более простым для разработки и реализации.

55 Превосходно.
Что там у нас на десерт?

Но эта тема может подождать до следующей главы.

Нейронные побрякушки

rectify стр. 196

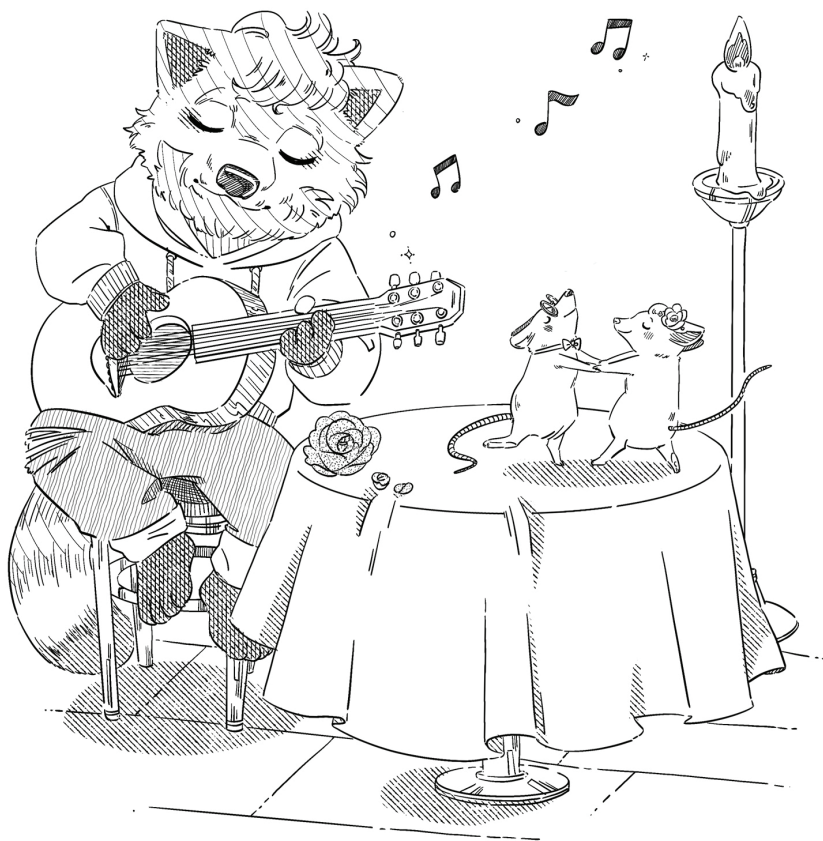
linear^{1,1} стр. 197

relu^{1,1} стр. 198

Нам срочно нужен
русский медовик!

ВСЕ ЕГО ОДИННАДЦАТЬ СЛОЕВ!

11. ПРЕКРАСНЫЕ ФОРМЫ RELU¹



¹ Глава названа в честь песни Shape of You, Эд Ширан (1991–).

Как вам медовик[†]?

1 Нет слов, просто ах и ох!

[†] Прекрасный медовик подавали в 20th Century Café (2013–2021), кондитер Мишель Пользин.

Мы выбрали именно медовик, потому что дальше речь пойдет о слоях.

2 О слоях какого рода?

В главе 10 говорилось про отдельные нейроны.

3 Что мы понимаем под слоем в этом случае?

В этой главе мы упорядочим нейроны в слои, чтобы строить более крупные нейронные сети.

Мы можем рассматривать слой как группу нейронов (наподобие $relu^{1,1}$), которые вместе работают с одним тензором.

4 Хорошо, но что такое функция слоя?

Говоря более техническим языком, слой реализует так называемую функцию слоя, а θ содержит параметры этой функции.

Функция слоя имеет форму:

5 Выглядит как обычная функция-цель.

$(\lambda(t)$

$(\lambda(\theta)$

... тело, генерирующее тензор

...))

Так и есть, и все функции слоя представляют собой функции-цели.

6 Но функции слоя нужны параметры θ , прежде чем она произведет новый тензор.

Тензорный аргумент t функции слоя называют входом слоя.

Верно. Функция слоя имеет смысл, только когда известны вход и θ и результирующие выходы каждого нейрона объединяются в тензор, который является результатом работы функции слоя.	7	Этот результат называют выходом слоя?
Ну конечно!	8	Пора показать пример, верно?
Да, самое время. Давайте начнем с примера слоя, который состоит из 4 нейронов. Как мы только что сказали, этот слой получает тензор и возвращает тензор. Допустим, он получает tensor^1 длиной 7.	9	Единственная разновидность нейронов, которую мы знаем, – это $\text{relu}^{1,1}$.
Правильно. Функция слоя затем вызывает 4 таких $\text{relu}^{1,1}$ для тензора t , у которого длина 7.	10	Должны ли мы при вызове также передать какие-либо веса (из фрейма 199:21) и смещения (из фрейма 199:22) в составе θ ?
Очень меткое замечание. Допустим, у нас есть 4 веса в форме tensor^1 : $w _0, w _1, w _2, w _3$	11	А что насчет смещений?
Пусть смещения для каждой из четырех $\text{relu}^{1,1}$ будут $b _0, b _1, b _2, b _3$	12	Хорошо.
Какова форма каждого из этих весов? $w _0, w _1, w _2, w _3$	13	Так как мы передаем их в $\text{relu}^{1,1}$, они должны иметь ту же форму, что и t , то есть (list 7)

Располагая этой информацией, что мы можем сказать о форме w ?

14 Мы можем сказать, что w имеет форму
(list 4 7)!

А что вы скажете о форме b ?

15 Так как смещения для любой $relu^{1,1}$ должны быть скалярами, b представляет собой тензор формы
(list 4)

Великолепно.

Теперь w и b становятся первыми двумя элементами θ , с которым мы вызываем функции $relu^{1,1}$.

Что возвращают эти четыре $relu^{1,1}$?

16 Они производят четыре скаляра.
Должны ли мы объединить эти скаляры в tensor¹?

Вы очень проникательны. Вот способ записи функции слоя:

```
(λ (t)
  (λ (θ)
    (let ((w θ0) (b θ1))
      [((relu1,1 t) (list w0 b0))
        ((relu1,1 t) (list w1 b1))
        ((relu1,1 t) (list w2 b2))
        ((relu1,1 t) (list w3 b3))]))))
```

Здесь мы вызываем $relu^{1,1}$ четыре раза с входным тензором t и строим четыре разных θ из элементов w и b .

Да, но скоро мы приведем ее в порядок.

Здесь важно то, что каждая $relu^{1,1}$ производит скаляр, а это означает, что выходом данного слоя будет $tensor^1$ формы

(list 4)

18 Итак, функция слоя получает вход t формы

(list 7)

и параметры θ

с весами формы

(list 4 7)

и смещения формы

(list 4)

чтобы произвести результат формы

(list 4)

Можно ли обобщить эту функцию?

Конечно можно.

Слой с такой структурой называется плотным[†].

[†] Также называется полностью связанным, или полносвязным.

19 Мне кажется, этот термин отражает отношение между формами тензоров в слое.

Так и есть!

В общем случае функция плотного слоя из m нейронов получает аргумент $tensor^1 t$ формы

(list n)

Далее она получает подходящий θ

с весами формы

(list $m n$)

и смещениями формы

(list m)

и вызывает функцию каждого из m нейронов для t , получая

m

скаляров в форме $tensor^1$.

Мы говорим, что ширина слоя равна m .

20 В нашем примере $m = 4$ и $n = 7$.

Поскольку плотный слой ширины m производит тензор формы

(list m),

в нашем примере тензор должен иметь форму

(list 4).

Для этого должен быть отдельный закон!

Закон о плотных слоях (начальная версия)

Функция плотного слоя из m нейронов получает
вход tensor^1 из n элементов и возвращает
выход tensor^1 из m элементов.

Судя по фрейму 17, выражение для результирующего тензора несколько неуклюже.

21 Есть более простой способ записать его?

Конечно есть!

Функция $*^{2,1}$ во фрейме 191:48 послужит хорошей отправной точкой. Это функция двух аргументов. Первый – tensor^2 , второй – tensor^1 .

22 Она перемножает каждый tensor^1 в составе первого аргумента со вторым аргументом (тоже tensor^1).
Получается, в нашем примере функция слоя во фрейме 17 должна использовать $*^{2,1}$ и получать тот же результат вместо использования четырех отдельных вызовов $\text{relu}^{1,1}$?

В самую точку.

Так как w представляет собой tensor^2 , он становится первым аргументом $*^{2,1}$, а входной тензор t становится вторым аргументом.

Если w имеет форму

$(\text{list } m \ n)$

и t имеет форму

$(\text{list } n)$

то результат вызова

$(*^{2,1} \ w \ t)$

также имеет форму

$(\text{list } m \ n)$

23 Но $\text{relu}^{1,1}$ выполняет не только умножение.

Разве мы не должны сделать что-то еще?

Закон о плотных слоях (финальная версия)

Функция плотного слоя из m нейронов получает вход tensor^1 из n элементов и возвращает выход tensor^1 из m элементов за один вызов $\bullet^{2,1}$.

Должны!

Функция $\text{relu}^{1,1}$ использует $\bullet$ и rectify , чтобы получить окончательный результат. Если мы хотим воспроизвести поведение m функций $\text{relu}^{1,1}$, то должны написать функцию $\bullet^{2,1}$, которая использует $\bullet^{2,1}$ для получения m скаляров:

```
(define  $\bullet^{2,1}$ 
  ( $\lambda$  (w t)
    (sum
      ( $\bullet^{2,1}$  w t))))†
```

[†] Возможно, кто-то узнал здесь произведение матрицы и вектора.

Да, это так.

Если

форма w ($\text{list } m \ n$)

и

форма t ($\text{list } n$)

получают форму результата
($\bullet^{2,1} w \ t$).

24 Определение такое же, как для функции $\bullet$, за исключением того, что мы используем $\bullet^{2,1}$ вместо $\bullet$.

25 Во фрейме 23 ($\bullet^{2,1} w \ t$) возвращает результат с формой ($\text{list } m \ n$). Затем для этого результата вызывается функция sum .

Функция sum понижает ранг каждого из m вложенных тензоров tensor^1 до одиночного скаляра, которые в итоге дают tensor^1 формы ($\text{list } m$).

Следовательно, результатом операции $\bullet^{2,1}$ между tensor^2 с формой

(list m n)
и tensor^1 с формой
(list n)
должен быть tensor^1 с формой
(list m).

Да, правильно.

Составьте логический столбец
для

$$(\bullet^{2,1} w t)$$

где w

$$\begin{bmatrix} 2.0 & 1.0 & 3.1 \\ 3.7 & 4.0 & 6.1 \end{bmatrix}_{(2\ 3)}$$

и t

$$[1.3\ 0.4\ 3.3]^\dagger$$

Вот начало логического столб-
ца:

$$1. \quad \left| \begin{array}{c} (\bullet^{2,1} w t) \end{array} \right|$$

Завершите его, но будьте
очень аккуратны.

[†] В этом и некоторых следующих при-
мерах m меньше n . Однако в общем
случае m и n независимы и не огра-
ничиваются друг другом.

26 Мы используем расширенные
операторы, поэтому должны
спускаться в тензоры, пока не
достигнем базового ранга:

2. $\left| \begin{array}{c} (\bullet^{2,1} \begin{bmatrix} 2.0 & 1.0 & 3.1 \\ 3.7 & 4.0 & 6.1 \end{bmatrix}_{(2\ 3)} \\ [1.3\ 0.4\ 3.3] \end{array} \right|$
3. $(\text{sum}_{(\ast^{2,1}} \begin{bmatrix} [2.0\ 1.0\ 3.1] & [3.7\ 4.0\ 6.1] \\ [1.3\ 0.4\ 3.3] \end{bmatrix}))$
4. $(\text{sum} \begin{bmatrix} [(\ast [2.0\ 1.0\ 3.1] [1.3\ 0.4\ 3.3]) \\ (\ast [3.7\ 4.0\ 6.1] [1.3\ 0.4\ 3.3])] \end{bmatrix}))$
5. $(\text{sum} \begin{bmatrix} [2.6\ 0.4\ 10.23] \\ [4.81\ 1.6\ 20.13] \end{bmatrix}))$
6. $[(\text{sum} [2.6\ 0.4\ 10.23]) (\text{sum} [4.81\ 1.6\ 20.13])]$
7. $[(\text{sum}^1 [2.6\ 0.4\ 10.23]) (\text{sum}^1 [4.81\ 1.6\ 20.13])]$
8. $[13.23\ 26.54]$

Отлично.

Вот объявление функции *linear*, которая идентична *linear*^{1,1} из фрейма 197:14, за исключением того, что она использует $\bullet^{2,1}$ вместо $\bullet$:

```
(define linear
  (λ (t)
    (λ (θ)
      (+ ( $\bullet^{2,1}$  θ0 t) θ1))))
```

27 Понятно.

Какой должна быть форма θ_1 ?

Теперь допустим, что

t

имеет форму

(list n)

и

θ_0

имеет форму

(list m n)

28 Во фрейме 25 сказано, что

($\bullet^{2,1}$ θ_0 t)

имеет форму

(list m)

Так как θ_1 представляет собой смещение, прибавляемое к выходу $\bullet^{2,1}$, и поскольку нам нужно, чтобы каждый нейрон имел отдельное смещение, этот параметр также должен иметь форму

(list m)

Что это говорит нам о форме

((linear t) θ)?

29 Его форма будет

(list m)

то есть это

tensor¹

длина которого совпадает с количеством нейронов слоя.

Точно!

Теперь определим функцию слоя *relu*, которая аналогична *relu*^{1,1} из фрейма 198:17, за исключением того, что она использует *linear* вместо *linear*^{1,1}:

```
(define relu
  (λ (t)
    (λ (θ)
      (rectify (linear t) θ))))
```

³⁰ Как ее можно использовать в нашем примере из фрейма 9?

Мы должны вызвать ее для

t

который имеет форму

(list 7)

и для

θ

образованного объединением

(list *w* *b*)

где

w

имеет форму

(list 4 7)

и

b

имеет форму

(list 4)

Какой будет форма выходного тензора?

³¹ В качестве функции слоя у нас выступает *relu*, поэтому количество нейронов определяется формой *θ*.

Поскольку *rectify* не влияет на форму

(*linear t*)

он имеет форму, аналогичную *tensor*¹, такой же длины, что и количество нейронов в слое.

В нашем примере это

(list 4)

Совершенно верно. Количество m нейронов в слое со входом длины n должно следовать из θ, где θ_0 имеет форму (list $m\ n$) а θ_1 имеет форму (list m) Какова форма выходного тензора этого слоя?	32 Она совпадает с количеством нейронов в слое (list m)
Очень хорошо. Мы используем эту связь между формой t, θ_0, θ_1 в <i>relu</i> для разработки наших нейросетей.	33 Можно ли ознакомиться с другими примерами этих форм?
Конечно. Допустим, у нас есть входной tensor¹ с формой (list 4) и мы хотим передать его слою с тремя нейронами, т. е. должны получить тензор с формой (list 3) Какую форму должен иметь θ_0?	34 В соответствии с фреймом 32 $n = 4$, $m = 3$. Следовательно, θ_0 должен иметь форму (list 3 4)
Какой должна быть форма θ_1?	35 Она должна быть такой же, как (list m) который в данном случае будет (list 3)

Список форм параметров tensor^2 и tensor^1 , необходимых слою, так и называется – списком форм слоя.

Каким должен быть список форм для слоя из предыдущего примера?

36 Он должен быть таким:

```
(list  
  (list 3 4)  
  (list 3))
```

А в общем виде для плотного слоя из m нейронов и входа длиной n ?

37 Вот общий вид:

```
(list  
  (list  $m$   $n$ )  
  (list  $m$ ))
```

Отлично.

Теперь в соответствии с подсказкой во фрейме 3 объединим простые сети, используя единственную знакомую нам функцию слоя *relu*.

38 Мы познакомимся с другими функциями слоя?

Всему свое время, а пока мы ограничимся только *relu*.

Вот простая сетевая функция:

39 Что такое сетевая функция?

```
| (define 1-relu  
|   ( $\lambda$  ( $t$ )  
|     ( $\lambda$  ( $\theta$ )  
|       ((relu  $t$ )  $\theta$ ))))†
```

Здесь у нас 1 слой, потому что только один вызов функции слоя *relu*.

[†] Это примечание можно пропустить.
Данное тело функции упрощается до *relu*.

Сетевая функция собирает вместе функции слоя таким образом, что выход одного слоя становится входом другого.

40 Но это также выглядит как обучаемая функция?

Так и есть.	41	Если они настолько похожи, зачем мы дали им разные имена?
Их конечные цели различны. Сетевые функции предназначены для использования в качестве обучаемых функций в процессах оптимизации градиентного спуска, где происходит обучение θ. Функции слоя, в свою очередь, используются для построения сетевых функций.	42	Значит, функция <i>1-relu</i> – это однослойная сетевая функция, построенная с использованием функции слоя <i>relu</i>. Хотелось бы взглянуть на сетевую функцию с более чем одним слоем.
Так выглядит каркасный код двухслойной сетевой функции: <pre>(define 2-relu (λ (t) (λ (θ) ((relu ((relu t) θ)) R))))</pre>	43	Почему это двухслойная сетевая функция?
Результирующий тензор первого (внутреннего) вызова <i>relu</i> передается во второй (внешний) вызов <i>relu</i>.	44	Точно, выход первого слоя становится входом второго слоя. Разве мы не собираемся найти R?
Буквально через минуту. Давайте поглубже вникнем в каркасный код. Каждая из <i>relu</i> требует два тензорных параметра в θ Сколько тензорных параметров должен содержать θ?	45	Так как в этой сетевой функции у нас два сетевых слоя, нам нужны два тензорных параметра для каждого слоя. Значит, набор θ должен содержать четыре тензорных параметра.

Правильно.

Первые два параметра, θ_0 и θ_1 , предназначены для первого слоя (т. е. для внутренней *relu*).

Внутренняя *relu* имеет к ним доступ напрямую в $\theta^\dagger$.

Имеет ли значение тот факт, что когда первая функция слоя обращается к θ , в нем присутствует 4 элемента?

† Немного более строгой версией этого кода было бы создание нового списка из θ_0 и θ_1 и передача его самой внутренней *relu*. Однако здесь мы используем более простой вариант.

Хорошо.

Оставшиеся параметры θ_2 и θ_3 являются аргументами внешней *relu*.

Теперь найдите R .

46 Во фрейме 27 мы видели, что *relu* включает в себя функцию *linear*, которая использует только элементы θ_0 и θ_1 .

Поэтому не имеет значения, что θ для первого слоя имеет больше двух элементов, ведь функция слоя не использует оставшиеся элементы.

47 R служит источником аргументов θ для внешней *relu* и должен возвращать список, состоящий из θ_2 и θ_3 . Очевидно, это $(\text{list } \theta_2 \theta_3)$?

Верно.

Но есть другая, более общая функция, которую можно здесь использовать.

48 Что это за функция?

Это функция, которая возвращает нам остаток списка, начиная с i -го элемента непустого списка l , где i – положительное число.

49 Покажите ее на примере.

Запишем ее:

l_{il}

Например:

1. $(\text{list } 2\ 4\ 8\ 9\ 6\ 3\ 7)_{4!}$
2. $(\text{list } 6\ 3\ 7)$

50 Ага!

Этот пример возвращает нам остаток списка, начиная с индекса 4.

Значит, вместо

$(\text{list } \theta_2\ \theta_3)$

мы можем использовать

$\theta_{2!}$

Замечательно.

Дадим определение функции *2-relu*:

```
| (define 2-relu
|   (λ (t)
|     (λ (θ)
|       ((relu
|         ((relu t) θ))
|         θ2!))))
```

Мы используем $\theta_{2!}$, чтобы получить остаток θ , начиная с индекса 2. А теперь определим трехслойную сетевую функцию *3-relu*.

51 Можем ли мы использовать *2-relu* для определения *3-relu*?

Конечно можем[†]!

Определите функцию *3-relu*, которая является трехслойной сетевой функцией, с помощью трех *relu*.

[†] Спасибо Кейту Чапману за создание мультсериала *Bob the Builder* (1997–).

52 Как насчет такого?

```
| (define 3-relu
|   (λ (t)
|     (λ (θ)
|       ((2-relu
|         ((relu t) θ))
|         θ2!))))
```


Очень хорошо.

Сначала мы вызываем *relu* для входа t , и она использует первые два элемента θ . Так получается выход первого слоя.

Таким образом, θ должен иметь два тензорных параметра для каждой *relu*.

Теперь объясните оставшуюся часть.

Конечно можно!

Но дойдем мы до этого в несколько шагов. Начнем с простой рекурсивной функции, которая делает следующее:

```
| (define k-relu
|   (λ (k t θ)
|     (cond
|       ((zero? k) t)
|       (else (k-relu (sub1 k)
|                     ((relu t) θ)
|                     θ2↓))))
|
```

Мы начинаем с тензора t и пропускаем его через k вызовов *relu*; при каждом вызове используем два элемента θ , чтобы в итоге получить выходной тензор.

53 Мы передаем выход первого слоя в *2-relu*, которая представляет собой двухслойную сетевую функцию, и передаем ей все элементы θ , кроме первых двух.

Можно ли обобщить эти функции на любое натуральное число k слоев?

54 Определение построено таким образом, чтобы хоть одно из значений выражения уменьшалось. Здесь k уменьшается из-за *(sub1 k)*, а θ уменьшается из-за $\theta_{2↓}$.

Но это не совсем правильно, не так ли?

Вот именно!

Проблема в том, что сетевые функции, такие как *1-relu*, *2-relu* и *3-relu*, показанные выше, должны принимать свои аргументы t и θ по одному за раз. Таким образом, *k-relu* нуждается во вложенных λ -выражениях. Кроме того, в клаузе `else` должно быть больше круглых скобок.

55 Можно поподробнее?

Да, конечно.

В предыдущей версии с пунктиром три аргумента. В следующей версии с пунктиром мы берем первый аргумент и отделяем в собственное λ -выражение:

```
(define k-relu
  (λ (k)
    (λ (t θ)
      (cond
        ((zero? k) t)
        (else ((k-relu (sub1 k))
                  ((relu t) θ)
                  θ2↓))))))
```

Единственным следствием этого является то, что когда мы вызываем *k-relu*, то должны сначала указать аргумент для k , а затем указать два других –

$((\text{relu } t) \theta)$

и

$\theta_{2↓}$

56 Другими словами, вместо вызова

$(k\text{-relu } (\text{sub1 } k) ((\text{relu } t) \theta) \theta_{2↓})$

мы выполняем вызов

$((k\text{-relu } (\text{sub1 } k)) ((\text{relu } t) \theta) \theta_{2↓})$

Мы можем также повторить это для

$(\lambda (t \theta) \dots)$

Правильно.

Вот финальная версия *k-relu*:

```
(define k-relu
  (λ (k)
    (λ (t)
      (λ (θ)
        (cond
          ((zero? k) t)
          (else (((k-relu (sub1 k))
                    ((relu t) θ)
                     θ21)))))))))†
```

57 У этого определения довольно сложный вид.

[†] Каждый шаг начиная с фрейма 54 называется «каррирование». Спасибо за идею Мозесу Шёнкелю и Хаскеллу Карри.

Давайте проанализируем его в каждом конкретном случае.

Когда $k = 0$, у нас нет слоев.

Поэтому выход совпадает со входом.

Когда $k > 0$, мы находим результат первого слоя

$((relu\ t)\ \theta)$

вызывая *relu* для входа t , а затем берем результат и применяем к нему θ . Здесь мы используем первые два элемента θ .

Что происходит дальше?

58 Выражение $(k-relu\ (sub1\ k))$ дает нам нейронную сеть, состоящую из $k - 1$ слоев. Мы вызываем эту чуть меньшую сеть на результате первого слоя

$((relu\ t)\ \theta)$

и передаем ей оставшиеся элементы

θ

Можно рассмотреть пример?

Вот пример, где $k = 4$, остальными аргументами являются t и θ :

1. $((((k\text{-relu } 4) \ t) \ \theta))$
2. $((((k\text{-relu } (\text{sub1 } 4)) ((\text{relu } t) \ \theta)) \theta_{2\downarrow}))$
3. $((((k\text{-relu } 3) ((\text{relu } t) \ \theta)) \theta_{2\downarrow}))$
4. $((((k\text{-relu } 2) ((\text{relu } ((\text{relu } t) \ \theta)) \theta_{2\downarrow})) \theta_{2\downarrow 2\downarrow}))$
5. $((((k\text{-relu } 1) ((\text{relu } ((\text{relu } ((\text{relu } t) \ \theta)) \theta_{2\downarrow})) \theta_{2\downarrow 2\downarrow})) \theta_{2\downarrow 2\downarrow 2\downarrow}))$

Завершите логический столбец.

Великолепно.

Давайте присмотримся более внимательно к θ , поступающему в сетевую функцию, созданную с использованием $k\text{-relu}$.

Как должен выглядеть θ для любого заданного k ?

⁵⁹ Вот завершение:

6. $((((k\text{-relu } 0) ((\text{relu } ((\text{relu } ((\text{relu } ((\text{relu } t) \ \theta)) \theta_{2\downarrow})) \theta_{2\downarrow 2\downarrow})) \theta_{2\downarrow 2\downarrow 2\downarrow})) \theta_{2\downarrow 2\downarrow 2\downarrow 2\downarrow}))$
7. $((\text{relu } ((\text{relu } ((\text{relu } ((\text{relu } t) \ \theta)) \theta_{2\downarrow})) \theta_{2\downarrow 2\downarrow})) \theta_{2\downarrow 2\downarrow 2\downarrow}))$

Итого $(k\text{-relu } 4)$ дает нам 4 вызова relu !

⁶⁰ В определении $k\text{-relu}$ мы видим, что каждый рекурсивный вызов $k\text{-relu}$ сопровождается операцией $\theta_{2\downarrow}$, которая убирает 2 тензорных параметра из θ .

Так как это случается k раз для k слоев, длина θ равна $2k$.

В этом θ какие элементы являются весами, а какие – смещениями?

⁶¹ Из определения *relu* мы знаем, что θ_0 – это вес, а θ_1 – смещение.

Это означает, что в θ для k слоев каждый элемент с четным индексом[†] является тензором весов, а с нечетным – тензором смещений.

[†] Поскольку индексация списков начинается с 0, мы условно относим 0 к четным числам.

Как найти тензор весов для i -го слоя?

⁶² Нужно взять элемент с индексом $2i$.

А тензор смещений?

⁶³ Нужно взять элемент с индексом $2i + 1$.

Превосходно.

Итак, допустим, что ширина i -го слоя нейросети равна

m

и длина его входа равна

n

Какой будет форма тензоров с индексами

$2i$

и

$2i + 1$?

⁶⁴ Элемент с индексом $2i$ является тензором весов и имеет форму

$(\text{list } m \ n)$

Элемент с индексом $2i + 1$ является тензором смещений и имеет форму

$(\text{list } m)$

Великолепно.

Пора рассмотреть несколько более крупный пример. Возьмем трехслойную сеть, где входной tensor^1 имеет форму

$(\text{list } 32)$

⁶⁵ Это функция

$(k\text{-relu } 3)$

и ширина

первого плотного слоя – 64,
второго плотного слоя – 45,
третьего плотного слоя – 26.

Какой будет сетевая функция
для этой сети?

И какова будет длина θ ?	66	Она равна удвоенному количеству слоев, т. е. 6.
Верно. Первый слой имеет ширину 64 нейрона, форма входа (list 32) Следовательно, θ_0 должен иметь форму (list 64 32) а θ_1 должен иметь форму (list 64) Что вы можете сказать о втором слое?	67	На вход второго слоя поступают данные с выхода первого слоя, поэтому он имеет форму (list 64) Ширина второго слоя равна 45, значит, θ_2 должен иметь форму (list 45 64) а θ_3 должен иметь форму (list 45)
Третий слой получает выходные данные второго слоя, поэтому форма его входа (list 45)	68	Какими будут θ_4 и θ_5 ?
Ширина этого слоя 26 поэтому θ_4 должен иметь форму (list 26 45) Что вы скажете про θ_5 ?	69	Форма θ_5 получается напрямую из ширины слоя (list 26)

Правильно.

Если мы объединим все перечисленные формы в общий список, то получим список форм (shape list) нейросети.

Каким будет список форм для текущей сети?

70 Вот он:

(list
(list 64 32)
(list 64)
(list 45 64)
(list 45)
(list 26 45)
(list 26))

Превосходно.

Выражение (k -relu 3) и список форм вместе полностью описывают нейросеть из текущего примера.

71 Как мы перейдем от этого описания нейронной сети к полностью работоспособному образцу?

Шаг за шагом!

Мы изучим следующий шаг в новой главе.

72 Ох... не пора ли нам подкрепиться?

Форменные побрякушки

•^{2,1} стр. 219

linear стр. 221

relu стр. 222

l_{i_l} (где l не пустой и $i > 0$) стр. 226

k-relu стр. 229

Как насчет тройного ягодного трайфла?

СО ВЗБИТЫМИ СЛИВКАМИ, КОНЕЧНО!

12. ПОСТРОИМ БЛОК, СЫГРАЕМ РОК¹



¹ Глава названа в честь многочисленных рок-музыкантов и композиторов 1950–1960-х годов.

Ну как, освежили свои силы? ¹ Трехслойный ягодный трайфл – то, чего мне не хватало.

Теперь вернемся к нашей трехслойной сети из фрейма [232](#):65, где входом служит `tensor1` с формой

² Да, для нее у нас есть сетевая функция

$(k\text{-relu } 3)$

`(list 32)`

и шириной

первого слоя – 64,

второго слоя – 45,.

третьего слоя – 26

Какой у нее список форм? ³ Это интересно.

Мы упоминали его во фрейме [234](#):70

```
(list
  (list 64 32)
  (list 64)
  (list 45 64)
  (list 45)
  (list 26 45)
  (list 26))
```

Правильно.

В данном случае мы построили нашу сетевую функцию отдельно от ее списка форм и построили список форм, учитывая только ширину каждого слоя.

⁴ В чем проблема такого подхода?

Хотя в таком разделении есть определенная выгода, при описании более крупных и сложных сетей удобнее

определять функции слоев и формы совместно для каждого слоя,

а затем

укладывать слои в стек, составляя из них единую сетевую функцию и общий список форм.

5 Хмм... довольно абстрактная идея.

Сейчас мы воплотим ее в реальность.

Начнем со знакомства с блоками[†].

[†] Так мы будем для краткости называть основные составляющие части сетей.

6 Что такое блок?

Блок связывает функцию слоя с ее списком форм.

7 Можно взглянуть на пример?

Пожалуйста!

Вот блок первого слоя в мере нашей нейросети:

```
(define layer1
  (block relu
    (list
      (list 64 32)
      (list 64))))
```

8 Первый аргумент – функция слоя *relu*, второй аргумент – список форм для 64 нейронов и *tensor*¹ длиной 32 из фрейма 233:67.

Должны ли мы аналогично определить второй слой?

Здесь мы вызываем функцию *block* с двумя аргументами.

Объясните эти два аргумента.

Да, должны.

Напишите код блока *layer2* для второго слоя с 45 нейронами.

9 Формы для второго слоя тоже указаны во фрейме 233:67. Так определяется блок *layer2*:

```
(define layer2
  (block relu
    (list
      (list 45 64)
      (list 45))))
```

А что насчет третьего слоя?

10 Он определяется так:

```
(define layer3
  (block relu
    (list
      (list 26 45)
      (list 26))))
```

Но мы по-прежнему не видим, что делает функция *block*!

Пора с этим разобраться. Вот определение функции *block*:

11 То есть мы просто объединяем их в список?

```
(define block
  (λ (fn shape-lst)
    (list fn shape-lst)))
```

Здесь

fn

обозначает

функцию блока (*block function*)

и

shape-lst

обозначает

список блока (*block list*)

Очень просто, не правда ли?

Определите функцию *block-fn*, которая получает блок и возвращает его функцию, и *block-ls*, которая получает блок и возвращает его список форм.

12 Вот эти определения:

```
(define block-fn
  (λ (ba)
    ba0))

(define block-ls
  (λ (ba)
    ba1))
```

А теперь немножко магии.

Как показано во фрейме 234:71, нейросеть можно полностью описать ее сетевой функцией и списком форм.

13 Ой, это значит, что нейросеть тоже может быть блоком! То есть, уложив блоки в стек, мы получаем новый блок.

Давайте определим таким способом нейросеть из примера во фрейме 232:65:

14 Что такое *stack-block*?

```
(define 3-layer-network
  (stack-blocks
    (list
      layer1
      layer2
      layer3)))
```

Здесь *stack-block* – это функция, которая получает список блоков и производит новый блок, функция которого

представляет собой комбинацию функций отдельных блоков

и список форм которого

получен объединением отдельных списков форм.

