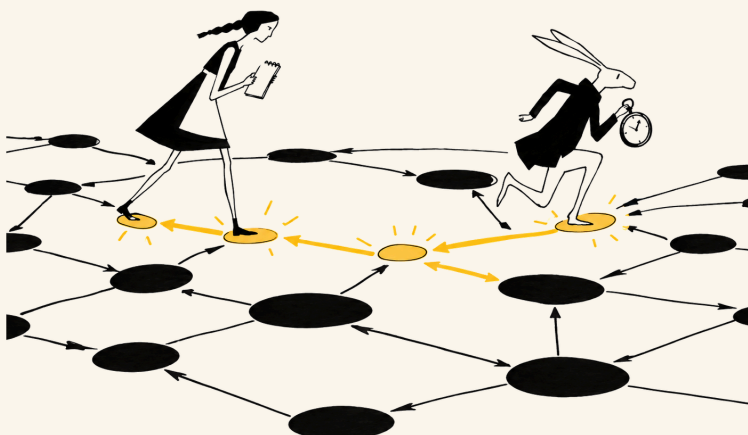


Приключения Алисы в Стране дифференцируемых чудес



Simone Scardapane · 2024

Иллюстрированные аудиолекции

Lecturized

Приключения Алисы в Стране дифференцируемых чудес

Иллюстрированные аудиолекции

Simone Scardapane · 2024

Пересказано в иллюстрированных лекциях Lecturized

lecturized.com

По оригиналу: Alice's Adventures in a Differentiable Wonderland, Volume I: A Tour of
the Land

Содержание

Цепочка кнопок

Одна длинная строка чисел

Три способа найти наклон

Идём по цепочке назад

Никогда не строить таблицу

Записки, оставленные по пути

Четверть, десять раз подряд

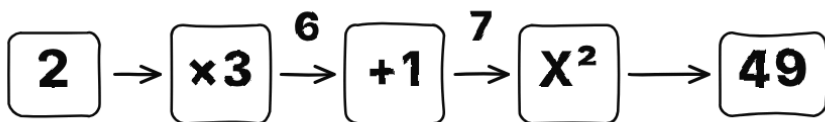
Линейка в углу

Цепочка кнопок

Возьмите калькулятор и нажмите короткую цепочку кнопок: начните с двух, умножьте на три, прибавьте один и возведите результат в квадрат. Получится сорок девять. Теперь начните снова с двух целых одной сотой, сдвинув число на одну сотую, и нажмите те же кнопки. Можно ожидать, что ответ тоже сдвинется примерно на сотую, но выходит около сорока девяти целых сорока двух сотых: ответ сдвинулся в сорок два раза сильнее, чем ваше число. Откуда берётся сорок два?

2.01

49.42



ЦЕПНОЕ ПРАВИЛО

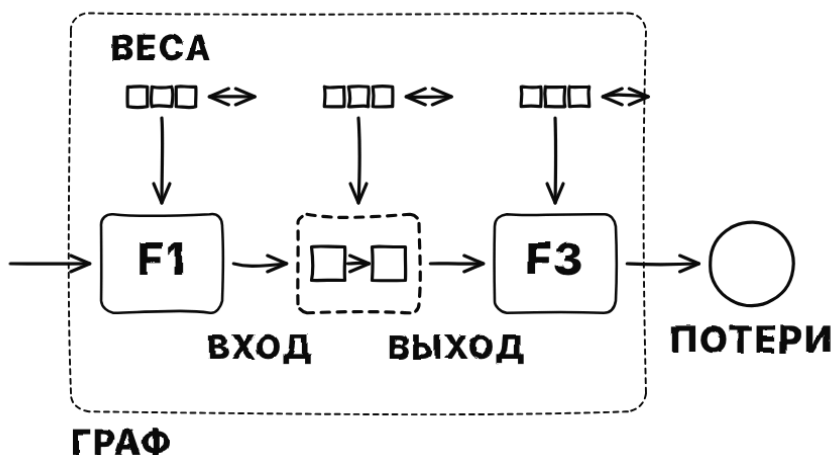
Цепочка кнопок калькулятора

Посмотрите на каждую кнопку отдельно. Умножение на три делает любое маленькое изменение в три раза больше.

Прибавление единицы пропускает изменение нетронутым, поэтому его скорость равна единице. Возведение в квадрат при значении семь делает маленькое изменение в четырнадцать раз больше — вдвое больше того значения, с которым оно работает.

Перемножьте три местные скорости — три на один на четырнадцать — и получится сорок два: скорость всей цепочки равна произведению скоростей её звеньев. Это и есть цепное правило, и вся эта глава о том, как делать именно это, и быстро, для программ с миллионами кнопок.

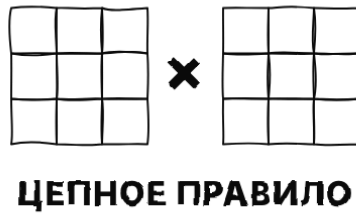
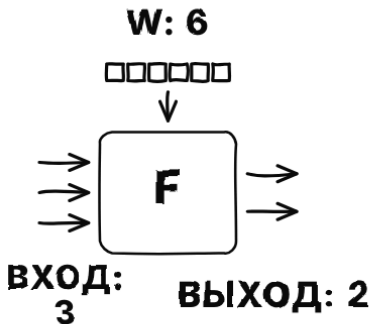
Когда обучают чат-ассистента или фотоприложение в вашем телефоне, программа — это длинная цепочка таких шагов, а в конце стоит одно число, потери, которое оценивает, насколько плохо программа справилась с пачкой примеров. Для обучения нужно знать, как сдвигается эта одна оценка, если чуть сдвинуть каждое из настраиваемых чисел программы — её веса. Вычислять все эти скорости автоматически, прямо по самой программе, называется автоматическим дифференцированием. Книга представляет программу как вычислительный граф: набор маленьких неделимых операций, которые программа действительно выполняет при запуске; их она называет примитивами. Для простоты глава рассматривает шаги как прямую цепочку, один за другим, но всё переносится и на программы, которые ветвятся и сливаются.



Программа как цепочка примитивов

Каждый примитив принимает список чисел — вход, — пользуется собственным списком настраиваемых чисел — своими весами — и передаёт дальше новый список — выход. Примитив может быть совсем маленьким, как одно умножение, большим, как целый слой сети, или даже блоком из многих слоёв: строить большие операции из маленьких — это то, чем программирование занимается всегда. О каждом примитиве книга предполагает только одно: мы знаем его местные скорости, собранные в две таблицы, которые называются якобианами. Входной якобиан говорит, насколько сдвигается каждое выходное число, если чуть сдвинуть каждое входное, поэтому в нём одна строка на каждый выход и один столбец на каждый вход. Якобиан по весам говорит

то же самое для весов: одна строка на каждый выход и один столбец на каждый вес.



Две таблицы примитива

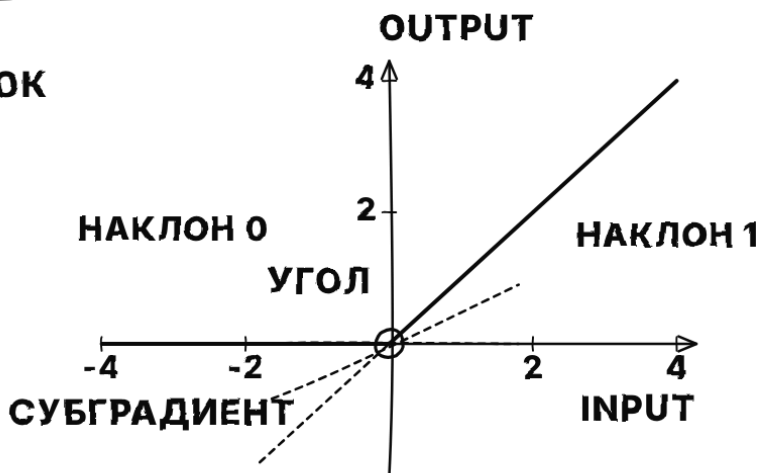
Представьте примитив, который принимает три числа, хранит шесть весов и выдаёт два числа. Его входной якобиан — таблица из двух строк и трёх столбцов, всего шесть скоростей; якобиан по весам — из двух строк и шести столбцов, двенадцать скоростей. А теперь представьте вход из тысячи чисел и выход из тысячи: один только входной якобиан содержит миллион скоростей, и это всего один слой скромной сети.

Поначалу вся задача выглядит тривиальной. Цепное правило, как вы видели на калькуляторе, говорит, что скорость цепочки —

это просто произведение таблиц её звеньев, перемноженных одна за другой. Трудная часть, и тема главы, — выполнять эти умножения эффективно. Ответ, к которому приходит глава, называется автоматическим дифференцированием в обратном режиме, или, на языке нейронных сетей, обратным распространением ошибки, и это краеугольный камень обучения сетей. Поняв его, вы поймёте и то, как устроены программы для обучения сетей, такие как PyTorch, JAX или TensorFlow, и почему они различаются.

Остаётся одно беспокойство: что, если у шага в какой-то точке нет скорости? ReLU, распространённый шаг, который оставляет положительные числа и превращает отрицательные в ноль, имеет в нуле острый угол. Такие шаги обрабатываются с помощью более слабого понятия наклона — субградиента, о котором речь пойдёт в конце главы. Шаги со скачками, например случайную выборку или отсечение по порогу, можно учесть, заменив их скорость гладкой заменой; это автор оставляет для следующего тома.

СКАЧОК



ReLU и её угол

Запомним

Автоматическое дифференцирование вычисляет прямо по программе, как меняется одна итоговая оценка, если чуть сдвинуть каждое настраиваемое число программы.

Программа рассматривается как цепочка примитивов, каждый из которых принимает вход и свои веса и выдаёт выход. Для каждого примитива нужны только две его таблицы местных скоростей: входной якобиан, выходы против входов, и якобиан по весам, выходы против весов. Цепное правило перемножает эти таблицы, как три на один на четырнадцать дало сорок два на калькуляторе; настоящая трудность в том, чтобы делать это эффективно, и ответ на неё — обратное распространение ошибки.

Вопросы

1. На калькуляторе итоговый ответ сдвинулся в сорок два раза сильнее, чем начальное число. Почему именно в сорок два?
 - a. Потому что сорок два — это сумма местных скоростей три, один и четырнадцать плюс начальное значение
 - b. Потому что местные скорости трёх кнопок — три, один и четырнадцать — в произведении дают сорок два
 - c. Потому что возведение в квадрат всегда умножает изменение на итоговый ответ, делённый на начальное число

- d. Потому что сдвиг на одну сотую всегда увеличивается во столько раз, сколько нажато кнопок
2. Примитив выдаёт два числа и хранит шесть весов. Какую форму имеет его якобиан по весам?
- a. Две строки и шесть столбцов: одна строка на каждый выход и один столбец на каждый вес
 - b. Шесть строк и два столбца: одна строка на каждый вес и один столбец на каждый выход
 - c. Две строки и два столбца, потому что считаются только выходы
 - d. Шесть строк и шесть столбцов, потому что каждый вес влияет на каждый вес
3. Если цепное правило просто перемножает таблицы, почему глава называет задачу трудной?
- a. Потому что цепное правило работает только для программ, которые ветвятся и сливаются
 - b. Потому что местные скорости большинства примитивов неизвестны и их приходится угадывать
 - c. Потому что таблицы бывают огромными, и настоящая трудность — выполнять умножения эффективно
 - d. Потому что итоговая оценка — список чисел, а не одно число
4. Почему в этой картине ReLU требует особого обращения?
- a. Потому что у неё есть веса, которые нельзя сдвигать
 - b. Потому что она случайно скачет, как при выборке

- c. Потому что она меняет длину списка чисел, который получает
- d. Потому что у неё в нуле острый угол, где нет обычной скорости

Ответы

- 1. **b)** Каждая кнопка меняет маленький сдвиг на свою местную скорость, а скорость всей цепочки — это произведение скоростей её звеньев: три на один на четырнадцать.
- 2. **a)** Якобиан по весам говорит, насколько сдвигается каждый выход, если чуть сдвинуть каждый вес, поэтому в нём одна строка на каждый выход и один столбец на каждый вес: два на шесть, двенадцать скоростей.
- 3. **c)** Уже у одного слоя с тысячей входов и тысячей выходов во входном якобиане миллион скоростей, так что трудная часть — выполнять умножения эффективно, и эту задачу решает обратное распространение ошибки.
- 4. **d)** ReLU оставляет положительные числа и превращает отрицательные в ноль, отчего в нуле остаётся острый угол; там используется более слабое понятие наклона — субградиент.

Разобранные примеры

Размеры двух таблиц Примитив принимает четыре числа, хранит десять весов и выдаёт три числа. Сколько скоростей в его входном якобиане и сколько в якобиане по весам?

