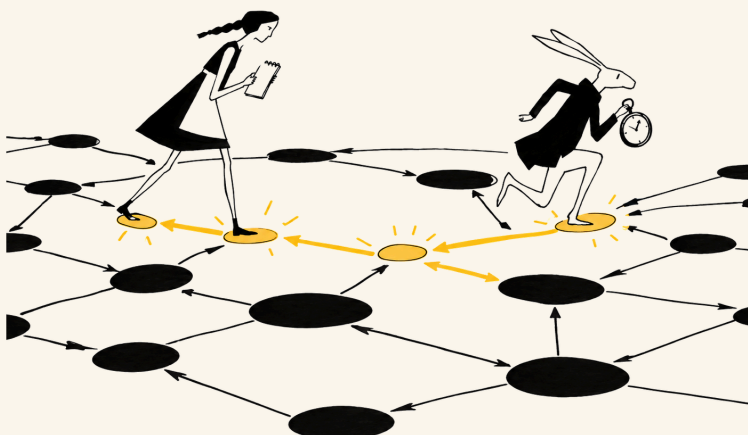


Приключения Алисы в Стране дифференцируемых чудес



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

Программа из маленьких шагов

Разворачиваем таблицы и ставим задачу

Три способа получить производную

Разворачиваем цепочку

Экономим память и просим только произведения

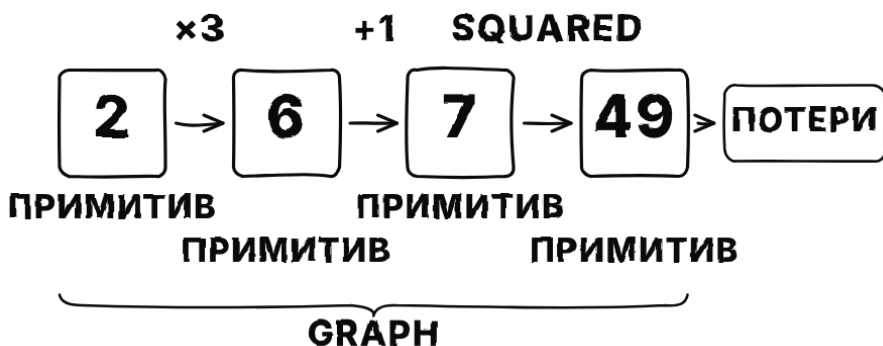
Обратный шаг слоя

Фреймворки — и почему побеждает ReLU

Угол ReLU

Программа из маленьких шагов

Откройте электронную таблицу и постройте крошечную цепочку ячеек. В первую ячейку введите число, скажем 2; во вторую — формулу, которая утраивает первую; в третью — формулу, которая прибавляет 1 ко второй; в последнюю — формулу, которая возводит третью в квадрат. В последней ячейке появится 49, потому что 2, умноженное на 3, — это 6, плюс 1 — это 7, а 7 в квадрате — 49. Теперь задайте вопрос поострее: если чуть-чуть увеличить первую ячейку, насколько быстро будет меняться последняя? Можно проверить вручную, набрав 2,01 и посмотрев на результат, но таблица и так знает каждый сделанный шаг, а каждый шаг по отдельности легко продифференцировать.



Цепочка маленьких шагов

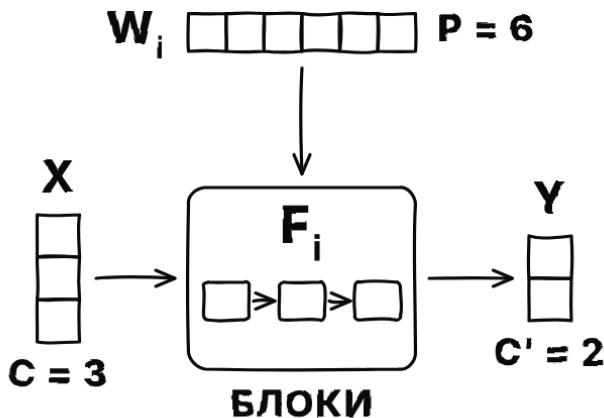
Именно этой задаче посвящена эта глава книги. Когда обучают чат-ассистента, огромная программа из таких шагов превращает предложение в одно число — потери, которые показывают, насколько ответ был неверен. Чтобы улучшить модель, компьютеру нужен градиент: как меняются потери, если чуть сдвинуть каждое из миллионов настраиваемых чисел. Задачу эффективно получать эти градиенты для любой программы книга называет автоматическим дифференцированием, сокращённо AD. У метода, к которому она ведёт, два имени: обратное распространение ошибки — у тех, кто занимается нейронными сетями, и автоматическое дифференцирование в обратном режиме — у специалистов по информатике.