15 Мне кажется, с этим нужно разобраться получше.

Тогда давайте разберем эту идею на нашем примере.

Нам нужно, чтобы сеть сначала вызывала *relu* из *layer1* для входного *tensor*¹ длины 32, используя первые два параметра θ , чьи формы задаются списком форм *layer1*.

Что произойдет с выходом при таком вызове?

Абсолютно точно.

Итак, в нейросети три *relu* из каждого из блоков слоев объединяются и образуют композитную функцию, получающую шесть параметров из заданного θ .

Что вы скажете о форме этих шести параметров?

Да, но небольшими частями.

Так выглядит функция *block-compose*. Она ожидает две блочные функции *f* и *g* в качестве двух первых аргументов. Третьим аргументом *j* является количество параметров θ , которые должна использовать *f*:

```
(define block-compose
  ( $\lambda$  (f g j)
    ( $\lambda$  (t)
      ( $\lambda$  ( $\theta$ )
        ((g
          ((f t)  $\theta$ ))
            $\theta_j$ ))))))
```

Поясните, как работает эта функция.

¹⁶ Выходом этого вызова *relu* будет *tensor*¹ длины 64.

Далее мы вызываем для него *relu* из *layer2*, используя следующие два параметра θ , формы которых заданы в списке форм *layer2*.

Потом мы поступаем аналогично с выходом этого вызова, но уже используя *layer3*.

¹⁷ Формы этих шести параметров получаются объединением списков форм каждого из трех блоков слоя.

Первые две из *layer1*, третья и четвертая из *layer2*, пятая и шестая из *layer3*.

Теперь мы можем объявить *stack-blocks*?

¹⁸ Она возвращает блочную функцию, которая ожидает тензор *t* в сопровождении θ , а затем вызывает *f* для *t* и θ .

Результат этого вызова поступает в качестве аргумента в *g* вместе с θ , из которого были удалены первые *j* параметров, т. к. они были предназначены для *f*.

Хотелось бы увидеть пример работы этой функции.

Пожалуйста.

Объединим две *relu*. Мы знаем, что *relu* ожидает два параметра, поэтому $j = 2$:

1. $(\text{block-compose } \text{relu } \text{relu } 2)$
2. $(\lambda (t)$
 $(\lambda (\theta)$
 $((\text{relu}$
 $((\text{relu } t) \theta))$
 $\theta_2)))$

¹⁹ Но эта функция совпадает с *2-relu*?

Так и есть!

Этим же способом можно определить *2-relu*:

```
(define 2-relu
  (block-compose relu relu 2))
```

²⁰ Вряд ли этого достаточно, чтобы определить *stack-blocks*.

Вы правы, недостаточно.

Нам еще нужно найти способ объединять два блочных списка.

²¹ Значит, нужно объявить новую функцию.

Мы назовем ее *append*. Вот как она работает:

1. $(\text{append } (\text{list } 3 \ 6 \ 1) (\text{list } 7 \ 2))$
2. $(\text{list } 3 \ 6 \ 1 \ 7 \ 2)$

²² Как она ведет себя с блочными списками *layer1* и *layer2*?

Давайте посмотрим.

²³ Вот завершение:

1. $(\text{append}$
 $(\text{list}$
 $(\text{list } 64 \ 32)$
 $(\text{list } 64))$
 $(\text{list}$
 $(\text{list } 45 \ 64)$
 $(\text{list } 45)))$

1. $(\text{list}$
 $(\text{list } 64 \ 32)$
 $(\text{list } 64))$
 $(\text{list } 45 \ 64)$
 $(\text{list } 45))$

Завершите логический столбец.

Итак, *append* сохраняет формы, но объединяет списки форм в порядке следования аргументов.

Правильно.

Мы можем использовать *block-compose* из фрейма 18 и *append* из фрейма 22 для определения функции *stack2*, которая укладывает в стек два блока – *ba* и *bb*. Так выглядит каркасный код:

```
(define stack2
  (λ (ba bb)
    (block
      (block-compose
        (block-fn ba)
        (block-fn bb)
        B
      )
      (append
        C
        D
      )
    )))
```

Найдите *B*, *C* и *D*.

²⁴ *B* является третьим аргументом *block-compose*, поэтому он представляет собой количество параметров, потребляемых блочной функцией

(block-fn ba),

то есть

|(block-ls ba)|

C и *D* – это списки форм, которые должны быть соединены для получения финального списка форм блока.

Следовательно,

C – это *(block-ls ba)*

и

D – это *(block-ls ba)*.

Отлично.

Так выглядит *stack2*:

```
(define stack2
  (λ (ba bb)
    (block
      (block-compose
        (block-fn ba)
        (block-fn bb)
        |(block-ls ba)|
      )
      (append
        (block-ls ba)
        (block-ls bb)
      )
    )))
```

²⁵ А теперь мы готовы объявить *stack-blocks*?

Теперь – да!

Функция *stack-blocks* получает один аргумент *bls*, представляющий собой список блоков, которые нужно уложить в стек:

```
(define stack-blocks
  (λ (bls)
    (stacked-blocks bls1↓ bls0↓)))
(define stacked-blocks
  (λ (rbls ba)
    (cond
      ((null? rbls) ba)
      (else
       (stacked-blocks rbls1↓
                        A
                        )))))
```

Здесь предикат *null?* проверяет, содержит ли список элементы. Укладка в стек выполняется с помощью функции *stacked-blocks*, первый аргумент которой представляет собой список блоков, а второй аргумент – блок, с которого начинается стек.

²⁶ Похоже, это определение следует правилу простой передачи аккумулятора.

Мы вызываем *stacked-blocks* с первым блоком в *bls* в качестве второго аргумента, а первым аргументом являются оставшиеся блоки в *bls*.

Мы должны найти *A*, верно?

Да, конечно!

Функция *stacked-blocks* получает два аргумента. Первым является

список блоков *rbls*

а вторым –

блок *ba*

начиная с которого должны быть стекированы блоки из *rbls*.

Второй аргумент *ba* можно рассматривать как аккумулятор, который хранит частично объединенный блок.

Теперь найдите А.

27 Если *rbls* пуст, мы не выполняем стекирование и возвращаем блок *ba*.

В противном случае мы должны объединить блок-аккумулятор

ba

с первым блоком

rbls₀

и использовать его как новое значение аккумулятора, пока мы продвигаемся вниз по остальным блокам с помощью *stacked-blocks*.

Итак, вместо А нужно подставить

(stack2 ba rbls₀)

Превосходно.

У нас получилось окончательное определение:

```
(define stack-blocks
  (λ (bls)
    (stacked-blocks bls1 bls0)))
(define stacked-blocks
  (λ (rbls ba)
    (cond
      ((null? rbls) ba)
      (else
       (stacked-blocks rbls1
                        (stack2 ba rbls0))))))
```

28 Как насчет примера, демонстрирующего функцию в действии?

С удовольствием.

²⁹ Вот завершение:

1. `(stack-blocks`
`(list`
`layer1`
`layer2`
`layer3))`
2. `(stacked-blocks`
`(list`
`layer2`
`layer3)`
`layer1)`
3. `(stacked-blocks`
`(list`
`layer3)`
`(stack2 layer1 layer2))`
4. `(stacked-blocks`
`(list)`
`(stack2`
`(stack2 layer1 layer2)`
`layer3))`
5. `(stack2`
`(stack2`
`(block relu`
`(list`
`(list 64 32)`
`(list 64)))`
`(block relu`
`(list`
`(list 45 64)`
`(list 45))))`
`layer3)`

6. `(stack2`
`(block ( $\lambda$  (t)`
`( $\lambda$  ( $\theta$ )`
`((relu`
`((relu t)  $\theta$ ))`
`$\theta_{21}$ )))`
`(list`
`(list 64 32)`
`(list 64)`
`(list 45 64)`
`(list 45)))`
`(block relu`
`(list`
`(list 26 45)`
`(list 26))))`
7. `(block ( $\lambda$  (t)`
`( $\lambda$  ( $\theta$ )`
`((relu`
`((relu t)  $\theta$ ))`
`$\theta_{21}$ )))`
`$\theta_{41}$ )))`
`(list`
`(list 64 32)`
`(list 64)`
`(list 45 64)`
`(list 45)`
`(list 26 45)`
`(list 26)))`

Завершите логический столбец.

Правило о блоках

Блоки можно укладывать в стек для создания более крупных блоков или полных сетей.

Превосходно.

Мы также можем определить функции, которые производят блоки определенного типа. Вот полезная функция:

```
(define dense-block
  (λ (n m)
    (block relu
      (list
        (list m n)
        (list m))))))
```

Здесь n – длина входного тензора, m – количество нейронов. Поясните, что делает *dense-block*.

³⁰ Она возвращает блок плотного слоя с *relu* в качестве функции блока и соответствующий список форм плотного слоя из m нейронов, работающего со входом *tensor*¹ длины n .

Верно.

Перепишите наши определения *layer1*, *layer2* и *layer3*, используя *dense-block*.

³¹ Теперь они будут выглядеть так:

```
(define layer1
  (dense-block 32 64))

(define layer2
  (dense-block 64 45))

(define layer3
  (dense-block 45 26))
```

Когда у нас есть сеть, определенная таким образом, и есть набор данных, мы можем найти точно обученный θ для сетевой функции, причем элементы θ имеют точно такие формы, какие заданы списком форм сети.

О том, как это сделать, мы расскажем в следующей главе.

³² Какой у нас следующий десерт?

Блочные побрякушки

block стр. 238

block-fn стр. 239

block-ls стр. 239

stack-blocks стр. 244

dense-block стр. 246

Как насчет стека
из блинчиков «Сюзетт»?

С ФЛАМБЕ!

Эта страница специально оставлена пустой
для капель карамели с блинчиков.

13. ВЗГЛЯНИТЕ НА ИРИС¹



¹ В оригинальном названии *An Eye for an Iris* игра слов: *iris* означает название цветка и радужки глаза.

Как вам стек с блинчиками 1 Восхитительно!
«Сюзетт»†?

† Спасибо Генри Карпентеру (1880–1961) за рецепт и Джулии Чайлд (1912–2004) за его популяризацию.

Что ж, пора за дело! 2 Сначала яблоки, теперь ирисы!
Познакомьтесь с дядюшкой
Эдгаром. Он любит выращи-
вать ирисы.

В саду у дядюшки Эдгара рас- 3 Как много растений!
тут 150 ирисов трех разных
сортов:

Iris Setosa (Ирис щетинистый)

Iris Versicolor (Ирис разноцветный)

Iris Virginica (Ирис виргинский)

У него по 50 растений каждого
сорта.

Дядюшка Эдгар буквально 4 Какого рода данных?
одержим сбором данных о
своих ирисах.

Он берет каждый цветок в 5 Что такое чашелистик?
своем саду и измеряет
ширину и длину чашелисти-
ка (sepal),
ширину и длину лепестка
(petal)

и записывает, на растении
какого сорта вырос этот цве-
ток†.

Дядюшка Эдгар уверен, что
этих четырех скалярных
значений достаточно, чтобы
корректно классифицировать
ирисы по сортам.

† Набор данных о цветах ириса собрал
Эдгар Андерсон (1897–1969).

Чашелистик – это одна из частей цветочной чашечки (calyx). Чашечка соединяет цветок со стеблем.

⁶ Как выглядит набор данных дядюшки Эдгара?

Вот пример записи в наборе данных дядюшки Эдгара:

⁷ Эти четыре числа образуют tensor^1 длиной 4.

Длина лепестка: 5.1 см

Ширина лепестка: 3.5 см

Длина чашелистика: 1.4 см

Ширина чашелистика: 0.2 см

Сорт: *Setosa*

Верно.

⁸ Но у не является тензором!

Перепишем эту запись набора данных иначе:

x: [5.1 3.5 1.4 0.2]

y: *Setosa*

Да, это так.

⁹ Как это сделать?

Мы не можем передать этот набор данных в наши функции напрямую. Сперва мы должны закодировать набор дискретных сортов как тензор.

Поскольку у нас есть 3 сорта, мы создадим tensor^1 длины 3, присвоив индекс каждому сорту[†].

¹⁰ Как мы закодируем, допустим, *Versicolor*?

Например, мы можем сказать, что

индекс 0 соответствует *Setosa*

индекс 1 соответствует *Versicolor*

индекс 2 соответствует *Virginica*

[†] Эти индексы могут быть присвоены произвольно, но мы выбрали алфавитный порядок сортов.

Так как у *Versicolor* индекс 1, мы можем сделать элемент тензора с индексом 1 равным 1.0, а остальные два индекса будут равны 0.0. Получившийся тензор будет выглядеть так:

```
[0.0 1.0 0.0]
```

Какими буду тензоры для *Setosa* и *Virginica*?

¹¹ Индекс *Setosa* равен 0; тензор будет следующим:

```
[1.0 0.0 0.0]
```

Индекс *Virginica* равен 2; тензор будет следующим:

```
[0.0 0.0 1.0]
```

Правильно.

Этот прием называется прямым унитарным кодированием (one-hot encoding)[†].

Это распространенный способ поставить тензор в соответствие классам. В данном случае у нас три класса, по одному на сорт.

[†] Изобрел Леопольд Кронекер (1823–1891).

¹² Почему мы используем именно 1.0?

Почему бы не использовать любое другое скалярное значение, например 328.9?

Очень хороший вопрос.

Дело в том, что здесь применяется степень доверия (degree of belief).

¹³ Что такое степень доверия?

Это мера нашей уверенности в каком-либо утверждении.

Допустим, мы точно знаем, что держим в руках ирис сорта *Versicolor*. Значит, мы на 100 % доверяем гипотезе, что ирис принадлежит этому классу:

$$100 \% = 100/100 = 1.0$$

¹⁴ Значит, если мы говорим, что определенные индексы представляют сорта значением 1.0, это означает нашу 100%-ную уверенность в принадлежности цветков к этим сортам.

Правильно. И в то же время если ирис принадлежит к какому-то сорту, мы на 0 % уверены, что он принадлежит к другим сортам.	15 Вот почему скалярные значения, соответствующие другим индексам, равны 0.0. Следовательно, выходной тензор отражает нашу уверенность в том, что цветок ириса принадлежит к определенному сорту.
Да, это точная интерпретация выходного тензора. Итак, как будет выглядеть запись данных дядюшки Эдгара из фрейма 8 в тензорной форме?	16 Поскольку эта запись соответствует сорту <i>Setosa</i>, а индекс данного сорта равен 0, запись выглядит так: x: [5.1 3.5 1.4 0.2] y: [1.0 0.0 0.0]
Превосходно. Мы перепишем все точки данных таким образом, чтобы набор данных состоял только из тензоров. Как можно определить функцию для автоматической классификации нового ириса, если нам известны только его измерения?	17 Наверное, нужно использовать любимые побрякушки из нашего сундучка?
Конечно! Мы должны найти обучаемую функцию, чьи параметры θ можно обучить, используя градиентный спуск на данных дядюшки Эдгара.	18 Здорово. С чего начнем?
Мы начнем с разработки сети, состоящей из плотных слоев, как делали это в главе 12.	19 Восхитительно!
Первым делом нужно решить, сколько слоев нам нужно и какой ширины должен быть каждый слой.	20 От чего зависит это решение?

Начнем с выходного слоя, который является последним слоем сети.

Мы знаем, что каждый u в наших данных представлен унитарным кодом длиной 3, поэтому чтобы иметь возможность сравнить предсказанное значение u с истинным, оно должно представлять собой `tensor1` с формой

`(list 3)`

Исходя из знаний о плотных слоях, скажите, какой ширины должен быть выходной слой?

Верно.

Сеть, которую мы строим, связывает каждый вход x с классом y при помощи тензора унитарного типа (`one-hot-like tensor`).

Он похож на тензор с унитарным кодированием, но степень уверенности в каждом отдельном классе может принимать любое значение между 0.0 и 1.0. Мы введем для этого типа кодирования обозначение

one-hot-like

Да, именно так.

21 Он должен быть ширины 3.

22 Что такое тензор унитарного типа?

23 Означает ли это, что наша степень уверенности в заданном классе находится где-то между 0 % и 100 %?

24 Как мы тогда решаем, какой класс представляет тензор?

В кодировании унитарного типа предсказанным является класс с наибольшей степенью уверенности в нем. Допустим, мы предсказали у [0.2 0.7 0.1] Какой класс мы предсказали?	25	Здесь наибольшая степень доверия у индекса 1, который соответствует классу <i>Versicolor</i> Но нам потребуется больше одного слоя, не так ли?
Для набора данных дядюшки Эдгара нам придется добавить еще один слой.	26	И это будет считаться глубокой нейросетью?
Да, конечно! Пусть у этого слоя будет ширина 6.	27	Почему 6? Почему не 2 или 8?
В общем случае слой, который ближе ко входу, всегда шире слоя, который ближе к выходу.	28	Почему же?
Слои, расположенные ближе ко входу, отвечают за изучение некоторых более примитивных характеристик набора данных, а слои, расположенные ближе к выходу, изучают более сложные характеристики на основе выходов, полученных предыдущими слоями. Здесь мы выбираем ширину 6, потому что она в достаточной степени больше, чем 3. Фактически ширину слоев часто выбирают опытным путем.	29	Вы хотите сказать, что нам могут понадобиться эксперименты?

Да, верно.

30 Понятно.

Конфигурация сети и выбор скаляров для гиперпараметров часто определяются путем экспериментов со всем набором данных или его меньшими подмножествами, хотя знание статистических параметров набора данных может натолкнуть на некоторые решения.

Вернемся к нашей задаче с ирисами.

31 Да.

Итак, мы определились с выбором структуры нашей глубокой нейросети.

У нее два слоя, первый – шириной 6 нейронов, второй – 3 нейрона.

Напишите определение этой сети, используя функцию *dense-block* из фрейма 246:30.

32 Вот это определение:

```
(define iris-network
  (stack-blocks
    (list
      (dense-block 4 6)
      (dense-block 6 3))))
```

Что дальше?

Теперь мы обучим сеть.

33 Что означает обучение сети?

Обучение – это процесс подбора наиболее подходящих параметров сетевой функции θ с использованием обучающего набора данных.

34 Значит, для обучения сети *iris-network* мы должны найти наиболее точные параметры θ ее сетевой функции, используя набор данных дядюшки Эдгара.

Да, но сначала нужно сделать еще кое-что.

35 Почему бы нам не взять все нули, как мы делали раньше?

Мы должны задать стартовые значения θ перед началом процесса обучения.

Мы будем использовать нули для всех параметров смещения в θ .

36 Понятно. Это значит, что все tensor^1 в нашем θ будут инициализированы значениями 0.0.

Что насчет весов tensor^2 в наборе параметров θ ?

С весами совсем другая история, потому что здесь просматривается большая проблема.

Когда все скалярные значения в tensor^2 одинаковые (в данном случае 0.0), все элементы tensor^1 в составе tensor^2 идентичны.

Что это означает для результата функции $*^{2,1}$, которая является частью *relu*?

37 Все tensor^1 в выходе $*^{2,1}$ будут идентичны, и, следовательно, результат, возвращаемый *relu*, будет содержать одинаковые скаляры.

Верно.

Таким образом, вместо того чтобы каждый нейрон принимал небольшое решение о своих входных данных, все нейроны в слое в конечном итоге учатся принимать одно и то же решение снова и снова.

38 Получается, что нам нужны различающиеся скалярные значения элементов tensor^2 в θ .

У сетей на основе *relu* есть одно важное преимущество – их веса в θ можно инициализировать случайными скалярными значениями.

39 Какая нам польза от случайной инициализации?

Так как случайная инициализация в общем случае гарантирует, что каждый вес большую часть времени имеет уникальное значение, каждый нейрон сети будет вести себя независимо.

40 То есть каждый нейрон будет учиться принимать независимые решения.

Точно.

Это делает нейросеть более эффективной (чего мы от нее и добиваемся). Такой подход называется устранением симметрии нейронов.

41 Отлично, давайте же займемся этим.

Пока не получится.

Используя случайные веса, можно столкнуться со второй проблемой, присущей сетям с большим количеством слоев.

42 Что это за проблема?

Представьте один скаляр входного тензора. По мере того как он проходит через каждый слой, он умножается на скаляр веса и добавляется к сумме, а когда слоев много, каждый из этих слоев косвенно умножает этот входной скаляр.

43 То есть на выходе мы наблюдаем эффект входного скаляра, прошедшего через череду умножений на веса.

Так и есть.

Что может произойти, если все эти веса имеют достаточно большие значения?

44 Ой, выходные значения могут стать очень большими.
Что это значит для нашей сети?

Это значит, что числовые значения слишком велики для осмысленного результата и могут привести к вычислительным ошибкам при работе программы. Это явление принято называть

45 А что, если значения весов очень маленькие?

взрывом (exploding).

В таком случае многократное умножение на маленькие веса приведет к появлению очень маленького выходного значения, близкого к нулю. Это явление называют

исчезновением (vanishing).

⁴⁶ Плохо и то, и другое, так?

Со слишком маленькими или большими весами связана еще одна проблема. Она заключается в том, что градиенты, которые мы вычисляем при использовании градиентного спуска, также могут стать очень большими или очень маленькими. Это так называемые *проблемы взрывающегося и исчезающего градиента*.

Вместе эти проблемы могут существенно затруднить обучение сети.

⁴⁷ Значит, веса нашей сети должны быть случайными, не очень большими, не очень маленькими.

Это длинный список!

Вовсе нет.
Начнем с первого требования.
Существует функция
random-tensor

которая получает три аргумента:

c – центральное значение;
 v – дисперсия;
 s – форма.

⁴⁸ Эта функция возвращает тензор формы s со случайными скалярными значениями?

В точку!

Аргументы c и v определяют природу случайно сгенерированных скаляров.

⁴⁹ Как они это делают?

Аргумент центрального значения s определяет среднее всех случайных чисел в тензоре, которое должно быть как можно ближе к s .

Другими словами, случайные числа распределены в окрестностях значения s .

50 Понятно.

Что означает дисперсия?

Аргумент дисперсии v определяет, как далеко от s может отклониться случайное значение.

Например, меньшее значение v будет означать, что большинство случайных чисел образуют относительно плотный кластер рядом с s , а большее значение приведет к тому, что многие случайные значения будут располагаться дальше от s .

51 Как это помогает нам инициализировать нейросеть?

Чтобы избежать проблем нестабильности и исчезновения, мы всегда должны инициализировать веса со средним значением 0.0.

52 Какой должна быть дисперсия?

Мы не будем приводить математическое доказательство, но лучшее значение дисперсии определяется выражением

53 Получается, у каждого слоя своя оптимальная дисперсия, основанная на длине входа. Это любопытная формула.

$$\frac{2}{n}$$

где n – длина входа слоя[†].

[†] Это свойство присуще сетям, использующим *rectify*, и называется инициализацией Хе (см. эпилог).

Да, эта формула гарантирует, что в глубоких сетях веса остаются в очень узком кластере около 0.0, поэтому когда скаляр из входного тензора умножается на веса в каждом слое, результат не взрывается и не исчезает.

⁵⁴ Хорошо.

Итак, мы должны взять список форм *iris-network* и применить следующее правило инициализации.

Правило инициализации слоя (начальная версия)

Смещения слоя tensor^1 инициализируются значениями 0.0.

Веса слоя tensor^2 инициализируются случайными скалярами с центральным значением 0.0 и дисперсией $2/n$, где n – длина входа слоя.

Да, мы оформим это в функцию.

Ниже показано определение функции *init- θ* , которая получает единственный аргумент *shapes*, представляющий собой список, соответствующий θ :

```
(define init- $\theta$ 
  ( $\lambda$  (shapes)
    (map init-shape shapes)))
```

Ее задача – вернуть случайно инициализированный θ на основе форм, найденных в списке форм.

⁵⁵ Она применяет функцию *init-shape* к каждой форме в *shapes*.

Это означает, что *init-shape* должна генерировать тензор, который нам нужен, учитывая форму аргумента.

Правильно.

Когда эта форма представлена в виде

`(list m)`

мы знаем, что это соответствует тензору смещения. Это значит, что мы должны создать нулевой тензор такой формы.

Существует!

Она называется *zero-tensor*, получает *shape* как аргумент и возвращает нулевой тензор этой формы.

Например:

1. `(zero-tensor (list 5))`
2. `[0.0 0.0 0.0 0.0 0.0]`

56 Для этой операции существует функция?

57 Судя по всему, это именно то, что нужно.

Что, если форма соответствует тензору весов?

В случае плотных слоев форма, соответствующая тензору весов, представлена в виде

`(list m n)`

где m – количество нейронов в слое, а n – длина входа слоя.

Как мы должны инициализировать тензор такой формы?

58 Согласно правилу, он должен быть инициализирован с центральным значением 0.0 и дисперсией, указанной во фрейме 53.

Правильно.

Взгляните на каркасный код *init-shape*:

```
(define init-shape
  (λ (s)
    (cond
      ((= |s| 1) (zero-tensor s))
      ((= |s| 2)
       (random-tensor 0.0
                     V s))))
```

59 V – это дисперсия весов, которая равна 2, деленной на длину входа. В случае когда речь идет о тензорах весов, длина входа является вторым элементом формы, т. е.

s_1

Следовательно, V

$(\div 2 s_1)$

Найдите V .

Обновленное определение
init-shape

```
(define init-shape
  (λ (s)
    (cond
      ((= |s| 1) (zero-tensor s))
      ((= |s| 2)
       (random-tensor 0.0
        (÷ 2 s1 s))))))
```

⁶⁰ Теперь-то мы готовы начать обучение нейросети?

Да, теперь готовы!
Нам нужен обучающий набор данных.

⁶¹ Мы будем использовать набор дядюшки Эдгара для 150 растений?

Да, но не полностью.
Мы отделим и сохраним несколько записей набора данных для тестирования нейросети.

⁶² Почему мы не поступали так с предыдущими наборами данных?

Отличный вопрос.
Наши предыдущие наборы данных служили для иллюстрации процесса градиентного спуска и способов улучшить его.

⁶³ Сколько тестовых записей нужно отделить от набора данных?

На практике, когда параметры θ обучены, мы должны протестировать их на точках данных, которые нейросеть никогда не встречала, чтобы оценить ее точность.

Мы зарезервируем 10 %, или 15 записей набора данных, выбрав случайным образом по 5 записей для каждого класса. У нас получится так называемый тестовый набор (test set).

⁶⁴ Этого достаточно?

В случае больших наборов данных принято выделять для тестирования 20 % набора. Поскольку наш набор довольно небольшой, мы ограничимся 10 %.

⁶⁵ Ясно.

Мы назовем тестовые значения x и y так:

(iris-test-xs, iris-test-ys)

Оставшиеся 135 записей образуют обучающий набор. Их мы обозначим так:

(iris-train-xs, iris-train-ys)

⁶⁶ Вполне разумные имена. Можно начинать обучение?

Начинаем!

Какая обучаемая функция нам нужна?

⁶⁷ Это сетевая функция *iris-network*:
(block-fn iris-network)

Прекрасно.

Давайте дадим ей имя:

```
(define iris-classifier  
  (block-fn iris-network))
```

⁶⁸ Наверное, нужно дать имя списку форм?

Это хорошая идея!

```
(define iris- $\theta$ -shapes  
  (block-ls iris-network))
```

⁶⁹ Ну а теперь не могли бы мы уже обучить нейросеть?

Ниже представлен каркасный код для нахождения θ с использованием стохастического градиентного спуска с *naked-gradient-descent* и *sampling-obj*

```
(define iris- $\theta$ 
  (with-hypers
    ((revs 2000)
     ( $\alpha$  0.0002)
     (batch-size 8))
    (naked-gradient-descent
     (sampling-obj
      L
      iris-train-xs iris-train-ys)
     L
    )))
```

Найдите L и I .

Хороший ответ.

Ниже показано, как обучить нашу сеть, чтобы получить *iris- θ* , представляющий собой точно подобранный θ для текущего обучающего набора:

```
(define iris- $\theta$ 
  (with-hypers
    ((revs 2000)
     ( $\alpha$  0.0002)
     (batch-size 8))
    (naked-gradient-descent
     (sampling-obj
      (l2-loss iris-classifier)
      iris-train-xs iris-train-ys)
     (init- $\theta$  iris- $\theta$ -shapes))))
```

Функции *iris-classifier* и *iris- θ* вместе образуют модель.

⁷⁰ L – это ожидающая функция. Она представляет собой вызов *l2-loss* для обучаемой функции, т. е. для *iris-classifier*. Следовательно, вместо L подставим

(l2-loss iris-classifier)

Вместо I должна быть инициализация θ случайными значениями, но с формой, заданной в *iris- θ -shapes*. Значит, подставляем

(init- θ iris- θ -shapes)

⁷¹ Что такое модель?

Модель – это приближение к идеализированной функции, представляющей набор данных. Идеализированная функция не только возвращает точное значение u для каждого x из набора данных, но и способна вернуть соответствующее значение u для x , не входящего в набор.

⁷² Почему мы называем эту функцию идеализированной?

Мы называем ее идеализированной, потому что допускаем ее существование за пределами набора данных, но доказательством служит только сам набор данных. Другими словами, у нас нет выражения

$(\lambda (x) \dots \text{некий } u \dots)$

определяющего эту функцию.

Для наших ирисов мы определили *iris-model*:

```
| (define iris-model
|   (λ (t)
|     ((iris-classifier t) iris-θ)))
|
```

Объясните, что представляет собой эта функция.

⁷³ Эта функция сначала вызывает *iris-classifier* для входного тензора t , а затем *iris-θ*. Результатом является выходной тензор, возвращенный функцией *iris-classifier*.

Мы можем обобщить *iris-model* в функцию *model*, которая конструирует модель из своих двух аргументов – обучаемой функции и θ :

```
(define model
  (λ (target θ)
    (λ (t)
      ((target t) θ))))
```

Объясните, как работает эта функция.

⁷⁴ Она получает *target* и θ , возвращает функцию, ожидающую аргумент, и вызывает *target* для этого аргумента и данного θ .

Другими словами, она генерирует модель, выведенную из *target* и θ .

Превосходно.

Теперь определите *iris-model* с помощью *model*.

⁷⁵ Вот определение:

```
(define iris-model
  (model iris-classifier iris-θ))
```

Превосходно.

Если наша модель правильно обучена, она должна давать результаты, максимально приближенные к идеализированной функции.

Например, если *iris-model* дать новый набор измерений, которых нет в обучающем наборе, она по-прежнему правильно классифицирует большую часть случаев.

⁷⁶ Откуда мы знаем, что θ для данной модели обучен достаточно хорошо?

Это превосходный вопрос, но ответ мы будем искать в следующей интерлюдии!

⁷⁷ И для этого нам нужно запастись лакомствами, верно?

Классные побрякушки

init- θ стр. 261

zero-tensor стр. 262

random-tensor стр. 263

(iris-test-xs, iris-test-ys) стр. 264

(iris-train-xs, iris-train-ys) стр. 264

iris-classifier стр. 264

iris- θ -shapes стр. 264

iris- θ стр. 265

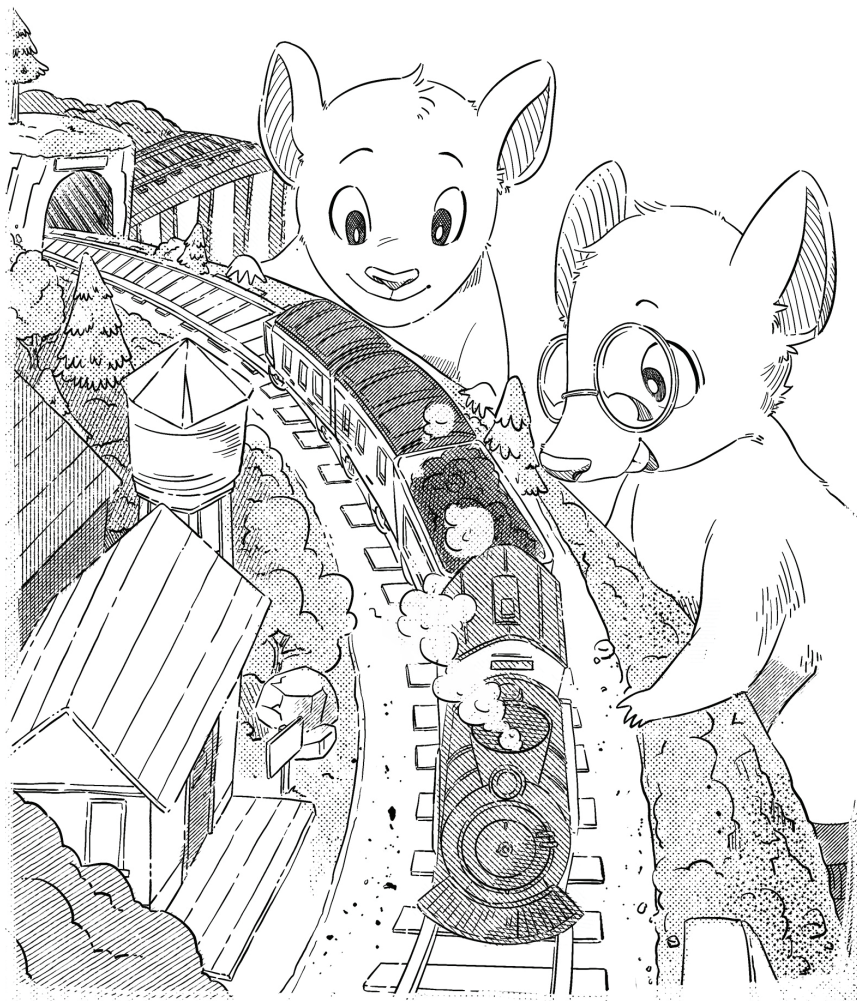
model стр. 267

iris-model стр. 267

Как насчет наисвежайшего милфея?