У входного якобиана одна строка на каждый выход и один столбец на каждый вход, поэтому в нём три строки и четыре столбца.

Три на четыре — двенадцать скоростей.

У якобиана по весам одна строка на каждый выход и один столбец на каждый вес, то есть три строки и десять столбцов.

Три на десять — тридцать скоростей.

Ответ: Двенадцать скоростей во входном якобиане и тридцать в якобиане по весам.

Цепочка из трёх кнопок Начните с единицы, умножьте на пять, прибавьте два и возведите в квадрат. Во сколько раз сильнее сдвигается итоговый ответ по сравнению с маленьким сдвигом начального числа?

Умножение на пять делает маленькое изменение в пять раз больше: местная скорость пять.

Прибавление двух пропускает изменение нетронутым: местная скорость один.

До возведения в квадрат доходит семь, а возведение в квадрат увеличивает маленькое изменение во столько раз, сколько дважды это значение: местная скорость четырнадцать.

Скорость цепочки — произведение местных скоростей: пять на один на четырнадцать.

Ответ: В семьдесят раз: сдвиг на одну сотую сдвигает ответ примерно на семь десятых, с сорока девяти примерно до сорока девяти целых семи десятых.

Задачи

1. Слой принимает сто чисел, выдаёт десять чисел и хранит тысячу десять весов. Сколько скоростей в его входном якобиане и в якобиане по весам?

Подсказка: Строки — всегда выходы; столбцы — входы в одной таблице и веса в другой.

Ответ: Тысяча и десять тысяч сто: десять строк на сто столбцов во входном якобиане и десять строк на тысячу десять столбцов в якобиане по весам.

2. В цепочке две кнопки: умножить на два, затем умножить на четыре. Если сдвинуть начальное число на одну десятую, насколько сдвинется итоговый ответ?

Подсказка: Сначала найдите местную скорость каждой кнопки, затем соедините их так, как велит цепное правило.

Ответ: На восемь десятых: местные скорости два и четыре в произведении дают восемь, а восемь на одну десятую — восемь десятых.

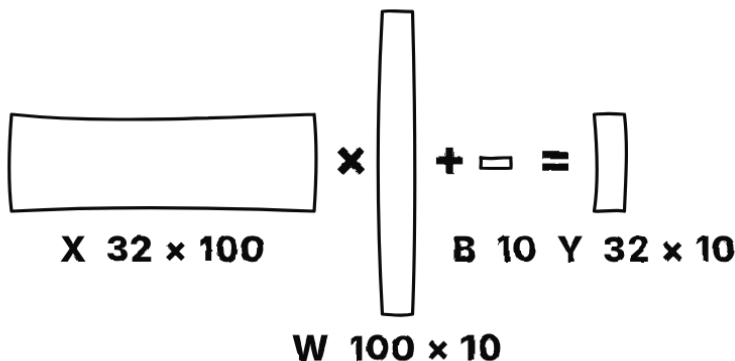
3. Объясните своими словами, почему главе нужно знать для каждого примитива только две таблицы местных скоростей, а не скорости всей программы.

Подсказка: Подумайте, что цепное правило делает со скоростями звеньев.

Ответ: Цепное правило строит скорость всей цепочки, перемножая скорости её звеньев. Поэтому, если известны входной якобиан и якобиан по весам каждого примитива, отсюда следуют и скорости итоговой оценки по каждому весу; остаётся лишь выполнить эти умножения эффективно.

Одна длинная строка чисел

Возьмите лист в клетку и заполните на нём блок числами: тридцать две строки по сто чисел. Теперь перепишите их на одну длинную полосу, строку за строкой: первую строку, сразу за ней вторую и так далее. Ничего не потеряно: на полоске те же три тысячи двести чисел, и любой, кто знает правило, сложит её обратно в блок. Этот маленький приём — развернуть таблицу в одну строку — и позволяет главе говорить о любом слое сети с помощью одних и тех же простых таблиц скоростей.



От блока к полоске, и слой

Вот зачем он нужен. Когда обучают фотоприложение в вашем телефоне, картинки проходят не по одной: они приходят мини-батчем, скажем по тридцать две сразу. Тогда обычный полносвязный слой работает с целой таблицей: тридцать две строки, по одной на картинку, и в каждой сто признаков этой картинки. Слой умножает эту таблицу на таблицу весов — сто строк на десять столбцов — и прибавляет десять дополнительных чисел, называемых смещениями, по одному на выход. На выходе — таблица из тридцати двух строк и десяти столбцов, по десять новых признаков на каждую картинку.

А теперь снова задайте вопрос с калькулятора: насколько сдвигается каждое выходное число, если чуть сдвинуть каждое входное? Выходное число задаётся двумя позициями — картинкой и признаком, — и входное тоже, поэтому у таблицы скоростей четыре направления вместо двух. Книга говорит, что у таких якобианов ранг четыре, и представить их трудно. Выход — полоска. Разверните блок входа в одну строку из трёх тысяч двухсот чисел, блок выхода — в одну строку из трёхсот двадцати, а веса и смещения тоже соберите в одну строку. Тогда входной якобиан — снова простая таблица, одна строка на каждое выходное число и один столбец на каждое входное: триста двадцать строк на три тысячи двести столбцов, чуть больше миллиона скоростей.

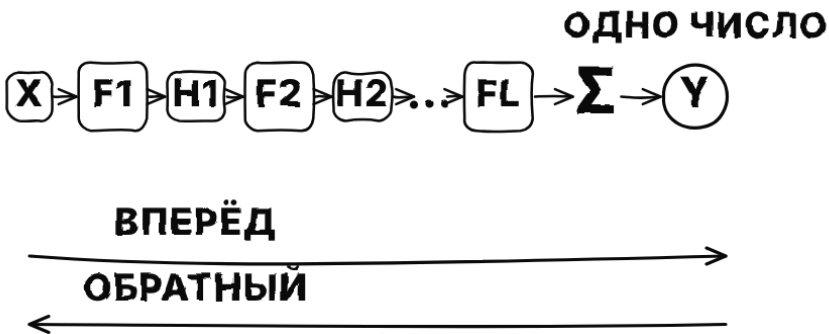
Отсюда следуют две вещи, и книга подчёркивает обе. Во-первых, отныне, говоря о размере входа, она имеет в виду количество всех его чисел вместе, включая батч, то есть здесь три тысячи двести, а не сто. Во-вторых, взгляните ещё раз на этот миллион — для одного скромного слоя и одного маленького батча. Мы можем точно знать, как вычислить такую таблицу, и всё же не хотеть

строить её в памяти, и эта забота определит всё, что последует дальше.

Книга добавляет замечание о том, как пишутся программы. Её способ описывать примитив — функция, которая получает вход и веса рядом, — это то, как работает функциональная библиотека вроде JAX. В PyTorch или TensorFlow слой — это объект, который хранит свои веса внутри себя, и вы вызываете этот объект только на входе. Такой стиль кое-что облегчает, например позволяет подождать с созданием весов, пока первый вход не покажет, какого они должны быть размера, — это называется ленивой инициализацией; но он добавляет обёртку, сквозь которую нужно смотреть, и в итоге два стиля будут вычислять скорости по-разному.

Теперь сама задача. Программа — это цепочка шагов, выписанных один за другим: первый примитив берёт вход и свои веса и делает первый промежуточный результат; второй берёт этот результат и свои веса и делает следующий; и так до последнего. Такой выписанный прогон называется трассой вычисления. Первые шаги — это слои модели, последний шаг оценивает каждый пример, например с помощью перекрёстной энтропии, а итоговая сумма складывает оценки по батчу в одно число. Поэтому программа всегда заканчивается одним числом: обучению нужно одно число, которое оно будет толкать вниз.

ТРАССА



Трасса вычисления заканчивается одним числом

Автоматическое дифференцирование, по определению книги, — это вычисление сразу и эффективно того, как это одно итоговое число зависит от каждого веса каждого шага, когда известны только цепочка и две местные таблицы каждого примитива. Для этого есть два больших семейства методов — прямой режим и обратный режим, — и различаются они только порядком, в котором перемножаются местные таблицы. Обратный, обратное распространение ошибки, окажется здесь гораздо эффективнее. То же рассуждение распространяется на программы, которые ветвятся и сливаются, и на веса, общие для нескольких шагов; к ним книга вернётся гораздо позже.

Запомним

Таблицы чисел с несколькими направлениями, например мини-батч картинок, разворачивают в одну длинную строку, чтобы каждый якобиан оставался простой таблицей из строк и столбцов. После развёртки размер входа означает все его числа, включая батч, и даже один скромный слой даёт таблицу примерно из миллиона скоростей, которую лучше не строить. Программа записывается как цепочка примитивов, заканчивающаяся суммой оценок по примерам, так что её результат всегда одно число. Автоматическое дифференцирование вычисляет, как это число зависит сразу от каждого веса, а прямой и обратный режимы различаются лишь порядком умножений.

Вопросы

1. Зачем книга разворачивает таблицу мини-батча в одну длинную строку чисел?
 - a. Чтобы каждый якобиан стал простой таблицей из строк и столбцов, а не блоком с четырьмя направлениями
 - b. Чтобы слою нужно было меньше весов и он быстрее работал на телефоне
 - c. Чтобы картинки пачки можно было снова обрабатывать по одной
 - d. Чтобы из слоя можно было убрать смещения, не меняя его выход

2. Что после развёртки книга понимает под размером входа слоя, который работает с тридцатью двумя картинками по сто признаков?
- a. Сто — число признаков одной картинки
 - b. Тридцать два — число картинок в пачке
 - c. Три тысячи двести — все числа вместе, включая пачку
 - d. Сто тридцать два — признаки плюс картинки
3. Почему программа в постановке задачи всегда заканчивается одним числом?
- a. Потому что каждый примитив цепочки выдаёт ровно одно число
 - b. Потому что итоговая сумма складывает оценки всех примеров пачки, а обучению нужно одно число, которое оно толкает вниз
 - c. Потому что трасса вычисления записывает только последний слой модели
 - d. Потому что веса всех шагов общие и складываются в одно число
4. Чем прямой режим отличается от обратного?
- a. Прямой режим работает в PyTorch, обратный — только в JAX
 - b. Прямому режиму нужны входные якобианы, обратному — только якобианы по весам
 - c. Прямой режим справляется с ветвящимися программами, обратный — только с прямыми цепочками
 - d. Только порядком, в котором перемножаются местные таблицы скоростей

Ответы

1. **a)** И выходное, и входное число задаются двумя позициями, поэтому у таблицы скоростей четыре направления; если развернуть оба в строки, она снова становится простой таблицей: одна строка на каждое выходное число и один столбец на каждое входное.
2. **с)** Отныне размер входа — это количество всех его чисел вместе, включая пачку: тридцать два на сто.
3. **b)** Последний шаг оценивает каждый пример, а итоговая сумма складывает оценки пачки в одно число, которое обучение и толкает вниз.
4. **d)** Оба вычисляют одни и те же скорости из одних и тех же местных таблиц; различаются они только порядком умножений, и обратный порядок окажется гораздо эффективнее.

Разобранные примеры

Таблицы одного слоя Полносвязный слой получает мини-батч из тридцати двух картинок по сто признаков и выдаёт по десять признаков на картинку, пользуясь таблицей весов сто на десять. Насколько велики после развёртки его входной якобиан и якобиан по весам, если считать только веса сто на десять, как в формах книги?

Разверните вход: тридцать два на сто — три тысячи двести чисел.

Разверните выход: тридцать два на десять — триста двадцать чисел.

У входного якобиана одна строка на каждое выходное число и один столбец на каждое входное: триста двадцать на три тысячи двести, то есть один миллион двадцать четыре тысячи скоростей.

Веса разворачиваются в сто на десять — тысячу чисел, так что якобиан по весам — триста двадцать на тысячу, триста двадцать тысяч скоростей.

Ответ: Во входном якобиане один миллион двадцать четыре тысячи скоростей, а в якобиане по весам — триста двадцать тысяч.

Задачи

1. Слой работает с мини-батчем из восьми примеров по пятьдесят признаков и выдаёт по двадцать признаков на пример, пользуясь таблицей весов пятьдесят на двадцать. Сколько скоростей после развёртки в его входном якобиане и сколько в якобиане по весам, если считать только эти веса?

Подсказка: Сначала сосчитайте все числа входа, выхода и весов, включая пачку.

Ответ: Шестьдесят четыре тысячи и сто шестьдесят тысяч: в выходе восемь на двадцать, сто шестьдесят чисел, во входе восемь на пятьдесят, четыреста, а весов тысяча; сто шестьдесят на четыреста — шестьдесят четыре тысячи, сто шестьдесят на тысячу — сто шестьдесят тысяч.