Автор предлагает представить вычислительный граф — набор маленьких элементарных операций, который получается, если на самом деле выполнить программу, как ячейки нашей таблицы. Каждая из этих операций называется примитивом. Для краткости книга следит за одной цепочкой примитивов, идущих друг за другом, и отмечает, что всё переносится на ветвящиеся графы без циклов и даже на более общие.

Поначалу кажется, что задача уже решена. Цепное правило для якобианов, знакомое по предыдущим главам книги, говорит, что производная композиции функций — это просто произведение их матриц Якоби, по одной на каждый шаг. Якобиан — это таблица частных производных: одна строка на каждое выходное число, один столбец на каждое входное, и каждая клетка говорит, насколько сдвинется этот выход, если чуть сдвинуть этот вход. Так зачем же целая глава? Потому что вычислить это произведение умно, не тратя впустую время и память, — вот настоящая трудность, а алгоритм, который с ней справляется, — краеугольный камень нейронных сетей и дифференцируемого программирования вообще. Автор добавляет, что, поняв его, вы поймёте и то, как устроены библиотеки вроде TensorFlow, PyTorch и JAX и почему они различаются.

Вот как книга вводит свои строительные блоки. Каждый примитив записывается как $y = f_i(x, w_i)$. Словами: итый примитив принимает входной вектор x и вектор настраиваемых чисел w_i , свои веса, и выдаёт выходной вектор y . Размеры записываются через тильду: $x \sim (c_i)$, $w_i \sim (p_i)$ и $y \sim (c'_i)$. Откуда берётся такая форма? Представьте линейную проекцию в слое сети: вход приходит от предыдущего шага, веса принадлежат самому шагу, а результат передаётся дальше. В числах: y

маленького слоя, который принимает 3 числа, выдаёт 2 числа и имеет 6 весов, с равно 3, c' равно 2, а p равно 6.

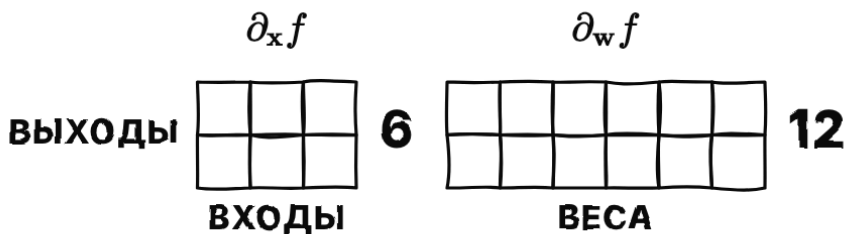


Один примитив: вход, веса, выход

Определение нарочно сделано свободным. Примитивом может быть одно умножение матриц, целый слой того вида, что строился в предыдущей главе, например полносвязный слой с функцией активации, или даже целый блок или модель. Блоки внутри блоков: такая рекурсивная составяемость — ключевое свойство программирования, и книга её сохраняет.

О каждом примитиве предполагается только одно: мы умеем вычислять его частные производные по двум его аргументам. Первая — входной якобиан, $\partial_x[f(x, w)] \sim (c', c)$: насколько

сдвигается каждый выход, если чуть сдвинуть каждый вход. Вторая — якобиан по весам, $\partial_{\mathbf{w}}[f(\mathbf{x}, \mathbf{w})] \sim (c', p)$: насколько сдвигается каждый выход, если чуть сдвинуть каждый вес. Размеры следуют из правила: одна строка на каждый выход и один столбец на каждое число, которое сдвигают. Для нашего маленького слоя входной якобиан — таблица из 2 строк и 3 столбцов, 6 чисел, а якобиан по весам — из 2 строк и 6 столбцов, 12 чисел.