НАГРОМОЖДЕНИЕ СЛОЕВ
ВКУСНЯТИНЫ!

ИНТЕРЛЮДИЯ VI. КАК ОБУЧАЮТСЯ МОДЕЛИ¹



¹ Рисунок в заставке отражает игру слов: train переводится как «поезда» и «обучаться».

Мильфей был великолепен.	¹ Особенно его слоистая структура!
В этой главе мы научимся определять, достаточно ли обучена модель.	² Значит, мы сможем определить, когда <i>iris-model</i> стала достаточно хороша.
Верно. Но сначала нужно численно выразить понятие «достаточно хороша».	³ Это было бы полезно.
Опираясь на заданное входное значение, классификатор создает унитарный код, как показано во фрейме 254:23. Этот выходной <code>tensor</code> ¹ кодирует класс, к которому принадлежит входной объект.	⁴ Да. Индекс наибольшего значения уверенности дает нам класс объекта.
Чтобы определить, насколько хороша <i>iris-model</i> , мы запускаем ее на значениях <i>x</i> из тестового набора данных. Во фрейме 264:66 мы сохранили отдельно несколько записей <i>iris-test-xs</i> и <i>ris-test-ys</i> для тестирования. Пора им вступить в игру. Напишите выражение, позволяющее запустить модель на этом наборе.	⁵ Мы можем напрямую вызвать <i>iris-model</i> для набора <i>iris-test-xs</i> : (<i>iris-model iris-test-xs</i>)

Отлично. Результату этого выражения присвоим имя <i>iris-pred-ys</i>. Затем мы сравним классы, представленные в <i>iris-pred-ys</i>, с известными классами, представленными в <i>iris-test-ys</i> из набора дядюшки Эдгара. Мы говорим, что модель точна, если эти наборы классов совпадают.	6 Что, если классы не совпадают?
Мы будем называть такую ситуацию ошибкой классификации. Отношение количества правильных классификаций к общему количеству тестовых записей называется точностью модели.	7 Итак, мы можем измерить точность <i>iris-model</i> и решить, насколько хорошо подобран <i>iris-θ</i>.
Правильно.	8 Какой должна быть хорошая оценка точности? Это 1.0 (т. е. модель не ошибается на всех тестовых записях)?
Обычно это зависит от задачи, которую мы решаем, и если она достаточно сложна, то никакая модель не будет точной при всех входных значениях. Даже человек допускает ошибки. В данном случае мы будем считать достаточно высокой оценку 0.9.	9 То есть если правильно классифицированы 9 входных объектов из 10, мы считаем, что набор параметров <i>iris-θ</i> достаточно хорош.
Да, вы правы. Это достаточно универсальный способ оценки модели, и его можно применять для других классификаторов.	10 Ура! Означает ли это, что мы можем определить обобщенную функцию для измерения точности классификаторов?

Да, можем.

И мы сможем использовать эту функцию, чтобы понять, достаточно ли хороша модель для решения нашей задачи.

11 Это впечатляет.

Можно начать разрабатывать определение этой функции?

Конечно.

Существует функция argmax^1 , которая находит в tensor^1 индекс наибольшего значения уверенности. Она получает tensor^1 в унитарном коде и определяет индекс наибольшего скалярного значения. Например:

$(\text{argmax}^1 [0.1 \ 0.3 \ 0.6])$

Напишите логический столбец для этого выражения.

12 Наибольшее значение tensor^1 – это 0.6, которое соответствует индексу 2:

1. $(\text{argmax}^1 [0.1 \ 0.3 \ 0.6])$
2. $\mid 2$

Как мы определим эту функцию?

Вот начало argmax^1 :

```
(define argmax1
  (λ (t)
    (let ((i (sub1 † t †)))
      (argmaxed t i i))))
```

13 Здесь мы вызываем вспомогательную функцию argmaxed с простой передачей аккумулятора в аргументе.

Как определена функция argmaxed ?

Так выглядит каркасный код функции argmaxed . Она получает $\text{tensor}^1 t$ и a – индекс наибольшего элемента, обнаруженного в tensor^1 к этому времени:

```
(define argmaxed
  (λ (t i a)
    (let ((â (next-a t i a)))
      (cond
        ((zero? i) â)
        (else
         M )))))
```

14 Нам нужно определить, должно значение a измениться на i или остаться прежним.

Если индексу i соответствует большее скалярное значение в tensor^1 , чем индексу a , то мы должны использовать i в качестве следующего a .

В противном случае мы оставляем a без изменения:

Определите *next-a*, которая применяется для нахождения $\hat{a}$ – следующего значения *a*.

```
(define next-a
  (λ (t i a)
    (cond
      ((> ti ta) i)
      (else a))))
```

Отлично.

Теперь найдите *M*.

- 15 Когда индекс, отсчитываемый в обратную сторону, больше 0, мы продолжаем со следующим меньшим индексом, но теперь используем $\hat{a}$ в качестве нового *a*. Следовательно, вместо *M* нужно подставить

(argmaxed t (sub1 i) $\hat{a}$)

Превосходно.

Так выглядит полная функция *argmaxed*:

```
(define argmaxed
  (λ (t i a)
    (let (( $\hat{a}$  (next-a t i a)))
      (cond
        ((zero? i)  $\hat{a}$ )
        (else
         (argmaxed t (sub1 i)  $\hat{a}$ ))))))
```

Следует ли она правилу на стр. 40?

- 16 Конечно следует.

Поскольку *next-a* и *argmax*¹ не являются рекурсивными, мы можем игнорировать их при определении того, следует ли *argmaxed* закону простой передачи аккумулятора. Вызов внутри *argmaxed* не обернут, поэтому нам нужно смотреть только на формальные аргументы *argmaxed*

t не меняется

i меняется при прохождении основного теста

a накапливает результат.

Как мы будем использовать *argmax*¹?

Если у нас есть два tensor^1 , закодированных по унитарному принципу, допустим s и u , представляющих один класс, что мы можем сказать про

$(\text{argmax}^1 t)$

и

$(\text{argmax}^1 u)$?

Правильно.

Мы можем использовать это свойство, чтобы подсчитать количество успешных тестов.

Рассмотрим функцию

$\text{class}=1$

которая получает

два унитарных tensor^1

и проверяет, представляют ли они одинаковый класс:

```
(define class=1
  (λ (t u)
    (cond
      ((= (argmax1 t) (argmax1 u)) 1.0)
      (else 0.0))))
```

Объясните, что делает эта функция.

Скоро узнаете.

Расширим эту функцию:

```
(define class=
  (ext2 class=1 1 1))
```

¹⁷ Мы можем предположить, что они равны. Другими словами, выражение

$(= (\text{argmax}^1 t) (\text{argmax}^1 u))$

должно быть истинным.

¹⁸ Она возвращает 1.0, если оба tensor^1 представляют один и тот же класс, и 0.0 в противном случае.

Для чего нужна такая функция?

¹⁹ Ага, значит, мы можем сравнить *iris-pred-ys* с *iris-test-ys* с помощью одиночного вызова функции *class=*

$(\text{class}= \text{iris-pred-ys } \text{iris-test-ys})$

Но такой вызов всего лишь вернет нам tensor^1 со значением 1.0 или 0.0.

Это именно то, что нам нужно, потому что мы собираемся применить *sum* к этому *tensor*¹, чтобы сосчитать количество входов, для которых *iris-model* дала правильный прогноз:

```
(sum
  (class= iris-pred-ys iris-test-ys))
```

Как после этого найти точность?

Великолепно.

Мы можем обобщить это решение в функцию проверки точности заданной модели на тестовых значениях *x* и *y*:

```
(define accuracy
  (λ (a-model xs ys)
    (÷ C
      D)))
```

Найдите *C* и *D*.

Все верно.

Так выглядит окончательное определение *accuracy*:

```
(define accuracy
  (λ (a-model xs ys)
    (÷ (sum (class= (a-model xs)
                    ys)) † xs †)))
```

Как нам применить эту функцию для измерения точности *iris-model*?

²⁰ Нам нужно разделить результат на общее количество тестовых входов:

```
(÷ (sum
  (class= iris-pred-ys iris-test-ys))
  † iris-test-xs †)
```

²¹ Мы можем найти предсказанные *ys*

```
(a-model xs)
```

и сравнить их с исходными *ys*

```
(class= (a-model xs) ys)
```

а затем суммировать результат в *C*

```
(sum (class= (a-model xs) ys))
```

D – это количество входных значений, которое равно † *xs* †.

²² Мы можем вызвать функцию проверки точности так:

```
▷ (accuracy
  iris-model
  iris-test-xs iris-test-ys)
```

```
▶ 1.0
```

Ого! Получается, точность нашей модели равна 100 %?

Она демонстрирует точность 100 % на нашем маленьком тестовом наборе из 15 записей. Это необычная ситуация, потому что наш набор данных очень прост.

В случае реальных (намного более крупных) наборов данных точность редко достигает такого высокого уровня.

Значения этих гиперпараметров были получены опытным путем, но есть более систематический способ.

При подборе гиперпараметров опытным путем мы просто пробуем их различные комбинации.

Если нам известна последовательность скалярных значений гиперпараметров, мы можем систематически перебирать в цикле каждое скалярное значение для каждого гиперпараметра, пока не получим удовлетворительный набор θ .

Да.

Мы можем ее использовать, если обучаемая функция является классификатором. Для других типов обучаемых функций применяются другие тесты.

Этот способ тестирования различных комбинаций для выбора лучшей из них называется поиском по сетке (*grid search*).

23 Как мы пришли к скалярным значениям гиперпараметров, которые применяются при обучении *iris-classifier*?

24 Звучит интригующе!
Что это за способ?

25 Пригодится ли нам функция ассигасы при подборе θ ?

26 Нужно ли нам что-то еще, чтобы облегчить поиск по сетке?

Конечно нужно!

Вот функция, которую мы применим для поиска по сетке:

```
(grid-search
  accurate-enough-iris- $\theta$ ?
  ((revs 500 1000 2000 4000)
   ( $\alpha$  0.0001 0.0002 0.0005)
   (batch-size 4 8 16))
  (naked-gradient-descent
   (sampling-obj
    (l2-loss iris-classifier)
    iris-train-xs iris-train-ys)
   (init- $\theta$  iris- $\theta$ -shapes)))
```

27 Она совпадает с `with-hypers`, но, похоже, здесь должно быть больше одного скаляра для каждого гиперпараметра.

Конечно.

Здесь присутствует последовательность, содержащая как минимум один скаляр (обычно больше), среди которых выполняется поиск гиперпараметров по сетке.

28 Как эти последовательности задействованы в `grid-search`?

В этом примере `grid-search` стартует с

```
revs = 500
 $\alpha$  = 0.0001
```

и

```
batch-size = 4.
```

Затем он находит θ , используя тело

```
(naked-gradient-descent
  (sampling-obj
   (l2-loss iris-classifier)
   iris-train-xs iris-train-ys)
  (init- $\theta$  iris- $\theta$ -shapes))
```

29 Тело функции совпадает с тем, которое было в `with-hypers`.

Так и есть.

Получив набор θ , тело функции проверяет его с помощью вспомогательной функции *accurate-enough-iris- θ ?*, которая возвращает $\#t$, если этот θ обеспечивает достаточную точность.

30 Как выглядит определение функции *accurate-enough-iris- θ ?*

Вот определение функции *accurate-enough-iris- θ ?*, проверяющей, достаточно ли велика точность при текущем θ и применении модели *iris-classifier* к тестовому набору (*iris-test-xs*, *iris-test-ys*)

31 Эта функция возвращает $\#t$, если точность больше или равна 0.9.
Что произойдет, если θ не обеспечивает достаточную точность?

```
(define accurate-enough-iris- $\theta$ ?
  ( $\lambda$  ( $\theta$ )
    ( $\geq$  (accuracy
          (model iris-classifier  $\theta$ )
            iris-test-xs iris-test-ys)
          0.9)))
```

Отличный вопрос.

Если *accurate-enough-iris- θ ?* при текущем θ возвращает $\#t$, то *grid-search* возвращает этот θ .

Если же *accurate-enough-iris- θ ?* при текущем θ возвращает $\#f$, то *grid-search* пробует другую комбинацию.

Он сохраняет прежние значения *revs* и α , но переходит к новому скалярному значению *batch-size* = 8.

32 То есть новая комбинация гиперпараметров будет следующей:

```
revs = 500
 $\alpha$  = 0.0001
batch-size = 8?
```

Совершенно верно.

После того как поиск по сетке достигает конечного скаляра в последовательности *batch-size*, а результат все еще недостаточно точен, он возвращается к началу последовательности *batch-size*, то есть к 4.

Но гиперпараметру α назначается новое значение последовательности $\alpha = 0.0002$, и поиск снова выполняет перебор всех остальных значений *batch-size*.

33 А если будут исчерпаны все значения α и *batch-size*?

Произойдет почти то же самое, что и раньше: поиск начнется с исходных значений α и *batch-size*, но гиперпараметру *revs* будет присвоено новое скалярное значение, и будет продолжен поиск приемлемого θ .

34 Что произойдет, если поиск по сетке переберет все доступные комбинации значений гиперпараметров, но так и не найдет подходящий θ ?

В таком случае *grid-search* вернет *#f*.

35 Это означает, что поиск по сетке завершился неудачей?

Верно.

36 Понятно.

Когда это происходит, мы должны попробовать разные последовательности скаляров для гиперпараметров в поиске по сетке или, возможно, попробовать другую обучаемую функцию (например, сеть с другим количеством слоев или разной шириной слоев) для использования в качестве классификатора. В некоторых ситуациях нам, возможно, придется согласиться на более низкую точность.

Так выглядит *grid-search* в обобщенном виде:
(*grid-search good-enough?*
 (*hyperparameter scalar scalar ...*)
 ...)
 body)

где *body* формирует θ , *good-enough?* проверяет, достигнута ли необходимая точность при текущем θ , а последовательность *scalar* представляет все различающиеся комбинации, которые мы должны испробовать для каждого

hyperparameter

³⁷ Следовательно, *grid-search* представляет собой систематический подход к определению скалярных гиперпараметров.

Совершенно верно!

А теперь пора перекусить!

³⁸ Снова что-то вкусное?

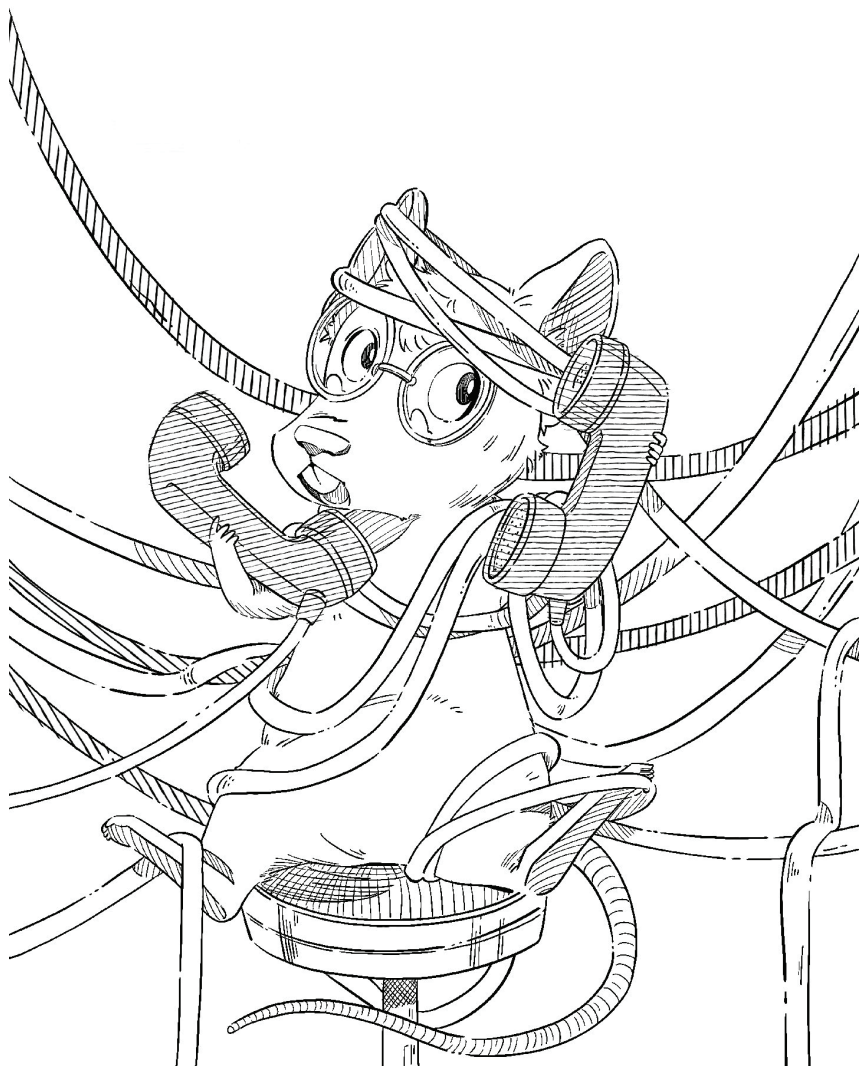
Обучающие побрякушки

accuracy стр. [275](#)
grid-search стр. [277](#)

Как насчет бельгийских
вафель?

ОКУНИТЕ ИХ В КЛЕНОВЫЙ СИРОП!

ИНТЕРЛЮДИЯ VII. ВЫ ГОТОВЫ ПРИНЯТЬ СИГНАЛ?



Бельгийские вафли[†] также вос- 1 Вкуснейшее лакомство!
хитительно вкусны с клубни-
кой и взбитыми сливками.

[†] Рецепт Мориса Реми-Пьера
Вермерха (1914–2001).

А теперь поговорим о сигналах. 2 Что такое сигналы?

Это лучше всего объяснить на 3 Что же делают эти болтливые
примере. ученики?

Пара болтливых учеников,
Алиса и Боб, – близкие друзья
и даже живут по соседству, но
им запрещено пользоваться
своими телефонами ночью.

Они решили воспользоваться 4 Как именно они это делают?
мигающим светом сигналь-
ных фонариков[†].

[†] Изобретение Дэвида Мизелла (1846–
1948).

Они используют международ- 5 Вы имеете в виду точки и
ную азбуку Морзе[†]. тире?

[†] У этого изобретения несколько
авторов: Альфред Вейл (1807–1859),
Сэмюэль Морзе (1791–1872) и
Фридрих Герке (1801–1888).

Конечно! 6 Как закодирована буква В?

Точке (·) соответствует
короткая вспышка, а тире (–)
передают длинной вспышкой.
Каждая буква алфавита зако-
дирована последовательно-
стью точек и тире, разделен-
ных коротким пробелом.

Например, буква А
закодирована так:

• –

Букве В английского алфавита соответствует код	7	Получается, буквы могут иметь разную длину?
--	---	---

— . . .

Да. Международная азбука Морзе содержит много разных символов, но мы будем использовать ровно 26 – по одному на каждую букву английского алфавита.	8	Значит, мы можем передать любое сообщение, используя только точки и тире?
---	---	---

Конечно! Теперь Алиса и Боб могут общаться ночи напролет, используя сигнальные фонарики. Алиса и Боб решили построить машину для декодирования этих сообщений.	9	Они должны изготовить одну машину для Алисы и другую для Боба.
--	---	--

Да, верно. Машина использует прикрепленный к окну оптический сенсор, чтобы улавливать вспышки света от фонарика другого ученика.	10	Что такое оптический сенсор?
---	----	------------------------------

Оптический сенсор преобразует падающий на него свет в электрический сигнал. Затем машина измеряет напряжение электрического сигнала.	11	Что такое напряжение?
---	----	-----------------------

Это мера величины электрического сигнала на выходе оптического сенсора. В отсутствие света напряжение на выходе сенсора равно

0.0 вольт (В)[†]

Когда свет падает на сенсор, напряжение на выходе

пропорционально силе света.

Для удобства возьмем сенсор, который выдает

0.0 (когда фонарик выключен)

и

1.0 (когда фонарик светит).

[†] Единица измерения названа в честь Алессандро Вольты.

Допустим, Боб отправил последовательность

• —

которая означает букву А.

Изначально фонарик Боба не горит. Он включается на короткое время для передачи точки (•), затем ненадолго гаснет. Затем он включается на более долгое время для передачи тире (—) и гаснет окончательно.

¹² Как себя ведет оптический сенсор, когда получает сообщение?

¹³ Как меняется выход оптического сенсора на протяжении этих событий?

Пока фонарь не светит, напряжение на выходе оптического сенсора Алисы равно 0.0 В.

¹⁴ Существует ли способ представить этот процесс в визуальной форме для лучшего понимания?

Затем, во время передачи точки, напряжение на выходе оптического сенсора возрастает до 1.0 В и сохраняется, пока светит фонарик.

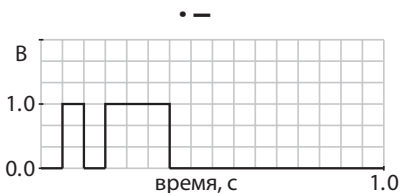
Когда Боб выключает фонарик после передачи точки, напряжение на выходе сенсора падает до 0.0 В.

Когда Боб включает фонарик для передачи тире, напряжение на выходе сенсора снова возрастает до 1.0 В и сохраняется на время передачи тире, а затем падает до 0.0 В.

Да, такой способ существует.

Допустим, Боб может передавать одну букву в секунду[†]. Мы можем построить график изменений напряжения на выходе сенсора Алисы, учитывая, что длина буквы не превышает одну секунду.

Ниже показан такой график для буквы А. Мы будем называть его графиком сигнала.



Здесь В – напряжение в вольтах.

[†] Это довольно быстро, но мы будем считать, что Боб справится.

¹⁵ На этом графике хорошо видно, как напряжение возрастает и падает в процессе передачи точки и тире.

Да, это так.

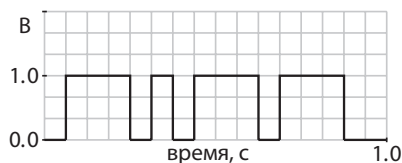
16 Можно еще один пример?

Конечно.

17 На этом графике видно, что напряжение на выходе сенсора на протяжении секунды меняется по-другому.

Вот график сигнала для буквы Y:

— • — —



Верно.

18 Значит, на выходе оптического сенсора мы наблюдаем сигнал?

Сигнал – это изменение физической величины (в данном случае – напряжения) с течением времени.

Пример, изображенный на предыдущем графике, – это аналоговый сигнал[†].

19 Как Алиса должна поступить с этим аналоговым сигналом?

[†] Он также называется непрерывным сигналом.

Поскольку Алиса хочет использовать нейросеть для декодирования сигнала, его необходимо преобразовать в tensor¹.

20 Как выглядит такой тензор?

Алиса использует аналого-цифровой преобразователь (АЦП) для преобразования этого электрического сигнала в tensor^{1†}.

[†] Читатели, знакомые с обработкой сигналов, могут пропустить описание преобразования аналогового сигнала в тензор и перейти к использованию готового тензора.

Сначала сделаем несколько упрощающих допущений.

Во фрейме 15 Боб отправляет 1 букву в секунду. Разобьем поток вспышек фонарика Боба на односекундные интервалы таким образом, чтобы каждый интервал содержал одну букву.

21 То есть передача сообщения из 10 букв займет 10 с?

Верно.

Теперь разобьем односекундный период на 16 сегментов, а затем свяжем скалярное значение с каждым из этих 16 сегментов, чтобы получить tensor^1 , соответствующий одиной букве в сообщении.

22 Какой скаляр будет связан с каждым элементом тензора?

Мы используем величину напряжения сигнала в середине i -го сегмента в качестве скалярного значения i -го элемента.

23 Не могли бы вы показать, как это выглядит на графике?

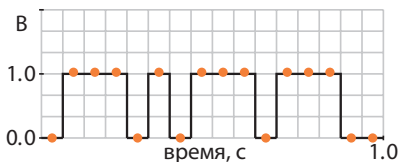
Охотно.

Вернемся к букве Y

— • — —

На этом графике оранжевыми точками отмечены скалярные значения, полученные для каждого сегмента графика.

24 Каждая оранжевая точка представляет скалярный элемент тензора tensor^1 .



Так выглядит полный tensor¹:

```
[0.0 1.0 1.0 1.0 0.0 1.0 0.0 1.0 1.0
1.0 0.0 1.0 1.0 1.0 0.0 0.0]
```

Ого! Как много беспорядочно разбросанных нулей и единиц!

²⁵ Да уж, это просто свалка из нулей и единиц!

Так и есть!

К счастью, подобные записи можно сократить. Применим многоточия для повышения читаемости сигнала. Ниже показана сокращенная запись предыдущего сигнала:

```
[0.0 1.0 2... 0.0 1.0 0.0 1.0 2... 0.0
1.0 2... 0.0 0.0]
```

Сигнал здесь, как и прежде, состоит из 16 сегментов, но смежные повторяющиеся значения обозначены многоточиями. Сегменты, представленные многоточием, повторяют значения сегмента непосредственно перед многоточием, а число над многоточием показывает, сколько еще последовательных сегментов у нас есть с тем же значением[†].

²⁶ Очень удобно.

Можно привести еще несколько примеров?

[†] Мы принимаем соглашение, что сокращенный тензор никогда не оканчивается многоточием. В данном случае можно видеть, что запись тензора заканчивается двумя значениями 0.0 подряд.

Конечно.

Немного поработаем с сокращениями. Рассмотрим сигнал, в котором

4 выкл., 4 вкл., 4 выкл., 4 вкл.

$[0.0 \dots 1.0 \dots 0.0 \dots 1.0 \dots 1.0]$

Создадим два похожих сигнала, где первый сигнал

3 выкл., 3 вкл., 3 выкл., 3 вкл.,
3 выкл., 1 вкл.

и второй

5 выкл., 5 вкл., 5 выкл., 1 вкл.

Цифровые сигналы, которые у нас были до сих пор, представляли собой tensor^1 , где каждый сегмент времени[†] был скаляром.

Обозначим их как signal^1 .

[†] Для краткости будем называть их просто сегментами.

Все правильно.

Это обозначение одномерного сигнала signal^1 .

Это означает, что сигнал состоит из значений, которые изменяются только вдоль одной оси, которая называется *измерением*. Измерением в данном случае является *время*.

Значит, когда мы упоминаем signal^1 , это означает одномерный сигнал, представленный в форме tensor^1 .

²⁷ Вот эти сигналы:

$[0.0 \overset{2}{\dots} 1.0 \overset{2}{\dots} 0.0 \overset{2}{\dots} 1.0 \overset{2}{\dots} 0.0 \overset{2}{\dots} 1.0]$

и

$[0.0 \overset{4}{\dots} 1.0 \overset{4}{\dots} 0.0 \overset{4}{\dots} 1.0]$

²⁸ Ой, мы используем такой же верхний индекс, как в обозначении тензора.

²⁹ Что означает термин «одномерный сигнал»?

³⁰ Понятно.

³¹ Существует ли signal^2 ?

Превосходный вопрос!

Да, signal^2 – это одномерный сигнал, который содержит tensor^1 в каждом сегменте. То есть фактически это tensor^2 .

³² Можно взглянуть на пример signal^2 ?

Пожалуйста.

Вот $\text{signal}^1 s$:

$[0.0 \dots 1.0 \ 0.0 \ \dots \ 0.0]$

Вот другой $\text{signal}^1 t$:

$[1.0 \ 0.0 \ \dots \ 1.0]$

³³ Мы составим signal^2 из этих двух signal^1 ?

Конечно.

Мы «упакуем» в signal^2 эти два сигнала s и t , извлекая по одному скаляру из s и t и помещая их в tensor^1 . Полученный signal^2 имеет 16 сегментов, каждый из которых – двухэлементный tensor^1 .

³⁴ А можно пример?

Конечно.

Скаляр с индексом 0 в s

0.0

Скаляр с индексом 0 в t

1.0

Значит, результирующий tensor^1

$[0.0 \ 1.0]$

Найдите tensor^1 для индекса 1 в signal^2 .

³⁵ Звучит вполне логично.

Скаляр с индексом 1 в s

0.0

Скаляр с индексом 1 в t

0.0

Значит, результирующий tensor^1

$[0.0 \ 0.0]$

Прекрасно.

Повторив эти шаги для всех сегментов, получим полный signal^2 :

$$\begin{bmatrix} 0.0 & 1.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 1.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ \vdots & \\ 0.0 & 1.0 \\ 0.0 & 1.0 \end{bmatrix}_{(16 \ 2)}$$

Превосходный вопрос.

В общем случае мы можем упаковать d сигналов, каждый формы

$(\text{list } n)$

и получить signal^2 формы[†]

$(\text{list } n \ d)$

Здесь d называется *глубиной* signal^2 .

[†] Упаковка signal^1 в signal^2 может быть знакома вам под названием транспонирования матриц.

³⁶ А если мы захотим упаковать больше, чем два сигнала?

³⁷ *Глубина* signal^2 – это количество упакованных в него signal^1 .

Можно ознакомиться с примером?

Закон упакованных сигналов

Signal² состоит из упакованных signal¹; и сам signal²,
и составляющие его signal¹ имеют одинаковое
количество сегментов.

Конечно.

³⁸ Она равна 3, так как signal²
составлен из 3 signal¹.

Возьмем два сигнала s и t из
фрейма 33 и упакуем их вмес-
те с сигналом u

[0.0 1.0 0.0 ...¹¹ 1.0 0.0]

Так будет выглядеть результат:

$$\begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ \vdots^9 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix}_{(16 \ 3)}$$

Какова здесь глубина signal²?

Отлично!

³⁹ Можно увидеть новый при-
мер?

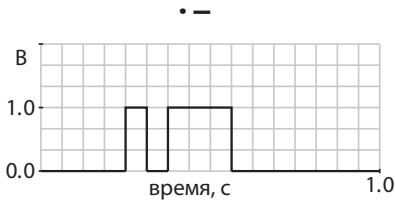
В примерах, которые мы виде-
ли до сих пор, сигнал начина-
ется с 0.0 и изменяется на 1.0
в следующем сегменте. Мы
называем началом сигнала
первый раз, когда видим это
изменение в сигнале.

Теперь мы опустим это пред-
положение, чтобы сделать
задачу более интересной.

Предположим, что начало
сигнала может находиться в
любом месте тензора, если весь
сигнал присутствует в преде-
лах 16-сегментного тензора.

Конечно.

Рассмотрим график сигнала для буквы А, только с более поздним стартом.



Какой тензор соответствует этому графику?

40 Вот он:

$[0.0 \overset{3}{\dots} 1.0 \ 0.0 \ 1.0 \overset{2}{\dots} 0.0 \overset{5}{\dots} 0.0]$

Правильно.

Подобный сдвиг «старта» сигнала называется переносом (*translation*).

41 Следовательно, наши болтливые ученики должны учитывать перенос сигнала в своих машинах?

Совершенно верно!

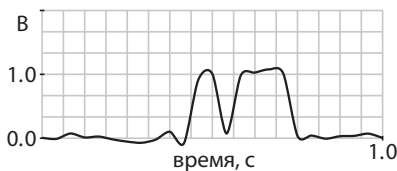
А теперь поговорим о шуме.

42 Что вы подразумеваете под шумом?

Графики сигнала, которые мы видели до сих пор, были идеальными. В реальности сигналы вовсе не так хороши. Они подвержены случайным изменениям.