2. Объясните своими словами, почему книга говорит, что мы можем уметь вычислять якобиан и всё же не хотеть строить его в памяти.

Подсказка: Подумайте, сколько скоростей даже у одного скромного слоя на маленькой пачке.

Ответ: Потому что после развёртки даже у одного скромного слоя на маленькой пачке якобиан примерно из миллиона скоростей. Вычислить его можно, но хранить такие таблицы для каждого слоя потребовало бы очень много памяти, поэтому хороший метод должен обходиться без того, чтобы строить их целиком.

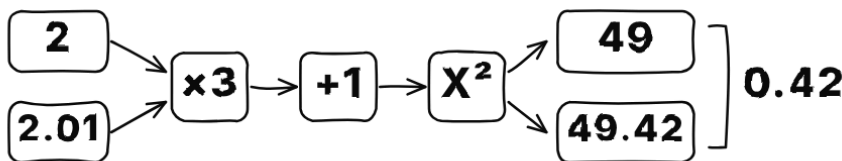
3. Опишите разницу между функциональным способом записи слоя, как в JAX, и объектным, как в PyTorch или TensorFlow.

Подсказка: Спросите себя, где живут веса в каждом стиле.

Ответ: В функциональном способе примитив — это функция, которая получает вход и веса рядом. В объектном слой хранит веса внутри себя и вызывается только на входе; это позволяет ленивую инициализацию — создавать веса, когда первый вход покажет их размер, — но добавляет обёртку, и два стиля вычисляют скорости по-разному.

Три способа найти наклон

Вернитесь к цепочке на калькуляторе из начала: два, умножить на три, прибавить один, в квадрат — получилось сорок девять. Допустим, вы не знаете цепного правила, но скорость вам всё равно нужна. Можно просто прогнать цепочку дважды — один раз с двух, другой раз с двух целых одной сотой — и разделить разность ответов на сдвиг. Ответы — сорок девять и около сорока девяти целых сорока двух сотых, так что скорость выходит около сорока двух, чуть больше, потому что сдвиг не бесконечно мал. Это работает, и это самый честный метод из всех.



$$0,42 \div 0,01 \approx 42$$

✓ ЛИШЬ ПРОВЕРКА

□□ □□ □□ □□ □□ ...

2 ПРОГОНА НА ВЕС

МИЛЛИАРДЫ

Сдвинуть и измерить

Это численное дифференцирование: прямо применить определение скорости — сдвинуть и измерить. Беда в цене. В простом варианте каждое число, для которого нужна скорость, стоит двух прогонов всей программы. У чат-ассистента миллиарды весов, так что одному кругу обучения понадобились бы миллиарды прогонов всей сети. Вот почему, как говорит книга, для обучения метод бесполезен и выживает только как проверка: вы вычисляете несколько скоростей сдвигом и сравниваете их с тем, что говорит ваш хитрый метод.

Второй способ — тот, которому учили в школе. Возьмите маленькую функцию: $a \cdot \sin(x) + b \cdot x \cdot \sin(x)$, умноженное на само число и на его синус. Программа для алгебры, например SymPy на Python, может точно вывести формулу её скорости, так же как вы на бумаге: $a \cdot \cos(x) + b \cdot (x \cdot \cos(x) + \sin(x))$, плюс $b \cdot x \cdot \cos(x)$, умноженное на число и на его косинус, плюс $b \cdot \sin(x)$. Это символьное дифференцирование. В нуле, при a и b , равных единице, формула даёт единицу: косинус нуля равен единице, а два других слагаемых обращаются в ноль.

$$f(x) = a \sin x + b x \sin x$$

SYMPY ↓

$$f'(x) = a \cos x + b x \cos x + b \sin x$$

$$X = 0, A = B = 1 \rightarrow 1$$

H = SIN X **ХРАНИТЬ РАЗ**

ТРУДНО УЛУЧШИТЬ

Формула и её повторы

Но взгляните на формулу: синус встречается в функции дважды, а косинус — дважды в скорости. Разумная программа вычислила бы каждый из них один раз, сохранила значение и использовала его везде, где нужно. Звучит просто, но книга предупреждает: найти самый лучший способ организовать такое вычисление, чтобы ничего не считалось дважды, — задача из того класса, для которых не известно быстрого общего метода; она называется оптимальным накоплением якобиана. Автоматическое дифференцирование обходит эти поиски, следуя структуре самой программы, шаг за шагом, и справляется гораздо лучше, чем выписывание полной формулы; на деле оно сводится к символьному дифференцированию, которому разрешено

хранить и повторно использовать свои промежуточные результаты.

Теперь к первому из двух автоматических методов — прямому режиму. Он исходит из цепного правила для двух шагов. Пусть первый примитив превращает вход в промежуточный результат, а второй превращает его в выход. Таблица скоростей выхода по входу — это таблица второго шага, умноженная на таблицу первого шага. Размеры должны подходить друг к другу: если во входе три числа, в промежуточном результате четыре, а в выходе два, то во второй таблице две строки и четыре столбца, в первой — четыре строки и три столбца, а в их произведении — две строки и три столбца, ровно по одной скорости для каждого выхода против каждого входа. Понимайте правило как рецепт: выполнив первый шаг и зная его таблицу, при выполнении второго шага вы обновляете эту таблицу, умножая её на новую.

$$\begin{array}{ccc}
 \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} & \times & \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} & = & \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \\
 2 \times 4 & & 4 \times 3 & & 2 \times 3
 \end{array}$$



J·T1

J·T2

T3



Σ СКОРОСТИ

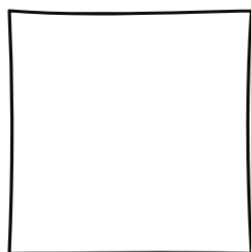
Прямой режим несёт касательные

Прямой режим превращает этот рецепт в алгоритм. Для каждого списка весов он хранит таблицу, называемую касательной: как текущий промежуточный результат зависит от этих весов. На первом шаге ещё ничего не сохранено, поэтому касательная первых весов — это просто якобиан первого шага по весам. На втором шаге происходят две вещи сразу: первая касательная обновляется умножением на входной якобиан второго шага, и для вторых весов заводится новая касательная, равная якобиану второго шага по весам. И так далее вниз по цепочке: на каждом шаге каждая уже хранящаяся касательная умножается на входной якобиан этого шага, один и тот же для всех, и рождается одна новая касательная из якобиана шага по весам. В самом конце программа складывает числа последнего результата, и

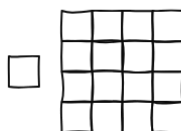
сложение строк каждой касательной даёт скорости итоговой оценки по каждому весу.

Затем книга взвешивает метод. С хорошей стороны, всю эту работу можно вплести в программу по ходу её выполнения, шаг за шагом, так что памяти нужно примерно столько же, сколько самой программе. С плохой стороны, сердце алгоритма — обновление касательной — перемножает две большие таблицы, а это очень дорого. Пусть каждый шаг принимает и выдаёт таблицу мини-батча из тридцати двух примеров по сто признаков, а одна касательная относится к тысяче весов. После развёртки входной якобиан — это три тысячи двести строк на три тысячи двести столбцов, а касательная — три тысячи двести строк на тысячу столбцов. Их произведение стоит примерно три тысячи двести, умноженные на три тысячи двести и на тысячу умножений: около десяти миллиардов на одно обновление одной касательной на одном шаге. Стоимость растёт как квадрат размера пачки и квадрат числа признаков, так что удвоение того и другого делает её в шестнадцать раз больше; для картинок с их огромным числом признаков это скоро становится невозможным.

ВХ. J КАСАТЕЛЬНАЯ



×



×2 ПАЧКА
×2 ПРИЗНАКИ
×16

3,200 × 3,200

3,200 × 1,000

≈ 10 000 000 000

✓ ПАМЯТЬ: МАЛО

Цена одного обновления касательной

Опыт

Попробуйте честный метод на калькуляторе на одном только возведении в квадрат: возведите в квадрат три, затем три целых одну тысячную. Ответы — девять и примерно девять целых шесть тысячных, так что скорость близка к шести, удвоенным трём, как и говорит школьное правило. А теперь представьте, что эти два прогона нужно повторить для каждого из миллиардов весов чат-ассистента, и вы поймёте, почему честный метод оставлен только для проверки.

Запомним

Численное дифференцирование сдвигает каждое число и заново прогоняет программу, поэтому стоит два прогона на вес и годится только для проверки других методов.

Символьное дифференцирование выписывает полную формулу скорости, которая повторяет работу, если не хранить промежуточные значения, а организовать это идеально — трудная задача; автоматическое дифференцирование вместо этого следует собственной структуре программы. Прямой режим хранит по одной касательной на каждый список весов, умножает каждую хранимую касательную на входной якобиан каждого нового шага и добавляет новую касательную из якобиана шага по весам. Памяти ему нужно мало, но каждое обновление перемножает две большие таблицы, и стоимость растёт как квадрат размера пачки и числа признаков — десять миллиардов умножений для одного скромного слоя.

Вопросы

1. Почему численное дифференцирование оставляют только для проверки и не используют для обучения?
 - a. Потому что оно даёт неверную скорость, когда в функции есть синус
 - b. Потому что ему нужна полная формула скорости, прежде чем оно сможет начать

- c. Потому что каждый вес стоит двух прогонов всей программы, а у сетей миллиарды весов
 - d. Потому что им нельзя пользоваться на калькуляторе, только внутри библиотеки
- 2. Что делает простую символьную формулу расточительной и что вместо этого делает автоматическое дифференцирование?
 - a. Формула повторяет слагаемые вроде синуса и косинуса; автоматическое дифференцирование следует собственным шагам программы и повторно использует промежуточные результаты
 - b. Формула всегда неверна в нуле; автоматическое дифференцирование вместо этого сдвигает вход
 - c. Формуле нужен калькулятор; автоматическому дифференцированию — только бумага
 - d. В формуле слишком мало слагаемых; автоматическое дифференцирование добавляет недостающие
- 3. Что в прямом режиме происходит с уже хранящимися касательными, когда программа выполняет новый шаг?
 - a. Их выбрасывают и строят заново из якобиана нового шага по весам
 - b. Каждая умножается на входной якобиан нового шага, один и тот же для всех, и заводится новая касательная
 - c. Обновляется только самая свежая касательная, а более старые остаются как были
 - d. Они складываются в одну касательную для всей программы

4. Как изменится стоимость одного обновления касательной в прямом режиме, если удвоить и размер пачки, и число признаков?
- a. Удвоится
 - b. Вырастет в четыре раза
 - c. Не изменится, потому что число весов прежнее
 - d. Вырастет в шестнадцать раз

Ответы

1. **c)** Сдвинуть и измерить — это работает, но каждое число, для которого нужна скорость, стоит двух прогонов всей программы, а у чат-ассистента миллиарды весов.
2. **a)** Синус встречается в функции дважды, а косинус — дважды в скорости; автоматическое дифференцирование следует структуре программы и сводится к символьному дифференцированию, которому разрешено хранить и повторно использовать промежуточные результаты.
3. **b)** На каждом шаге каждая уже хранящаяся касательная умножается на входной якобиан этого шага, общий для всех, и рождается одна новая касательная из якобиана шага по весам.
4. **d)** Стоимость растёт как квадрат размера пачки и квадрат числа признаков, так что удвоение того и другого умножает её на четыре и ещё раз на четыре.

Разобранные примеры

Скорость функции с синусом Функция — это a , умноженное на синус числа, плюс b , умноженное на само число и на его синус. Её скорость, выведенная символьно, — это a , умноженное на

косинус, плюс b , умноженное на число и на косинус, плюс b , умноженное на синус. Чему равна скорость в нуле при a , равном двум, и b , равном единице?

Первое слагаемое — a , умноженное на косинус нуля; косинус нуля равен единице, так что оно даёт два.

Во втором слагаемом множителем стоит само число, а оно равно нулю, так что оно даёт ноль.

Третье слагаемое — b , умноженное на синус нуля; синус нуля равен нулю, так что оно даёт ноль.

Сложите три слагаемых.

Ответ: Два.

Цена одного обновления касательной Шаг принимает и выдаёт мини-батч из тридцати двух примеров по сто признаков, а касательная относится к тысяче весов. Примерно сколько умножений стоит одно обновление этой касательной?

После развёртки во входе и в выходе по тридцать два на сто — три тысячи двести чисел.

Значит, входной якобиан — три тысячи двести на три тысячи двести, а касательная — три тысячи двести на тысячу.

Перемножение двух таблиц стоит примерно произведение трёх размеров: три тысячи двести на три тысячи двести на тысячу.

Три тысячи двести в квадрате — десять миллионов двести сорок тысяч; умноженное на тысячу, это около десяти миллиардов.