Два якобиана слоя

Эти допущения разумны, говорит автор, потому что книга ограничивается дифференцируемыми моделями. Прimitives, который непрерывен, но в нескольких точках не имеет производной, например ReLU, оставляющая положительные

числа и обнуляющая отрицательные, встраивается позже с помощью инструмента под названием субградиент. Операции, которые вовсе не дифференцируемы, например случайная выборка или отсечение по порогу, тоже можно учесть, если ослабить их градиент или взять эквивалентную оценку; это автор оставляет для следующего тома.

Запомним

Для обучения сети нужен градиент одного числа, потерь, по каждому весу, а автоматическое дифференцирование — это задача эффективно вычислить их все для любой программы. Выполненная программа — это цепочка маленьких примитивов, каждый из которых записывается как $y = f_i(\mathbf{x}, \mathbf{w}_i)$: вход, собственные веса, выход. Для каждого примитива нужны только две таблицы частных производных: входной якобиан размера s' на s и якобиан по весам размера s' на p . Цепное правило уже говорит, что ответ — это произведение якобианов; вся трудность, и вся причина, почему обратное распространение ошибки так важно, в том, чтобы вычислить это произведение дёшево.

Вопросы

1. Цепное правило уже говорит, что производная композиции — это произведение якобианов. Зачем же книге целая глава об автоматическом дифференцировании?

- a. Потому что цепное правило не выполняется для нейронных сетей
 - b. Потому что настоящая трудность — вычислить это произведение умно, не тратя впустую время и память
 - c. Потому что якобианы примитивов обычно неизвестны
 - d. Потому что потери — вектор, а цепному правилу нужен скаляр
2. Примитив принимает вход из s чисел, имеет r весов и выдаёт выход из s' чисел. Какую форму имеет его якобиан по весам?
- a. s строк и r столбцов
 - b. s' строк и s столбцов
 - c. s' строк и r столбцов
 - d. r строк и s' столбцов
3. Что книга предполагает известным о каждом примитиве?
- a. Как вычислять его частные производные по входу и по весам
 - b. Только как вычислять его выход, ничего о производных
 - c. Его полную символьную формулу, чтобы с ней могла работать система компьютерной алгебры
 - d. Что это одно умножение матриц
4. Как книга вписывает в эту схему примитив вроде ReLU, у которого есть точка без производной?
- a. Она отказывается от ReLU и использует только гладкие функции

- b. Она откладывает ReLU до следующего тома вместе с выборкой
- c. Для точек без производной она использует инструмент под названием субградиент
- d. Она заменяет ReLU численным приближением её производной

Ответы

1. **b)** В принципе ответ — это произведение якобианов; трудность, и причина, почему так важно обратное распространение ошибки, в том, чтобы вычислять это произведение эффективно.
2. **c)** У якобиана одна строка на каждый выход и один столбец на каждое число, которое сдвигают; для якобиана по весам сдвигают p весов, поэтому его форма — c' на p .
3. **a)** Единственное допущение — каждый примитив приходит вместе со своим входным якобианом и якобианом по весам.
4. **c)** Непрерывные примитивы с несколькими точками недифференцируемости встраиваются с помощью субградиентов; до следующего тома откладываются только полностью недифференцируемые операции вроде выборки.

Разобранные примеры

Размеры двух якобианов слоя Примитив $y = f(x, w)$ принимает вход из 3 чисел, имеет 6 весов и возвращает 2 числа. Сколько элементов в его входном якобиане и в якобиане по весам?

Считываем размеры: $c = 3$ входа, $p = 6$ весов, $c' = 2$ выхода.

У входного якобиана одна строка на каждый выход и один столбец на каждый вход, поэтому его форма — $(c', c) = (2, 3)$.