43 Как выглядит график реального сигнала?

Приблизительно так выглядит реальный сигнал при передаче буквы А:



44 Ох, теперь элементы сигнала очень редко, если вообще когда-либо, точно равны 0.0 или 1.0.

Разумеется, поскольку на него накладывается случайный шум, он каждый раз будет выглядеть немного иначе.

Да, это так.

Например, наш `signal`¹ для буквы Y может выглядеть приблизительно так:

[0.05 1.2 0.96 -0.03 1.1 0.09 1.0 0.9
-0.02 1.1 1.2 0.2 -0.01 -0.04 0.1 0.08]

Приемники наших болтливых учеников могут декодировать сообщения, пропуская такие сигналы через нейросеть.

Но нейросеть должна сохранять способность декодировать сигналы даже при наличии шумов и помех.

45 Определенно, он выглядит зашумленным!

Как же нам работать с таким сигналом?

46 Это впечатляет.

Мы научимся создавать такую нейросеть?

Да, но немного позже.

47 Не могу дождаться!

Побрякушки на молнии

упакованные сигналы стр. 291

Как насчет профитролей?

ХРУСТЯЩИХ, С НЕЖНЫМ
ЗАВАРНЫМ КРЕМОМ?

14. ВСЕ ОЧЕНЬ СИЛЬНО ПЕРЕПУТАНО...



Ну что, профитроли оправдали ожидания?	1	Да, и еще они были хрустящие!
--	---	-------------------------------

Теперь приступим к изучению новой разновидности слоя.	2	Этот слой поможет Алисе и Бобу обмениваться сообщениями?
---	---	--

Конечно. Для декодирования их сообщений сначала нужно изучить понятие корреляции между двумя одномерными сигналами.	3	Что такое корреляция?
--	---	-----------------------

<i>Корреляция</i> – это способ обнаружить определенную закономерность (паттерн) в произвольном месте сигнала.	4	Зачем нам обнаруживать паттерны в сигнале?
---	---	--

Обнаружение паттернов в сигнале помогает нам определить, какую букву представляет этот сигнал. Мы обнаруживаем паттерн путем сканирования сигнала от начала до конца и измерения сходства между текущей частью сигнала и паттерном.	5	А что такое сканирование?
--	---	---------------------------

<i>Сканирование</i> – это процесс проверки каждого сегмента сигнала, от начала и до конца, по одному сегменту за раз.	6	Можете показать пример?
---	---	-------------------------

Конечно.

Рассмотрим signal^1 из 16 сегментов, который содержит единственную точку (·) и кодирует букву E, сдвинутую на 3 сегмента:

$$[0.0 \overset{2}{\dots} 1.0 \ 0.0 \ \overset{10}{\dots} 0.0]$$

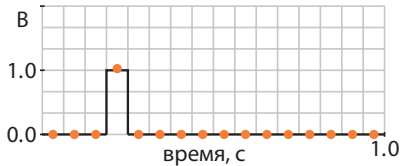
Таковы исходные условия нашего примера. Здесь мы рассматриваем идеальный сигнал, но тот же принцип применим и к зашумленным сигналам.

7 Можно ли также взглянуть на график?

Вот буква E:

·

Вот график сигнала:



8 Оно такое же, как в предыдущем фрейме:

$$[0.0 \overset{2}{\dots} 1.0 \ 0.0 \ \overset{10}{\dots} 0.0]$$

Запишите сокращенное представление сигнала.

А теперь возьмем другой signal^1 , но намного короче, чем s :

$$[0.0 \ 1.0 \ 0.0]$$

9 Этот signal^1 выглядит очень похоже на интересующую нас часть более длинного сигнала s .

Короткий signal^1 является паттерном, который мы используем для поиска совпадения с сигналом-источником. Этот короткий сигнал мы будем называть фильтром filter^1 †.

10 Длина этого фильтра равна 3. Все фильтры короче, чем источник, и имеют длину 3?

† Его также часто называют ядром (*kernel*).

Нет, это не обязательно так. Обычно мы выбираем их как небольшое нечетное число. Мы используем маленькие числа, потому что маленькие шаблоны легче сопоставить. Более сложные шаблоны сопоставляются с использованием множества небольших фильтров.

¹¹ Как сопоставляют фильтры с сигналом?

Вот где вступает в дело коррекция.

¹² Теперь фильтр совмещен с первыми тремя элементами источника.

Сначала мы выравниваем фильтр и источник по индексу 0 источника:

[0.0 1.0 0.0]

[0.0 0.0 0.0 1.0 0.0 ... 0.0]

Три элемента источника, которые перекрываются фильтром, образуют tensor^1 , в данном случае

[0.0 0.0 0.0]

¹³ Перекрытие определяет элементы из источника, которые перекрываются с фильтром, когда он выровнен с источником в позиции 0.

Мы называем его
перекрытие в позиции 0

Правильно.

¹⁴ Но что это значит?

Теперь найдем скалярное произведение ($\bullet$) между фильтром и этим перекрытием[†]:

1. $(\bullet [0.0 \ 1.0 \ 0.0] [0.0 \ 0.0 \ 0.0])$
2. $(\text{sum} \ (* [0.0 \ 1.0 \ 0.0] [0.0 \ 0.0 \ 0.0]))$
3. $(\text{sum} \ [0.0 \ 0.0 \ 0.0])$
4. 0.0

[†] По соглашению первым аргументом $\bullet$ всегда является фильтр.

Это скалярное произведение векторов возвращает скалярную меру сходства между фильтром и перекрытием.	15	Нам важно получить на выходе именно скалярное значение?
Да. Скаляр – это простой показатель того, насколько хорошо совпадение.	16	Что означает мера 0.0?
Она означает, что фильтр совершенно не похож на перекрытие.	17	И что нам теперь делать?
Теперь мы сдвигаем фильтр на один элемент вправо, чтобы перекрытие начиналось с позиции 1: $\begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & \dots & 0.0 \end{bmatrix}$	18	Теперь оно выглядит так: $\begin{bmatrix} 0.0 & 0.0 & 1.0 \end{bmatrix}$
Каким теперь будет перекрытие?		
Верно. Что нужно сделать дальше?	19	Мы должны найти скалярное произведение между фильтром и перекрытием: $\begin{array}{l l} 1. & (\bullet [0.0 \ 1.0 \ 0.0] [0.0 \ 0.0 \ 1.0]) \\ 2. & (sum \\ & \quad (* [0.0 \ 1.0 \ 0.0] [0.0 \ 0.0 \ 1.0])) \\ 3. & (sum \\ & \quad [0.0 \ 0.0 \ 0.0]) \\ 4. & 0.0 \end{array}$
		Скалярное произведение снова равно 0.0.
Правильно. В позиции 1 опять нет совпадения с фильтром.	20	Значит, нужно сдвинуть фильтр еще на одну позицию вправо?

Именно. Мы переходим к перекрытию в позиции 2: $[0.0 \ 1.0 \ 0.0]$ $[0.0 \ 0.0 \ 0.0 \ 1.0 \ 0.0 \ \overset{10}{\dots} \ 0.0]$ Каким теперь будет скалярное произведение?	21 Мы легко найдем его: 1. $(\bullet [0.0 \ 1.0 \ 0.0] [0.0 \ 1.0 \ 0.0])$ 2. $(sum$ $\quad (* [0.0 \ 1.0 \ 0.0] [0.0 \ 1.0 \ 0.0]))$ 3. $(sum$ $\quad [0.0 \ 1.0 \ 0.0])$ 4. 1.0 Теперь оно равно 1.0!
Так и должно быть. У нас получилось положитель- ное значение (т. е. не 0.0). Это говорит о том, что мы нашли совпадение между источни- ком и фильтром в перекрытии с позицией 2.	22 Что можно сказать про остав- шиеся позиции перекрытия?
Все остальные позиции, вклю- чая 13, дадут нам 0.0.	23 Почему мы остановимся на позиции 13?
Потому что при смещении дальше позиции 13 последний элемент фильтра выйдет за границу сигнала.	24 Что нам делать дальше?
Мы собираем скалярные результаты умножения на каждой позиции перекрытия от 0 до 13 и формируем из них 14-элементный $signal^1$: $[0.0 \ 0.0 \ 1.0 \ 0.0 \ \overset{9}{\dots} \ 0.0]$ Этот $signal^1$ является результа- том корреляции между источ- ником s и фильтром.	25 Итак, корреляция определя- ет скалярное произведение фильтра и перекрытия в каждой позиции с 0 по 13 и формирует $signal^1$ из этих произведений!

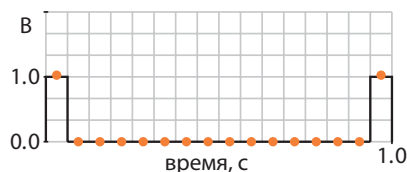
Верно. В общем случае если источник состоит из n сегментов, а фильтр – из m сегментов, то корреляция имеет длину $n - m + 1$	26 Так вот почему результирующий сигнал во фрейме 25 имеет $16 - 3 + 1 = 14$ элементов.
Точно. Результат корреляционной функции имеет «пики» там, где фильтр напоминает источник, т. е. коррелирует с ним.	27 Наподобие 1.0 в позиции перекрытия 2 во фрейме 21?
Да. Способ, которым мы до сих пор определяли наши позиции перекрытия от 0 до $n - m + 1$, неудобен по двум причинам. Во-первых, более короткий результат корреляции означает, что некоторая информация была потеряна. Чтобы предотвратить эту потерю информации, было бы хорошо, если бы исходный и результирующий сигналы имели одинаковую длину.	28 Какова вторая причина?
Вторая причина заключается в том, что частичные совпадения между источником и фильтром на границах источника не могут быть обнаружены.	29 Почему?
Потому, что мы начинаем отсчет позиции перекрытия с 0 и заканчиваем, когда последний элемент фильтра совпадет с последним элементом источника.	30 Можно пример?

Конечно.

В качестве примера возьмем источник t

$$[1.0 \ 0.0 \ \dots^{12} \ 0.0 \ 1.0]$$

с соответствующим графиком:



Правильно.

Чтобы исправить оба этих недостатка, мы начинаем перекрытия с позиции -1 :

$$\begin{aligned} &[0.0 \ 1.0 \ 0.0] \\ &[1.0 \ 0.0 \ \dots^{12} \ 0.0 \ 1.0] \end{aligned}$$

Верное замечание!

В этом случае мы считаем, что источник дополнен нулями с двух сторон:

$$\begin{aligned} &[0.0 \ 1.0 \ 0.0] \\ &0.0 \ [1.0 \ 0.0 \ \dots^{12} \ 0.0 \ 1.0] \ 0.0 \end{aligned}$$

Чтобы у нас была возможность рассматривать позицию 14, когда фильтр выходит за последний элемент источника:

$$\begin{aligned} &[0.0 \ 1.0 \ 0.0] \\ &0.0 \ [1.0 \ 0.0 \ \dots^{12} \ 0.0 \ 1.0] \ 0.0 \end{aligned}$$

Это означает, что в результате корреляции получается сигнал, длина которого совпадает с длиной источника.

31 В таком случае наши перекрытия пропускают частичные совпадения между источником и фильтром в начале и в конце источника.

32 Но ни в одном тензоре нет элемента с индексом -1 !

33 Это позволяет нам рассматривать перекрытия источника с позицией -1 .
Но почему с двух сторон?

34 После добавления позиций с индексами -1 и 14 к имеющимся 14 позициям перекрытия у нас получается в общей сложности 16 позиций.

35 Хотелось бы увидеть пример того, как работает дополнение.

Пожалуйста.

Найдем корреляцию между фильтром

$$[0.0 \ 1.0 \ 0.0]$$

и источником

$$[1.0 \ 0.0 \ \dots \ 0.0 \ 1.0]$$

Как выглядит перекрытие в позиции –1?

³⁶ Перекрытие должно включать добавленное значение 0.0 в начале, так что средний элемент фильтра совпадает с нулевым элементом источника:

$$[0.0 \ 1.0 \ 0.0]$$

Найдем скалярное произведение фильтра и перекрытия

1. $(\bullet [0.0 \ 1.0 \ 0.0] [0.0 \ 1.0 \ 0.0])$
2. $(\text{sum} \quad (* [0.0 \ 1.0 \ 0.0] [0.0 \ 1.0 \ 0.0]))$
3. $(\text{sum} \quad [0.0 \ 1.0 \ 0.0])$
4. 1.0

Как будет выглядеть скалярное произведение в позиции 14?

³⁷ Вот так:

1. $(\bullet [0.0 \ 1.0 \ 0.0] [0.0 \ 1.0 \ 0.0])$
2. $(\text{sum} \quad (* [0.0 \ 1.0 \ 0.0] [0.0 \ 1.0 \ 0.0]))$
3. $(\text{sum} \quad [0.0 \ 1.0 \ 0.0])$
4. 1.0

Что насчет остальных позиций перекрытия?

Они все равны нулю, давая нам корреляцию

$$[1.0 \ 0.0 \ \dots \ 1.0]$$

Мы будем называть ее результирующим сигналом.

³⁸ Почему мы называем ее сигналом?

Мы называем этот результат сигналом, потому что теперь он может служить источником для следующих фильтров, позволяя нам построить каскадную корреляцию в виде отдельных фильтров.

³⁹ Что означает каскадная корреляция?

Это означает, что мы выполняем последовательность корреляций таким образом, что результат одной операции корреляции служит входным сигналом для следующей корреляции, использующей другой фильтр.

40 Почему это важно?

Закон о корреляции (версия с одним фильтром)

Результатом корреляции фильтра длины m с источником signal^1 длины n , где m нечетное ($2p + 1$), является signal^1 длины n , полученный скольжением фильтра от позиции перекрытия $-p$ до позиции $n - p - 1$, где каждый сегмент результата signal^1 получен путем скалярного умножения фильтра и перекрытия источника для каждой позиции перекрытия.

Это полезно при обнаружении сложных паттернов, т. е. состоящих из других паттернов, как во фрейме 11.

41 Да, это определенно будет полезно.

Но результат выглядит точно так же, как источник.

Это из-за фильтра, который мы выбрали. Он обладает уникальным свойством копировать признаки, найденные в источнике, в результат корреляции[†]. В общем случае это не так. Другие фильтры могут быть более избирательными при выявлении других признаков, что приводит к получению отличающихся результатов.

42 Что такое признаки?

[†] Этот фильтр также называют дельтой Кронекера в честь математика Леопольда Кронекера.

Признаки – это особые паттерны, которые мы ищем в источнике.

Например, переход от низкого значения (близкого к 0.0) в одном сегменте к высокому значению (близкому к 1.0) в следующем сегменте является признаком «восходящего участка». Аналогично мы можем определить признак «нисходящего участка».

43 Разные признаки требуют применения разных фильтров.

Да, это так.

44 Найдутся ли у нас различные признаки и соответствующие фильтры для декодирования сообщений наших болтливых учеников?

Да, но вместо того чтобы определять фильтры вручную, мы воспользуемся нейросетью, которая выучит эти фильтры.

45 Вы хотите сказать, что фильтры являются частью θ для нейронных сетей, использующих корреляцию?

Правило фильтров

Фильтры являются тензорами параметров в θ .

Именно так. Мы рассматриваем фильтры как параметры[†] слоя корреляции (т. е. слоя нейронов, выполняющих операцию корреляции) в сети, и мы обучаем эти фильтры на наборах данных, где роль входа x играют сигналы $\text{signal}^{1\dagger}$.

[†] Эти параметры также называют весами фильтра.

[‡] На данный момент. Позже мы увидим, как преобразовываются x .

46 Да, как в случае signal^1 наших болтливых учеников. Но что можно сказать про выходы y ?

Выходами у являются соответствующие буквы, закодированные унитарным кодом, как показано во фрейме 252:12.

Например, унитарный `tensor`¹ для буквы В выглядит так:

$$[0.0 \ 1.0 \ 0.0 \ \dots \ 0.0]$$

Двум тензорам в этом наборе данных мы дадим имена

morse-xs

и

morse-ys

Следовательно, наша нейросеть классифицирует каждый `signal`¹ по одному из 26 классов, что соответствует 26 буквам английского алфавита.

Верно.

Но мы должны немного усовершенствовать нашу функцию корреляции, прежде чем использовать ее в многослойной нейросети.

Типичный нейросетевой классификатор пытается одновременно обнаружить несколько признаков в одном источнике.

Да, существует.

Чтобы сделать это, нам нужна аналоговая операция, выполняющая корреляцию относительно нескольких фильтров за один вызов функции.

⁴⁷ Означает ли это, что нейросеть использует функции корреляции для обучения других фильтров, необходимых для распознавания каждой буквы?

⁴⁸ Зачем нам это нужно?

⁴⁹ Существует ли способ сделать это за один вызов, как мы делали для `*2,1` во фрейме 218:22?

⁵⁰ Как же нам это сделать?

Представьте, что у нас есть b различных фильтров. Каждый фильтр представляет собой signal^1 и имеет длину m .

Что вы можете сказать об этих b различных фильтрах?

51 Вместе эти b фильтров образуют tensor^2 с формой $(\text{list } b \ m)$

Да, конечно.

Мы будем называть этот tensor^2 банк фильтров (или просто банк).

52 Как можно выполнить операцию корреляции источников с целым банком фильтров?

Мы берем каждый фильтр в банке и сопоставляем его с одним источником длины n , что дает нам b разных результатов, где каждый результат представляет собой tensor^1 той же длины, что и источник.

53 Означает ли это, что выход может быть упакован в tensor^2 с формой $(\text{list } b \ n)$?

Это заманчиво, но тогда наш результат не будет представлять собой signal^1 , как мы ожидаем на выходе шага корреляции. Он также не будет signal^2 , потому что у него не будет такого же количества сегментов, как у исходного сигнала, то есть n . Вместо этого в нем будет b signal^1 .

54 Разве это проблема?

Да.

Это лишает нас возможности каскадной обработки результатов корреляции другими слоями корреляции нейронной сети.

55 Значит, эта проблема мешает обнаружению более сложных признаков (паттернов, построенных из других паттернов), как показано во фрейме 41.

Так как же нам сделать вывод нашей корреляции в банке фильтров похожим на signal^2 ?

Мы упаковываем b результатов от b фильтров из банка, один за другим, в signal^2 формы

$(\text{list } n \ b)$

Снова рассмотрим банк из двух фильтров

$[[0.0 \ 2.0 \ 1.0] \ [1.0 \ 2.0 \ -1.0]]_{(2 \ 3)}$

и исходный сигнал

$[0.0 \ \overset{2}{\dots} \ 1.0 \ 0.0 \ \overset{10}{\dots} \ 0.0]$

Найдите корреляцию каждого фильтра в банке с исходным сигналом.

56 Можно пример?

57 Выполним корреляцию с исходного сигнала с первым фильтром

$[0.0 \ 2.0 \ 1.0]$

мы получим

$[0.0 \ 0.0 \ 1.0 \ 2.0 \ 0.0 \ \overset{10}{\dots} \ 0.0]$

а корреляция со вторым фильтром

$[1.0 \ 2.0 \ -1.0]$

дает нам

$[0.0 \ 0.0 \ -1.0 \ 2.0 \ 1.0 \ 0.0 \ \overset{9}{\dots} \ 0.0]$

Теперь упакуем оба этих сигнала в signal^2 , как делали во фрейме 291:37:

$$\begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 1.0 & -1.0 \\ 2.0 & 2.0 \\ 0.0 & 1.0 \\ \vdots & \vdots \\ 0.0 & 0.0 \end{bmatrix}_{(16 \ 2)}$$

58 Понятно.

Но если мы захотим организовать каскадную корреляцию с использованием signal^2 , будет ли она работать правильно?

Отличный вопрос.

Давайте посмотрим, как корреляция работает с signal^2 .

Допустим, наши signal^2 имеют n сегментов (в нашем примере $n = 16$), каждый представляет собой tensor^1 длины d .

59 То есть мы имеем signal^2 длиной n и глубиной d .

Для корреляции с signal^2 наш фильтр сам должен иметь форму signal^2 , так что обозначим его как filter^2 , и он должен иметь такую же глубину, как источник.

Какую глубину должен иметь фильтр?

⁶⁰ Он тоже должен иметь глубину d .

Правильно.

Так же, как и раньше, у нас есть банк, состоящий из filter^2 , где каждый filter^2 содержит m сегментов.

Корреляция производит результирующий signal^2 глубиной b .

⁶¹ Иначе говоря, в общем случае источник, фильтр и результат представляют собой signal^2 .

Да, конечно.

Опишем формы наших сигналов. Источник имеет форму

$(\text{list } n \ d)$

и у нас есть банк из b фильтров формы

$(\text{list } b \ m \ d)$,

где m – ширина каждого фильтра.

⁶² Здесь тоже не помешает пример.

Конечно.

Здесь signal^2 из фрейма 291:36 получен упаковкой

signal^1 из фрейма 290:33

с

signal^1 тоже из фрейма 290:33.

⁶³ В данном случае $n = 16$, $d = 2$.

Назовем полученный сигнал st :

$$\begin{bmatrix} 0.0 & 1.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 1.0 & 0.0 \\ 0.0 & 0.0 \\ \vdots & \\ 0.0 & 1.0 \end{bmatrix}_{(16 \times 2)}$$

Каковы n и d сигнала st ?

Поскольку глубина источника равна 2, глубина наших фильтров также должна быть равна 2. Вот банк из 4 filter^2 :

```
[[[ 0.0 0.0 ][ 0.0 1.0 ][ 0.0 0.0 ]]  
[[ 0.0 0.0 ][ 1.0 0.0 ][ 0.0 0.0 ]]  
[[ 0.0 1.0 ][ 1.0 0.0 ][ 0.0 0.0 ]]  
[[ 0.0 0.0 ][ 1.0 0.0 ][ 1.0 0.0 ]]]
```

Назовите значения b , m и d .

Очень хорошо.

Возьмем из банка первый filter^2

```
[[ 0.0 0.0 ][ 0.0 1.0 ][ 0.0 0.0 ]]
```

и начнем с позиции -1 .

Мы полагаем, что дополняющие элементы имеют ту же форму, что и основные элементы, и состоят из 0.0.

⁶⁴ Так как у нас есть 4 filter^2 , то $b = 4$. Каждый filter^2 имеет 3 сегмента, значит, $m = 3$, и глубина $d = 2$, как раньше.

⁶⁵ Если мы начинаем с позиции -1 , то как нам дополнить signal^2 ?

⁶⁶ Итак, в этом случае вместо подстановки на дополняющую позицию скаляра 0.0 мы используем нулевой tensor^1 .

Поэтому мы дополним *st* так:

$$\begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 1.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 1.0 & 0.0 \\ 0.0 & 0.0 \\ \vdots & \\ 0.0 & 1.0 \\ 0.0 & 0.0 \end{bmatrix}_{(16 \times 2)}$$

Каким будет перекрытие на позиции -1?

⁶⁷ Вот таким:

$[[0.0 \ 0.0] \ [0.0 \ 1.0] \ [0.0 \ 0.0]]$

Правильно.

Теперь найдем произведение фильтра и перекрытия:

⁶⁸ Результат представляет собой *tensor*¹ из 3 элементов, что совпадает с шириной наших фильтров.

Ох, но ведь это означает, что результат не является скаляром, как мы установили во фрейме 16.

1. $(\bullet [[0.0 \ 0.0] \ [0.0 \ 1.0] \ [0.0 \ 0.0]] \ [[0.0 \ 0.0] \ [0.0 \ 1.0] \ [0.0 \ 0.0]])$
2. $[(\bullet [0.0 \ 0.0][0.0 \ 0.0]) \ (\bullet [0.0 \ 1.0][0.0 \ 1.0]) \ (\bullet [0.0 \ 0.0][0.0 \ 0.0])]$
3. $[0.0 \ 1.0 \ 0.0]$

Что вы скажете о результате?

Правильно.

Как нам преобразовать *tensor*¹ в скаляр?

⁶⁹ Согласно закону на стр. 50, функция *sum* понижает ранг тензора на 1.

Превосходно.

Выполняя корреляцию signal^2 , мы дополнительно применяем к результату функцию sum на каждой позиции перекрытия:

1. $(\text{sum} \quad (\bullet \quad [[0.0 \ 0.0] \ [0.0 \ 1.0] \ [0.0 \ 0.0]] \quad [[0.0 \ 0.0] \ [0.0 \ 1.0] \ [0.0 \ 0.0]]))$
2. $(\text{sum} \quad [0.0 \ 1.0 \ 0.0])$
3. 1.0

⁷⁰ Теперь у нас получился скаляр!

Оставшаяся часть корреляции с этим фильтром такая же. Мы сдвигаем фильтр на одну позицию вниз и повторяем до тех пор, пока не достигнем перекрытия на позиции 14, где также применяется дополнение тензором

$[0.0 \ 0.0]$

В конечном итоге это дает нам результирующий сигнал

$[1.0 \ 0.0 \ \overset{13}{\dots} \ 1.0]$

формы

(list 16).

⁷¹ Мы должны повторить эту корреляцию для всех оставшихся фильтров?

Да, должны.

Это даст нам в итоге 4 результирующих signal^1 :

$[1.0 \ 0.0 \ 0.0 \ 0.0 \ 0.0 \ \overset{10}{\dots} \ 1.0]$
 $[0.0 \ 0.0 \ 0.0 \ 1.0 \ 0.0 \ \overset{10}{\dots} \ 0.0]$
 $[0.0 \ 1.0 \ 0.0 \ 1.0 \ 0.0 \ \overset{10}{\dots} \ 0.0]$
 $[0.0 \ 0.0 \ 1.0 \ 1.0 \ 0.0 \ \overset{10}{\dots} \ 0.0]$

Найдите финальный signal^2 .

⁷² Чтобы получить signal^2 из этих signal^1 , мы упакуем их вместе и получим

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ \vdots & & & \\ 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}_{(16 \ 4)}$$

Теперь у нас есть способ принять signal^2 и создать signal^2 , а также каскадно передавать этот signal^2 в многослойной нейронной сети!

73 Это наша финальная версия функции корреляции?

Да, это она.

74 В нашем примере

В нейронной сети мы обучаем фильтры (т. е. фильтры – это параметры внутри θ).

$$n = 16$$

$$m = 3$$

Нейронная сеть учится распознавать закономерности во входных данных, которые затем можно классифицировать при обучении банка фильтров, связанного с каждым корреляционным слоем.

и

b – количество фильтров в банке.

Но что такое d ?

В нашем примере болтливые ученики используют корреляцию, чтобы классифицировать каждый сигнал как соответствующую букву.

Хороший вопрос.

75 Есть ли у корреляции собственное имя, которое можно использовать для нашей функции?

В многослойной нейтральной сети d – это глубина сигнала, произведенного предыдущим слоем. Позже мы увидим, как именно корреляция работает в нейронной сети.

Закон о корреляции (версия с банком фильтров)

Корреляция банка фильтров формы ($\text{list } b \ m \ d$) с исходным signal^2 формы ($\text{list } n \ d$) представляет собой signal^2 формы ($\text{list } n \ b$), сформированный в результате упаковки $b \ \text{signal}^1$, полученных в результате корреляции $b \ \text{filter}^2$ с источником.

Да!

Мы назовем ее

correlate

Это расширенная функция,
аналогичная •^{2,1} из фрей-
ма 219:24.

В следующей главе вы узнаете,
как использовать функцию
correlate.

⁷⁶ Фух, это было утомительное
приключение.

Может быть, немного переку-
сим?

Скользящие побрякушки

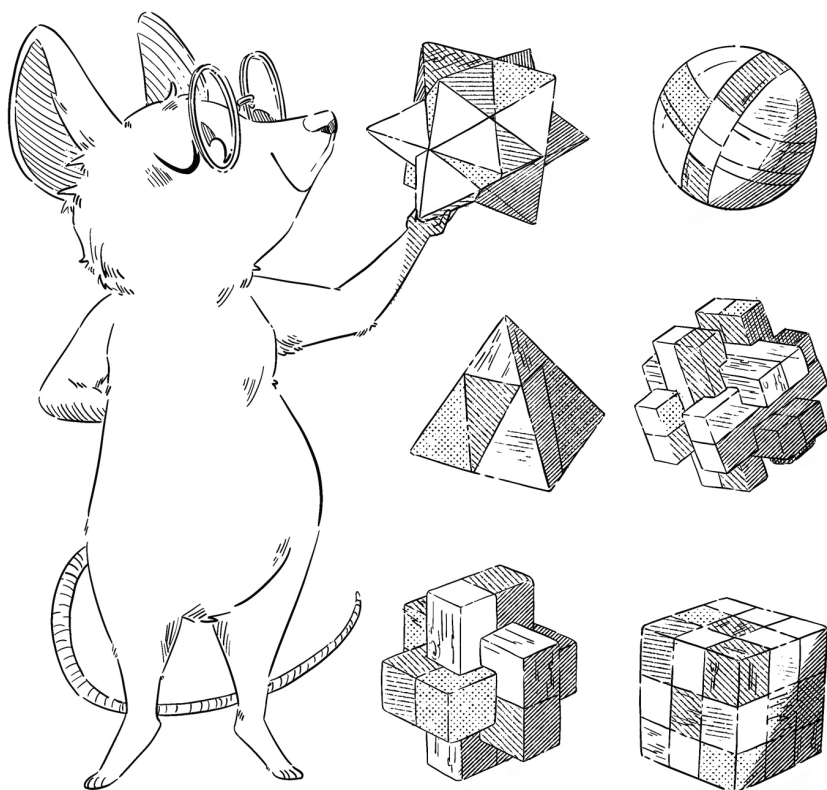
(*morse-xs*, *morse-ys*) стр. 306

correlate стр. 314

Как насчет кусочка торта «Муравейник»?

«МУРАВЕЙНИКИ»
ТАКИЕ ЗАПУТАННЫЕ!

15.НО ПРИ ЭТОМ ВЗАИМОСВЯЗАНО!



Вам не кажется, что мы обье- лись «Муравейником»?	¹ Он очень сладкий и вкусный, но несколько тяжеловат!
Теперь, когда мы изучили кор- реляцию, пора применить ее в нейросети!	² Мы должны сделать из нее функцию слоя, как мы посту- пили с $\bullet^{2,1}$, не так ли?
Да, точно! Является ли корреляция линейной функцией, опреде- ление которой дано во фрей- ме 101:11?	³ Интересный вопрос. В ней применяется много ска- лярных произведений, так что, скорее всего, она линейная. Верно?
Конечно! Хотя мы не будем приводить здесь доказательство, корреляция может быть полностью описана линейными функ- циями. Их можно рассматри- вать просто как связку между тензорами. Вот определение функции <i>corr</i> , которая комбинирует тензор t с двумя тензорами θ_0 и θ_1 из θ :	⁴ Если θ_0 – банк фильтров, то ожидается, что t представляет собой <i>tensor</i> ² . Что такое θ_1 ?
<pre>(define corr (λ (t) (λ (θ) (+ (correlate θ₀ t) θ₁))))</pre>	
θ_1 называется параметром смещения и представляет собой <i>tensor</i> ¹ , который содер- жит одно скалярное значение для каждого фильтра в банке.	⁵ Зачем нам нужен параметр смещения в <i>corr</i> ?
Как и в случае функции <i>linear</i> во фрейме 199:22, параметр смещения определяет точку, в которой спрямляющая функ- ция <i>rectify</i> начинает прини- мать значение 0.0.	⁶ Значит ли это, что мы должны комбинировать <i>corr</i> и <i>rectify</i> аналогично тому, как делали это с <i>linear</i> и <i>rectify</i> , чтобы получить функцию <i>relu</i> во фрейме 198:17?

Да, конечно!

Теперь мы готовы определить блок корреляции *resu*[†], аналогичный *relu* для плотных слоев.

[†] В других нейросетевых системах эту функцию также часто называют *conv* или *conv1D*. Но здесь мы называем ее *resu*, чтобы подчеркнуть сходство с *relu*.

7 Откуда взялось название *resu*?

Хороший вопрос.

Это сокращение от *rectifying correlational unit* (блок спрямляющей корреляции):

```
(define accuracy
  (λ (t)
    (λ (θ)
      (rectify ((corr t)θ)))))
```

8 Очевидно, что это слегка измененная *relu*.

Будем ли мы применять *resu* так же, как *relu*?

В основном да.

Основное различие заключается в форме параметров в соответствующем θ . Для *resu* θ_0 должен быть банком фильтров, который представляет собой *tensor*³, тогда как θ_0 для *relu* – это *tensor*².

Определите список форм для *resu*, где

b – количество фильтров,
m – ширина каждого фильтра

и

d – глубина входного сигнала.

9 Список форм для *resu* состоит из одной формы банка фильтров и одной формы тензора смещения.