Ответ: Около десяти миллиардов умножений на одно обновление одной касательной на одном шаге.

Задачи

1. У небольшой модели тысяча весов. Сколько прогонов всей программы нужно простому численному дифференцированию, чтобы получить все их скорости?

Подсказка: Каждое число, для которого нужна скорость, стоит определённого числа прогонов.

Ответ: Две тысячи прогонов: по два прогона, со сдвигом и без, на каждый из тысячи весов.

2. Первый примитив превращает пять чисел в шесть, а второй превращает эти шесть в два. Каковы формы двух таблиц в цепном правиле и их произведения?

Подсказка: В каждой таблице одна строка на каждый выход её шага и один столбец на каждый вход её шага.

Ответ: Таблица второго шага — два на шесть, таблица первого — шесть на пять, а их произведение — два на пять: по одной скорости для каждого из двух выходов против каждого из пяти входов.

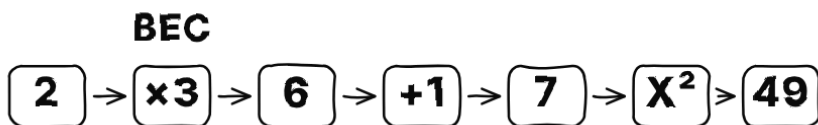
3. В прямом режиме программа выполняет четвёртый шаг. Сколько хранящихся касательных обновляется на этом шаге и сколько новых заводится?

Подсказка: Сосчитайте касательные, которые оставили первые три шага.

Ответ: Обновляются три, по одной на каждый прежний шаг, и все умножаются на входной якобиан четвёртого шага, а одна новая касательная заводится из якобиана четвёртого шага по весам.

Идём по цепочке назад

Возьмите цепочку на калькуляторе ещё раз: два, умножить на три, прибавить один, в квадрат — сорок девять. На этот раз прочитайте её с конца. Последняя кнопка, возведение в квадрат при семи, увеличивает маленькое изменение в четырнадцать раз, так что ответ сдвигается в четырнадцать раз сильнее, чем значение прямо перед квадратом. На кнопку раньше прибавление единицы пропускает изменения без изменений, так что ответ по-прежнему сдвигается в четырнадцать раз сильнее, чем шестёрка, вышедшая из умножения. Ещё на кнопку назад — умножение на три, — и ответ сдвигается в сорок два раза сильнее, чем начальная двойка. Идя назад, вы на каждой остановке узнавали, насколько итоговому ответу важно стоящее там число.



42 ← **14** ← **14**



НАЗАД

$$14 \times 2 = 28$$

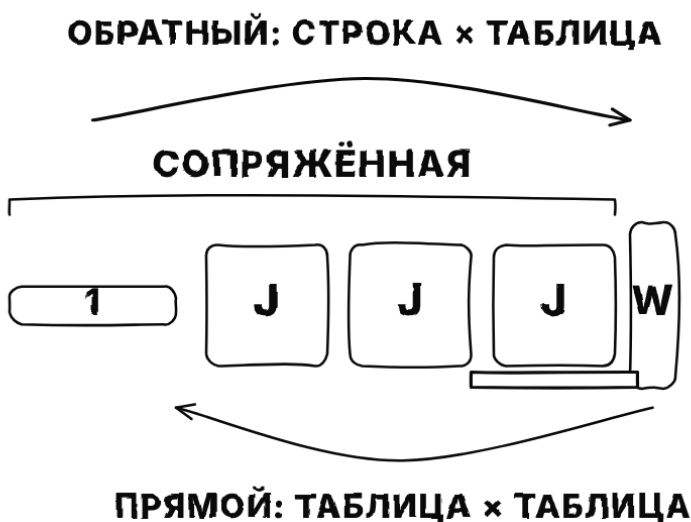
$$3,01 \rightarrow 49,28$$

Читаем цепочку с конца

Теперь представьте, что тройка в первой кнопке — настраиваемый вес. Насколько сдвинется ответ, если чуть сдвинуть этот вес? Выход этой кнопки — вес, умноженный на два, так что сдвиг веса сдвигает выход вдвое сильнее; а мы уже знаем, что ответ сдвигается в четырнадцать раз сильнее, чем этот выход. Четырнадцать на два: ответ сдвигается в двадцать восемь раз сильнее, чем вес. Можно проверить: с тремя целыми одной сотой вместо трёх цепочка даёт около сорока девяти целых двадцати восьми сотых. Этот маленький путь назад — вся идея этой части.

Книга выписывает скорость итоговой оценки по весам одного шага где-то в середине цепочки. Это длинное произведение. Сначала идёт строка единиц, которая обозначает итоговую

сумму: каждое число последнего результата входит в оценку один раз. Затем идут входные якобианы всех последующих шагов, от последнего назад до шага сразу после нашего, и наконец якобиан нашего шага по весам. Первой части этого произведения — строке единиц, умноженной на все эти последующие входные якобианы, — книга даёт имя: это скорость итоговой оценки по промежуточному результату нашего шага.

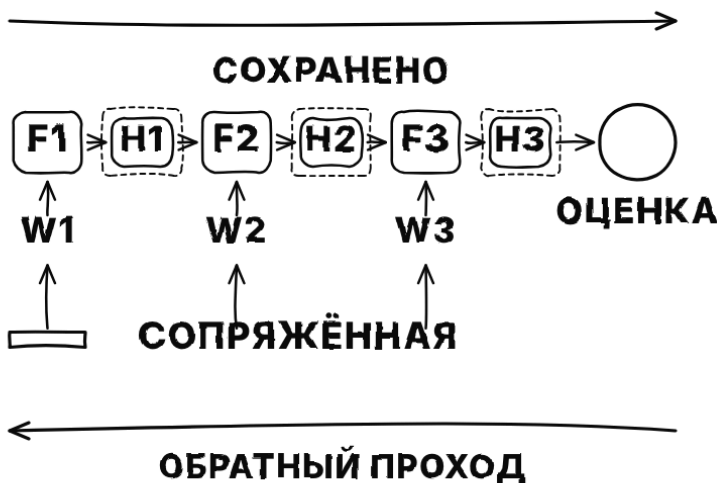


С какого конца начинать

В произведении таблиц можно умножать в любом порядке и получать тот же результат. Прямой режим умножал от собственного конца шага наружу, в порядке работы программы, и расплачивался большими произведениями таблицы на таблицу. Но взгляните на другой конец. Он начинается с одной

строки, а строка на таблицу гораздо дешевле, чем таблица на таблицу; и выходит снова одна строка. А строку для одного шага можно получить из строки следующего шага ещё одним умножением на входной якобиан, идя назад ровно так, как вы делали на калькуляторе.

Так получается автоматическое дифференцирование в обратном режиме. Сначала прогоните всю программу, от начала до конца, и сохраните каждый промежуточный результат. Затем начните со строки единиц и идите назад от последнего шага к первому. На каждом шаге умножьте текущую строку на якобиан шага по весам — это скорость оценки по весам этого шага, готово. Затем умножьте строку на входной якобиан шага и несите новую строку на шаг назад.



Обратный проход

Этот путь назад — вторая программа, почти зеркальная первой; её называют обратной, или двойственной, программой. Строки, которые она несёт, называются сопряжёнными переменными: каждая говорит, как итоговая оценка зависит от каждого числа одного промежуточного результата, тогда как касательные прямого режима говорили, как промежуточный результат зависит от весов. На языке сетей прогон исходной программы — это прямой проход, что не то же самое, что прямой режим, а путь назад — обратный проход.

Выигрыш велик. Каждый обратный шаг умножает одну строку на таблицу, и стоимость растёт лишь пропорционально участвующим размерам. Для прежнего слоя — тридцать два

примера по сто признаков и тысяча весов — два произведения одного обратного шага стоят около десяти миллионов и около трёх миллионов умножений, примерно в тысячу раз меньше, чем десять миллиардов одного обновления в прямом режиме. Именно это и позволяет вообще обучить фотоприложение в вашем телефоне или чат-ассистента.



Дешёвые шаги, дорогая память

Плата — память. Весь прямой проход должен закончиться, прежде чем начнётся обратный, и каждый промежуточный результат нужно хранить, пока путь не вернётся к нему. Контрольные точки градиента меняют часть этой памяти на дополнительную работу. На рисунке книги четыре блока в ряд: во время прямого прохода сохраняются только выходы первого,

второго и четвёртого блоков — как контрольные точки. На обратном пути проход застревает на третьем блоке, результаты которого выброшены; тогда он заново выполняет короткий прямой проход от ближайшей контрольной точки, второго блока, восстанавливает недостающее и идёт дальше. Это стоит на четверть больше вычислений, чем простой обратный проход, и в общем случае чем меньше контрольных точек хранится, тем больше приходится вычислять заново; обучение очень больших моделей опирается на этот обмен каждый день.

Запомним

Скорость итоговой оценки по весам одного шага — это длинное произведение, которое начинается со строки единиц и идёт через входные якобианы последующих шагов к якобиану шага по весам. Обратный режим умножает со стороны строки: сначала прогоняет всю программу, сохраняет каждый промежуточный результат, а затем идёт назад, неся одну строку — сопряжённую переменную. На каждом шаге сопряжённая переменная, умноженная на якобиан по весам, даёт скорости этого шага, а сопряжённая переменная, умноженная на входной якобиан, становится следующей сопряжённой переменной назад — как четырнадцать на два дало двадцать восемь на калькуляторе. Произведения строки на таблицу делают его примерно в тысячу раз дешевле прямого режима для одного скромного слоя ценой памяти, которую контрольные точки градиента уменьшают, пересчитывая часть результатов.

Вопросы

1. Почему обратный режим перемножает длинное произведение, начиная со строки единиц?
 - a. Потому что строка единиц — самая большая таблица и её нужно израсходовать первой
 - b. Потому что строка на таблицу гораздо дешевле таблицы на таблицу, а результат — снова одна строка
 - c. Потому что таблицы можно перемножать только в порядке работы программы
 - d. Потому что строка единиц избавляет от нужды в якобианах по весам
2. О чём говорит сопряжённая переменная?
 - a. Как промежуточный результат сдвигается, если чуть сдвинуть веса прежних шагов
 - b. Сколько промежуточных результатов нужно хранить во время прямого прохода
 - c. Как итоговая оценка зависит от каждого числа одного промежуточного результата
 - d. Сколько памяти нужно шагу во время обратного прохода
3. Почему обратный проход не может начаться до окончания прямого?
 - a. Потому что обратный путь начинается с последнего шага, и ему нужны промежуточные результаты, которые производит и сохраняет прямой проход

- b. Потому что веса создаются только в конце прямого прохода
 - c. Потому что строка единиц вычисляется сложением всех весов
 - d. Потому что библиотека сначала должна перевести программу в формулу
4. Что происходит, когда хранится меньше контрольных точек?
- a. Обратный проход становится дешевле и требует меньше памяти
 - b. Скорости становятся менее точными, потому что часть результатов угадывается
 - c. Прямой проход приходится вовсе пропустить
 - d. Памяти тратится меньше, но больше результатов приходится вычислять заново во время обратного прохода

Ответы

1. b) Произведение можно брать в любом порядке; если начать со стороны строки, каждое умножение — одна строка на таблицу, а это дёшево и снова даёт одну строку.
2. c) Сопряжённые переменные — это строки, которые несёт обратный путь; каждая говорит, как итоговая оценка зависит от чисел одного промежуточного результата, а касательные прямого режима шли в другую сторону.
3. a) Путь начинается с конца цепочки и на обратном ходу использует каждый промежуточный результат, поэтому вся программа должна сначала отработать и сохранить их все.

4. d) Контрольные точки меняют память на работу: выброшенные результаты восстанавливаются повторным прогоном части прямого прохода, так что меньше контрольных точек — больше пересчёта.

Разобранные примеры

Цена четырёх блоков В сети четыре блока. При прямом проходе сохраняются только выходы первого, второго и четвёртого блоков. Насколько больше вычислений это требует по сравнению с простым обратным проходом и почему?

Прямой проход выполняет каждый из четырёх блоков по разу.

На обратном пути проход застревает на третьем блоке, результаты которого не сохранены.

Он заново выполняет прямое вычисление от ближайшей контрольной точки, выхода второго блока, то есть выполняет третий блок ещё раз.

Значит, нужно пять прогонов блоков вместо четырёх — один лишний на каждые четыре.

Ответ: На четверть больше вычислений — в одну целую двадцать пять сотых раза.

Путь назад с весом Цепочка начинается с единицы, умножает на вес, равный пяти, прибавляет два и возводит в квадрат — получается сорок девять. Идя назад, найдите, насколько сдвигается ответ, если чуть сдвинуть вес.

Квадрат работает с семёркой и увеличивает маленькое изменение в четырнадцать раз: ответ сдвигается в четырнадцать раз сильнее, чем семёрка.