Это $2 \times 3 = 6$ элементов, и каждый говорит, насколько сдвигается один выход, если чуть сдвинуть один вход.

У якобиана по весам одна строка на каждый выход и один столбец на каждый вес, поэтому его форма — $(c', p) = (2, 6)$, то есть $2 \times 6 = 12$ элементов.

Ответ: Во входном якобиане 6 элементов, в якобиане по весам — 12.

Задачи

- 1.** Прimitив принимает 4 числа, имеет 12 весов и выдаёт 3 числа. Запишите формы его входного якобиана и якобиана по весам.

Подсказка: Одна строка на каждый выход; один столбец на каждый вход в первом случае и на каждый вес во втором.

Ответ: Входной якобиан $(3, 4)$, якобиан по весам $(3, 12)$: выходы дают строки, входы или веса — столбцы.

- 2.** Объясните своими словами, почему книга спокойно называет «прimitивами» и одно умножение матриц, и целый слой, и всю модель.

Подсказка: Подумайте, что методу на самом деле нужно знать о каждом шаге.

Ответ: Методу нужны для каждого шага только его выход и два его якобиана, поэтому неважно, насколько шаг велик. Блок из блоков поменьше — всё равно шаг со входом, весами и выходом; эта рекурсивная составляемость, ключевое свойство программирования, позволяет применять одно и то же рассуждение в любом масштабе.

- 3.** При обучении чат-ассистента предложение превращается в одно число — потери. Что автоматическое дифференцирование

должно получить для этих потерь и почему именно это нужно обучению?

Подсказка: Что компьютер должен знать о каждом настраиваемом числе, прежде чем сможет улучшить модель?

Ответ: Оно должно получить градиент потерь по каждому весу — для всех весов сразу и эффективно. Обучение меняет каждый вес в ту сторону, где потери уменьшаются, а градиент как раз и говорит, как сдвигаются потери, если чуть сдвинуть каждый вес.

Разворачиваем таблицы и ставим задачу

Представьте электронную таблицу из 32 строк и 100 столбцов: каждая строка — одна фотография с вашего телефона, каждый столбец — одно число, которое фотоприложение о ней измерило. Приложение не смотрит на фотографии по одной; оно берёт весь лист сразу — мини-батч. Теперь спросите себя, сколько частных производных описывают один слой, работающий с этим листом. Можно ожидать несколько сотен. Честный подсчёт даёт около миллиона, и понять почему — первый урок этой части.