Форма банка фильтров:

(list *b m d*)

и форма тензора смещения:

(list *b*)

Следовательно, список форм имеет вид:

```
(list
  (list b m d)
  (list b))
```

Рассмотрим формы входов и выходов *recu*.

Если входной *signal*² имеет форму

$$(\text{list } n \ d)$$

где θ_0 имеет форму

$$(\text{list } b \ m \ d)$$

и где θ_1 имеет форму

$$(\text{list } b)$$

то какую форму будет иметь выход слоя *recu*?

¹⁰ Функция *rectify* не меняет форму своего аргумента, поэтому форма выхода будет совпадать с формой

$$((\text{corr } t) \ \theta)$$

или, иначе говоря, с формой

$$(+ \ (\text{correlate } \theta_0 \ t) \ \theta_1)$$

Форма результата функции + определяется формой

$$(\text{correlate } \theta_0 \ t)$$

которая, согласно закону корреляции на стр. 313, имеет вид

$$(\text{list } n \ b)$$

Отлично.

Теперь запишите определение функции *recu-block*, которая получает одиночный слой *recu*, а также заданные *b*, *m* и *d* как аргументы.

¹¹ Это легко:

```
(define recu-block
  (λ (b m d)
    (block recu
      (list
        (list b m d)
        (list b))))))
```

Великолепно.

Сети, использующие корреляцию, имеют собственное название. Это сверточные нейронные сети (convolutional neural network, CNN).

¹² Но почему их называют сверточными, а не корреляционными?

Свертка – это функция, похожая на корреляцию, за исключением того, что фильтры зеркально отображаются (то есть переворачиваются) перед вызовом корреляции. CNN впервые были характеризованы с использованием этой функции.

Так как фильтры обучаются во время обучения сети, не имеет значения, отражены они зеркально или нет. Таким образом, мы можем избежать зеркального отображения, оставив только корреляцию.

Именно этим мы и займемся. Давайте начнем двигаться со стороны выхода, как мы это делали во фрейме 254:21 для сети *iris-network*.

Что наша сеть должна производить на выходе?

Правильно.

Давайте воспользуемся этим в качестве руководства к проектированию и предложим, чтобы наш выходной слой *resci* имел 26 фильтров, по одному для каждой буквы, которую мы хотим обнаружить.

Замечательное наблюдение.

Нам придется конвертировать этот signal^2 в tensor^1 .

15 Итак, несмотря на то что сеть называется сверточной нейронной сетью, мы ограничиваемся корреляциями.

Хорошо, что при этом не нужно менять привычную аббревиатуру CNN.

Не пора ли объявить нашу CNN?

14 Она должна выдавать унитарный вектор длиной 26. Это означает, что выходной слой должен производить tensor^1 с формой (list 26).

15 Вполне разумная идея.

Но кажется, это не совсем правильно. Из фрейма 10 мы знаем, что этот слой *resci* будет выдавать signal^2 с формой

(list 16 26)

16 Как это сделать?

Рассмотрим пример.

Пусть входной сигнал представляет букву Q. Если допустить, что наша сеть обучена правильно, один из 26 фильтров слоя *геси* выдаст выход с высоким уровнем сигнала (т. е. пик) в одном из сегментов, в то время как на выходе всех остальных фильтров будет присутствовать очень низкое значение.

Например, один из сегментов будет иметь tensor^1 длины 26, выглядящий примерно так:

```
[0.1 0.0 0.2 0.0 0.1 0.0 0.1 0.0
 0.2 0.0 0.1 0.0 0.1 0.0 0.2 0.0
 0.8 0.0 0.1 0.0 0.2 0.0 0.1 0.0
 0.1 0.0]
```

Там будут tensor^1 , не имеющие заметных пиков.

Например:

```
[0.2 0.1 0.2 0.0 0.1 0.0 0.1 0.0
 0.0 0.0 0.1 0.0 0.1 0.1 0.0 0.0
 0.0 0.0 0.0 0.1 0.1 0.0 0.0 0.0
 0.2 0.0]
```

Нужно сложить все эти сегменты.

В результате сложения мы получим tensor^1 , где по-прежнему будет пик в позиции Q.

17 Мы видим пиковое значение 0.8 в позиции, соответствующей букве Q.

Что будет в остальных сегментах?

18 Как нам сгенерировать tensor^1 из этих signal^2 ?

19 Это очень простое решение. Как мы выполним сложение?

Мы определим функцию sum^2 , которая складывает тензоры первого ранга, составляющие $tensor^2$:

```
(define sum2
  (λ (t)
    (summed2 t (sub1 † t †) 0.0)))

(define summed2
  (λ (t i a)
    (cond
      ((zero? i) (+ t0 a))
      (else
       (summed2 t (sub1 i) (+ ti a))))))
```

²⁰ Она выглядит точно так же, как sum^1 .

Да, фактически они идентичны.

Но разница в том, что результатом выражения

$t|_0$

является $tensor^1$, а не скаляр.

Поскольку $+$ является расширенным оператором, $summed^2$ аккумулирует $tensor^1$, который становится результатом sum^2 .

²¹ Получается, достаточно написать

```
(define sum2 sum1)
```

Впечатляет, правда?

Теперь расширим функцию:

```
(define sum-cols
  (ext1 sum2 2))
```

²² Это просто магия!

Мы можем использовать эту функцию на выходе для конвертации $signal^2$ в $tensor^1$.

Почти.

²³ Зачем нужно это делать?

Сначала мы преобразуем сумму в среднее значение, разделив ее на количество сегментов. Это так называемая *группировка по общему среднему* (*global average pooling*).

При использовании только *sum-cols* сумма небольших частичных совпадений может в конечном итоге превысить фактический пик, что приведет к снижению точности сети.

24 Понятно.

Можем ли мы извлечь количество сегментов в t , например, так:

$$(shape\ t)_0?$$

Нужно учесть еще один фактор.

Так как *sum-cols* расширенная, она может получать пакет $signals^2$, делая ранг t больше 2. Следовательно, мы должны извлечь форму вложенных $signal^2$, например, так:

$$(shape\ t)_{(- (rank\ t)\ 2)\downarrow}$$

25 Ага, и затем его первый элемент будет количеством сегментов сигнала

$$(shape\ t)_{(- (rank\ t)\ 2)\downarrow 0}$$

Итак, если t имеет форму $(list\ 8\ 16\ 4)$, мы получим:

1. $(list\ 8\ 16\ 4)_{(-\ 3\ 2)\downarrow 0}$
2. $(list\ 8\ 16\ 4)_{1\downarrow 0}$
3. $(list\ 16\ 4)_0$
4. 16

Правильно, вот функция слоя *signal-avg*:

```
(define signal-avg
  (λ (t)
    (λ (θ)
      (÷ (sum-cols t)
         (shape t)(- (rank t) 2)↓0))))
```

26 Так как эта функция не использует параметры из θ , список форм будет пустым: $(list)$

Как выглядит список форм для этого слоя?

Замечательно.

27 Вот он:

Теперь определите блок для этого слоя.

```
(define signal-avg-block
  (block signal-avg
    (list)))
```

Это будет выходной слой сети?

Да, верно. Теперь построим остальные слои, которые будут слоями <i>геси</i> .	28	Сколько должно быть таких слоев?
Пусть их будет 8.	29	Это значение, как всегда, найдено опытным путем?
Да, но, как говорится, в нашем безумии есть свой смысл. Мы ожидаем, что более ранние уровни в сети будут обнаруживать простые закономерности. Например, это могут быть такие признаки, как нарастающие и спадающие фронты сигнала.	30	Сколько слоев нужно отвести для этого?
Давайте выделим на обнаружение простых паттернов два слоя. Следующие два слоя будут обнаруживать группы таких признаков, выделяя из них признаки более высокого уровня, такие как точки, тире и пробелы.	31	Так вот почему мы видели более сложные признаки во фрейме 304:41.
Конечно. Затем следующие четыре слоя обнаруживают группы предыдущих признаков.	32	Значит, фильтры в этих слоях учатся обнаруживать комбинации точек и тире?
Сложно сказать точно, какие именно паттерны будут изучать фильтры, потому что окончательное распределение происходит в процессе обучения. Идея иерархии признаков помогает при начальной разработке архитектуры сети, а потом мы уточняем ее экспериментально.	33	Понятно.

Устройство нашей сети во фрейме 30 описано двумя слоями одновременно. Для удобства мы можем определить блок из двух слоев и использовать этот блок снова и снова.

Вот каркасный код для *fcn-block*:

```
(define fcn-block
  (λ (b m d)
    (stack-blocks
      (recu-block b m d)
      (recu-block b m B))))
```

Эта функция определяет блок из двух слоев *recu*, каждый из которых содержит *b* фильтров и имеет ширину *m*, и первый слой получает *signal*² глубиной *d*.

Найдите *B*.

Здесь *fcn* означает *fully convolutional network* (полностью сверточная сеть), потому что мы используем только слои *recu*, за исключением выходного слоя *signal-avg*, и в ней нет плотных слоев.

Так выглядит окончательное определение *fcn-block*:

```
(define fcn-block
  (λ (b m d)
    (stack-blocks
      (recu-block b m d)
      (recu-block b m b))))
```

34 Выходом первого слоя *recu* будет сигнал глубины *b*. Поскольку выход первого слоя *recu* становится входом для следующего слоя *recu*, вместо *B* нужно подставить *b*

потому что третьим аргументом *recu-block* является глубина его входного сигнала.

Почему дали этому блоку название *fcn-block*?

35 Так много имен...

Теперь можно дать определение всей сети?

Так выглядит определение сети:

```
(define morse-fcn
  (stack-blocks
    (list
      (fcn-block 4 3 1)
      (fcn-block 8 3 4)
      (fcn-block 16 3 8)
      (fcn-block 26 3 16)
      signal-avg-block)))
```

36 Как мы передали аргументы в 4 вызова *fcn-block*?

Для фильтров мы выбрали ширину 3, чего достаточно для нашей сети.

Последний вызов *fcn-block* требует 26 фильтров, как сказано во фрейме 15.

В первом вызове задействовано 4 фильтра в соответствии с 4 разными признаками входного сигнала. Он возвращает выходной сигнал глубины 4.

37 Ясно, вот почему в следующем вызове аргумент *d* равен 4.

Правильно.

И мы видим, что при каждом втором вызове *fcn-block* глубина сигнала равна количеству фильтров в предыдущем блоке.

Мы используем 8 и 16 фильтров, соответственно, для двух других блоков, чтобы обеспечить достаточную мощность для нахождения совпадений с требуемыми паттернами признаков, соответствующих более ранним блокам.

38 Как мы будем обучать эту сеть?

Начнем с инициализации θ .

39 Мы можем использовать *init- θ* !

Да, но пока эта функция не знает, как обращаться с банками фильтров, имеющих форму с длиной 3.

40 Значит, мы должны обновить *init-shape*?

Да, должны. Вспомним объявление *init-shape* из фрейма 264:60.

41 В нем нет клаузы `cond` для форм длины 3.

```
(define init-shape
  (λ (s)
    (cond
      ((= |s| 1) (zero-tensor s))
      ((= |s| 2)
       random-tensor 0.0 (÷ 2 s1) s))))
```

Давайте добавим ее:

42 Разве нам не нужно найти A ?

```
(define init-shape
  (λ (s)
    (cond
      ((= |s| 1) (zero-tensor s))
      ((= |s| 2)
       random-tensor 0.0 (÷ 2 s1) s))
      ((= |s| 3)
       random-tensor 0.0
       (÷ 2 A) s))))
```

Скоро найдем, но сперва нам нужно определить правило инициализации банков фильтров.

43 Да, и n – это длина входного tensor^1 этого слоя.

Согласно правилу инициализации слоя на стр. 261, аргумент дисперсии для *random-tensor* задается формулой

$$\frac{2}{n}$$

Так и есть.

Однако реальная цель этого заключается в том, что n – это количество скаляров из входных данных, которые умножаются на веса в слое для получения одного скаляра. Это так называемый множитель входов (*fan-in*).

Плотные слои строятся с помощью $\ast^{2,1}$ и *sum*, следовательно, множитель входов представляет собой входной *tensor*¹.

В слоях *recu* мы используем *correlate*.

Чтобы получить один скаляр на выходе, корреляция использует только

$$m \times d$$

скаляров, потому что перекрытие с любым заданным фильтром имеет длину m сегментов, и мы суммируем скалярные произведения на каждом из этих сегментов в перекрытии. Каждое из этих скалярных произведений использует d скаляров из входных данных.

Правильно.

Если s описывает форму банка фильтров, как нам найти m и d ?

44 В чем заключается отличие от слоев *recu*?

45 Ага, значит, для банка фильтров множитель входов всегда равен

$$(* m d)$$

46 Форма банка фильтров следующая:

$$(list\ b\ m\ d)$$

значит, m равно

$$s_1$$

и d равно

$$s_2$$

Прекрасно.

Теперь найдите A .

47 A представляет собой множитель входов для слоя *resci* и равняется

$$(* s_1 s_2)$$

Очень хорошо.

Ниже приведено полное определение *init-shape*:

48 Теперь мы будем обучать нейросеть?

```
(define init-shape
  (λ (s)
    (cond
      ((= |s| 1) (zero-tensor s))
      ((= |s| 2)
       (random-tensor 0.0 (÷ 2 s1) s))
      ((= |s| 3)
       (random-tensor 0.0
        (÷ 2 (* s1 s2)) s))))
```

Иногда для процесса обучения полезно выполнить предварительное преобразование данных, которое позволяет сделать архитектуру нейросети проще. Эта операция называется *предобработкой* (*preprocessing*).

49 В каком преобразовании нуждается наш набор данных?

Во-первых, наш набор данных во фрейме 309:47 представляет *morse-xs* как *signal*¹. Как показано в последующих фреймах, *correlate* и, следовательно, слои *resci* работают с *signal*². Значит, мы должны преобразовать *morse-xs* в *signal*² путем обертывания каждого скаляра в элементе *morse-xs* в *tensor*¹.

50 Вот оно:

```
[[[-0.26] [ 0.02] [-0.17] [0.0]
  [-0.02] [-0.10] [ 0.03] [0.0]
  [-0.16] [ 1.16] [-0.13] [0.97]
  [ 1.01] [ 0.93] [-0.18] [0.13]]]
```

Вот, для примера, signal^1 :

```
[ -0.26  0.02 -0.17 0.0
  -0.02 -0.10  0.03 0.0
  -0.16  1.16 -0.13 0.97
   1.01  0.93 -0.18 0.13]
```

Покажите преобразование
этого signal^1 в signal^2 .

Превосходно.

51 Как решают эту проблему?

Теперь о другом преобразовании. Данные сигнала колеблются между низким уровнем около 0.0 и высоким около 1.0.

Наличие на входе значений, близких к 0.0, всегда является проблемой, поскольку такие значения сводят на нет влияние любых весовых коэффициентов слоя.

Мы масштабируем данные таким образом, чтобы сигнал колебался между минимальным значением около -0.5 и максимальным около 0.5.

52 Ага, я понял, мы достигаем этого вычитанием 0.5 из входного сигнала!

Таким образом мы избегаем состояния, когда входное значение слишком близко к нулю.

Верно.

53 Должны ли мы предварительно обработать *morse-ys*?

Мы будем считать, что входные данные *morse-xs* предварительно обработаны таким образом. Чтобы продолжить наш пример, воспользуемся следующим предобработанным тензором:

```
[[ -0.76] [-0.48] [-0.67] [-0.50]
 [ -0.52] [-0.60] [-0.47] [-0.50]
 [ -0.66] [ 0.66] [-0.63] [ 0.47]
 [  0.51] [ 0.43] [-0.68] [-0.37]]
```

Нет, в этом нет необходимости, так как значения *morse-ys* нужны нам только для вычисления потерь.

54 Понятно.

Теперь-то мы можем приступить к обучению нейросети?

Давайте определим обучающую функцию для обучения нейросети для классификации сигналов кода Морзе:

55 Хорошо.

Мы будем использовать метод стохастического градиентного спуска?

```
(define train-morse
  (λ (network)
    (with-hypers
      ((α 0.0005)
       (revs 20000)
       (batch-size 8)
       (μ 0.9)
       (β 0.999))
      (trained-morse
       (block-fn network)
       (block-ls network))))))
```

Здесь мы предоставляем значения для гиперпараметров, а затем используем вспомогательную функцию *trained-morse* для создания модели из заданной сети.

Теперь нам нужно определить функцию *trained-morse*.

Да.

56 А еще нам нужно инициализировать θ с помощью *init- θ* .

Мы будем использовать *adam-gradient-descent*.

Чтобы сделать его стохастическим, нужно применить *sampling-obj* с *l2-loss* в качестве функции потерь и взять набор данных

(*morse-train-xs*, *morse-train-ys*)

Ниже приводится определение функции *trained-morse*. Она получает функцию сети *classifier* и список форм *θ-shapes* и возвращает модель, использующую *classifier* и обученные *θ*:

```
(define trained-morse
  (λ (classifier θ-shapes)
    (model classifier
      (adam-gradient-descent
        (sampling-obj
          (l2-loss classifier)
          morse-train-xs
          morse-train-ys)
        (init-θ θ-shapes))))))
```

57 Как мы нашли все эти гиперпараметры?

Методом поиска по сетке, разумеется.

58 Какова точность этой сети?

Давайте проверим:

59 Точность 94 %.
Это хорошо?

```
▷ (accuracy
   (train-morse morse-fcn)
   morse-test-xs morse-test-ys)
▶ 0.94
```

Это очень хорошо.
Но можно сделать лучше.

60 Интересно.
Как?

При помощи пропускных соединений (skip connection).

61 Что такое пропускные соединения?

Пропускные соединения подмешивают входной сигнал блока к выходу этого блока.

62 Зачем?

Как описано во фрейме 258:43, в очень глубоких сетях, по мере того как входное скалярное значение перемещается по слоям, каждый слой оказывает свое влияние на роль этого скаляра для выхода.

63 Да, помню, это приводит к проблеме вычислительного взрыва или исчезновения данных, что затрудняет обучение нейросети. Вот почему мы должны тщательно инициализировать сеть.

Правильно.

Даже когда сеть тщательно инициализирована и обучение проходит так, как ожидалось, некоторые веса все равно становятся очень маленькими и подавляют скаляры входных данных.

Подмешивание входных данных к выходам восстанавливает эффект этих скаляров, позволяя им оказывать большее влияние на последующие слои.

Это улучшает обучение сети и делает ее более эффективной в решении целевой задачи.

Мы начнем с определения функции *skip*, которая получает блочную функцию *f* и прибавляет ее вход к выходу:

```
(define skip
  (λ (f)
    (λ (t)
      (λ (θ)
        (+ ((f t) θ)
           t))))))
```

Объясните, как работает эта функция.

⁶⁴ Это звучит великолепно.

Как нам добавить обходные связи в *morse-fcn*?

⁶⁵ Когда *skip* получает блочную функцию *f*, она возвращает новую блочную функцию, которая прибавляет к выходу *f*

$((f\ t)\ \theta)$

вход блочной функции

t

Почему эта функция помечена пунктиром?

Пунктир присутствует, потому что это сложение между входом и выходом не работает, когда различается их глубина. Например, вход имеет форму

$(\text{list } n\ d)$

а выход имеет форму

$(\text{list } n\ b)$

и значения *d* и *b* различаются.

⁶⁶ Как устранить эту проблему?

Мы конвертируем входной сигнал из формы

(list n d)

в форму

(list n b)

67 Мы можем сделать это при помощи самой *rescu*, используя b фильтров, разве нет?

Определение функции *rescu* во фрейме 8 содержит *rectify*. Мы не используем *rescu*, потому что *rectify* может подавить некоторые скалярные значения входа.

Зато мы можем использовать *correlate*, которая не подавляет никакие входные скаляры.

68 Понятно.
Как в таком случае должен выглядеть соответствующий банк фильтров?

Мы уже знаем, что необходимо b фильтров и что глубина входного сигнала равна d .

Мы используем значения от 1 до m таким образом, чтобы фильтры не объединяли соседние значения друг с другом.

Какова форма такого банка фильтров?

69 Использование фильтра шириной 1 гарантирует, что перекрытие будет занимать ровно один сегмент, поэтому соседние сегменты не повлияют на него.
Форма банка фильтров будет такой:

(list b 1 d)

Правильно.

Этот банк фильтров, очевидно, добавляет дополнительный параметр в θ для каждого пропускного соединения.

70 Настало время для серьезной магии!

Да, пора. Ниже дано определение функции *skip*, которая теперь получает второй аргумент j – количество параметров из θ , потребляемых f :

```
(define skip
  (λ (f)
    (λ (t)
      (λ (θ)
        (+ ((f t) θ)
           ((correlate θ) t)))))))
```

Сколько параметров результирующий блок потребляет из θ ?

71 Он потребляет $j + 1$ параметров, потому что f получает j из них, и еще один получает *correlate*.

Закон о пропускных соединениях

Пропускное соединение для блока, имеющего вход с глубиной d и выход с глубиной b , требует наличия в θ банка фильтров с формой $(\text{list } b \ 1 \ d)$.

Верно.

Вот определение функции *skip-block*, которая получает блок ba с глубиной входа d и глубиной выхода b . Она возвращает новый блок с обходной связью вокруг *block*:

```
(define skip-block
  (λ (ba d b)
    (let ((shape-list (block-ls ba)))
      (block
        (skip (block-fn ba) |shape-list|)
        (append
          shape-list
          (list
            (list b 1 d)))))))
```

72 Сначала мы присваиваем списку форм ba имя *shape-list*. Функция возвращенного блока использует *skip* для создания пропускного соединения. Первым аргументом *skip* является блочная функция $(\text{block-fn } ba)$ а вторым аргументом является количество потребляемых ей аргументов, которое равно $|shape-list|$

Объясните, какую роль здесь играет функция *skip*.

Великолепно.

Теперь поясните, как определяются формы возвращенного блока.

73 Мы добавляем новую форму для банка фильтров пропускного соединения
(*list b 1 d*)
в конец списка форм блока
shape-list

Очень хорошо.

Теперь определим *residual-block*:

```
(define residual-block
  (λ (b m d)
    (skip-block
     (fcn-block b m d)
     d b)))
```

74 Это объясняет имя функции. Как *residual-block* участвует в объявлении нашей сети?

Пропускное соединение иногда называют *остаточным* (*residual*).

Эта функция идентична *morse-fcn*, за исключением того, что мы используем *residual-block* вместо *fcn-block*:

```
(define morse-residual
  (stack-blocks
   (list
    (residual-block 4 3 1)
    (residual-block 8 3 4)
    (residual-block 16 3 8)
    (residual-block 26 3 16)
    signal-avg-block)))
```

75 Вот она, долгожданная чистая, неподдельная, прекрасная магия. Она также заметно повысила точность!

```
▶ (accuracy
   (train-morse morse-residual)
   morse-test-xs morse-test-ys)
▶ 0.98
```

Эта маленькая частица магии подводит нас к концу нашей прямой линии к глубокому обучению.

76 Но ведь линия не имеет конца!

Конечно, и обучение никогда не заканчивается.

Но продолжать обучение лучше после того, как вы прочитаете эпилог этой книги.

77 Тогда скорее к эпилогу!

Какой десерт мы возьмем в дорогу?

Корреляционные побрякушки

corr стр. 316

recu стр. 317

skip стр. 334

skip-block стр. 334

Как насчет
джалеби (जलेबी)?

СО ВЗБИТЫМИ СЛИВКАМИ!



¹ Название главы напоминает нам о песнях в исполнении Карен Карпентер (1950–1983) и ее брата Ричарда Карпентера (1946–) и музыкальных аранжировщиках Роджере Николсе (1940–) и Поле Уильямсе (1940–).

Любой короткий путь по сложному рельефу обязательно обходит вниманием множество интересного: водовороты, ручейки и замысловатые изгибы дорог, которые и делают эти края увлекательными. В нашем путешествии по миру нейронных сетей мы намеренно двигались кратчайшим путем и упустили многие особенности ландшафта. В этом эпилоге мы познакомим вас с направлениями, которые стоит посетить тем, кто заинтересован в дальнейшем изучении темы, потому что на самом деле мы только начали.

1. Математическая база

Наше представление о глубоком обучении в предыдущих главах опиралось на минимально необходимое количество математики, но серьезное изучение существующей литературы по машинному обучению требует понимания математических основ этой области. Двумя наиболее важными областями являются линейная алгебра и теория вероятности со статистикой. Они часто сопровождаются изрядной порцией векторного исчисления.

Такие книги, как *Deep Learning* [1], содержат очень сжатое изложение необходимого математического аппарата, но предполагают определенный уровень математической подготовки. Чтобы достичь этого уровня подготовленности, можно воспользоваться такими книгами, как *Coding the Matrix* [2] (линейная алгебра) и *Bayesian Statistics the Fun Way* [3] (теория вероятностей). Они хороши для изучения упомянутых предметов с нуля.

2. Распределения генерации данных

Наборы данных – любопытная сущность. Они выглядят так, как будто в них нет ничего особенно интересного. Они просто содержат большое количество точек. Но на самом деле наборы данных предоставляют нам выборку результатов процесса генерации данных (data-generating process). Например, в примере с азбукой Морзе из главы 14 мы описываем, как используемые нами тензоры получают-ся из физического процесса нажатия кнопки фонарика.

Эти процессы генерации данных можно рассматривать с точки зрения вероятности. Мы связываем их с идеальным *распределением вероятностей* (probability distribution), которое сопоставляет каждую точку, генерируемую процессом, с вероятностью ее возникновения. Мы называем это *распределением генерации данных* (data generating distribution).

Процесс обучения параметров θ нейронной сети строит внутреннюю аппроксимацию этого распределения, генерирующего данные, и применяет его к конкретной текущей задаче.

3. Задачи

Смысл глубокого обучения заключается в том, чтобы делать всякие интересные штуки. Специалисты дали этим интересным штукам несколько скучное название. Они называют их задачами (task).

Одним из интересных применений многих приложений является прогнозирование рыночных цен (например, на акции, товары, недвижимость) на основе набора данных, собранных из предыдущих транзакций. В этих приложениях задача состоит в том, чтобы предсказать один скаляр. Этот тип задач называется *задачами регрессии* (regression).

В главе 3 показан пример регрессии на основе линейной модели, т. е. линейной функции и соответствующего ей набора параметров θ . Этот тип модели просто реализует математическую функцию, но на самом деле математическая функция моделирует некоторый аспект реального мира, например цену некоторого актива.

Еще одна разновидность интересных вещей, которые можно сделать с помощью нейронных сетей, – это идентифицировать объекты и лица на изображениях, идентифицировать слова в речи или ставить медицинские диагнозы по рентгеновским снимкам. Это примеры задачи, называемой *классификацией* (classification). Мы встречались с классификацией в главах 13 и 15.

Если нам понадобятся услуги языкового перевода, нейронные сети снова готовы прийти на помощь. Существуют специальные нейронные сети для перевода письменного текста с одного языка на другой. Некоторые нейронные сети дополнительно могут распознавать слова, встроенные в изображения, и переводить их, что особенно полезно при чтении табличек с помощью камеры телефона. Такого рода задачи называются *задачами перевода* (translation).

Задачи перевода можно применять и к изображениям. Особенно увлекательным примером такого перевода является перенос стиля (style transfer), когда визуальный стиль художника переносится на входное изображение для воссоздания того же изображения в стиле этого художника.

Другой класс задач связан с генерацией новых точек путем выборки данных, генерирующих распределение, которое нейрон-

ная сеть изучила из набора данных. Например, мы можем создавать оригинальные художественные изображения или музыкальные композиции, которые никогда раньше не существовали, используя специально разработанные нейронные сети, обученные на этих типах наборов данных.

Нейронные сети всегда разрабатываются для конкретной задачи, которая определяет выбор базовых функций и структур. Здесь мы обсудим некоторые идеи, влияющие на дизайн пространства.

4. Другие функции потерь

Одним из важнейших решений при проектировании нейронной сети для конкретной задачи является выбор функции потерь. Хотя *l2-loss* является широко применяемой функцией потерь [4], в различных приложениях и задачах широко используются и другие функции потерь. Давайте кратко рассмотрим некоторые из них.

Одной из популярных функций потерь для задач классификации является *функция кросс-энтропийной потери* (*cross-entropy loss*). Эта функция потерь имеет два интересных свойства. Во-первых, каждый аргумент `tensor1` должен в сумме давать 1.0, а во-вторых, каждый элемент в `tensor1` представляет вероятности для соответствующего класса и, следовательно, должен находиться в диапазоне от 0.0 до 1.0. Эта потеря определяется следующим образом:

```
(define cross-entropy
  (λ (target num-classes)
    (λ (xs ys)
      (λ (θ)
        (let ((pred-ys ((target xs) θ)))
          (* -1
            (÷ (• ys (log pred-ys))
              num-classes)))))))
```

Когда `ys` является унитарным вектором, *cross-entropy* выбирает самый высокий предсказанный класс. Умножение на -1 связано с тем, что вероятности всегда находятся в диапазоне от 0.0 до 1.0, а их логарифм всегда отрицательный, что дает отрицательный результат скалярного произведения. В соответствии с принятым соглашением потеря всегда положительная, поэтому мы умножаем ее на -1 .

Кросс-энтропийная потеря обычно используется в сочетании с решателем *softmax* (вы скоро с ним познакомитесь). Вместе эти две

функции позволяют обучать сеть с большей точностью и с меньшим количеством прогонов (обновлений).

Так выглядит функция потерь, похожая на *l2-loss*, за исключением того, что она использует функцию абсолютного значения вместо возведения в квадрат, чтобы избавиться от отрицательных разностей.

```
(define l1-loss
  (λ (target)
    (λ (xs ys)
      (λ (θ)
        (let ((pred-ys ((target xs) θ)))
          (sum (abs (- ys pred-ys))))))))
```

Эту функцию потерь можно использовать, когда мы не хотим, чтобы выбросы в наборе данных непропорционально влияли на потери.

5. Другие решатели

В предыдущих главах мы использовали единственный решатель – *rectify*. Несмотря на то что он обладает рядом очень полезных свойств и чрезвычайно прост, *rectify* относится к относительно недавним разработкам. Ранее в нейронных сетях применялись нелинейные функции, например *логистическая сигмоида* (logistic sigmoid)¹ и *гиперболический тангенс* (tanh).

```
(define logistic-sigmoid
  (λ (x)
    (let ((ex (exp x)))
      (÷ ex (+ 1 ex)))))

(define tanh
  (λ (x)
    (let ((e2x (exp (* 2 x)))
          (div (- e2x 1) (+ e2x 1))))))
```

Они все еще применяются в некоторых особых случаях, но их использование значительно сократилось в пользу решателя *rectify* и его вариантов.

¹ Описана в работе Пьера-Франсуа Верхальста (1804–1849).

Одним из вариантов *rectify* является *спрямление с утечкой* (*leaky rectify*) [5], которое использует небольшой ненулевой наклон для входных данных, меньших 0.0. Этот решатель можно определить так:

```
(define leaky-rectify
  (λ (m)
    (λ (x)
      (cond
        ((< x 0) (* m x))
        (else x))))))
```

Основное преимущество спрямления с утечкой заключается в том, что оно позволяет пропустить небольшое количество отрицательного градиента при расчете градиентов. Это полезно в глубоких сетях, поскольку ускоряет обучение и позволяет избежать некоторых патологических состояний, которые иногда могут возникать при исправлении ошибок в больших сетях. В этом определении значение *m* должно быть предоставлено в качестве аргумента, но существуют и другие варианты, когда *m* становится частью *θ* и, таким образом, может быть обучено [6].

Рассмотрим еще один пример решателя, который несколько ограничен в вариантах использования, но зато широко применяется в таких ситуациях. Это функция *softmax*.

```
(define softmax
  (λ (x)
    (let ((expd (- (exp x) (max x))))
      (÷ expd (sum expd)))))
```

Данная функция всегда возвращает tensor^1 , отвечающий критерию распределения вероятностей – каждый элемент в нем находится в диапазоне от 0.0 до 1.0, а сумма элементов равна 1.0. Из-за этого свойства решатель *softmax* используется в сочетании с функцией потерь *cross-entropy* (которая ожидает распределения вероятностей), что делает его идеальным для последнего слоя нейронной сети-классификатора. Кроме того, вычитание *max x* необходимо для численной стабилизации суммы, чтобы она никогда не равнялась 0.0, но это не меняет окончательное значение функции.

6. Многомерные сигналы

Чтобы иметь дело с более сложными применениями нейросетей, связанными с изображениями и видео, нам нужно расширить наше понятие о сигналах. До сих пор мы имели дело только с сигналами, имеющими отсчеты в одном измерении (это были наши сигналы азбуки Морзе из фрейма 287:29). Физические явления, такие как звуки, сейсмические колебания и изменчивость температуры, часто служат источниками сигналов такого типа, которые полезны во многих различных применениях.