Прибавление двух пропускает изменения: ответ сдвигается в четырнадцать раз сильнее, чем пятёрка, вышедшая из умножения.

Эта пятёрка — вес, умноженный на начальную единицу, так что сдвиг веса сдвигает её ровно настолько же: скорость один.

Умножьте принесённую скорость на эту местную: четырнадцать на один.

Ответ: В четырнадцать раз сильнее, чем сдвиг веса.

Задачи

1. Цепочка начинается с трёх, умножает на вес, равный четырём, прибавляет один и возводит в квадрат. Во сколько раз сильнее сдвигается ответ по сравнению с маленьким сдвигом веса?

Подсказка: Идите назад от квадрата: сначала найдите, насколько ответу важно число, входящее в квадрат.

Ответ: В семьдесят восемь раз: квадрат работает с тринадцатую, так что скорость там двадцать шесть; прибавление единицы её пропускает; выход умножения — вес, умноженный на три, так что сдвиг веса сдвигает его втрое сильнее; двадцать шесть на три — семьдесят восемь.

2. В сети восемь блоков, и выходы всех блоков, кроме пятого, сохраняются как контрольные точки. Сколько дополнительных вычислений нужно обратному проходу?

Подсказка: Сосчитайте, сколько прогонов блоков делает прямой проход и сколько нужно повторить.

Ответ: На одну восьмую больше: прямой проход выполняет восемь блоков, и заново нужно выполнить только пятый — от контрольной точки четвёртого блока, — так что нужно девять прогонов вместо восьми.

3. Объясните своими словами, почему обратный режим настолько дешевле прямого и чем он за это платит.

Подсказка: Сравните, что перемножается на каждом шаге в двух методах.

Ответ: Прямой режим на каждом шаге перемножает большие таблицы на большие таблицы, а обратный только и делает, что умножает одну строку на таблицу, и стоимость этого растёт лишь пропорционально размерам; для скромного слоя это примерно в тысячу раз меньше работы. Плата — память: весь прямой проход нужно хранить, пока обратный путь не дойдёт до каждого результата, и контрольные точки облегчают это, пересчитывая часть из них.

Никогда не строить таблицу

Нарисуйте на бумаге маленькую таблицу умножения. Сверху напишите два числа, два и три; слева сбоку — два других, один и четыре. Заполните каждую клетку произведением числа её строки на число её столбца: в первой строке получится два и три, во второй — восемь и двенадцать. Эта маленькая сетка, на которую ребёнку нужна минута, — ровно тот набор скоростей, который нужен программе обучения для весов слоя, и к концу этой части вы поймёте почему.

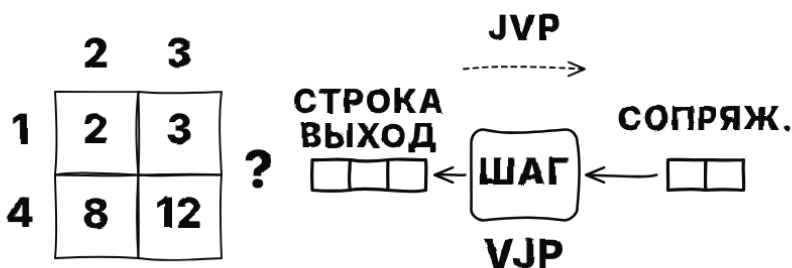
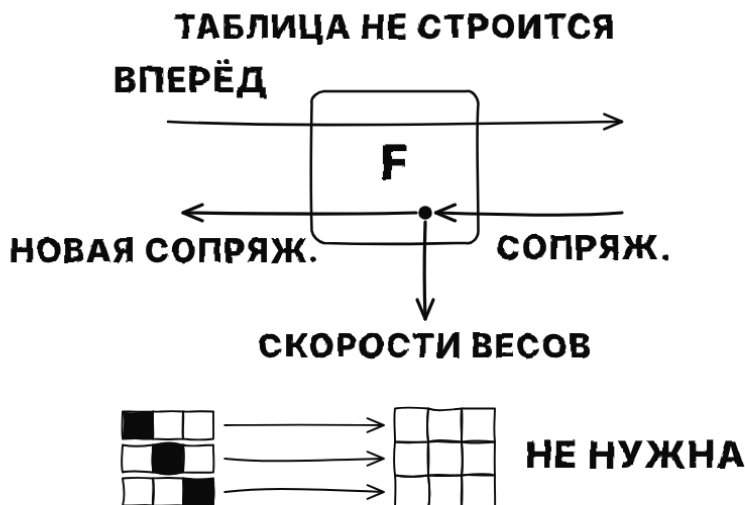


Таблица умножения и «строка вход — строка выход»

Взгляните ещё раз на обратный путь обратного режима. На каждом шаге происходят только две вещи: сопряжённая переменная, одна строка, умножается на якобиан шага по весам, и та же строка умножается на входной якобиан шага. Больше от шага никогда ничего не требуется. Книга даёт этим двум операциям имя: произведение вектора на якобиан.

Произведение вектора на якобиан — это функция: вы даёте ей строку с одним числом на каждый выход шага, и она возвращает строку с одним числом на каждый вход или на каждый вес. У прямого режима есть своя зеркальная операция — произведение якобиана на вектор, — которая для функции с одним выходом есть знакомая скорость вдоль выбранного направления.

Так что обратный шаг — это всего два вызова. Вызовите произведение вектора на якобиан по весам с сопряжённой переменной — и получите скорости оценки по весам этого шага. Вызовите входное произведение вектора на якобиан с той же сопряжённой переменной — и получите новую сопряжённую переменную, готовую к тому, чтобы её несли на шаг назад. Рисунок книги изображает примитив как блок: прямой проход входит с одной стороны и выходит с другой, а две обратные стрелки, по одной на каждое из этих произведений, несут сопряжённую переменную назад сквозь него. И вот смысл всего раздела: полная система автоматического дифференцирования — это набор примитивов, каждый в паре с двумя своими произведениями вектора на якобиан, и ни один якобиан никогда не нужно строить в памяти.

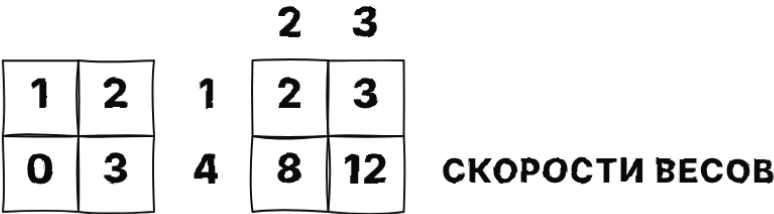


Примитив с двумя обратными правилами

При этом ничего не теряется. Если вам действительно нужен полный якобиан, можно вызвать произведение один раз со строкой, в которой на первом месте единица, а в остальных нули, затем с единицей на втором месте и так далее, и каждый вызов даст одну строку таблицы. Но программе обучения таблица никогда не нужна; ей нужны только произведения с сопряжёнными переменными.

Теперь награда — для двух частей, из которых состоит полносвязный слой. Первая — линейная проекция: выход — это таблица весов, умноженная на вход. Её входной якобиан — просто сама таблица весов, а вот якобиан по весам — блок с тремя направлениями: выход, строка весов и столбец весов. Входному

произведению блок не нужен: каждое входное число собирает элементы сопряжённой переменной для тех выходов, которые оно питало, каждый взвешенный весом, который их соединял, — это то же самое, что пропустить сопряжённую переменную назад через таблицу весов, повернутую набок. Возьмите таблицу весов, первая строка которой — один и два, а вторая — ноль и три, и сопряжённую переменную один и четыре. Первый вход питал выходы через веса один и ноль, поэтому он собирает один на один плюс четыре на ноль, то есть один; второй питал их через два и три, поэтому он собирает один на два плюс четыре на три, то есть четырнадцать.



СКОРОСТИ ВХОДА: 1, 14

Два произведения линейного шага

Произведение по весам ещё приятнее — это ваша таблица умножения. Скорость оценки по весу в данной строке и данном столбце — это элемент сопряжённой переменной этой строки, умноженный на элемент входа этого столбца. Рассуждение книги словами: оценка, как её видит этот шаг, — это сумма произведений, где каждый вес умножен на свой элемент сопряжённой переменной и на свой элемент входа, так что сдвиг одного веса сдвигает оценку ровно на его элемент сопряжённой переменной, умноженный на его элемент входа. При входах два и три и сопряжённой переменной один и четыре сетка — два и три, затем восемь и двенадцать: та самая таблица, которую вы нарисовали в начале. Так что каждый раз, когда линейная проекция работает вперёд, обратный проход отправляет сопряжённую переменную назад через её повернутые веса и заполняет таблицу умножения для её весов, и нигде не появляется блок с тремя направлениями.

Вторая часть — функция активации, применяемая к каждому числу отдельно, например ReLU; весов у неё нет, так что важно только входное произведение. Каждый выход зависит только от входа на том же месте, поэтому входной якобиан — таблица, нулевая всюду, кроме диагонали, где стоит наклон активации при каждом входе. Умножение строки на такую таблицу просто масштабирует каждый элемент сопряжённой переменной на наклон на том же месте. Для ReLU при входах минус один, два и три наклоны равны нулю, единице и единице; сопряжённая переменная пять, шесть и семь возвращается как ноль, шесть и семь. Для слоя из тысячи чисел полная таблица содержала бы миллион элементов, все, кроме тысячи, нулевые, а произведению она никогда не нужна. Вот как фотоприложение в вашем

телефоне получает свои скорости: каждая операция в библиотеке обучения несёт своё маленькое обратное правило такого рода.



1000 × 1000: ПОЧТИ НУЛИ

ReLU только масштабирует

Запомним

Обратный путь только и делает, что умножает строку на якобиан, и эти операции — произведения вектора на якобиан, одно для входа и одно для весов каждого примитива. Система автоматического дифференцирования — это набор примитивов, у каждого по два произведения, и она никогда не строит якобиан, хотя могла бы, по одной строке. Для линейной проекции входное произведение отправляет сопряжённую переменную назад через повернутую таблицу весов, а произведение по весам — это таблица умножения элементов сопряжённой переменной на элементы входа. Для активации, применяемой к каждому числу отдельно, входное произведение лишь масштабирует каждый элемент сопряжённой переменной на наклон активации при соответствующем входе — как ReLU превратила пять, шесть и семь в ноль, шесть и семь.

Вопросы

1. Почему системе автоматического дифференцирования никогда не нужно строить якобиан в памяти?
 - а. Потому что обратный путь только и делает, что умножает строку на якобиан, а каждый примитив может сделать это напрямую с помощью двух своих произведений вектора на якобиан
 - б. Потому что якобианы всегда нулевые, кроме диагонали

- c. Потому что скорости находят, сдвигая каждый вес
 - d. Потому что прямой проход уже хранит якобианы каждого шага
2. Каковы для линейной проекции скорости оценки по её весам?
- a. Сама таблица весов, повёрнутая набок
 - b. Сложенные элементы сопряжённой переменной, по разу на каждый вес
 - c. Таблица умножения: элемент сопряжённой переменной строки веса на элемент входа его столбца
 - d. Блок с тремя направлениями, который нужно построить до обратного прохода
3. Что входное произведение вектора на якобиан у ReLU делает с сопряжённой переменной?
- a. Прибавляет наклон ReLU к каждому её элементу
 - b. Умножает каждый её элемент на наклон ReLU при входе на том же месте
 - c. Заменяет её входами слоя
 - d. Умножает её на полную таблицу наклонов — миллион элементов для тысячи чисел
4. Как получить полный входной якобиан примитива из его произведения вектора на якобиан, если он действительно нужен?
- a. Вызвать его один раз со строкой из единиц
 - b. Вызвать его с весами вместо сопряжённой переменной
 - c. Прогнать прямой проход дважды со сдвигом

- d. Вызывать его со строкой, где на одном месте единица, а в остальных нули, — по разу для каждого места, по одной строке таблицы за вызов

Ответы

1. a) Обратный шаг требует лишь сопряжённую переменную на якобиан по весам и сопряжённую переменную на входной якобиан, так что достаточно примитивов в паре с этими двумя произведениями.
2. c) Оценка, как её видит шаг, — это сумма произведений каждого веса на его элемент сопряжённой переменной и на его элемент входа, так что сдвиг одного веса сдвигает оценку на этот элемент сопряжённой переменной, умноженный на этот элемент входа.
3. b) Входной якобиан активации, применяемой к каждому числу отдельно, нулевой вне диагонали, так что произведение просто масштабирует каждый элемент сопряжённой переменной на наклон на том же месте, и никакая таблица не строится.
4. d) Каждый вызов со строкой, где одна единица, даёт одну строку таблицы, так что повторение для каждого места восстанавливает весь якобиан.