БАТЧ ЧЕРЕЗ СЛОЙ

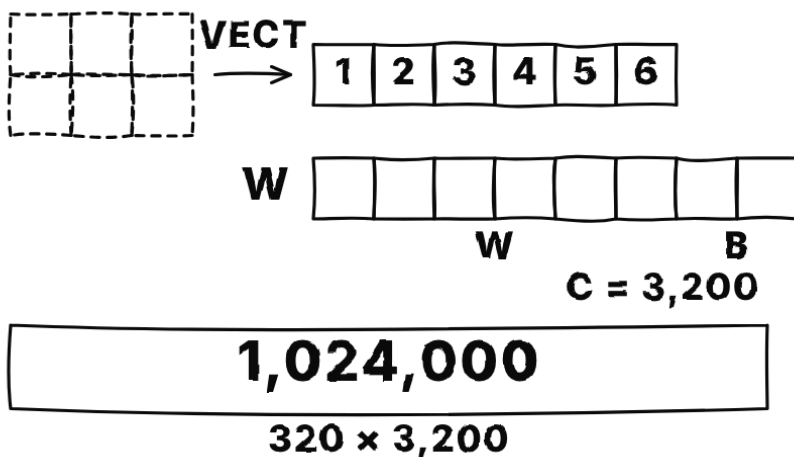


Батч проходит через слой

До сих пор книга говорила о векторах, так что каждый якобиан был плоской таблицей — матрицей. Настоящие примитивы часто принимают и выдают массивы более высокого ранга, то есть с более чем двумя направлениями. Пример книги — полносвязный слой на мини-батче: $f(\mathbf{X}, \mathbf{W}) = \mathbf{XW} + \mathbf{b}$. Словами: батч $\mathbf{X}$ умножается на матрицу весов $\mathbf{W}$, и к каждой строке прибавляется вектор смещения $\mathbf{b}$. Вход $\mathbf{X}$ имеет форму (n, c) : n примеров по c чисел; $\mathbf{W}$ имеет форму (c, c') , а $\mathbf{b}$ — форму (c') , где c' , число выходов, — гиперпараметр, размер, который мы выбираем сами; выход имеет форму (n, c') .

Откуда берутся формы якобианов? Якобиан сопоставляет каждое выходное число с каждым числом, которое сдвигают. У выхода два направления, n и c' ; у входа два, n и c ; значит, входной якобиан имеет форму (n, c', n, c) — это массив ранга 4. Якобиан по весам, сопоставляющий выход с матрицей $\mathbf{W}$, имеет форму (n, c', c, c') , тоже ранг 4. Для листа с фотографиями n равно 32, а c равно 100; возьмём c' равным 10. Тогда во входном якобиане 32 на 10 на 32 на 100 элементов, то есть 1 024 000; в якобиане по весам 32 на 10 на 100 на 10, то есть 320 000.

Книга уходит от высоких рангов простым приёмом — развёрткой. Прочитайте массив строку за строкой и выложите все его числа в один длинный вектор; это записывается как $\mathbf{x} = \text{vect}(\mathbf{X})$. Веса тоже становятся одним вектором — развёрнутая матрица, а за ней смещение: $\mathbf{w} = [\text{vect}(\mathbf{W}); \mathbf{b}]$. Теперь якобианы снова обычные матрицы: входной якобиан формы (nc', nc) и якобиан по весам формы (nc', cc') . Проверим на фотографиях: 320 строк на 3200 столбцов — снова 1 024 000, а 320 на 1000 — снова 320 000; ничего не потеряно, только переставлено.



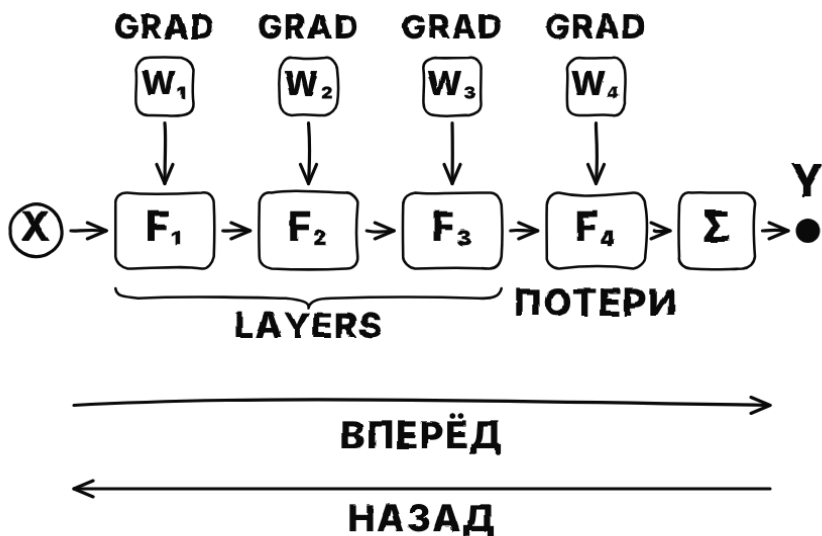
Развёртка таблицы

Для всего дальнейшего важны два следствия. Во-первых, когда книга говорит «размер входа c », она имеет в виду произведение всех размеров входа, включая мини-батч; для нашего листа это 3200, а не 100. Во-вторых, миллион элементов для одного скромного слоя — это предупреждение: даже когда мы умеем вычислить якобиан, мы можем не захотеть его материализовать, то есть на самом деле построить в памяти. Запомните это предупреждение: глава превратит его в метод.