К сигналам более высокой размерности относятся изображения, видео и другие типы данных, встречающиеся в очень большом количестве визуальных приложений, таких как классификация изображений, преобразование фотографий и т. д. Принципы сверточных нейронных сетей, такие как корреляция, аналогичны в более высоких измерениях, за исключением того, что они имеют дело с корреляцией более чем в одном измерении вместо одного. Следовательно, функция *correlate* расширяется до большего количества измерений.

Сигналы более высокой размерности порождают сети с гораздо большими наборами параметров θ , поскольку теперь мы имеем дело с очень обширными данными. Нередко можно встретить θ , содержащие сотни миллионов скалярных параметров. На примерах таких сетей, как AlexNet [7], ResNet [8] и VGG [9], чрезвычайно интересно исследовать архитектурные решения.

7. Системы для обработки естественного языка

Текстовые задачи, такие как перевод и классификация текстов, требуют обработки естественных языков. Нейронная сеть, предназначенная для обработки естественного языка, обычно основана на очень специфических структурах слоев, которые позволяют сети изучать взаимосвязь между словами в разных частях фрагмента текста.

Эти сети могут изучать, как идеи (смысл) проходят через последовательность предложений, и воспроизводить их по мере необходимости в таких задачах, как перевод и обобщение. Такие сети, как GPT-3 [10], содержат миллиарды скаляров в своих наборах θ и способны решать очень сложные задачи.

Предложения в системе обработки текста представлены последовательностью токенов. Нейронные сети обработки текста строят модели вероятности того, что одно слово следует за другим в последо-

вательности. Затем обученные модели используются для таких задач, как ответы на вопросы, диалоги «человек–машина» (так называемые чат-боты), генерация текста и т. д. Нейронные сети обработки текста можно комбинировать с нейронными сетями обработки речи, чтобы построить все чаще встречающиеся голосовые помощники.

Существует два основных типа архитектур, используемых для обработки текста. Одна из них – это *архитектура рекуррентной (recurrent) нейронной сети*, которая использует выходные данные, связанные с предыдущим токеном, для обработки будущего токена. Это позволяет сети получать информацию о контексте, в котором появляется данный токен.

Но сейчас рекуррентные сети быстро заменяются более новой архитектурой, основанной на идее *внимания (attention)* [11]. Отличие сетей, основанных на внимании, заключается в том, что они работают с целыми последовательностями, а не с одним токеном за раз. Они включают в себя блок обработки, так называемый трансформер (transformer), использующий внимание для реализации механизма, с помощью которого сеть может узнать, как каждый токен во входной последовательности влияет на другие токены.

8. Генеративные сети

В приложениях, требующих генерации контента, используются два основных типа архитектур нейронных сетей.

Первый тип – это *вариационный автокодер (variational auto-encoder)*. Чтобы понять, как работает эта сеть, сначала нужно разобраться, что такое автокодер.

Автокодер – это нейронная сеть, состоящая из двух частей, которые обучаются вместе, но могут использоваться по отдельности. Эти две части называются кодер и декодер.

Кодер отображает каждый ввод в более компактную форму того же ввода. Обычно входными данными являются тензоры, состоящие из очень большого количества скаляров. Компактная форма выходных данных обычно представляет собой tensor^1 , состоящий из очень небольшого числа скаляров. Например, можно создать кодер, который будет брать изображение букв алфавита в определенном шрифте и представлять его в виде тензора tensor^1 , состоящего из 12 скаляров, так называемого кода. Обучив этот кодер изображениям разных шрифтов, мы можем заставить его создавать отдельные коды для каждого шрифта.

Декодер выполняет обратное преобразование. Он принимает компактную форму (код) в качестве входных данных и производит выходные данные, максимально приближенные к исходным входным данным.

Вариационный *автокодер* [12] гарантирует, что коды, создаваемые кодирующей частью, обладают уникальными математическими свойствами, чтобы их можно было рассматривать как выборки из генератора случайных чисел. Затем, предоставляя новые случайные числа соответственно обученному декодеру, можно сгенерировать новые точки данных, которые по характеристикам будут аналогичны исходному набору входных данных, на которых обучалась сеть.

Например, предоставляя нашему декодеру шрифтов новые случайные 12-скалярные коды, мы можем создавать изображения совершенно новых шрифтов. Одни и те же принципы могут быть использованы для создания выходных данных самых разных типов.

Другой тип архитектуры, используемый в генеративных приложениях, – *генеративно-сопоставительная сеть* (generative adversarial network, GAN) [13]. Этот тип нейронной сети также состоит из двух частей.

Первая – это генератор, который отвечает за преобразование случайного числа в результат с нужными характеристиками, например изображение шрифта. Вторая – дискриминатор, который отвечает за принятие решения о том, проходит ли выходной сигнал генератора проверку на достоверность. В нашем примере дискриминатор решит, является ли генератор изображений генератором шрифта.

Изюминка GAN заключается в том, что в процессе обучения «наказывают» как дискриминатор, если он ошибается в своих суждениях, так и генератор, если он генерирует неправильный вывод.

Совместное обучение дискриминатора и генератора гарантирует, что генератор научится создавать выборки, которые будет принимать дискриминатор, а дискриминатор научится принимать только выборки, удовлетворяющие требованиям приложения, и отбраковывать остальные.

9. Прикладные соображения

Одной из наиболее важных характеристик архитектуры нейронной сети является ее *емкость* (capacity), которую можно грубо представить как количество скаляров в ее наборе θ . Если параметров

слишком мало, сеть не сможет достаточно хорошо изучить обучающий набор, что называется *недообучением* (underfitting). Если их слишком много, то сеть «выучит наизусть» обучающий набор и будет плохо работать с незнакомыми данными – состояние, известное как *переобучение* (overfitting).

Даже если сеть имеет подходящую емкость, ее обучение на слишком малом количестве *revs* может привести к недообучению, а на слишком большом количестве – к переобучению.

Либо недообучение, либо переобучение, либо и то, и другое может привести к тому, что сеть не будет работать оптимально. Сеть не сможет обобщить точки, лежащие за пределами обучающей выборки. Выбор сетей с нужной емкостью и тщательная настройка гиперпараметров – очень важная часть процесса обучения сети.

Понятию правильной емкости трудно дать точное определение, потому что оно зависит от архитектуры сети. Например, обходные связи могут значительно повысить точность сети без существенного влияния на количество скаляров в θ .

Как и в других ситуациях, о которых говорилось в этой книге, эксперименты помогают разработать эвристики и правила проектирования, позволяющие создавать сети с адекватной емкостью.

Для правильного обучения нейронных сетей определенным задачам необходимо владеть разнообразными практическими приемами.

Во-первых, всегда полезно тщательно отслеживать метрики сети по мере ее обучения. Отслеживание потерь при обучении, то есть значения, полученного функцией потерь на каждом из обновлений, помогает следить за тем, насколько хорошо обучается сеть.

Во-вторых, использование надлежащего проверочного набора для отслеживания соответствующей метрики качества сети (например, точности прогнозов) помогает определить, правильно ли происходит обучение.

В-третьих, для управления эффективностью обучения используются такие методы, как регуляризация. Регуляризация – это название, данное методу добавления дополнительного члена к функции потерь, который влияет на направление градиентного спуска. Примеры регуляризации приведены в работе Гудфеллоу и соавт. [1]. Регуляризация помогает обучать сеть без недообучения или переобучения, что улучшает общее качество сети с точки зрения поставленной перед ней задачи.

В-четвертых, важной частью обучения является инициализация весов. Хе и соавт. [6] предложили схему инициализации, используемую в этой книге. Эта схема полезна для сетей на основе спрямляющих функций. Другие схемы, такие как предложенная Глоротом и соавт. [14], помогают в работе с иными типами сетей.

10. Вперед, отважные ученики!

В этой книге мы вихрем промчались по темам и источникам знаний для любопытных читателей. Мы надеемся, что благодаря пониманию нейронных сетей, полученному из нашей книги, эти темы станут менее сложными.

À bientôt¹!

¹ До скорой встречи – фр.

ПРИЛОЖЕНИЕ А. ПРИЗРАК В МАШИНЕ¹



¹ «Призрак в машине» – понятие, введенное британским философом Гилбертом Райлом (1900–1976); послужило названием песни группы The Police: Эндрю Саммерс (1942–), Стинг (1951–) и Стюарт Коплденд (1952–).

Начиная с главы 4 почти все, что мы делали, зависит от существования

ТАИНСТВЕННОГО



Это и следующее приложение, предназначенные для тех, кому интересно, как все устроено, представляют собой полное объяснение происходящего. Упомянутые в приложениях функции обеспечивают полное автоматическое дифференцирование, составляющее семантическую основу для всех программ, продемонстрированных в основной части этой книги, то есть в главах с 1 по 15.

Это надмножество механизма, традиционно известного в истории нейронных сетей как обратное распространение. Алгоритм обратного распространения – это пример обратного режима автоматического дифференцирования, который мы описываем здесь.

В основной части книги все определения даны в строгом логическом порядке, обеспечивающем цельность восприятия. Мы тщательно избегали определений Scheme, которые могут нарушить это состояние ума. Однако общий характер автоматического дифференцирования, а также наше желание сделать определения простыми для понимания требуют понимания некоторых частей Scheme, которых мы избегали.

Несмотря на то что эти определения написаны на метаязыке, мы попытались сохранить некоторые обозначения со стр. **xxiii**, особенно для членов списка и элементов векторов. Для читателей, которые самостоятельно попытаются определить эти функции по ходу чтения, мы предлагаем следующие обозначения:

Фрейм	Обозначение метаязыка	Перевод в Scheme
31	$[e\ es\ \dots]$	(vector <i>e es</i> ...)
32	$v _i$	(vector-ref <i>v i</i>)
51	$\dagger v \dagger$	(vector-length <i>v</i>)
51	l_0	(list-ref <i>l 0</i>) или (car <i>l</i>)
51	l_{1i}	(drop <i>l 1</i>) или (cdr <i>l</i>)
72	$(\text{let}^2 ((a\ b)\ E)\ B)$	(let-values (((<i>a b</i>) <i>E</i>)) <i>B</i>)
75	$\langle e1\ e2 \rangle$	(values <i>e1 e2</i>)

Такие символы, как ∇ , σ , ρ , κ , могут быть введены непосредственно как символы Unicode или записаны в английской транслитерации (т. е. *nabla*, *sigma*, *rho*, *kappa*).

Остальная часть этого приложения представлена в формате «маленьких коробочек», но стиль отличается. Вместо уже знакомого вам стиля «вопрос–ответ» приложение больше похоже на комиксы. Его следует читать по одному фрейму за раз, слева направо внутри фрейма.

1. Пролог

Вот определение ∇ :

```
(define  $\nabla$ 
  ( $\lambda$  ( $f$   $\theta$ )
    (let ((wrt ( $map^*$   $dual^*$   $\theta$ )))
      ( $\forall once$  ( $f$  wrt) wrt))))
```

¹ Сбиты с толку? Не бойтесь.

Цель этого приложения – полностью объяснить, о чем идет речь в данном определении.

2. Градиент и все остальное

Градиент представляет собой скорость изменения результата функции по отношению к аргументу этой функции.

² Пусть нам дана функция f , получающая один скалярный аргумент. Вызовем ее для скаляра a :

$(f\ a)$

Для лучшего понимания того, что представляет собой градиент f , мы начнем с метода, который использовали во фрейме 63:30[†]. Мы вызываем f для аргумента, который отличается от a на небольшую величину $\Delta a^\ddagger$, которую можно сделать произвольно маленькой:

³ Мы ожидаем, что результат тоже изменится на некоторую величину. Мы ссылаемся на эту величину при помощи записи

Δf_a

Здесь нижний индекс представляет аргумент(ы) в вызове f .

$(f\ (+\ a\ \Delta a))$

[†] С той разницей, что здесь мы используем более абстрактное значение вместо определенного 0.0099 во фрейме 63:30.

[‡] Произносится «дельта а».

Градиент f относительно a представляет собой скорость изменения результата ($f a$) по отношению к изменению a . Отсюда следует отношение

4

$$\frac{\Delta f_a}{\Delta a}$$

Это отношение[†] можно расширить до функции более, чем одного аргумента, так что каждый аргумент можно будет рассматривать независимо. Пусть g получает два скалярных аргумента, и мы вызываем ее для a и b .

5

Мы находим скорость изменения по отношению к a и b независимо следующим образом:

$$\frac{\Delta g_a}{\Delta a}$$

и

$$\frac{\Delta g_b}{\Delta b}$$

где Δg_a (и, соответственно, Δg_b) представляют собой изменения в

($g a b$)

когда a (и, соответственно, b) изменяется на Δa (Δb).

Градиент g относительно a и b представляет собой список[‡]

$$\left(\text{list } \frac{\Delta g_a}{\Delta a} \frac{\Delta g_b}{\Delta b} \right)$$

[†] Для тех, кто знаком с дифференциальным исчислением: пределы неявны, и выражения с использованием Δa следует неявно понимать как находящиеся в пределе, когда Δa приближается к 0.0. Для функций, определенных, но имеющих разрыв в точке a , результаты автоматического дифференцирования не определены и могут быть получены предельным приближением либо слева, либо справа.

[‡] Примером может служить фрейм 76:20.

Любая функция, которая принимает тензоры и возвращает скаляр (например, функция потерь), обрабатывается так же, как и функция с несколькими аргументами.

Каждую скалярную величину s в тензорах аргументов изменяют на Δs и наблюдают изменение результирующей скалярной величины.

6

Вместо упаковки градиентов в список, как было сделано для g во фрейме 5, мы определяем градиент как тензор той же формы, что и аргумент, но со значениями градиента, заменяющими скаляры в этом аргументе.

Например, пусть h будет функцией, которая принимает тензор формы (list 2 3), и мы вызываем ее для этого тензора:

$$\begin{bmatrix} ax & ay & az \\ bx & by & bz \end{bmatrix}_{(2\ 3)}$$

Мы находим градиенты h относительно каждого скаляра ax , ay , az , bx , by , bz и записываем их вместе, как показано справа:

$$\begin{bmatrix} \frac{\Delta h_{ax}}{\Delta ax} & \frac{\Delta h_{ay}}{\Delta ay} & \frac{\Delta h_{az}}{\Delta az} \\ \frac{\Delta h_{bx}}{\Delta bx} & \frac{\Delta h_{by}}{\Delta by} & \frac{\Delta h_{bz}}{\Delta bz} \end{bmatrix}_{(2\ 3)}$$

Функция, которая принимает списки (тензорные или вложенные), обрабатывается аналогично, так что градиент по отношению к аргументу списка представляет собой список с той же структурой, но со скалярными градиентами в местах, где аргумент списка имеет скаляры.

Другими словами, для функции, которая производит скаляр, ее градиент по отношению к аргументу в виде списка или тензора имеет ту же структуру, что и этот аргумент, но любой скаляр в аргументе заменяется соответствующим градиентом по отношению к этому скаляру.

3. Примитивы, композиции и цепочки

Мы рассматривали абстрактные градиенты, записывая их просто как отношение двух абстрактных величин. Однако для конкретных примитивов, таких как сложение и умножение скаляров (и многих других), соотношения можно привести к фиксированной формуле.[†]

В качестве примера рассмотрим градиент функции «+» относительно скаляров a и b .

[†] В этом суть дифференциального исчисления.

В случае функции «+», если мы меняем a (или b) на Δa (или на Δb), результат также меняется на Δa (или на Δb). Следовательно, мы можем записать для градиента функции «+» такой вывод:

1. $\left(\text{list } \frac{\Delta + a}{\Delta a} \quad \frac{\Delta + b}{\Delta b} \right)$
2. $\left(\text{list } \frac{\Delta a}{\Delta a} \quad \frac{\Delta b}{\Delta b} \right)$
3. $\left(\text{list } 1.0 \ 1.0 \right)$

Градиенты равны 1.0 и 1.0 и не зависят от a и b .

Остальные формулы выводятся аналогичным образом (хотя вывод может быть намного сложнее этого) для всех используемых нами примитивов, так что каждый градиент определяется простой формулой, основанной на аргументах примитива.

¹¹ Например, градиент функции «*» относительно скаляров a и b записывается так:

$$(\text{list } b \ a)$$

Другими словами, нам никогда не нужно явно находить отношения изменения результатов к изменению аргументов.

¹² Опираясь на эти формулы градиентов примитивов, мы применяем фиксированное правило для нахождения градиентов функций, построенных из примитивов.

Когда результат вызова одной функции, скажем g , является аргументом другой функции, скажем f , тогда говорят, что результат f возникает из

композиции функций
 f и g .

¹³ Например, пусть f и g – две функции, каждая из которых принимает один скалярный аргумент и выдает скалярный результат. Применим композицию f и g к скаляру a так же, как делали это для

$$(f (g \ a))$$

Запишем это в более явном виде:

$$(\text{let } ((b \ (g \ a))) \\ (f \ b))$$

Предположим, что значение a получается применением другой функции, скажем h , к другому скаляру, скажем 3.72.

¹⁴ Мы можем «развернуть» этот вызов h и добавить его в выражение let :

$$(\text{let } ((a \ (h \ 3.72))) \\ (\text{let } ((b \ (g \ a))) \\ (f \ b)))$$

Это выражение <code>let</code> эквивалентно $(f(g(h\ 3.72)))$ В этом случае мы говорим, что f, g и h образуют цепочку вызовов где вызов h является самым глубоким (<i>innermost</i>) а вызов f является самым внешним (<i>outermost</i>)	15 Мы можем повторять этот процесс развертывания функций компонентов до тех пор, пока вся цепочка не будет состоять только из примитивных функций.
Скаляр, передаваемый в качестве аргумента (здесь 3.72) в самую глубокую функцию, называется аргументом цепочки	16 Результат, возвращаемый самой внешней функцией, называется результатом цепочки.
Наиболее простая из возможных композиций[†] двух примитивов f и g может быть записана в виде выражения <code>let</code>, где скаляр a является константой: $(\text{let } ((b\ (g\ a)))\ (f\ b))$ ----- [†] Также во фрейме 14.	17 Чтобы найти градиент композиции f и g, изменим a на Δa.
Результат выражения $(g\ a)$ изменится на Δg_a. Поскольку результатом $(g\ a)$ является скаляр b, мы дадим Δg_a имя Δb. Изменение выражения в целом можно записать как Δf_b.	18 Градиентом выражения относительно a будет отношение $\frac{\Delta f_b}{\Delta a}$

Перепишем это выражение как ¹⁹

$$\frac{\Delta f_b}{\Delta b} \times \frac{\Delta b}{\Delta a}$$

что аналогично

$$\frac{\Delta f_b}{\Delta b} \times \frac{\Delta g_a}{\Delta a}$$

Здесь просматривается закономерность. Правый член – это градиент g относительно a , а левый член – это градиент f относительно b .

Следовательно, градиент композиции f и g относительно аргумента цепочки можно расписать как

$$\begin{aligned} &(\text{градиент } f \text{ относительно} \\ &\quad \text{ее аргумента}) \\ &\quad \times \\ &(\text{градиент } g \text{ относительно} \\ &\quad \text{ее аргумента}) \end{aligned}$$

²⁰ Это так называемое *цепное правило (chain rule)*. Оно дает нам способ получить градиент композиции функций, используя индивидуальные градиенты каждой из функций-компонентов.

Теперь рассмотрим цепочку из более чем двух примитивов:

$$(f0 (f1 \dots (fn a)))$$

²¹ Градиент композиции можно расписать аналогичным образом:

$$\begin{aligned} &(\text{градиент } f0 \text{ относительно} \\ &\quad \text{ее аргумента}) \\ &\quad \times \\ &(\text{градиент } f1 \text{ относительно} \\ &\quad \text{ее аргумента}) \\ &\quad \times \\ &(\text{градиент } fn \text{ относительно} \\ &\quad \text{ее аргумента}) \end{aligned}$$

Это можно представить как прохождение по цепочке при накопительном умножении градиента каждого примитива по отношению к его аргументу. Аккумулятор, который мы используем, называется

мультипликативным аккумулятором.

²² Начальное значение мультипликативного аккумулятора равно 1.0, и мы умножаем его на один градиент на каждом шаге цепочки.

Когда мы пройдем всю цепочку, мультипликативный аккумулятор будет содержать градиент результата цепочки относительно ее аргумента.

Как и раньше, мы можем обобщить этот процесс, включив в него примитивы, принимающие более одного аргумента. Для этого мы должны индивидуально накопить градиенты по каждому скалярному аргументу. Поскольку теперь существует много скаляров, относительно которых мы находим градиенты, существует столько же различных мультипликативных аккумуляторов, каждый из которых участвует только в цепи, связанной со своим скаляром.

В первом способе начинаем с мультипликативного аккумулятора = 1.0
и умножаем его на
градиент h относительно a
затем умножаем на
градиент g относительно $(h a)$
и наконец умножаем на
градиент f относительно $(g(h a))$

Первый способ, когда начинают с h и заканчивают f , называется

прямым режимом (forward mode)

автоматического дифференцирования.

В этом режиме мы начинаем с мультипликативного аккумулятора = 1.0.

23 Определения в этом приложении имеют дело с этой полной общностью, но в данном разделе мы рассматриваем только цепочки с одним аргументом.

Поскольку умножение ассоциативно, существует два способа пройти по этой цепочке. Например, если у нас есть цепочка

$$(f(g(h a)))$$

то градиент этой композиции относительно a можно найти двумя способами.

24 Во втором способе начинаем с мультипликативного аккумулятора = 1.0
и умножаем его на
градиент f относительно $(g(h a))$
затем умножаем на
градиент g относительно $(h a)$
и наконец умножаем на
градиент h относительно a

25 Затем мы берем самый глубокий примитив в цепочке, умножаем его градиент на исходный мультипликативный аккумулятор и продолжаем обрабатывать цепочку справа налево, перенося результат самого примитива, а также мультипликативного аккумулятора.

-
- | | |
|--|--|
| <p>В прямом режиме нам не нужно явно строить цепочки – градиенты находятся по мере того, как мы определяем результаты примитивов. Это хорошо работает, когда градиенты, которые мы ищем, исходят от функций, которые имеют очень мало скаляров в своих аргументах.</p> | <p>26 Когда аргументы этих функций состоят из очень большого количества скаляров, таких как встречающиеся в тензорах и θ, с которыми мы сталкиваемся в нейронных сетях, нам приходится отслеживать очень большое количество аккумуляторов. Когда наши цепочки включают бинарные примитивы (а они всегда есть), это приводит к необоснованному увеличению числа мультипликативных аккумуляторов, которые нам приходится обрабатывать.</p> |
| <p>Второй способ, альтернативный прямому режиму, называется</p> <p style="padding-left: 40px;"><i>обратным режимом (reverse mode)</i></p> <p>автоматического дифференцирования.</p> <p>Здесь мы проходим наши цепочки слева направо, т. е. от самого внешнего примитивного вызова до самого глубокого.</p> | <p>27 В нашем примере с f, g и h этот режим рассмотрен во фрейме 24.</p> <p>Однако, в отличие от прямого режима, это заставляет нас явно создавать цепочку примитивов. Мы строим эту цепочку во время вычисления результирующего скаляра, но начинаем проходить цепочку только тогда, когда нам действительно нужен градиент.</p> |
| <p>Преимущество автоматического дифференцирования в обратном режиме заключается в отсутствии непропорционального увеличения числа мультипликативных аккумуляторов для функций, которые производят скаляр из составных аргументов.</p> | <p>28 Это позволяет нам обрабатывать фиксированное количество мультипликативных аккумуляторов, по одному для каждого скаляра в аргументе цепочки, и обновлять их по мере обхода цепочек.</p> |
-

В оставшейся части этого приложения подробно описано, как происходит автоматическое дифференцирование в обратном режиме.

²⁹ Процесс начинается с представления скаляров таким образом, чтобы явно выстраивались цепочки примитивов, затем формируется описание того, как обрабатываются составные структуры данных, и, наконец, как конструируются цепочки и извлекаются градиенты.

4. Представление скаляров

Наша автоматическая дифференциация происходит во время выполнения. Мы не пытаемся перевести функции в их эквивалентные дифференциальные формы. Вместо этого мы оцениваем результат функции для заданных аргументов, а затем определяем градиент этого результата по отношению к этим аргументам.

³⁰ Это означает, что числовые примитивы, такие как сложение и умножение, не только возвращают свои числовые результаты, но и организуют цепочку так, чтобы градиенты можно было определять всякий раз, когда это требуется.

Для этого нам нужен способ представления скаляров, состоящий из двух частей. Первая, его

вещественная (*real*)

часть, обозначаемая *r*, представляет собой числовое значение скаляра. Вторая, его

звеньевая (*link*)[†]

часть, обозначаемая *k*, представляет собой функцию, которая управляет цепочкой, создавшей этот скаляр, и вызывается для прохода по цепочке.

³¹ Каждая двойственная форма представлена вектором. Функция *dual* строит вектор из 3 элементов. 0-й элемент является тегом, позволяющим отличить этот вектор от остальных. 1-й элемент – это *r*, 2-й – *k*

```
(define dual
  (λ (r k)
    [dual r k ]))
```

Чтобы предоставить уникальный тег, мы используем саму функцию *dual* в качестве ее тега.

Мы называем это представле-
ние

двойственным (dual)[†]

[†] Название *link* (звено цепи) появилось здесь, потому что эта часть является одним звеном в цепочке примитивов, которые, возможно, произвели данный скаляр.

[‡] Термин предложил Уильям Клиффорд. Здесь наше понятие двойственности немного отличается от того, которое ввел Клиффорд, но суть аналогична, поэтому мы используем такой же термин.

В дальнейшем имя переменной *d* и имена переменных, начинающиеся с *d*, обозначают двойственные числа.

Предикат *dual?* использует этот уникальный тег, чтобы проверить, является ли данное значение двойственным

```
(define dual?
  (λ (d)
    (cond
      ((vector? d) (eq?† d|0 dual))
      (else #f))))
```

[†] Этот предикат истинен, если его аргументы находятся в том же месте в памяти. Здесь это верно, когда две функции одинаковы.

Интересный взгляд на *eq?* можно найти в главе 18 «We Change Therefore We Are the Same!» в книге *The Seasoned Schemer*, MIT Press, 1996.

³² Выражения, используемые в основной части книги, часто содержат литеральные константы, например для гиперпараметров или когда наборы данных считываются из файлов. Эти константы представлены в виде вещественных чисел. Однако для целей автоматической дифференциации в обратном режиме мы рассматриваем их как двойственные.

Теперь мы будем называть скалярами как вещественные, так и двойственные числа.

Это именно скаляры из основной части книги, как во фрейме 30:7.

Следовательно, мы определяем функции доступа, которые возвращают вещественную часть и ссылку из скаляра, независимо от того, является это вещественным числом или двойственным.

Функция ρ возвращает вещественную часть скаляра. Она начинается с проверки, является ли скаляр двойственным. Если это так, его вещественная часть возвращается путем обращения к вектору, представляющему двойственность.

Точно так же κ возвращает звеньевую часть скаляра. Если d является двойственным, его звено возвращается путем обращения к вектору, который представляет двойственный скаляр:

```
(define  $\kappa$ 
  ( $\lambda$  ( $d$ )
    (cond
      (( $dual?$   $d$ )  $d|_2$ )
      (else  $end-of-chain$ ))))
```

33 Исходя из этого, дадим определение предиката *scalar*?:

```
(define scalar?
  ( $\lambda$  ( $d$ )
    (cond
      (( $number?$   $d$ ) #t)
      (else ( $dual?$   $d$ ))))
```

34 Если d не является двойственным, мы считаем, что это вещественное число:

```
(define  $\rho$ 
  ( $\lambda$  ( $d$ )
    (cond
      (( $dual?$   $d$ )  $d|_1$ )
      (else  $d$ ))))
```

35 Если скаляр не является двойственным, то цепи, которая его произвела, не существует. В этом случае звеньевая часть должна быть функцией, завершающей цепочку. Мы называем это функцией

end-of-chain

и вскоре дадим ей определение.

5. Дифференцируемые функции

Функции, для которых нас интересует нахождение градиентов, обычно образуют скалярные потери, как во фрейме 56:9. Мы находим градиенты потери относительно аргумента θ этих функций.

Для наших целей мы называем такие функции

дифференцируемыми функциями

Здесь мы не нуждаемся в таком обобщении. Поэтому выполним упрощающую оптимизацию. Мы напрямую создаем один θ -образный градиент, который представляет собой сумму всех отдельных θ -образных градиентов, фактически не создавая эти отдельные θ -образные градиенты[†].

[†] Для тех, кому понятно значение этих слов: мы находим построчную сумму якобиана.

³⁶ При представлении пакета, как во фрейме 116:12, наши дифференцируемые функции производят `tensor`¹ скаляров с одним скаляром для каждого элемента пакета.

В общем случае автоматическое дифференцирование создает θ -образный градиент для каждого скаляра в результате.

³⁷ Преимущество этой оптимизации заключается в том, что она упрощает градиентный спуск, устраняя необходимость сохранять все отдельные θ -образные градиенты.

6. Дифференцируемые объекты

Исходя из того, как мы выстраиваем двойственные скаляры, мы должны еще раз подчеркнуть здесь, что двойственное число само содержит породившую его цепь.

³⁸ Поэтому нас больше интересуют результаты, получаемые дифференцируемыми функциями, чем сами функции.

Но двойственные числа встраиваются в другие структуры, такие как списки и векторы (например, в θ). Эти списки и векторы, в свою очередь, передаются в качестве аргументов дифференцируемым функциям и создаются как результаты данных функций.

39 Мы называем скаляры, списки и векторы

дифференцируемыми объектами

Дифобъекты определяются как

скаляры

списки дифобъектов

или

векторы дифобъектов

Мы можем рассматривать дифобъекты как древовидные[†] структуры, на листьях которых есть скаляры. Узлы в этой структуре – списки и векторы. Кроме того, скаляры на листьях хранят цепи, которые их произвели.

40 Мы часто сталкиваемся с необходимостью пройти по структуре дифобъектов, чтобы добраться до скаляров на листьях.

[†] Точнее, направленно-ациклично-графоподобные структуры.

Одной из полезных функций обхода дифобъектов является *tap**. Эта функция возвращает новый дифобъект с той же структурой, что у аргумента, но листья содержат другие скаляры.

41 Первый аргумент *tap** – это функция *f*, которая принимает один скаляр для произведения другого скаляра. Второй аргумент *tap** – это дифобъект *u*.

В итоге *tap** возвращает новый дифобъект, где каждый скаляр, скажем *d*, в *u* заменяется на *(f d)*.

Определение *map** может быть таким:

```
(define map*
  (λ (f y)
    (cond
      ((scalar? y) (f y))
      ((list? y)
       (map (λ (lm†)
              (map* f lm))
            y))
      ((vector? y)
       (vector-map (λ (ve‡)
                     (map* f ve))
                    y)))))
```

† От *list member* (член списка).

‡ От *vector element* (элемент вектора).

Следующая функция *dual** преобразует любой скаляр в усеченное двойственное представление. Усеченное представление – это двойственное число, звеном которого всегда является функция *end-of-chain*.

```
(define dual*
  (λ (d)
    (dual (ρ d)
          end-of-chain)))
```

42 Эта функция рекурсивно проходит структуру *y*. В основном тесте, где *y* – скаляр, мы вызываем *f* для *y*, тем самым создавая новый скаляр.

Для списков (векторов) мы рекурсивно обходим отдельные элементы[§].

[§] Это определение включает в себя возможность того, что векторы могут содержать списки, хотя мы не используем такие векторы в основной части книги.

43 Рассмотрим пример использования функции *map**. Если *y* является дифференцируемым объектом, выражение

$(map^* dual^* y)$

возвращает дифобъект, содержащий в своих листьях только усеченные двойственные числа. Мы будем называть эту разновидность дифобъектов *усеченными дифобъектами*

Теперь мы запаслись достаточными знаниями, чтобы начать разбираться с ∇ .

```
(define  $\nabla$ 
  ( $\lambda$  ( $f$   $\theta$ )
    (let ((wrt ( $map^* dual^* \theta$ )))
      ( $\nabla once$  ( $f$  wrt) wrt))))
```

⁴⁴ Аргумент f – это функция, для которой ищется градиент, а θ – аргумент функции f , относительно которой мы ищем градиент.

Поскольку θ представляет собой список тензоров и поскольку тензоры являются либо скалярами, либо, возможно, вложенными векторами, θ сам по себе является дифференцируемым объектом.