Разобранные примеры

Скорости по весам проекции Линейная проекция получает входы один и пять, а назад приходит сопряжённая переменная два и три. Каковы скорости оценки по её четырём весам?

Скорость для веса в данной строке и данном столбце — элемент сопряжённой переменной этой строки, умноженный на элемент входа этого столбца.

Первая строка, элемент два: два на один — два, два на пять — десять.

Вторая строка, элемент три: три на один — три, три на пять — пятнадцать.

Разложите их сеткой: строки — по элементам сопряжённой переменной, столбцы — по входам.

Ответ: Первая строка — два и десять, вторая — три и пятнадцать.

Назад через ReLU Слой ReLU получил входы четыре, минус два, одну вторую и минус три. Назад приходит сопряжённая переменная один, два, три и четыре. Какую сопряжённую переменную он передаёт дальше?

Наклон ReLU равен единице там, где вход положителен, и нулю там, где отрицателен.

Значит, наклоны — один, ноль, один и ноль.

Масштабируйте каждый элемент сопряжённой переменной на наклон на том же месте.

Ответ: Один, ноль, три и ноль.

Задачи

1. У линейной проекции таблица весов, первая строка которой — два и один, а вторая — один и три. Назад приходит сопряжённая переменная один и два. Какую новую сопряжённую переменную передаёт входное произведение?

Подсказка: Каждый вход собирает элементы сопряжённой переменной для выходов, которые он питал, каждый взвешенный весом, который их соединял; первый вход питал выходы через первый элемент каждой строки.

Ответ: Четыре и семь: первый вход получает один на два плюс два на один, то есть четыре; второй получает один на один плюс два на три, то есть семь.

2. У примитива пять выходов и восемь входов. Сколько вызовов его входного произведения вектора на якобиан нужно, чтобы восстановить весь входной якобиан, и сколько чисел даёт каждый вызов?

Подсказка: Каждый вызов с одной единицей даёт одну строку таблицы; спросите себя, сколько в таблице строк.

Ответ: Пять вызовов, по одному на каждый выход, и каждый даёт строку из восьми чисел, по одному на вход.

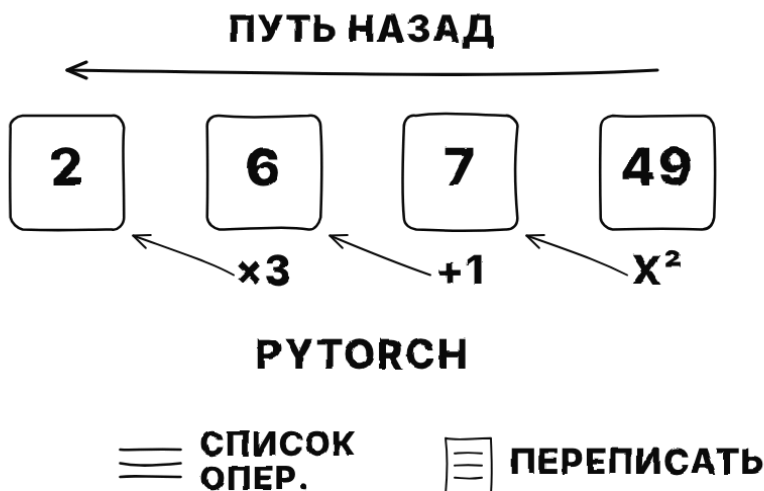
3. Функция активации применяется к слою из пятисот чисел. Сколько элементов было бы в её полном входном якобиане и сколько из них были бы нулями?

Подсказка: В таблице одна строка на выход и один столбец на вход, и ненулевая только диагональ.

Ответ: Двести пятьдесят тысяч элементов, из которых двести сорок девять тысяч пятьсот — нули; наклоны хранят только пятьсот элементов на диагонали, — вот почему произведение вместо этого просто масштабирует сопряжённую переменную.

Записки, оставленные по пути

Пройдите цепочку на калькуляторе в последний раз, но на стикерах. Запишите исходную двойку на стикере. Каждый раз, нажимая кнопку, записывайте результат на новый стикер, а под ним помечайте, чем он получен: «умножить на три, из предыдущего стикера», «плюс один, из предыдущего стикера», «в квадрат, из предыдущего стикера». Дойдя до сорока девяти, вы найдёте дорогу назад, ничего не запоминая: каждый стикер говорит, какая операция его дала и откуда она взялась. Эта цепочка стикеров — почти буквально то, как обратный режим устроен в PyTorch.

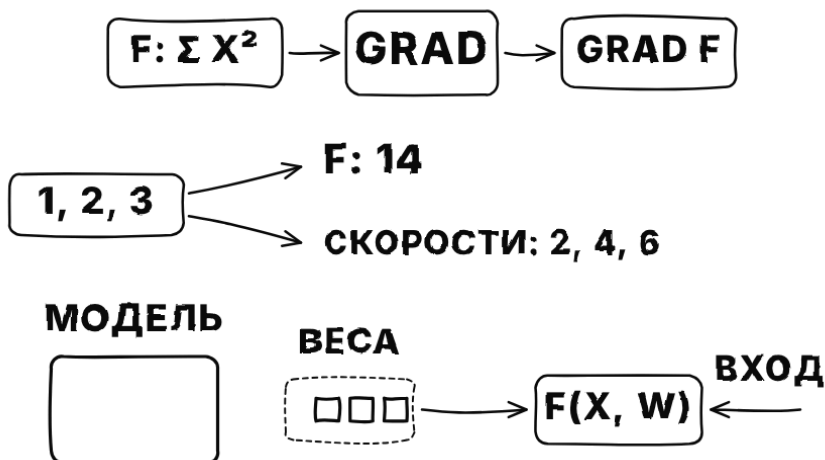


Цепочка стикеров

Сначала книга говорит, что систему обратного режима можно построить многими способами: от ведения текущего списка всех выполненных операций, как делает TensorFlow, до переписывания собственного кода программы во вторую программу, которая вычисляет скорости. Затем она рассматривает два стиля, которые мы уже встречали: функции и объекты.

В функциональной библиотеке вроде JAX всё — функция, и дифференцирование тоже. Произведение вектора на якобиан становится функцией, которая делает функции. Вы даёте ей функцию и точку, на которую нужно посмотреть; она возвращает вам новую функцию, которая берёт строку с одним числом на

каждый выход и возвращает эту строку, умноженную на якобиан в вашей точке, — по одному числу на каждый вход. Для функции с одним выходом есть кое-что ещё проще: дайте ей функцию, и она вернёт вторую функцию, которая вычисляет градиент — скорости по каждому входу — в любой точке, какой захотите. Пример книги — сумма квадратов десяти чисел: функция-градиент, получив десять чисел, возвращает десять скоростей. Попробуйте с тремя числами — один, два и три: сумма квадратов равна четырнадцати, а скорости равны двум, четырём и шести, удвоенному каждому числу, потому что каждый квадрат растёт вдвое быстрее своего числа. JAX предоставляет такие функции, делающие функции, в готовом виде, PyTorch повторяет тот же интерфейс в собственном модуле, а результат можно скомпилировать в оптимизированный граф; хитро сочетая два вида произведений, можно получить даже вторые скорости — скорости скоростей.



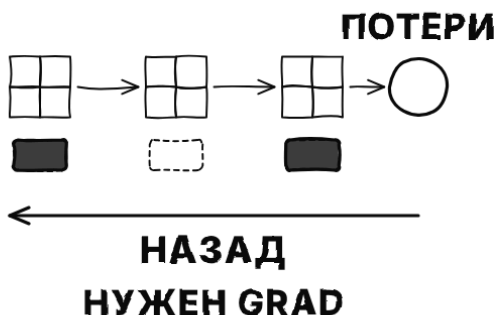
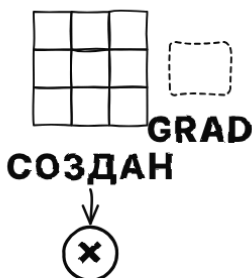
Функция, которая делает функции

Однако настоящие модели в основном пишут как объекты, хранящие свои веса внутри. Один выход — очистить их: вынуть веса из объекта в простую коллекцию и вызывать модель как функцию, которая получает вход и эти веса рядом друг с другом, — именно этого и хочет функциональный стиль.

PyTorch умеет делать эту работу и напрямую — со стикерами. Каждый тензор — так PyTorch называет таблицу чисел — несёт две дополнительные вещи: пустую ячейку для своего градиента и пометку об операции, которая его создала. Рисунок книги показывает это слева: тензор, его пустая ячейка для градиента и стрелка к операции, из которой он получился. Когда вы просите обратный проход для одного-единственного числа, например для

потерь на пакете фотографий, PyTorch идёт по пометкам назад через вычислительный граф, применяя по пути обратное правило каждой операции. Справа на рисунке ячейки заполняются: градиенты хранятся внутри самих тензоров, но только в тех, владеец которых сказал, что им нужен градиент, чтобы не тратить память на остальные.

ТЕНЗОР



☰ MICROGRAD

Тензоры носят свои пометки с собой

Книга признаёт, что это лишь набросок, и отсылает за подробностями к официальной документации. Её самый тёплый совет — построить небольшую систему обратного режима самому, и она называет одну из многих учебных версий в сети, *micrograd*, которая уместается в нескольких коротких файлах.

Опыт

Запишите цепочку «два, умножить на три, плюс один, в квадрат» на стикерах, и пусть каждый стикер говорит, чем он получен. Теперь начните с последнего стикера и идите назад, записывая на каждом, насколько итоговые сорок девять сдвигаются вместе с ним: четырнадцать на семёрке, четырнадцать на шестёрке, сорок два на двойке. Вы только что вручную сделали то, что делает PyTorch, когда вы просите у него обратный проход.

Запомним

Обратный режим можно построить многими способами — от текущего списка операций до переписывания кода программы. В функциональном стиле JAX дифференцирование — это функция, которая делает функции: получив функцию и точку, она возвращает произведение вектора на якобиан, а для единственного выхода возвращает функцию-градиент — так у суммы квадратов единицы, двойки и тройки скорости равны двум, четырём и шести. Объекты, хранящие веса внутри, можно очистить, превратив в такие функции. PyTorch же даёт каждому тензору пометку об операции, которая его создала, и пустую ячейку для градиента, по запросу проходит по пометкам назад и заполняет ячейки тех тензоров, которые попросили градиенты.

Вопросы

1. Что вы получаете в функциональном стиле JAX, когда просите произведение вектора на якобиан функции в точке?
 - a. Полный якобиан функции в этой точке, в виде таблицы
 - b. Новую функцию, которая берёт строку с одним числом на каждый выход и возвращает эту строку, умноженную на якобиан
 - c. Значение функции в этой точке, вычисленное дважды
 - d. Список операций, которые выполнила функция, для дальнейшего использования
2. Откуда PyTorch знает, как пройти вычисление в обратном направлении?
 - a. Он заново выполняет прямой проход, слегка сдвигая каждый вес
 - b. Он переписывает код модели во вторую программу ещё до начала обучения
 - c. Каждый тензор несёт пометку об операции, которая его создала, и обратный проход идёт по этим пометкам
 - d. Он просит у объекта модели формулу её скоростей
3. Почему PyTorch заполняет ячейку градиента только у тех тензоров, которые попросили градиент?
 - a. Чтобы не тратить память на градиенты, которые никому не нужны

- b. Потому что у остальных тензоров нет создавшей их операции
 - c. Потому что обратный проход не может пройти через тензоры без градиента
 - d. Чтобы прямой проход шёл быстрее
4. Что значит очистить модель?
- a. Убрать функции активации, чтобы модель стала линейной
 - b. Удалить веса с нулевым градиентом
 - c. Скомпилировать модель в оптимизированный граф
 - d. Вынуть веса из объекта, чтобы модель можно было вызвать как функцию от входа и весов

Ответы

1. **b)** Дифференцирование там — это функция, которая делает функции: получив функцию и точку, она возвращает новую функцию, умножающую любую строку на якобиан в этой точке.
2. **c)** Каждый тензор хранит пометку о создавшей его операции, поэтому, когда вы просите обратный проход для одного числа, PyTorch идёт по пометкам назад через граф.
3. **a)** Градиенты хранятся внутри самих тензоров, и только там, где владелец сказал, что они нужны, — чтобы не тратить память на остальные.
4. **d)** Объекты хранят веса внутри; очистка вынимает их в простую коллекцию, чтобы модель можно было вызвать по-функциональному — вход и веса рядом.

Разобранные примеры

Градиент суммы квадратов Пример книги просит функцию-градиент суммы квадратов десяти чисел. Сколько чисел возвращает эта функция-градиент и что она вернёт для трёх чисел — три, минус один и два, — если взять три числа вместо десяти?