Последнее замечание об обозначениях. Запись каждого примитива как функции от входа и весов соответствует функциональным библиотекам вроде JAX, где всё — функция. Объектно-ориентированные библиотеки, такие как TensorFlow и

PyTorch, делают слой объектом, который хранит свои параметры внутри, а вместо вызова функции — метод. Такой стиль позволяет ленивую инициализацию — создавать параметры, только когда стала известна форма входа, — но добавляет слой абстракции между формулами книги и работающим кодом, и два стиля к тому же по-разному реализуют автоматическое дифференцирование.

Теперь можно поставить задачу. Возьмём программу из l примитивов подряд: $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$ и так далее, вплоть до $\mathbf{h}_l = f_l(\mathbf{h}_{l-1}, \mathbf{w}_l)$, и наконец $y = \Sigma \mathbf{h}_l$. Такая запись называется трассой вычисления программы. Грубо говоря, первые $l - 1$ шагов — это слои модели, шаг l — потери на каждом примере, например перекрёстная энтропия, а итоговая сумма складывает потери по мини-батчу. Поэтому программа всегда заканчивается одним числом, скаляром: численной оптимизации нужно одно число, которое она будет уменьшать; всю программу книга обозначает $F(\mathbf{x})$. С тремя слоями l равно 4: три слоя, один шаг потерь, затем сумма.



Трасса вычисления

Дальше следует определение. Автоматическое дифференцирование — это задача одновременно и эффективно вычислить все якобианы программы по весам, зная её граф и входные якобианы и якобианы по весам каждого примитива: $AD(F(x))$ — это набор $\partial_{w_i} y$ для $i = 1, \dots, l$. Поскольку y — одно число, каждая из них того же размера, что и её веса: по одному элементу на вес, и каждый говорит, как сдвигаются потери, если чуть сдвинуть этот вес. Есть два семейства алгоритмов — прямой режим и обратный режим, — и различаются они только порядком, в котором перемножаются отдельные части. Книга объявляет, что обратный режим, который для нейронных сетей называется обратным распространением ошибки, здесь гораздо эффективнее и что вывод распространяется на ветвящиеся

графы и на разделение весов, когда одним набором весов пользуются несколько слоёв, как в главе 13.

Запомним

Примитивы часто работают с массивами ранга выше двух, так что их якобианы имели бы ранг 4; развёртка каждого массива в один длинный вектор снова превращает их в обычные матрицы. Отныне «размер входа» означает произведение всех размеров входа, включая мини-батч, и числа растут быстро: около миллиона элементов для одного маленького слоя на 32 фотографиях. Именно из-за такого размера мы можем вовсе не хотеть строить якобианы в памяти, даже когда умеем их вычислять. Программа — это трасса вычисления из 1 примитивов, которая заканчивается суммой, так что её выход у всегда одно число.

Автоматическое дифференцирование — это вычисление всех градиентов у по весам сразу и дёшево; прямой и обратный режимы различаются лишь порядком умножений, и обратный выигрывает.

Вопросы

1. Зачем книга разворачивает $\mathbf{X}$ в $\text{vect}(\mathbf{X})$, а веса — в $[\text{vect}(\mathbf{W}); \mathbf{b}]$?
 - а. Чтобы уменьшить число частных производных слоя

- b. Чтобы снова превратить якобианы ранга 4 в обычные матрицы, не потеряв ни одного элемента
 - c. Потому что тогда размерностью мини-батча можно пренебречь
 - d. Потому что функциональные библиотеки не умеют хранить матрицы
2. Слой работает с мини-батчем из 32 фотографий по 100 чисел в каждой. Чему здесь равно s , когда книга говорит о «размере входа s »?
- a. 100, числу чисел на фотографию
 - b. 32, размеру мини-батча
 - c. 3200, произведению всех размеров входа
 - d. 132, сумме двух размеров
3. Почему программа в трассе вычисления всегда заканчивается одним числом y ?
- a. Потому что каждый слой сводит свой вход к одному числу
 - b. Потому что численной оптимизации нужно одно число, которое она будет уменьшать, поэтому потери по примерам суммируются
 - c. Потому что иначе якобианы нельзя было бы развернуть
 - d. Потому что дифференцирование в прямом режиме работает только со скалярами
4. Чем, по словам книги, различаются автоматическое дифференцирование в прямом и в обратном режиме?
- a. Порядком, в котором перемножаются отдельные части