Сначала преобразуем θ в усеченный дифобъект

$(map^* dual^* \theta)$

⁴⁵ Эта операция отбрасывает любые предыдущие звенья, которые могут содержать скаляры в θ . Выгода заключается в ограничении определения градиента исключительно тем, что выполняет f , а не историей θ до вызова f .

Этому усеченному дифобъекту присвоим имя *wrt* как сокращение от математического термина

with respect to (относительно ...)

как напоминание, что θ является аргументом f , определяющим градиент.

⁴⁶ Стоит еще раз подчеркнуть, что *wrt* по структуре идентичен θ , но каждый скаляр из θ в *wrt* теперь стал усеченно-двойственным.

Также стоит отметить, что, в отличие от θ , *wrt* всегда содержит усеченные двойственные числа на листовых позициях.

Теперь мы готовы вызвать f с аргументом wrt , который на самом деле является упрощенным θ . Усеченные двойственные числа на листьях wrt становятся аргументами цепочек, которые строятся при вызове f . Другими словами, все градиенты результата этого вызова определяются относительно этих усеченных двойственных чисел в wrt .

⁴⁷ Вызовем функцию f с аргументом wrt . Мы предполагаем, что этот вызов успешно завершается и создает дифобъект, и затем определяем градиенты для этого дифобъекта относительно wrt . Фактическое определение градиента осуществляется функцией ∇_{once} , которую мы сейчас опишем.

7. Градиентное состояние и ∇_{once}

Когда мы идем по цепочке примитивов, нам нужна структура, которая отслеживает градиенты.

⁴⁸ Кроме того, эта структура должна помнить один градиент для каждого скаляра в аргументе[†].

[†] Эти скаляры являются подмножеством скаляров в wrt .

Для этого мы используем *градиентное состояние* (`gradirnt state`). Оно ассоциирует каждый скаляр d с аккумулятором, который представляет текущий градиент результата цепочки относительно d .

⁴⁹ Технически он представляет собой сумму всех градиентов каждого скаляра в результате относительно d , как мы заметили во фрейме 37, но здесь мы игнорируем это различие.

Градиентное состояние удобно представлять в виде хеш-таблицы.

Теперь определим ∇_{once} :

```
(define  $\nabla_{\text{once}}$ 
  ( $\lambda$  (y wrt)
    (let (( $\sigma$  ( $\nabla_{\sigma}$  y (hasheq))))
      (map* ( $\lambda$  (d)
              (hash-ref  $\sigma$  d 0.0))
            wrt))))
```

Здесь мы определяем градиентное состояние $\sigma^{\dagger}$, вызывая ∇_{σ} для y и пустого состояния, созданного с помощью (hasheq). Теперь σ содержит градиенты y относительно каждого скаляра в wrt.

[†] Идею хеш-таблиц предложил Ганс Лун (1886–1967). Здесь σ представляет собой первое вхождение в хеш-таблицу.

Определим ∇_{σ} :

```
(define  $\nabla_{\sigma}$ 
  ( $\lambda$  (y  $\sigma$ )
    (cond
      ((scalar? y)
       (let ((k ( $\kappa$  y)))
         (k y 1.0  $\sigma$ )))
      ((list? y)
       ( $\nabla_{\sigma}$ -list y  $\sigma$ ))
      ((vector? y)
       ( $\nabla_{\sigma}$ -vec y (sub1  $\dagger$  y  $\dagger$   $\sigma$ ))))))
```

50 Как определено во фрейме 6, градиент у относительно wrt должен иметь ту же структуру, что и wrt, но с градиентами вместо скаляров в листовых узлах.

Итак, мы используем map*, чтобы заменить все скаляры в wrt соответствующим градиентом. Мы вызываем map* для функции, которая ищет градиент скаляра в σ . Здесь мы используем значение по умолчанию 0.0, когда скаляр еще не присутствует в σ . Это позволяет нам создать градиент у именно той структуры, которая нам нужна.

51 Здесь мы рекурсивно обходим структуру дифобъекта у и накапливаем градиенты в σ .

Что касается списков, мы накапливаем градиенты от каждого члена списка, рекурсивно вызывая ∇_{σ} при обходе списка, используя вспомогательную функцию ∇_{σ} -list.

Что касается векторов, мы накапливаем градиенты от каждого элемента, рекурсивно вызывая ∇_{σ} при обходе вектора, используя опорную функцию ∇_{σ} -vec.

Это подводит нас к основному циклу, когда у является скаляром. Здесь мы начинаем проходить цепочку, которая произвела этот скаляр.	52 Сначала мы определяем звено k скаляра y (κy)
Звено k принимает три аргумента. Первый – это скаляр, по цепи которого мы хотим пройти. В данном случае это скаляр y.	53 Второй аргумент – это начальное значение мультипликативного аккумулятора, когда мы начинаем движение по цепочке.
Поскольку мы только начинаем движение по цепочке y, мы предоставляем 1.0, т. е. мультипликативную идентичность, в качестве начального значения для этого мультипликативного аккумулятора.	54 Третий аргумент – это градиентное состояние, которое мы будем обновлять значениями градиентов. Вызов функции возвращает градиентное состояние, содержащее градиенты, полученные проходом по цепочке.
Так мы обходим списки, по одному члену за раз:	55 Работая с вектором, мы проходим его, начиная с последнего элемента и отсчитывая индексы до 0-го элемента:

```
(define  $\nabla_{\sigma}$ -list
  ( $\lambda$  ( $y$   $\sigma$ )
    (cond
      ((null?  $y$ )  $\sigma$ )
      (else
       (let (( $\hat{\sigma}$  ( $\nabla_{\sigma}$   $y_0$   $\sigma$ )†))
         ( $\nabla_{\sigma}$ -list  $y_{1:}$   $\hat{\sigma}$ ))))))
```

```
(define  $\nabla_{\sigma}$ -vec
  ( $\lambda$  ( $y$   $i$   $\sigma$ )
    (let (( $\hat{\sigma}$  ( $\nabla_{\sigma}$   $y_i$   $\sigma$ )))
      (cond
        ((zero?  $i$ )  $\hat{\sigma}$ )
        (else ( $\nabla_{\sigma}$ -vec  $y$  (sub1  $i$ )  $\hat{\sigma}$ ))))))
```

[†] Это использование ∇_{σ} представляет собой первый случай взаимно рекурсивного вызова, являющийся частью Scheme, которую мы не использовали в основной части книги, как отмечено на стр. 349.

8. Звенья

Теперь перейдем к звеньям. Они представляют собой функции, которые помогают в обходе цепочек скаляра.

56 Начнем с простейшего случая, когда скаляр имеет звено *end-of-chain* для вещественных чисел и усеченных двойственных чисел.

Звено *end-of-chain* всегда вызывается в конце цепочки. К тому времени, когда мы закончим обход цепочки и достигнем *end-of-chain*, у нас будет градиент, связанный с этой цепочкой, в мультипликативном аккумуляторе, который мы всегда называем z .

57 Конец цепочки ассоциирован со скаляром d , который является аргументом цепочки. Наша задача в конце цепочки – запомнить градиент z для скаляра d в градиентном состоянии σ .

Учитывая это, определим *end-of-chain*:

```
(define end-of-chain
  (λ (d z σ)
    (let ((g (hash-ref σ d 0.0)))
      (hash-set σ d (+ z g)))))
```

58 В общем случае одна скалярная величина d из *wrt* может появиться в конце нескольких цепочек, которые дают единственный результат u . Это может произойти, например, когда аргумент функции используется более одного раза.

В этом случае каждое появление d в конце цепочки вносит свой собственный вклад в градиент u относительно d . Конечный градиент здесь представляет собой сумму всех этих вкладов.

59 В качестве примера допустим, что мы имеем функцию f

```
(λ (x)
  (+ x x))
```

Результатом вызова

$(f d)$

будет

$2 \times d$

60 Поскольку x используется дважды в теле f , это считается двумя вхождениями d .

Следовательно, градиент сложения относительно d будет равен	61	$1.0 + 1.0 = 2.0$
Это требование включено в <i>end-of-chain</i> . Сначала мы ищем текущий градиент g для d в σ , используя <i>hash-ref</i> со значением по умолчанию 0.0, если d отсутствует в σ .	62	Затем мы прибавляем мультипликативный аккумулятор z к текущему градиенту g и запоминаем эту сумму как градиент для d с помощью <i>hash-set</i> .
Рассмотрим звено примитивной операции. Сложение двух скаляров, $^{+0.0}$ из фрейма 188:40, определяется следующим образом. Он принимает два скалярных аргумента и возвращает двойственное число, где вещественная часть является суммой вещественных частей двух его аргументов.	63	Здесь $+$ и $*$ представляют обычные арифметические операции с вещественными числами: <pre>(define $^{+0.0}$ (λ (da db) (dual (+ (ρ da) (ρ db)) (λ (dz σ) (let ((σ̂ ((κ da) da (* 1.0 z) σ)) ((κ db) db (* 1.0 z) σ̂))))))</pre>
Поскольку $^{+0.0}$ является примитивной операцией, ее градиенты относительно аргументов da и db могут быть определены в виде формулы.		Давайте посмотрим, что произойдет при вызове этого звена.
Сначала мы вызываем звено $(κ da)$ для скаляра da мультипликативного аккумулятора $(* 1.0 z)$ который мы объясним ниже, и градиентного состояния σ чтобы получить новое градиентное состояние $\hat{\sigma}$.	64	Затем вызываем звено $(κ db)$ для скаляра db db мультипликативного аккумулятора $(* 1.0 z)$ который мы объясним ниже, и градиентного состояния $\hat{\sigma}$ получая в итоге результатирующее градиентное состояние, возвращаемое звеном.

Рассмотрим подробнее мультипликативные аккумуляторы. Когда мы идем по звеньям для da и db , мультипликативный аккумулятор является произведением текущего аккумулятора z и градиента примитива.

Итак, эта ссылка начинает обход цепочек для da и для db с мультипликативными аккумуляторами

$(* 1.0 z)$ для da

и

$(* 1.0 z)$ для db .

Нужно учесть, что

$(* 1.0 z)$

можно записать просто как z .

Мы используем аналогичный подход в случае exp^0 , которая представляет собой $e^{(\rho da)}$, где e – трансцендентная константа и da – скаляр. Здесь градиентом также является $e^{(\rho da)}$:

```
(define exp0
  (λ (da)
    (dual (exp (ρ da))
          (λ (d z σ)
            ((κ da) da (* (exp (ρ da)) z) σ))))
```

⁶⁵ Градиенты сложения относительно аргумента, по цепочке которого мы движемся, как было сказано во фрейме 10, оба равны 1.0.

⁶⁶ Кроме того, поскольку это звено не завершает цепочку, нам не нужно отслеживать какие-либо градиенты для d . Следовательно, мы игнорируем d .

```
67 (define +0,0
  (λ (da db)
    (dual (+ (ρ da) (ρ db))
          (λ (d z σ)
            (let ((ô ((κ da) da z σ)))
              ((κ db) db z ô))))))
```

⁶⁸ Аналогично определим операцию $*^{0,0}$ из фрейма 189:41, представляющую собой перемножение двух скаляров:

```
(define *0,0
  (λ (da db)
    (dual (* (ρ da) (ρ db))
          (λ (d z σ)
            (let ((ô ((κ da) da (* (ρ db) z) σ)))
              ((κ db) db (* (ρ da) z) ô))))))
```

Стандартные способы определения примитивов очень похожи и могут быть обобщены в шаблоны для определений всех наших примитивов.

Начнем с примитива с одним аргументом. Мы предлагаем ввести функцию *prim1*, которую можно использовать для определения, например, *exp*⁰.

69

```
define exp0
  (prim1 exp
    (λ (ra z)
      (* (exp ra) z))))
```

Здесь *exp* – это функция, которая получает один вещественный числовой аргумент и возвращает вещественную часть ответа, которая будет двойственной.

В этом определении *prim1* получает два аргумента-функции. Первый – это функция, определяющая вещественную часть двойственного числа. Поэтому мы называем ее *ρ*-функцией примитива.

70

Вторая функция определяет, как должно вести себя тело функции звена. Другими словами, она содержит формулу для градиента. Поэтому мы называем ее *∇*-функцией примитива.

Вторая функция получает два аргумента и отвечает за вычисление градиента. Первый аргумент *ra* является тем же вещественным аргументом, который передается при вычислении вещественной части. Второй – мультипликативный аккумулятор, который будет передан в звено.

71

Вызов *prim1* возвращает функцию, которая получает двойственный аргумент и возвращает соответствующий двойственный аргумент с правильно построенным звеном.

Вот определение *prim1*:

72

```
(define prim1
  (λ (ρ-fn ∇-fn)
    (λ (da)
      (let ((ra (ρ da)))
        (dual (ρ-fn ra)
          (λ (d z σ)
            (let ((ga (∇-fn ra z)))
              ((κ da) da ga σ))))))))
```

Два аргумента – *ρ-fn* (для *ρ*-функции) и *∇-fn* (для *∇*-функции) – используются для получения вещественной части и градиента соответственно. Функция *prim1* возвращает функцию, которая получает один двойственный аргумент *da* и возвращает двойственный результат.

Вещественная часть полученного двойственного результата получается вызовом функции $\rho\text{-fn}$ для вещественной части da .

⁷³ Звеньевая функция данного результата имеет тело, которое вызывает $\nabla\text{-fn}$ с аргументами в виде da и z и передает результат дальше.

Аналогично устроена prim2 . Самая большая разница, помимо дополнительного аргумента для $\rho\text{-fn}$ и $\nabla\text{-fn}$, заключается в том, что мы используем возможность возврата составного значения Scheme для получения двух градиентов от $\nabla\text{-fn}$, по одному для da и $db^\dagger$.

Затем они передаются следующему звену с помощью $\hat{\sigma}$ как во фрейме 63.

⁷⁴

```
(define prim2
  (λ (ρ-fn ∇-fn)
    (λ (da db)
      (let ((ra (ρ da))
            (rb (ρ db)))
        (dual (ρ-fn ra rb)
              (λ (d z σ)
                (let2 ((ga gb) (∇-fn ra rb z))
                  (let ((σ̂ ((κ da) da ga σ))
                    ((κ db) db gb σ̂))))))))))
```

[†] См. обозначения для значений `let` на стр. 349.

Переопределим $+^{0,0}$ и $*^{0,0}$ с $\text{prim2}^\dagger$:

```
define +0,0
  (prim2 +
    (λ (ra rb z)
      (z z))))
```

⁷⁵

```
(define *0,0
  (prim2 *
    (λ (ra rb z)
      ((* rb z) (* ra z)))))
```

[†] Каждая из функций $+^{0,0}$ и $*^{0,0}$ вызывает prim2 , вторым аргументом которой является функция, возвращающая два значения. См. обозначения на стр. 351.

Для определения операций сравнения мы используем функцию поддержки:

```
(define comparator
  (λ (f)
    (λ (da db)
      (f (ρ da) (ρ db))))))
```

76 Затем дадим определения:

```
(define <0,0 (comparator <))
(define >0,0 (comparator >))
(define ≤0,0 (comparator ≤))
(define ≥0,0 (comparator ≥))
(define =0,0 (comparator =))
```

9. Некоторые распространенные числовые примитивы

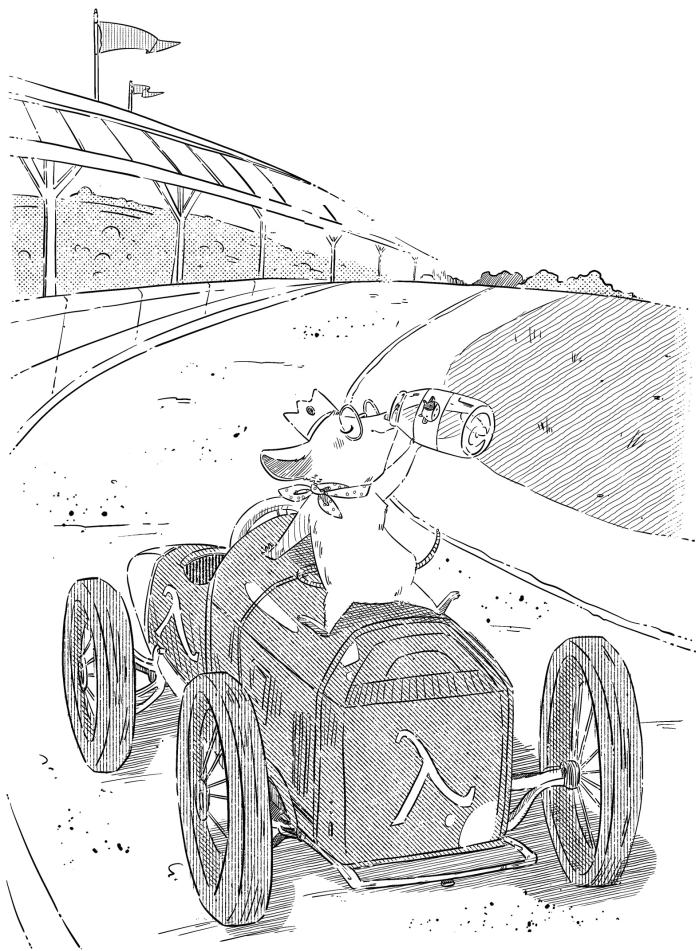
```
(define -0,0
  (prim2 -
    (λ (ra rb z)
      (z (- z))))))
(define ÷0,0
  (prim2 ÷
    (λ (ra rb z)
      (< (* (÷ 1 rb) z)
        (* (÷ (* -1 ra) (* rb rb)) z))))))
(define log0
  (prim1 log
    (λ (ra z)
      (* (÷ 1 ra) z))))
```

77

```
define expt0,0
  (prim2 expt
    (λ (ra rb z)
      (< (* z (* z (expt ra (- rb 1))))
        (* (* (expt ra rb) (log ra)) z))))))
(define sqrt0
  (prim1 (λ (ra)
    (expt ra  $\frac{1}{2}$ )))
    (λ (ra z)
      (* ( $\frac{1}{2}$  (expt ra  $-\frac{1}{2}$ )) z))))
```

Мы называем эту первую версию автоматического дифференцирования «Системой А». В приложении В мы покажем, как улучшить Систему А, переопределив некоторые функции.

ПРИЛОЖЕНИЕ В. Я ТАК ХОЧУ, ЧТОБЫ ГОНКА НЕ КОНЧАЛАСЬ¹



¹ Оригинальное название главы *I could have Raced All Day* отсылает нас к песне Алана Джея Лернера (1918–1986) *I Could Have Danced All Night* из мюзикла *My Fair Lady* (1956). Фамилия автора либретто (Alan Learner) перекликается с исходным названием книги (Little Learner).

В книгах из серии «The Little» все основные идеи упакованы в аккуратные маленькие коробочки. Автоматическое дифференцирование, описанное в приложении А (далее мы будем говорить Система А), вполне логично отражает все идеи, о которых говорилось в основной части книги. Увы, иногда упаковка в маленькие коробочки влечет за собой проблемы, особенно когда мы сталкиваемся с суровой реальностью.

Например, при использовании Системы А обучение сетей, таких как *morse-fcn* и *morse-residual*, может занимать непомерно много времени. По мере перехода к запуску и обучению все более крупных моделей возникает насущная необходимость избавиться от этих врожденных недостатков.

Иногда нам нравится представлять себе, как мог размышлять Огюстен Луи Коши в 1847 году: «Когда мои идеи в далеком будущем дождутся полного понимания, человечество будет потрясено тем, чего можно достичь с помощью открытых мной алгоритмов». Что ж, это далекое будущее уже наступило.

Функции, работающие с очень большими тензорами, могут быть намного более эффективными по времени выполнения, если использовать оборудование, которое выполняет их «параллельно». Эти так называемые *массово-параллельные машины* (*massively parallel machine*) наиболее эффективны, когда мы предоставляем им правильно организованные определения и структуры данных.

В этом приложении мы продолжаем придерживаться своего традиционного стиля «Little Book», чтобы полностью соответствовать этим требованиям. Мы переходим от простого подхода в Системе А к усовершенствованному, который столь же ловко справляется с запутанными реалиями. Мы продолжим использовать стиль комиксов и уже знакомые обозначения из приложения А.

1. Тензоры и их слабые места

Система А неявно предполагает, что тензоры ранга выше 0 представлены вложенными векторами. Уточним это предположение.

¹ Рассмотрим набор функций, который позволяет нам манипулировать тензорами ранга выше 0.

<pre>(define tref (λ (t i) (vector-ref t (ρ i)))) (define tlen (λ (t) (vector-length t)))</pre>	2 <pre>(define tensor vector) (define tensor? (λ (t) (cond ((scalar? t) #t) ((vector? t) #t) (else #f))))</pre>
Кроме того, у нас есть <i>ext1</i> и <i>ext2</i>, определения которых точно такие же, как во фреймах 181:16 и 185:35. Вместе эти функции образуют абстрактный интерфейс для наших тензоров.	3 В общем случае все скаляры в наших тензорах в Системе А двойственны. У каждого двойственного значения есть вещественная часть, представляющая собой значение скаляра, и ссылка на звено цепочки, содержащая необходимую информацию для извлечения градиента из этого скаляра.
Когда наши тензоры велики и наши сети тоже велики (со многими слоями), общее количество двойственных скаляров и их звеньев, с которыми нам приходится иметь дело, также становится большим.	4 Более того, многие из этих двойственных чисел имеют очень длинные цепочки, что делает обход этих цепочек довольно трудоемким. А их большое количество еще сильнее замедляет Систему А.

2. Уменьшение количества звеньев

Давайте сначала устраним это узкое место и найдем способ уменьшить количество двойственных скаляров и звеньев в системе. Ключевым наблюдением здесь является то, что почти каждый скаляр, созданный внутри Системы А, находится в тензоре.	5 Все подобные скаляры в этом тензоре создаются одинаковым образом, обычно посредством вызова расширенной функции.
---	--

Это означает, что их звенья идентичны по структуре. Например, вот как ведет себя $*^{2,1}$:

$$1. \left(*^{2,1} \begin{bmatrix} 2.0 & 3.0 & 1.1 \\ 1.6 & 2.3 & 4.1 \end{bmatrix} \right)_{(2 \ 3)} \\ \quad [1.0 \ 0.1 \ 3.2]) \\ 2. \left[\begin{bmatrix} 2.0 & \underline{0.3} & 3.5 \\ 1.6 & 0.23 & 13.12 \end{bmatrix} \right]_{(2 \ 3)}$$

Разница между звеньями заключается в конкретных значениях da и db . Например, подчеркнутый скаляр 0.3 будет иметь в своем звене значение $(\rho \ da)$, равное 0.3

Учитывая это, мы можем иметь представление, в котором звено ассоциировано со всем тензором результата, а эквивалентные значения a и b в этом звене будут тензорами всего аргумента.

6 Все скаляры в результирующем тензоре будут содержать звено, произведенное $*^{0,0}$, которое выглядит примерно так:

```
(λ (dz σ)
  (let ((σ((κ da) da (* (ρ db) z) σ)))
    ((κ db) db (* (ρ da) z) σ)))
```

7 Соответственно, значение $(\rho \ db)$ будет 0.1

Другие двойственные числа в тензоре будут иметь соответствующие скаляры для da и db в своих звеньях.

8 Таким образом, вместо того чтобы следовать по отдельным звеньям в скалярах, мы могли бы использовать одно звено для всего тензора.

3. Доработка Системы А

Для этого нужно расширить наше определение двойственных чисел, включив в их вещественную часть тензоры, а не только скаляры. Мы называем двойственные значения, не являющиеся скалярами, двойственными тензорами.

9 Начнем с нового определения *scalar*? – когда у нас есть двойственное число, мы проверяем, является ли его вещественная часть числом.

```
(define scalar?
  (λ (d)
    (cond
      ((number? d) #t)
      ((dual? d) (number? (ρ d)))
      (else #f))))
```

Мы называем вещественную часть двойственного тензора просто тензором, а иногда и недвойственным тензором, чтобы подчеркнуть, что он не является двойственным.

Наше определение двойственных чисел в Системе А останется прежним, но мы внесем изменения в некоторые другие определения.

Определим функцию *dual-like?*, которая является предикатом, содержащим проверку векторов и допускающим теперь вызовы для ρ и κ .

```
(define dual-like?
  (λ (d)
    (cond
      ((scalar? d) #t)
      ((dual? d) #t)
      (else (vector? d)))))
```

Далее модифицируем ∇_σ , чтобы реализовать аналогичную возможность нахождения константных тензоров в базовом случае с помощью *dual-like?*. Мы больше не спускаемся рекурсивно в вектор. Мы просто ожидаем, что векторы (т. е. как тензоры) будут нести свои собственные звенья, и начинаем проход по ним.

¹⁰ Клауза *else (vector? d)* допускает возможность работы с константными векторами, подобно тому, как мы обрабатываем константные скаляры во фрейме 360:33.

¹¹

```
(define ∇σ
  (λ (y σ)
    (cond
      ((dual-like? y)
        ((κ y) y (one-like (ρ y)) σ))
      ((list? y) (∇σ-list y σ))
      (else (error)))))
```

В отличие от предыдущего определения $\nabla\sigma$, здесь u может быть двойственным тензором. Поэтому при вызове функции звена мы передаем в аккумулятор тензор аналогичной формы

$(\rho\ u)$

но состоящий из значений 1.0.

Другое большое отличие, с которым мы сталкиваемся, когда допускаем наличие тензоров внутри двойственных значений, состоит в том, что градиентное состояние σ больше не отображает двойственные значения только в скаляры.

¹² Мы генерируем этот тензор с помощью функции *one-link*, определение которой вы скоро увидите.

¹³ Назначение σ состоит в том, чтобы сопоставить двойственное значение с его градиентом. Когда наши двойственные значения содержат тензоры, градиенты, соответствующие этим двойственным значениям, также будут тензорами той же формы.

Это означает, что нам требуется немного модифицированное определение *end-of-chain*.

```
(define end-of-chain
  (λ (d z σ)
    (let ((g (hash-ref σ d 0.0)))
      (hash-set σ d (+ρ g z)))))
```

¹⁴ В этом определении $+^{\rho}$ – это расширенная функция, которая просто складывает вместе два (недвойственных) тензора, подобно расширенной версии $+$, описанной во фрейме 45:11, но с некоторыми отличиями. Вскоре мы разберем это определение.

Аналогичным образом обновим *map**.

```
(define map*
  (λ (f y)
    (cond
      ((dual-like? y) (f y))
      ((list? y)
       (map (λ (yi)
              (map* f yi))
            y))
      (else y))))
```

¹⁵ Это была относительно несложная часть.

Теперь мы рассмотрим последствия для расширенных функций от обертывания тензоров внутрь двойственных значений.

4. Расширение функций на двойственные тензоры

Наши определения *ext1* и *ext2* в интерлюдии V предполагают тензоры, в которых каждый вложенный скаляр является двойственным. Именно это позволяет нам легко находить градиенты расширенных функций.

¹⁶ Но теперь, поскольку эти тензоры обернуты в двойственную форму, и *ext1*, и *ext2* обновляются для создания двойственного тензора, где звено имеет четко определенный способ прохождения по цепочке двойственных тензоров.

Давайте сначала разберемся с этой задачей. Начнем с переименования существующего определения *ext1* в *ext1^p* и *ext2* в *ext2^p*. Верхний индекс *p* напоминает нам, что это расширение будет использоваться для создания вещественной части двойственного тензора.

¹⁷ Из этого следует, что *ext1^p* и *ext2^p* являются реальными путями *ext1* и *ext2* соответственно, где определена вещественная часть двойственного числа. Далее приведены переопределения.

```
(define ext1p
  (λ (f n)
    (λ (t)
      (cond
        ((of-rank? n t) (f t))
        (else (tmap (ext1p f n) t))))))
```

¹⁸

```
(define ext2p
  (λ (f n m)
    (λ (t u)
      (cond
        ((of-ranks? n t m u) (f t u))
        (else
         (desc (ext2p f n m)
              n t m u))))))
```

Остальные определения вспомогательных функций остаются такими же, как в интерлюдии V.

¹⁹ В определениях этих функций *t* и *u* являются недвойственными тензорами, т. е. они являются вещественной частью двойственных тензоров, и они также производят недвойственные тензоры, т. е. они производят только вещественную часть результирующего двойственного тензора.

Соединим новые знания и определим функцию *one-like* с помощью *ext1^p*.

```
(define one-like
  (ext1p (λ (s) 1.0) 0))
```

20 Определим *ext2^p*, используя *ext2^p*.

```
(define +p
  (ext2p + 0 0))
```

5. Модификация расширенных функций

Теперь посмотрим, что происходит внутри *ext1* и *ext2*. Начнем с *ext1*.

21 Функция *ext1* ожидает, как и раньше, расширяемую функцию *f* и базовый ранг *n*. В данном случае мы рассчитываем, что *f* получает и возвращает двойственное значение.

Затем *ext1* возвращает функцию, которая принимает один двойственный тензорный аргумент *da* (ранга *n* или выше) и возвращает двойственное число, вещественная часть которого определяется с помощью *ext1^p*.

22 Так выглядит каркасный код

```
(define ext1
  (λ (fn)
    (λ (da)
      (dual
        ((ext1p R) n) (ρ da))
        (λ (dz σ)
          (let ((ga ( N da z)))
            ((κ da) da ga σ)))))))
```

Звено этого возвращенного двойственного числа отвечает за определение расширенного градиента *f*. Мы делаем это с помощью выражения *N*, которое возвращает функцию, определяющую градиент *ga* с помощью *da* и мультипликативного аккумулятора *z*. Продолжая идти по цепочке, пройдем по *ga* до звена *da*.

Немного позже мы поясним, что такое расширенный градиент функции.

23 Здесь стоит обратить внимание на то, что эта функция структурно идентична для вызовов *prim1* по *R* и *N*! Теперь можно определить *ext1*.

```
(define ext1
  (λ (fn)
    (prim1 (ext1p R) n)
    N)))
```

6. В игру вступают примитивы

Теперь наша задача состоит в том, чтобы определить R и N . При использовании Системы А аргументом f функции $ext1$ может быть любая функция, которая принимает в качестве аргумента один тензор ранга n и в результате возвращает один тензор любого ранга.

²⁴ Но в двойственно-тензорной системе мы вводим дополнительное ограничение. Оно состоит в том, что мы можем расширять только примитивы, т. е. мы можем расширять функции, созданные с использованием лишь $prim1$ или $prim2$.

Это не слишком существенное ограничение, так как большинство наших расширений до сих пор затрагивали только примитивные функции.

²⁵ Кроме того, поскольку наше определение $ext1$ является вызовом $prim1$, функции, расширенные с помощью $ext1$, автоматически подчиняются этому ограничению.

Примитивы можно рассматривать как состоящие из двух подфункций: ρ -функции, которая является первым аргументом $prim1$, и ∇ -функции, которая является вторым аргументом[†].

²⁶ В случае расширенных функций каждая из этих подфункций расширяется независимо.

[†] Это соображение применимо и к двум аргументам $prim2$.

Расширение ρ -функции вызывается для вещественной части двойственного тензора. Мы видим это вскользь, когда вызываем $ext1^p$ для получения ρ -функции $ext1$.

²⁷ Расширенный градиент из фрейма 23 определяется путем расширения ∇ -функции и применения ее к da и z .

Мы оговариваем существование двух функций-аксессоров (accessor function, функция доступа) с именами ρ -function и ∇ -function, которые извлекают ρ - и ∇ -функции соответственно из любого заданного примитива. Мы определим их в ближайшее время.

Теперь нам понятно, что R выглядит так:

$(\rho\text{-function } f)$

28 Можно подставить это выражение в каркасный код $ext1$:

```
(define ext1
  (λ (f n)
    (prim1 (ext1ρ (ρ-function f) n)
      N)))
```

Точно так же, как у нас есть $ext1^ρ$ для расширений ρ -функций, мы предполагаем наличие функции $ext1^{\nabla}$ для расширений ∇ -функций.

29 Следовательно, N будет выглядеть как-то так:

$(ext1^{\nabla}$ G $n)$

Поскольку $ext1^{\nabla}$ расширяет ∇ -функцию f , вместо

G

мы подставляем

$(\nabla\text{-function } f)$

30 Теперь можно записать полную версию $ext1$:

```
(define ext1
  (λ (f n)
    (prim1 (ext1ρ (ρ-function f) n)
      (ext1∇ (∇-function f) n))))
```

7. Добавление вызывающих функций к примитивам

Теперь займемся работой над $ext1$.

Нам нужно определить аксессоры

ρ -function и ∇ -function

и расширитель

$ext1^{\nabla}$

31 Начнем с ρ -function и ∇ -function. Аргументы этих двух функций всегда представляют собой примитивную функцию, сгенерированную с использованием $prim1$.