У функции один выход, поэтому её градиент содержит по одной скорости на каждый вход.

Если на входе десять чисел, функция-градиент возвращает десять скоростей.

Каждый квадрат растёт вдвое быстрее своего числа, а остальные квадраты не меняются, когда сдвигают одно число.

Значит, каждая скорость вдвое больше своего числа: шесть, минус два и четыре.

Ответ: Десять чисел для десяти входов; для трёх, минус единицы и двух скорости равны шести, минус двум и четырём.

Задачи

1. Каковы скорости функции-градиента суммы квадратов в числах одна вторая и четыре?

Подсказка: Во сколько раз быстрее своего числа растёт квадрат?

Ответ: Один и восемь: каждая скорость вдвое больше своего числа.

2. Функция принимает шесть чисел и выдаёт два. Вы просите у функциональной библиотеки её произведение вектора на якобиан в некоторой точке. Сколько чисел принимает полученная функция и сколько возвращает?

Подсказка: Строка, которую она принимает, содержит по одному числу на каждый выход исходной функции.

Ответ: Она принимает два числа, по одному на выход, и возвращает шесть, по одному на вход: строку, умноженную на якобиан размером два на шесть в этой точке.

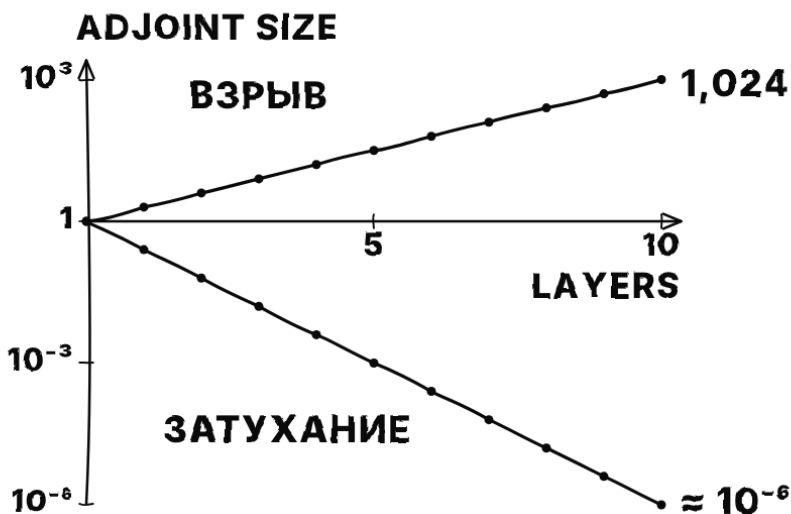
3. Объясните своими словами, как цепочка стикеров позволяет выполнить обратный проход, не помня программу.

Подсказка: Подумайте, что каждый стикер говорит о том, откуда взялось его число.

Ответ: Каждый стикер записывает своё значение, создавшую его операцию и стикер, из которого он получен. Начав с итогового результата, можно идти по этим связям назад и на каждом стикере применять обратное правило его операции, чтобы найти, насколько итоговый результат сдвигается вместе с ним. PyTorch делает то же самое с пометками, которые прикрепляет к каждому тензору.

Четверть, десять раз подряд

Наберите на калькуляторе единицу и умножьте её на четверть десять раз подряд. Вы, возможно, ждёте чего-то маленького, но не настолько: на экране примерно одна миллионная. Теперь сбросьте, снова наберите единицу и умножьте на два тридцать раз подряд: на экране больше миллиарда. Эти две серии нажатий — вся суть этой части.



Сжатие и рост, слой за слоем

Вспомните, что делает функция активации при обратном проходе: каждый элемент сопряжённой переменной умножается

наклон функции активации при соответствующем входе. В глубокой сети функция активации стоит в каждом слое, поэтому на обратном пути сопряжённая переменная умножается на такой наклон снова и снова, по разу на слой. Если наклоны всюду меньше единицы, сопряжённая переменная сжимается с каждым слоем, как ваша четверть, и после достаточного числа слоёв скорости для ранних весов практически равны нулю: это проблема затухающего градиента. Если наклоны всюду больше единицы, сопряжённая переменная растёт с каждым слоем, как ваше удвоение, и скорости уносятся в бесконечность: это проблема взрывающегося градиента.

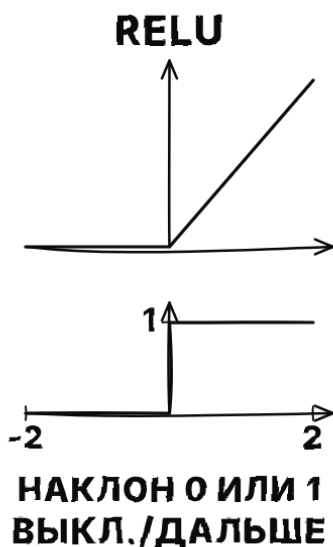
Это не просто курьёзы, потому что компьютер хранит числа с ограниченной точностью, обычно тридцатью двумя двоичными разрядами или меньше. Число, которое достаточно сжалось, просто округляется до нуля, а число, которое достаточно выросло, больше не помещается, и при многих слоях это происходит быстро. Книга добавляет любопытный поворот: из-за этих крошечных эффектов округления стопка чисто линейных слоёв на компьютере не вполне линейна, и этим даже пользовались, чтобы обучать глубокие сети совсем без функций активации.

Сигмоида показывает, как появляются затухающие градиенты. В прошлом это была распространённая функция активации, сглаженная ступенька: S-образная кривая, плавно поднимающаяся от нуля до единицы. Её наклон в любой точке равен её значению, умноженному на единицу минус это значение. Поскольку значение лежит между нулём и единицей, это произведение наибольшее, когда значение равно половине, и тогда оно равно половине от половины, четверти; при значении

девять десятых оно лишь девять сотых. Рисунок книги показывает S-образную кривую и рядом её наклон — невысокий бугорок, который никогда не поднимается выше четверти. Значит, каждый сигмоидный слой уменьшает сопряжённую переменную по меньшей мере до четверти, а десять таких слоёв — примерно до миллионной, как в вашей первой серии на калькуляторе.



$$\frac{1}{4} \times 10 \approx 10^{-6}$$



Наклоны сигмоиды и наклоны ReLU

Может ли у функции активации быть наклон, всюду равный ровно единице? Только у функции, которая вообще ничего не делает и возвращает свой вход без изменений, — и тогда слои вокруг неё сливаются в один линейный слой. Нам нужно что-то достаточно линейное, чтобы наклоны оставались безвредными, и

при этом достаточно изогнутое, чтобы слои оставались раздельными. ReLU подходит: её наклон равен нулю для отрицательных входов и единице для положительных. Её рисунок показывает линию, лежащую ровно вдоль нуля, а затем поднимающуюся под сорок пять градусов, и рядом её наклон — ступеньку от нуля до единицы. Так что каждый элемент сопряжённой переменной либо выключается, что оставляет много нулей и экономит работу, либо передаётся дальше без изменений, и ничто не сжимается и не взрывается.

Есть и бонус для памяти. ReLU оставляет положительные числа и превращает отрицательные в ноль, поэтому её выход говорит так же хорошо, как и вход, где наклон равен единице, а где нулю. Поэтому обратный проход может работать по выходу, и вход хранить не нужно: слой может затереть его при прямом проходе, сэкономяв немного памяти и не испортив ни одной скорости, и PyTorch предлагает ровно это как опцию работы на месте.

Запомним

Каждая функция активации на обратном пути умножает сопряжённую переменную на свой наклон, так что в глубокой сети эти наклоны перемножаются. Наклоны меньше единицы заставляют скорости ранних слоёв затухать, наклоны больше единицы — взрываться, а ограниченная точность компьютеров превращает и то и другое в настоящие сбои. Наклон сигмоиды никогда не больше четверти, так что десять сигмоидных слоёв могут сжать сопряжённую переменную до миллионной. Наклон ReLU равен нулю или единице, так что сопряжённая переменная либо выключается, либо передаётся без изменений, а поскольку выход показывает, где наклон равен единице, вход можно затереть ради экономии памяти.

Вопросы

1. Почему глубокой сети с сигмоидными функциями активации грозят затухающие градиенты?
 - a. Потому что выход сигмоиды может быть больше единицы
 - b. Потому что наклон сигмоиды никогда не больше четверти, а эти наклоны перемножаются слой за слоем
 - c. Потому что сигмоида выключает каждый отрицательный вход
 - d. Потому что у сигмоиды нет наклона в нуле

2. Почему затухающие и взрывающиеся градиенты — настоящие сбои, а не просто очень маленькие или очень большие числа?
 - a. Потому что скорости тогда указывают не в ту сторону
 - b. Потому что потери перестают быть одним числом
 - c. Потому что компьютеры хранят числа с ограниченной точностью, так что крошечные числа становятся нулём, а огромные больше не помещаются
 - d. Потому что тогда прямой проход приходится запускать дважды
3. Почему бы не взять функцию активации с наклоном, всюду равным ровно единице?
 - a. Потому что такой наклон есть только у функции, возвращающей вход без изменений, и тогда слои вокруг неё сливаются в один линейный слой
 - b. Потому что такая функция заставила бы каждый градиент взрываться
 - c. Потому что её наклон был бы равен нулю для отрицательных входов
 - d. Потому что компьютер не может хранить единицу точно
4. Почему слой ReLU может затереть свой вход во время прямого прохода?
 - a. Потому что у ReLU вообще нет обратного правила
 - b. Потому что вход больше не нужен ни одному слою
 - c. Потому что затираание делает скорости точнее
 - d. Потому что её выход так же хорошо, как вход, показывает, где наклон равен единице, а где нулю

Ответы

1. **b)** На обратном пути сопряжённая переменная в каждом слое умножается на наклон функции активации, а наклон сигмоиды не больше четверти, так что десять слоёв могут сжать её примерно до миллионной.
2. **c)** Числа хранятся тридцатью двумя двоичными разрядами или меньше, поэтому число, которое достаточно сжалось, округляется до нуля, а число, которое достаточно выросло, больше не помещается.
3. **a)** Единственная функция с наклоном, всюду равным единице, вообще ничего не делает, и без изгиба линейные слои сливаются в один; ReLU достаточно линейна, но всё же изогнута.
4. **d)** ReLU оставляет положительные числа и обнуляет отрицательные, поэтому обратный проход может прочесть наклоны по выходу, а вход можно затереть ради экономии памяти.

Разобранные примеры

Наибольший наклон сигмоиды Наклон сигмоиды равен её значению, умноженному на единицу минус это значение. Каков наклон, когда значение равно половине, и когда оно равно восьми десятым? И во сколько раз самое большее пять сигмоидных слоёв могут изменить элемент сопряжённой переменной?

При значении одна вторая: половина на половину — это четверть.

При значении восемь десятых: восемь десятых на две десятых — шестнадцать сотых, меньше четверти.

Наклон наибольший при одной второй, так что он никогда не превышает четверти.

Через пять слоёв элемент сопряжённой переменной умножается самое большее на четверть пять раз: четверть от четверти от

четверти от четверти от четверти — одна тысяча двадцать четвёртая часть.

Ответ: Четверть и шестнадцать сотых; пять слоёв сжимают элемент сопряжённой переменной самое большее примерно до одной тысячной.

Задачи

1. Каков наклон сигмоиды, когда её значение равно двум десятым, и когда оно равно девяноста девяти сотым?

Подсказка: Умножьте значение на единицу минус значение.

Ответ: Шестнадцать сотых и примерно одна сотая: две десятых на восемь десятых, и девяносто девять сотых на одну сотую, то есть девяносто девять десятитысячных.

2. Элемент сопряжённой переменной, равный единице, проходит назад через три слоя, у функций активации которых наклон везде полтора. Чем он станет? А через три слоя ReLU, у которых все входы были положительными?

Подсказка: На обратном пути каждый слой умножает элемент сопряжённой переменной на свой наклон.

Ответ: Три и три восьмыми, ведь полтора, умноженное на себя трижды, — это три целых триста семьдесят пять тысячных, начало взрыва; через три слоя ReLU с положительными входами он остаётся единицей, потому что каждый наклон равен единице.