- b. Прямой режим использует численное дифференцирование, обратный — символьное
- c. Прямому режиму нужны входные якобианы, обратному — только якобианы по весам
- d. Обратный режим работает только в объектно-ориентированных библиотеках

Ответы

1. **b)** Развёртка лишь переставляет числа: массивы ранга 4 становятся матрицами формы (nc', nc) и (nc', cc') ровно с тем же числом элементов.
2. **c)** Размер входа — это произведение всех размеров входа, включая мини-батч: 32 на 100 — это 3200.
3. **b)** Последние шаги вычисляют потери для каждого примера, а затем складывают их по мини-батчу, и получается скаляр, который обучение уменьшает.
4. **a)** Оба вычисляют одни и те же градиенты по весам; различаются они только порядком композиции, и обратный режим здесь оказывается гораздо эффективнее.

Разобранные примеры

Считаем элементы якобиана слоя Полносвязный слой $f(X, W) = XW + b$ работает с мини-батчем из $n = 8$ примеров по $c = 4$ числа и выдаёт $c' = 3$ выхода на пример. Сколько элементов в его входном якобиане и какую форму он принимает после развёртки?

До развёртки входной якобиан сопоставляет каждое выходное число с каждым входным, поэтому его форма — $(n, c', n, c) = (8, 3, 8, 4)$.

Его элементов: $8 \times 3 \times 8 \times 4 = 768$.

После развёртки выход — вектор из $n \cdot c' = 24$ чисел, а вход — из $n \cdot c = 32$ чисел, так что якобиан — матрица формы (24, 32).

Проверка: $24 \times 32 = 768$, то же число, потому что развёртка лишь переставляет числа.

Ответ: 768 элементов; после развёртки форма (24, 32).

Задачи

1. Слой работает с $n = 16$ примерами по $c = 50$ чисел и выдаёт $c' = 5$ выходов на пример. Найдите размер входа в смысле книги и развёрнутые формы входного якобиана и якобиана по весам ($\mathbf{W}$ формы (c, c')).

Подсказка: Размер входа — произведение всех размеров входа; развёрнутые якобианы имеют формы (nc', nc) и (nc', cc') .

Ответ: Размер входа $16 \times 50 = 800$. Входной якобиан (80, 800), то есть 64 000 элементов; якобиан по весам (80, 250), то есть 20 000 элементов.

2. В модели 5 слоёв, за ними функция потерь для каждого примера и сумма по мини-батчу. Запишите, чем заканчивается её трасса вычисления, и скажите, чему равно l .

Подсказка: Сосчитайте примитивы до итоговой суммы: слои и шаг потерь.

Ответ: Трасса идёт от $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$ до $\mathbf{h}_6 = f_6(\mathbf{h}_5, \mathbf{w}_6)$, где шестой шаг — потери, и заканчивается $y = \Sigma \mathbf{h}_6$. Значит, l равно 6, а y — одно число.