Поскольку примитив, созданный с использованием *prim1*, представляет собой функцию, над ним возможна только операция вызова (invocation).

Так выглядит *ρ-function*:

```
(define ρ-function
  (λ (prim)
    (prim ρ-function)))
```

32 А так – *∇-function*:

```
(define ∇-function
  (λ (prim)
    (prim ∇-function)))
```

Но подождите, примитивы не предназначены для приема функций в качестве аргументов.

Они ожидают только двойственные тензоры.

33 Что ж, давайте исправим это, переопределив *prim1*, чтобы примитивы также могли принимать функции *ρ-function* или *∇-function* в качестве аргумента.

Так выглядит обновленное определение *prim1*:

```
(define prim1
  (λ (ρ-fn ∇-fn)
    (λ (daf)
      (cond
        ((eq? daf ρ-function) ρ-fn)
        ((eq? daf ∇-function) ∇-fn)
        (else
         (prim1-dual ρ-fn ∇-fn daf))))))
```

34 Функция *prim1*, вызванная с аргументами *ρ-* и *∇-function*, возвращает функцию, которая тестирует только этот соответствующий аргумент[†].

[†] Эта функция предназначена для создания сущности, которая может рассматриваться как экземпляр с возможностью функционального вызова и может быть написана на других языках как запись, которую можно вызвать, или объект, который наследуется от класса функции.

Если *daf* является функцией *ρ-function*
 будет возвращен результат *ρ-fn*

35 Если *daf* является функцией *∇-function*
 будет возвращен результат *∇-fn*

Если аргумент не является ни тем, ни другим, он выполняет то, что делает исходная функция *prim1*, а именно создает двойную функцию, которая вызывает требуемую подфункцию по мере необходимости. Мы запишем это как вызов функции *prim1-dual*.

³⁶ Вот определение функции *prim1-dual*:

```
(define prim1-dual
  (λ (ρ-fn ∇-fn da)
    (let ((ra (ρ da)))
      (dual (ρ-fn ra)
        (λ (d z σ)
          (let ((ga (∇-fn ra z)))
            ((κ da) da ga σ)))))))
```

Это определение становится идентичным *prim1* из фрейма 371:72, когда функция получает как аргумент *ρ-fn*, *∇-fn* и *da*.

Мы даем вещественной части *da* имя *ra* и используем его как в вещественной части двойственного числа, так и в его звене.

Теперь определим *ext1[∇]*:

```
(define ext1∇
  (λ (∇-fn n)
    (λ (t z)
      (cond
        ((of-rank? n t) (∇-fn t z))
        (else
         (tmap (ext1∇ ∇-fn n) t z))))))
```

³⁷ Эта функция получает два аргумента: *∇-fn*, представляющий собой *∇*-функцию примитива, и *n* – требуемый базовый ранг *∇-fn*.

Она возвращает функцию, которая принимает недвойственный тензор *t* и *z*, и возвращает тензор, представляющий градиент расширенной версии *f*.

³⁸ Это определение похоже на *ext1^ρ*, но главное отличие здесь в том, что *∇-fn* ожидает, что два аргумента образуют градиент: элемент *t* с заданным индексом и элемент из *z* с тем же индексом.

Подобно $ext1^p$, если ранг t соответствует его базовому рангу n , мы вызываем

$\nabla\text{-fn}$ по t и z

Этот вызов возвращает значение градиента f от t .

Определение $ext2$ аналогично таковому для $ext1$:

```
(define ext2
  (λ (f n m)
    (prim2 (ext2p (ρ-function f) n m)
      (ext2v (∇-function f) n m))))
```

Перед тем как разработать следующее определение, необходимо ввести понятие функции с переменным количеством аргументов (variable-arity function)[†]. Такая функция может быть вызвана с произвольным количеством аргументов, которые собраны в список, передаваемый в функцию.

Ниже приводится пример такой функции:

```
(define bizarre
  (λ xs
    |xs|))
```

У xs после λ нет круглых скобок.

³⁹ Рекурсивный случай аналогичен $ext1^p$, где мы вызываем map , используя

$(ext1^v \nabla\text{-fn } n)$

для каждого элемента t и z , и собираем их результаты в новый вектор градиента.

⁴⁰ Но чтобы завершить это определение, нам нужно определить $prim2$ и $ext2^v$.

⁴¹ Здесь мы вызываем *bizzare* с 3 и 5 аргументами:

```
1. | (bizarre 17 45 81)
2. | 3
```

```
1. | (bizarre 83 22 16 41 94)
2. | 5
```

Мы можем начать с *prim2*:

```
(define prim2
  (λ (ρ-fn ∇-fn)
    (λ ds
      (let ((daf ds0))
        (cond
          ((eq? daf ρ-function) ρ-fn)
          ((eq? daf ∇-function) ∇-fn)
          (else
           (prim2-dual ρ-fn ∇-fn
            daf dsi)))))))
```

⁴² И с функции поддержки *prim2-dual*:

```
(define prim2-dual
  (λ (ρ-fn ∇-fn da db)
    (let ((ra (ρ da))
          (rb (ρ db)))
      (dual (ρ-fn ra rb)
        (λ (d z σ)
          (let2 ((ga gb) (∇-fn ra rb z))
            (let ((σ((κ da) da ga σ)))
              ((κ db) db gb σ))))))))
```

Из *prim2* возвращается функция с переменным числом аргументов, потому что она получила бы только один аргумент, если бы была вызвана из *ρ-function* или *∇-function*, но получила бы два аргумента, если бы была вызвана как примитив.

⁴³ Функция *prim2-dual* также ведет себя идентично *prim2* из фрейма 372:74, если у нее есть 4 аргумента.

8. Расширение градиентов бинарных примитивов

Это подводит нас к определению *ext2[∇]*. Исходя из сходства *ext1[∇]* с *ext1^ρ*, можно предположить, что *ext2[∇]* должна быть похожа на *ext2^ρ*, но нужно будет учитывать дополнительный аргумент *z*.

⁴⁴ Так выглядит определение *ext2[∇]*:

```
(define ext2∇
  (λ (∇-fn n m)
    (λ (t u z)
      (cond
        ((of-ranks? n t m u) (∇-fn t u z))
        (else
         (desc∇ (ext2∇ ∇-fn n m)
          n t m u z))))))
```


Когда ранг t соответствует n , а ранг u соответствует m , мы создаем пару градиентов, вызывая

$(\nabla\text{-fn } t \ u \ z)$

В противном случае мы спускаемся в один или оба тензора. Мы делаем это, вызывая desc^∇ , которая структурно очень похожа на desc из фрейма 186:37. Главное отличие здесь в том, что desc^∇ дополнительно принимает мультипликативный аккумулятор z .

Здесь вы можете видеть три вспомогательные функции, которые не встречали до сих пор:

$\text{desc}\text{-}t^\nabla$

$\text{desc}\text{-}u^\nabla$

и

tmap2

Начнем с функции tmap2 .

Здесь g создает два градиента при каждом вызове. Эти градиенты собираются в два отдельных вектора для создания двух тензоров градиента, по одному относительно каждого из t и u .

45 Так выглядит определение desc^∇ :

```
(define desc∇
  (λ (g n t m u z)
    (cond
      ((of-rank? n t) (desc-u∇ g t u z))
      ((of-rank? m u) (desc-t∇ g t u z))
      ((= † t † † u †) (tmap2 g t u z))
      ((rank> t u) (desc-t∇ g t u z))
      (else (desc-u∇ g t u z)))))
```

46 Она применяет g к t , u и z .

```
(define tmap2
  (λ (g t u z)
    (build-gt-gu † t †
      (λ (i)
        (g ti ui zi))))
```

47 Эту работу выполняет функция build-gt-gu , которую вызывают с необходимым размером векторов градиента (здесь длина t) и функцией инициализации, которая создает два значения для каждого индекса i .

Определим *build-gt-gu*.

48

```
(define build-gt-gu
  (λ (tn init)
    (let ((gt (make-vector tn))
          (gu (make-vector tn)))
      (fill-gt-gu gt gu init (sub1 tn))))))
```

Здесь мы сначала создаем два вектора *gt* и *gu* длиной *tn*, где каждый элемент имеет некоторое значение по умолчанию (обычно 0).

Затем *fill-gt-gu* вызывает функцию *init* с индексом *i*, чтобы получить значение для $gt|_i$ и $gu|_i$, и заменяет там значения по умолчанию. Наконец, она возвращает *gt* и *gu*.

```
(define fill-gt-gu
  (λ (gt gu init i)
    (let2 ((gti gui) (init i))
      (vector-set! gt i gti)†
      (vector-set! gu i gui)
      (cond
        ((zero? i) (gt gu))
        (else
         (fill-gt-gu gt gu g (sub1 i)))))))
```

[†] Мы используем *vector-set!* с побочным эффектом, и тело *let²* оценивается последовательно. Некоторых это может шокировать, но здесь это оправдано как однократная инициализация ранее не инициализированных векторов. Нам достаточно, что *build-gt-gu* представляет собой абстракцию, свободную от побочных эффектов, даже если ее реализация может быть таковой. Версия без побочных эффектов может быть определена с помощью *list*→*vector* за счет большего выделения памяти.

Теперь определим *desc-t^v*:

49

```
(define desc-tv
  (λ (g t u z)
    (build-gt-acc-gu † t †
      (λ (i)
        (g t|i u z|i))))))
```

При спуске в *t*, но не в *u*, мы по-разному обрабатываем градиенты, возвращаемые *g*. Мы делаем это, вызывая опорную функцию *build-gt-acc-gu*, которая, опять же, вызывается с длиной вектора, который нам нужно построить, и функцией инициализации, которая создает два значения при заданном индексе *i*.

Весь *u* используется с каждым элементом *t*, когда мы спускаемся в *t*, но не в *u*. Это означает, что *u* повторяется для каждого элемента *t*.

50

Из-за этого многократного использования мы суммируем *gus*, полученный для каждого *i*, аналогично фрейму 368:60.

Другими словами, мы накапливаем *gus*. Вот почему наша опорная функция здесь называется *build-gt-acc-gu*.

51 Так выглядит ее определение:

```
(define build-gt-acc-gu
  (λ (tn init)
    (let ((gt (make-vector tn))
          (gu 0.0))
      (fill-gt-acc-gu gt
        init (sub1 tn) gu))))
```

Она принимает два аргумента: длину создаваемого вектора *tn* и функцию инициализации *init*. Затем она создает вектор *gt* длины *tn*, где каждый элемент, опять же, имеет некоторое значение по умолчанию.

52 Чтобы организовать накопление, мы начинаем с *gu*, инициализированного значением 0.0. Затем мы вызываем *fill-gt-acc-gu*, которая многократно вызывает *init* для каждого индекса *i* и присваивает значение каждому элементу *gt* при накоплении *gu*.

Определение *fill-gt-acc-gu* дано ниже:

53 Эта функция по своей структуре идентична *fill-gt-gu*, но мы аккумулируем *gu* с помощью $+^p$ вместо записи значений в вектор.

```
(define fill-gt-acc-gu
  (λ (gt init i gu)
    (let2 ((gti gui) (init i))
      (vector-set! gt i gti)†
      (let ((gū (+p gu gui)))
        (cond
          ((zero? i) (gt gū))
          (else
           (fill-gt-acc-gu gt
             init (sub1 i) gū)))))))
```

[†] Примечание во фрейме 48 справедливо и здесь.

Оставшаяся функция *desc-u*^v очень похожа на *desc-t*^v, но опорная функция вместо *gu* аккумулирует *gt*:

```
(define desc-uv
  (λ (g t u z)
    (build-gu-acc-gt † u †
      (λ (i)
        (g t ui zi))))))
(define build-gu-acc-gt
  (λ (n init)
    (let ((gu (make-vector n))
          (gt 0.0))
      (fill-gu-acc-gt gu
        init (sub1 n) gt))))
```

54 Вот определение функции *fill-gu-acc-gt*:

```
(define fill-gu-acc-gt
  (λ (gu init i gt)
    (let2 ((gti gui) (init i))
      (vector-set! gu i gui)†
      (let ((gt̂ (++ gt gti)))
        (cond
          ((zero? i) (gt̂ gu))
          (else
           (fill-gu-acc-gt gu
             init (sub1 i) gt̂)))))))
```

† ... И здесь.

С этими изменениями наши определения скалярных примитивов и их расширений остаются идентичными тем, которые используются в основной части книги и в приложении А.

55 Нескалярные примитивы или примитивы, принимающие тензоры ранга выше 0, например *sum*¹ и *argmax*¹, переопределяются с учетом ограничения во фрейме 24.

9. Нескалярные примитивы

Начнем с переопределения *sum*¹ как *ρ*-функции *sum*^{1ρ}:

```
(define sum1ρ
  (λ (t)
    (summedρ t (sub1 † t †) 0.0)))
```

56 Далее определим *summed*^ρ:

```
(define summedρ
  (λ (t i a)
    (let ((â (+ρ a ti)))
      (cond
        ((zero? i) â)
        (else
         (summedρ t (sub1 i) â))))))
```

Вот соответствующая ∇ -функция $sum^{1\nabla}$:

```
(define sum1∇
  (λ (t z)
    (tmap (λ (t) z) t)))
```

57 Эта функция является предопределенной формулой для градиента суммы скаляров внутри `tensor1`.

Теперь мы можем определить $sum^{1\dagger}$:

```
(define sum1†
  (prim1 sum1ρ sum1∇))
```

[†] Аналогично определяются и другие векторные операции, причем их предопределенные формулы для ∇ -функций определяются индивидуально. Мы отсылаем читателя к сайту www.thelittlelearner.com.

58 и расширить ее как обычно, поскольку сейчас это примитив. Так мы получаем `sum` и `sum-cols`:

```
(define sum
  (ext1 sum1 1))

(define sum-cols
  (ext1 sum1 2))
```

Эти новые определения для `ext1` и `ext2` полностью способны обрабатывать тензоры внутри двойственных чисел, чтобы на много порядков улучшить узкие места для таких программ, как *morse*.

59 Теперь мы рассмотрим другую оптимизацию, которая еще больше повышает производительность наших моделей.

10. Плоские тензоры

Расположение тензоров в памяти является ключевым фактором, от которого зависит, насколько быстро скаляры в этих тензорах могут быть извлечены для работы с ними.

60 В большинстве случаев аппаратное обеспечение лучше работает с тензорами, которые расположены в памяти «непрерывно», а это означает, что все скаляры в тензоре размещаются в смежно расположенных ячейках памяти.

В большинстве реальных задач глубокого обучения приходится иметь дело с очень большими тензорами.

61 В таких случаях имеет смысл позаботиться о непрерывном представлении тензоров в памяти, в том числе для массивно-параллельного оборудования.

Тензоры в виде векторов, с которыми мы здесь работали, не обладают свойством непрерывного расположения в памяти.

62 Поскольку мы используем вложенное представление, разные части одного и того же тензора (при ранге выше 1) могут оказаться в разных несмежных местах в памяти.

Теперь мы рассмотрим другое представление тензоров, которое избавляется от вложенности и использует один полностью непрерывный участок памяти для представления всего тензора.

63 Мы называем это плоским тензорным представлением. Мы дадим здесь лишь краткий обзор и отсылаем читателя к определениям, которые можно найти на

www.thelittlelearner.com.

Гибридные представления, сочетающие плоские и вложенные тензорные представления, конечно, тоже возможны и иногда имеют смысл на определенных типах массивно-параллельного оборудования. Однако здесь мы ограничиваемся полностью плоскими тензорными представлениями.

Плоский тензор состоит из трех определяющих частей. Первая – это его форма, представляющая собой список, идентичный форме любого тензора, который вы встречали в основной части книги.

64 В отличие от естественно рекурсивного определения формы во фрейме 37:37, которое опирается на вложенность тензора, здесь мы предоставляем форму тензора при его построении.

Вторая часть – хранилище, представляющее собой непрерывный участок памяти для всех элементов тензора.

⁶⁵ Все скаляры в тензоре (которые теперь являются просто вещественными числами) размещаются в этом хранилище непрерывно[†].

[†] Аналогично размещению массивов в таких языках, как Fortran и C.

Третья часть – смещение, представляющее собой натуральное число, указывающее, со скольких позиций в хранилище начинается этот тензор.

⁶⁶ Это необходимо, чтобы иметь возможность выбирать субтензоры из заданного плоского тензора без необходимости копировать элементы исходного хранилища в новое хранилище.

В качестве примера возьмем следующий *tensor*³:

$$\left[\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{bmatrix} \right]_{(2\ 3\ 4)}$$

⁶⁷ Во вложенном представлении данный тензор устроен так:

```
(vector
  (vector
    (vector 1 2 3 4)
    (vector 5 6 7 8)
    (vector 9 10 11 12))
  (vector
    (vector 13 14 15 16)
    (vector 17 18 19 20)
    (vector 21 22 23 24)))
```

Плоское представление выглядит так:

```
(flat (list 2 3 4)
  (store 1 2 ... 24)
  0)
```

⁶⁸ Здесь функция *store* выделяет непрерывный сегмент памяти и заполняет его предоставленными значениями.

Дадим этому хранилищу временное имя:

```
(define t-store
  (store 1 2 ... 24))
```

⁶⁹ А теперь определим *t* как *tensor*³ из фрейма 68, используя *t-store*:

```
(define t
  (flat (list 2 3 4)
    t-store
    0))
```

Как нам представить различные субтензоры данного тензора? Например, мы хотим выделить субтензор

$$t|_1$$

Это должен быть тензор

$$\begin{bmatrix} 13 & 14 & 15 & 16 \\ 13 & 14 & 15 & 16 \\ 21 & 22 & 23 & 24 \end{bmatrix}_{(3\ 4)}$$

70 В случае плоского тензорного представления мы можем повторно использовать *t-store*:

```
(flat (list 3 4)
  t-store
  12)
```

Аргумент смещения 12 указывает, что этот тензор начинается с 12-й позиции в *t-store*. Мы называем часть хранилища, занятую скалярами в тензоре, его экстендом (*extent*).

В данном случае экстенд тензора начинается с 12-й позиции и доходит до 23-й (предполагая, конечно, что нумерация позиций начинается с 0).

Функция *shape* теперь просто возвращает форму, связанную с плоским тензором.

71 Вспомним определение ранга из фрейма 41:42:

```
(define rank
  (λ (t)
    |(shape t)|))
```

Операции расширения *ext1* и *ext2* над плоскими представлениями также становятся проще, потому что не требуется спуск в субтензоры.

72 Например, рассмотрим скалярное расширение, такое как *log*. Теперь функция *ext1^ρ* выделяет новое хранилище и заполняет его, проходя экстенд тензора и вызывая *ρ*-функцию *log⁰* для соответствующего входного элемента.

Мы называем эту операцию циклическим обходом (looping). Количество элементов, обрабатываемых на каждом шаге, называется шагом обхода. В случае скалярных операций обход происходит с шагом в один элемент. Этот обход используется в основном для заполнения правильными значениями предварительно выделенных тензорных хранилищ.

Именно этот упрощенный обход дает нам преимущества в скорости при выполнении на массивно-параллельных машинах.

Для $ext2^o$ и $ext2^v$ формы тензоров аргументов определяют шаблон циклического обхода и шаги, необходимые для каждого аргумента, а также форму заполняемого выходного тензора.

⁷³ Для расширенных функций, таких как sum , $ext1^o$, придется вызывать функцию sum^1 с шагом, равным длине тензора $tensor^1$, который необходимо суммировать.

⁷⁴ Нам, конечно, нужно иметь соответствующую $ext1^v$, которая является зеркальной операцией $ext1$, но вызывает ∇ -функцию.

⁷⁵ Например, когда один из тензоров аргументов (скажем, t) повторяется, потому что он соответствует своему базовому рангу, а другой тензор (скажем, u) нет, то цикл продолжает обход u , но начинается снова для t каждый раз, когда достигнут конец t .

11. Функции формы

Поскольку расширения в плоском тензорном представлении должны предварительно выделять память, возникает необходимость в том, чтобы примитивы могли сообщать о формах, которые потребуются для их выходных данных.

⁷⁶ Например, sum^1 производит только один скаляр каждый раз, когда он вызывается, но требует $tensor^1$ для создания этого скаляра. Точно так же $+^{0,0}$ производит один скаляр для одного скаляра в каждом из своих входов, но один из входов может использоваться повторно.

Для правильного предварительного распределения памяти необходимо, чтобы примитивы также имели функцию формы (*shape function*). Функция формы для унарных операций принимает входную форму (его базового ранга) и создает выходную форму.

77 Например, функция формы *sum*¹ имеет вид

```
(define shape-sum1
  (λ (st)
    (list)))
```

Это говорит о том, что форма вывода *sum*¹ для любой заданной входной формы ранга 1 представляет собой пустой список, то есть вывод является скаляром.

Аналогично, функция формы для всех скалярных примитивов двух аргументов будет такой:

```
(define shape0,0
  (λ (st su)
    (list)))
```

78 Функция формы для **^{2,1}* такова:

```
(define shape*2,1
  (λ (st su)
    st))
```

Из этого определения следует, что выходные данные, произведенные примитивом **^{2,1}*, имеют ту же форму, что и его аргумент ранга 2 *st*.

Теперь примитивы определяются их функциями формы. Например:

```
(define sum1
  (prim1 sum1ρ sum1∇ shape-sum1))
(define +0,0
  (prim2 +
    (λ (ra rb z)
      (z z))
    shape0,0))
```

79 И *prim1*, и *prim2* соответственно потребуют проверки аргумента *shape-fn*, определенной таким образом:

```
(define shape-fn
  (λ (prim)
    (prim shape-fn)))
```

Поскольку *ext1* и *ext2* также определены как примитивы, нам необходимо предоставить для них функции формы. Однако они могут использовать функции формы своих примитивов и автоматически выводить из них функцию формы для расширения.

⁸⁰ Оба определения функций формы для *ext1* и *ext2* отражают структуру *ext1^o* и *ext2^o*, но вместо создания тензоров они возвращают только формы.

12. И наконец...

Мы намеренно абстрагировались от того, как представлены хранилища. Простой способ представить их в Scheme – использовать функцию *vector*:

```
(define store  
  vector)
```

⁸¹ Некоторые языки допускают наличие собственных библиотек, обычно называемых интерфейсами внешних функций. Они позволяют создавать хранилища в собственных представлениях операционной системы, которые гораздо более дружелюбны к аппаратному обеспечению.

Многие машины с поддержкой массового параллелизма предоставляют собственные библиотеки, которые можно использовать для повышения эффективности параллельного выполнения.

⁸² Этот подход к использованию массивно-параллельных машин также требует тщательной реконструкции *ext1*, *ext2* и всех примитивов, чтобы воспользоваться преимуществами параллелизма, но мы оставляем это вам в качестве домашнего задания!

На этом мы остановимся.

Теперь, когда вы познакомились со всеми персонажами, не пропуская глав, пора заканчивать наше маленькое путешествие. Мы прощаемся с вами словами из песни:

83 *You will see light in the darkness*
You will make some sense of this
And when you've made your little
journey
You will find the fun you've missed[†].

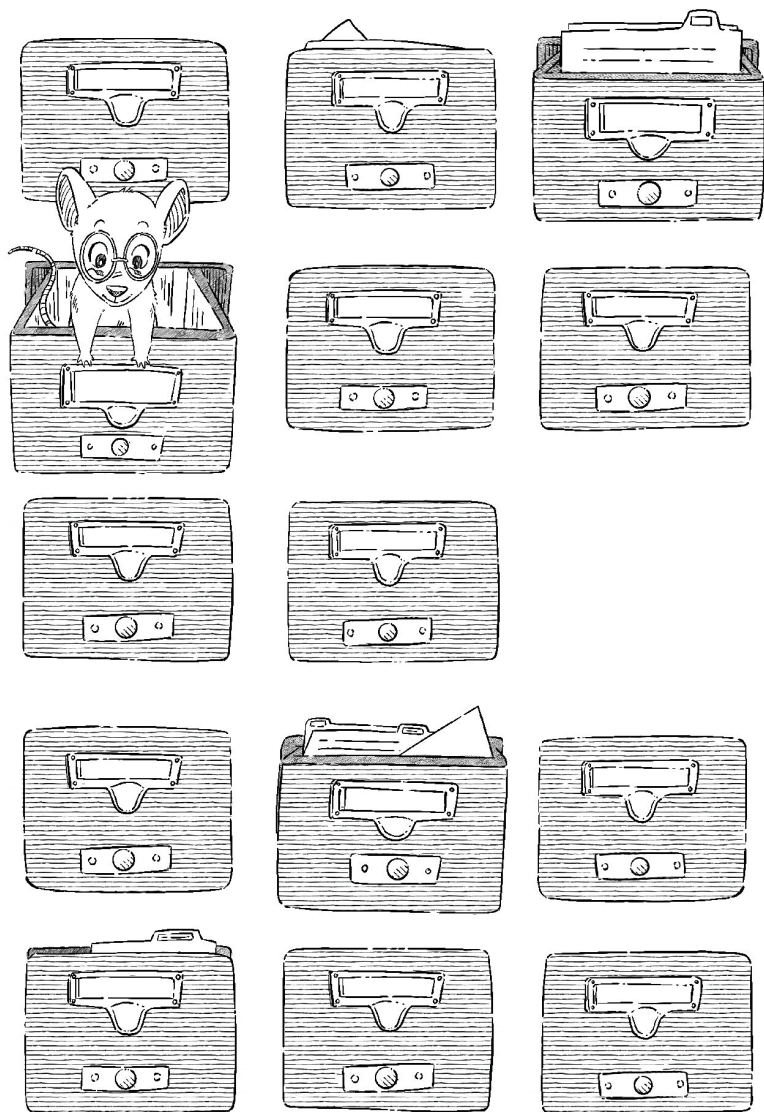
[†] Ты увидишь свет во тьме,
Ты узришь в нем свой смысл,
И, лишь завершив паломничество,
Ты найдешь радость, которой был лишен.

Припев из написанной Стингом песни *Secret Journey* в исполнении группы *The Police*. В песне припев заканчивается словами «...*You will find this love you miss*» – «... ты найдешь любовь, в которой нуждался». <https://www.lyrics.com/lyric/27342575/The+Police/Secret+Journey>.

Литература

- [1] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*. MIT Press, 2016.
- [2] P. N. Klein, *Coding the Matrix: Linear Algebra Through Applications to Computer Science*. Newtonian Press, 2013.
- [3] W. Kurt, *Bayesian Statistics the Fun Way*. No Starch Press, Inc., 2019.
- [4] L. Hui, M. Belkin, *Evaluation of neural architectures trained with square loss vs cross-entropy in classification tasks*, in 9th International Conference on Learning Representations, ICLR 2021, OpenReview.net, 2021.
- [5] A. L. Maas, *Rectifier nonlinearities improve neural network acoustic models*, 2013.
- [6] K. He, X. Zhang, S. Ren, J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, in Proceedings of the IEEE International Conference on Computer Vision (ICCV), December 2015.
- [7] A. Krizhevsky, I. Sutskever, G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in Advances in neural information processing systems, 2012.
- [8] K. He, X. Zhang, S. Ren, J. Sun, *Deep residual learning for image recognition*, in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
- [9] K. Simonyan, A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, in International Conference on Learning Representations, 2015.
- [10] T. Brown, B. Mann, N. Ryder, et al., *Language models are few-shot learners*, in Advances in Neural Information Processing Systems (H. Larochelle et al., eds.), vol. 33, Curran Associates, Inc., 2020.
- [11] A. Vaswani, N. Shazeer, N. Parmar, et al., *Attention is all you need*, in Advances in Neural Information Processing Systems (I. Guyon et al., eds.), vol. 30, Curran Associates, Inc., 2017.
- [12] D. P. Kingma and M. Welling, *Auto-Encoding Variational Bayes*, in 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14–16, 2014, Conference Track Proceedings, 2014.
- [13] I. Goodfellow, J. Pouget-Abadie, et al., *Generative adversarial nets*, in Advances in Neural Information Processing Systems (Z. Ghahramani, M. Welling, et al., eds.), vol. 27, Curran Associates, Inc., 2014.
- [14] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR, 2010.

Предметный указатель



Символы β -редукция 6**S**

Softmax 342

A

Абстрактный интерфейс 376

Автоматическое
дифференцирование 75

Аккумулятор 39

мультипликативный 355

Аналого-цифровой
преобразователь 286**B**

Выборка 115

пакета 116

Г

Гиперпараметры 90

Градиент 73, 350

взрывающийся 259

исчезающий 259

Градиентное состояние 365

Градиентный спуск

импульсный 151

стохастический 125

Д

Двойственное представление 359

Дисперсия 260

Дифференцируемый объект 362

усеченный 363

И

Инициализация Хе 260

Искусственный нейрон 200

К

Каррирование 230

Классификация 339

Клауза 7

Ключевое слово 6

Корреляция 296

каскадная 303

Кусочно-линейная

аппроксимация 210

Л

Лепесток 250

Логический столбец 15

М

Массово-параллельные машины 375

Машинное обучение 27

Машинный перевод 339

Метод последовательных

приближений 54

Модель 266

точность 271

Н

Набор

данных 23

тестовый 263

параметров 25

Начало координат 18

Нейронная сеть. См. Нейросеть

Нейросеть 201

вариационный

автокодировщик 344

генеративно-состязательная 345

недообучение 346

обучение 256

переобучение 346

регуляризация 346

рекуррентная 344

сверточная 318

трансформер 344

О

Обратное распространение 349

Обходная связь 331

Оптимизация градиентным

спуском 83

Оптический сенсор 283

Остаточная связь. См. Обходная
связь

Ошибка

классификации 271

среднеквадратичная 59

П

Пакет 116

Пакетный индекс 118

Перекрестная энтропия 340

Поиск по сетке 276

Предобработка 328

Признак 305

Произведение Адамара 46

Р

Распределение вероятностей 338

Регрессия 339

Режим цепочки вычислений

обратный 357

прямой 356

С

Свертка 319

Сигнал 282

напряжение 283

перенос 293

сканирование 296

Скаляр 30

Скорость

обучения 66

спуска 147

Смещение 199

Список форм 234

Спрямление с уткой 342

Степень доверия 252

Т

Тензор 31

двойственный 377

плоский 393

ранг 33

унитарного типа 254

форма 35

экстент 395

У

Унитарное кодирование 252

Ф

Фильтр 297

банк 307

Функция

аксесор 383

активации 196

дифференцируемая 361

естественно-рекурсивная 34

композитная 240

композиция 353

нелинейная 98

обучаемая 60

ожидающая 61

параметризованная 22

параметры 21

расширение 45

рекурсивная 8

решатель 196

сетевая 224

скорость изменения 64

формальный аргумент 3

формы 397

целевая 62

Ц

Цепное правило 355

Цепочка вызовов 354

аргумент 354

результат 354

Циклический обход 396

Ч

Чашелистик 250

Я

Ядро. См. Фильтр

Книги издательства «ДМК ПРЕСС» можно купить оптом
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;

Тел.: +7(499) 782-38-89.

Электронная почта: **books@aliens-kniga.ru**.

Дэниел П. Фридман, Анураг Мендхекар

The Little Learner:
чудесное машинное обучение

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Редактор *Иванов П. Б.*
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90¹/₁₆.
Печать цифровая. Усл. печ. л. 18.69.
Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**

В этой книге из знаменитой серии The Little Book охвачены все концепции, необходимые для интуитивного понимания работы глубоких нейронных сетей, включая тензоры, расширенные операторы, алгоритмы градиентного спуска, искусственные нейроны, полносвязные, сверточные сети и остаточные сети, а также автоматическое дифференцирование.

- С юмором и сократовским подходом, характерными для всех книг серии, объясняется работа глубоких нейронных сетей. Читатель начнет с азов глубокого обучения и познакомится с полной реализацией полезного приложения: распознавателя зашумленных сигналов азбуки Морзе.
- Разговорный стиль, постепенное движение от простого к сложному, забавные иллюстрации, формат вопросов и ответов делают обучение доступным и увлекательным.
- Ключевые идеи машинного обучения представлены с использованием небольшого подмножества языка Scheme.



Дэниел П. Фридман – профессор информатики в Школе информатики и вычислительной техники университета штата Индиана и автор многих книг, опубликованных издательством MIT Press, в том числе «The Little Schemer», «The Seasoned Schemer», «The Little Prover» и «The Reasoned Schemer».



Анураг Мендхекар – соучредитель и президент компании Paper Culture, занимающейся разработкой искусственного интеллекта для творчества, предприниматель.

 The MIT Press


издательство
www.dmk.rf



ISBN 978-5-93700-233-4



9 785937 002334 >