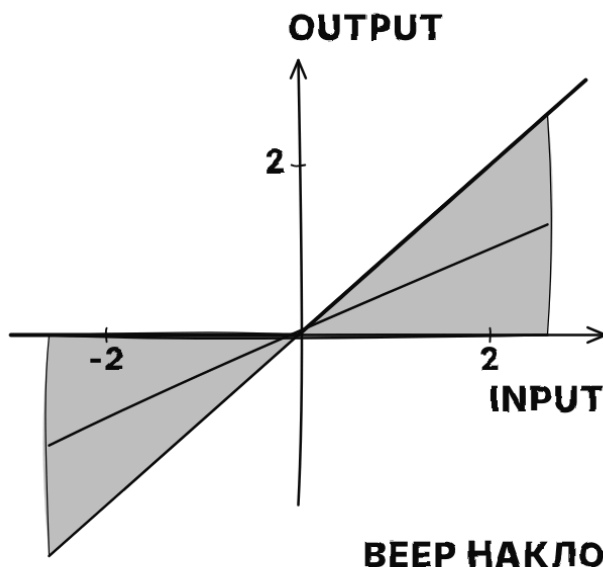
3. Объясните своими словами, почему ReLU избегает и затухания, и взрыва градиента для тех элементов, которые она пропускает.

Подсказка: Каковы единственные два наклона, которые может иметь ReLU?

Ответ: Наклон ReLU равен нулю для отрицательных входов и единице для положительных. Элемент либо выключается, либо умножается ровно на единицу, так что пропущенные элементы никогда не сжимаются и не увеличиваются, сколько бы ни было слоёв.

Линейка в углу

Нарисуйте ReLU на листе бумаги: слева линия, лежащая ровно вдоль горизонтальной оси, а затем от угла в нуле — линия, поднимающаяся вправо под сорок пять градусов. Теперь положите через угол карандаш и медленно наклоняйте его. Горизонтальный — он везде остаётся под рисунком; поднятый до сорока пяти градусов — всё ещё под ним; наклонённый чуть круче — он пересекает поднимающуюся ветвь, а наклонённый вниз — пересекает ровную ветвь слева. Значит, в углу не один наклон, а целый их веер — любой наклон от горизонтального до сорока пяти градусов.

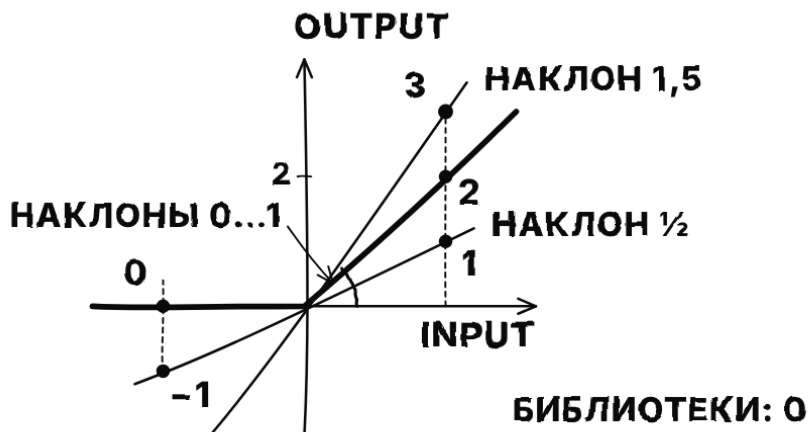


Карандаш, наклонённый в углу

Это та маленькая деталь, которую книга до сих пор откладывала. У ReLU нет обычного наклона в нуле, поэтому сеть, построенная из ReLU, не везде гладкая, и правила предыдущих частей в этой точке как будто ломаются. Практический ответ: это почти никогда не важно. Обучение начинается со случайных весов и движется маленькими случайными шагами, так что шанс, что вход попадёт ровно в ноль, практически нулевой, а на волосок в любую сторону наклон совершенно определён.

Более строгий ответ использует веер, который вы только что нарисовали. Для функции, изгибающейся вверх, — выпуклой функции — субградиент в точке это наклон прямой через эту точку, которая везде остаётся ниже всей функции. Там, где

функция гладкая, такая прямая только одна — обычная касательная, и её наклон — обычная скорость. В углу таких прямых много, и совокупность их наклонов называется субдифференциалом. Для ReLU субдифференциал равен нулю при отрицательных входах, единице при положительных, а в нуле это любое число от нуля до единицы. Проверьте один из них: возьмите прямую через угол с наклоном одна вторая. При входе два она даёт единицу, а ReLU даёт два, так что прямая ниже; при минус двух она даёт минус единицу, а ReLU даёт ноль, так что она снова ниже. Наклон полтора не проходит: при входе два он даёт три, выше двойки ReLU. На практике большинство библиотек просто выбирают ноль как наклон ReLU в нуле, а обучение, которое на каждом шаге следует одному выбранному субградиенту, называется субградиентным спуском.



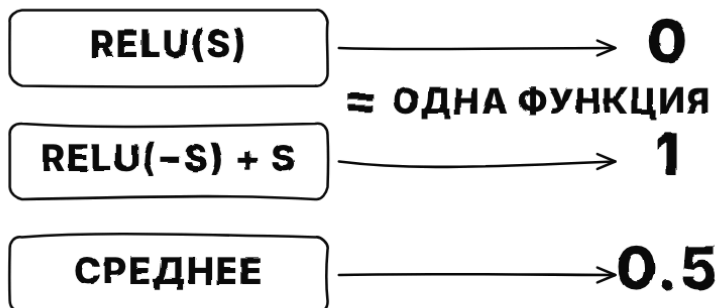
Проверка субградиента

Книга признаёт, что история ещё запутаннее. Для функций, которые не изгибаются вверх, субградиента может вообще не быть, и математики используют более свободные версии, смотрящие только на малую окрестность точки, — например, субдифференциал Кларка. И само автоматическое дифференцирование может здесь споткнуться, потому что два способа записать одну и ту же функцию могут дать в углу разные наклоны. Пример книги: возьмите ReLU от минус числа и прибавьте само число обратно; для любого числа это снова даёт ровно ReLU. Усредните это с обычной ReLU — и у вас по-прежнему ReLU. И всё же в нуле обратный проход PyTorch даёт ноль для первой, единицу для второй и одну вторую для третьей: три версии одной функции — три ответа, и поэтому для

доказательства правильности таких программ нужны более тонкие правила цепочки.

○ КЛАРК

НАКЛОН В 0



Одна функция — три ответа

Глава заканчивается тем, что отправляет вас практиковаться. Постройте заново небольшую систему обратного режима — например, разобрав `micrograd`, которая работает с отдельными числами; как только граф ветвится и сливается, его шаги нужно сначала расставить в порядке, учитывающем, кто кого питает, — это дешёвая сортировка, — иначе обратный проход пойдёт по неверным путям. Затем напишите собственный примитив в `PyTorch`, задав его прямое вычисление и его произведения вектора на якобиан, — например, одну из обучаемых функций активации из предыдущей главы, хотя `PyTorch` мог бы вычислить

его обратный проход и сам. В JAX есть те же шаги: учебная версия JAX под названием autodidax, собственные обратные правила для новых примитивов и сборник рецептов для продвинутых задач вроде вторых скоростей.

На этом глава закрывается. Начав с цепочки на калькуляторе и её сорока двух, вы проследили скорости обучающей программы вперёд и назад, увидели, почему побеждает обратный путь, почему он никогда не строит своих таблиц и как обращаться с безвредными наклонами ReLU и даже с её единственным острым углом.

Запомним

У ReLU в нуле угол, но при обучении вход почти никогда не попадает точно туда, так что обычный наклон почти всегда определён. Субградиент выпуклой функции — это наклон прямой через точку, которая остаётся ниже всей функции; в углу их множество, субдифференциал, для ReLU в нуле — это любой наклон от нуля до единицы, и библиотеки обычно выбирают ноль. Для других функций субградиентов может и не быть, а равносильные записи ReLU заставляют PyTorch возвращать в нуле ноль, единицу или одну вторую, так что в доказательствах угол требует осторожности. Книга заканчивается призывом к практике: построить заново небольшую систему вроде micrograd, упорядочив шаги ветвящегося графа, и писать примитивы с собственными произведениями вектора на якобиан в PyTorch или JAX.

Вопросы

1. Почему угол ReLU в нуле на практике почти никогда не важен?
 - a. Потому что библиотеки перед обучением убирают из входов все нули
 - b. Потому что обучение начинается со случайных весов и движется маленькими случайными шагами, так что вход почти никогда не попадает ровно в ноль
 - c. Потому что наклон ReLU в нуле всегда равен единице
 - d. Потому что обратный проход целиком пропускает слой ReLU
2. Какие наклоны являются субградиентами ReLU в нуле?
 - a. Только ноль — значение, которое выбирают библиотеки
 - b. Только одна вторая — посередине между двумя ветвями
 - c. Любой наклон от нуля до единицы, потому что такие прямые через угол остаются ниже всей ReLU
 - d. Любой наклон от минус единицы до единицы
3. Что показывает пример книги с тремя способами записать ReLU?
 - a. Что равносильные записи одной и той же функции могут заставить автоматическое дифференцирование возвращать в углу разные наклоны
 - b. Что два из трёх способов на самом деле не ReLU
 - c. Что PyTorch неправильно вычисляет ReLU для положительных входов

- d. Что у ReLU нет субградиентов
4. Что должна сделать небольшая система обратного режима вроде `micrograd` перед обратным проходом, когда граф программы ветвится и сливается?
- a. Заменить каждую ReLU сигмоидой
 - b. Дважды запустить прямой проход, чтобы проверить результаты
 - c. Построить полный якобиан каждого шага
 - d. Расставить шаги в порядке, учитывающем, кто кого питает, — дешёвой сортировкой

Ответы

1. **b)** При случайных начальных весах и маленьких случайных шагах шанс, что вход попадёт ровно в ноль, практически нулевой, а на волосок в любую сторону наклон определён.
2. **c)** Прямая через угол везде остаётся ниже ReLU ровно тогда, когда её наклон лежит между горизонталью и сорока пятью градусами, так что субдифференциал в нуле — это любое число от нуля до единицы.
3. **a)** Все три — одна и та же функция, но обратный проход PyTorch даёт в нуле ноль, единицу и одну вторую, поэтому в доказательствах угол требует более тонких правил цепочки.
4. **d)** Как только граф ветвится и сливается, его шаги нужно упорядочить, чтобы обратный проход не пошёл по неверным путям; `micrograd` делает это дешёвой сортировкой.

Разобранные примеры

Три ReLU — три наклона Вторая версия — это ReLU от минус числа плюс само число; третья — среднее обычной ReLU и второй

версии. Проверьте, что вторая версия равна ReLU в точках три и минус три, и найдите наклоны, которые обратный проход PyTorch даёт в нуле, зная, что наклон обычной ReLU в нуле он считает равным нулю.

В точке три: ReLU от минус трёх равна нулю, плюс три даёт три — столько же, сколько ReLU от трёх.

В точке минус три: ReLU от трёх равна трём, плюс минус три даёт ноль — столько же, сколько ReLU от минус трёх.

В нуле наклон обычной ReLU берётся равным нулю.

Наклон второй версии — это минус наклон ReLU, из-за минуса внутри, плюс единица от прибавленного числа: минус ноль плюс один — это единица.

Третья версия — среднее двух первых, так что её наклон — среднее нуля и единицы.

Ответ: Вторая версия равна ReLU в обеих точках; наклоны в нуле — ноль, единица и одна вторая.

Проверка субградиента Является ли наклон одна четвёртая субградиентом ReLU в нуле? Проверьте прямую через угол с этим наклоном при входах четыре и минус четыре.

При четырёх прямая даёт четверть от четырёх, то есть единицу; ReLU даёт четыре, значит, прямая ниже.

При минус четырёх прямая даёт минус единицу; ReLU даёт ноль, значит, прямая снова ниже.

Прямая через угол с наклоном между нулём и единицей остаётся ниже на обеих ветвях, а значит, ниже везде.

Ответ: Да, одна четвёртая — субградиент ReLU в нуле.

Задачи

1. Является ли наклон восемь десятых субградиентом ReLU в нуле? Проверьте прямую через угол при входах пять и минус пять.

Подсказка: Сравните значение прямой со значением ReLU при каждом входе.

Ответ: Да: при пяти прямая даёт четыре, ниже пятёрки ReLU; при минус пяти она даёт минус четыре, ниже нуля ReLU; и восемь десятых лежит между нулём и единицей.

2. Является ли наклон минус одна вторая субградиентом ReLU в нуле? Найдите вход, при котором прямая через угол поднимается выше ReLU.

Подсказка: Посмотрите налево, где ReLU лежит ровно вдоль нуля.

Ответ: Нет: при минус двух прямая даёт единицу, выше нуля ReLU, так что она не остаётся ниже всей функции.

3. Объясните своими словами, что такое субградиент и почему в гладкой точке он только один.

Подсказка: Представьте карандаш, положенный через точку и наклоняемый.

Ответ: Для функции, изгибающейся вверх, субградиент в точке — это наклон прямой через эту точку, которая остаётся ниже всей функции. В гладкой точке так может только касательная, поэтому единственный субградиент — обычный наклон; в углу карандаш можно наклонять через целый веер положений, что даёт много субградиентов.