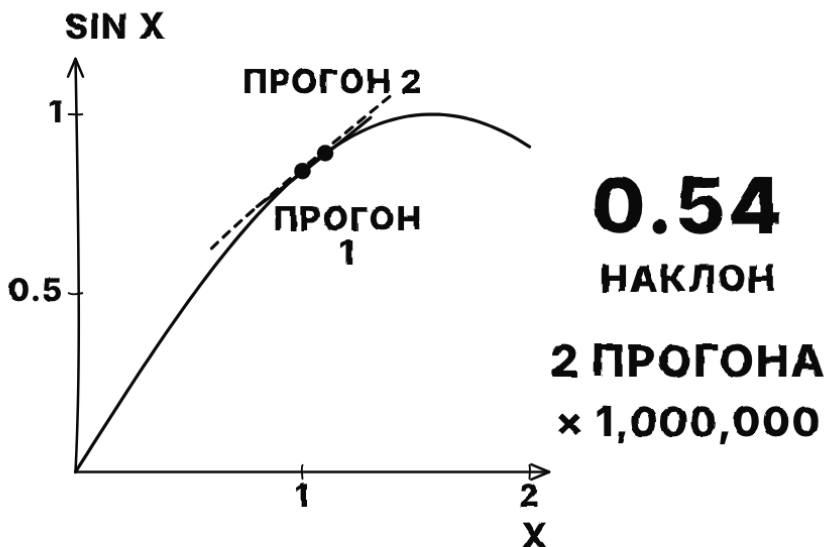
3. Объясните своими словами, почему книга предупреждает, что мы можем не захотеть материализовать якобианы, даже умея их вычислять.

Подсказка: Вспомните, сколько элементов получилось для одного скромного слоя на мини-батче.

Ответ: Когда в размер входа входит мини-батч, развёрнутые якобианы становятся огромными: около миллиона элементов для одного маленького слоя на 32 фотографиях. Строить такие таблицы в памяти для каждого слоя было бы расточительно, поэтому метод должен получать градиенты, ни разу не храня их целиком.

Три способа получить производную

Возьмите карманный калькулятор и функцию синус. Вычислите синус 1, затем синус 1,001, вычтите одно из другого и разделите на 0,001. Получится около 0,54, а косинус 1 — настоящая производная — равен 0,5403. Это работает, и на одну производную ушло два прогона функции. Запомните эти два прогона: именно они решают судьбу первого метода.

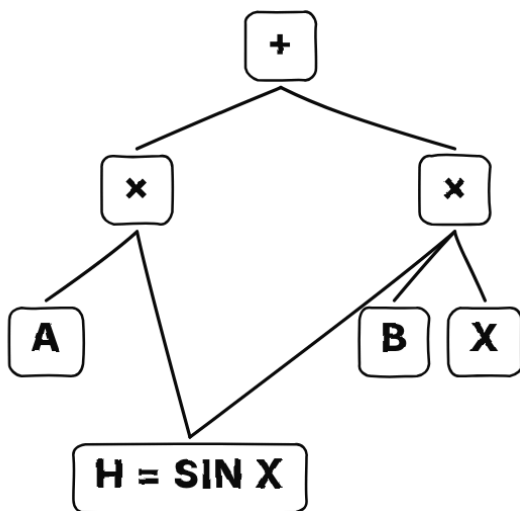


Наклон по двум прогонам

То, что вы сделали, — это численное дифференцирование: прямое применение определения производной с маленьким

сдвигом, чтобы получить приближение. Книга указывает на его цену: в простой реализации каждое число, по которому вы хотите дифференцировать, требует двух вызовов функции. Модели с миллионом весов понадобилось бы два миллиона прогонов всей программы на один градиент, а у чат-ассистента весов гораздо больше. Поэтому автор оставляет численному дифференцированию только одну работу — время от времени проверять, что более быстрая реализация даёт правильные числа.

Второй способ — символьное дифференцирование. Возьмём маленькую функцию из книги: $f(x) = a \sin(x) + b x \sin(x)$. Символьная система, программа для алгебры, может заранее вывести всю формулу производной, и книга показывает это на библиотеке SymPy для Python: вы объявляете x , a и b символами, вводите функцию, просите её производную, и программа печатает результат. Результат такой: $f'(x) = a \cos(x) + b x \cos(x) + b \sin(x)$. Откуда он берётся? Первое слагаемое превращает синус в косинус; второе — произведение, x на синус, поэтому оно даёт два слагаемых, по одному на каждый множитель. Проверим числами: возьмём a равным 2, b равным 3, а x равным $\pi/2$, где синус равен 1, а косинус 0; производная равна 2 на 0, плюс 3 на $\pi/2$ на 0, плюс 3 на 1, то есть 3.



NP-ПОЛНАЯ

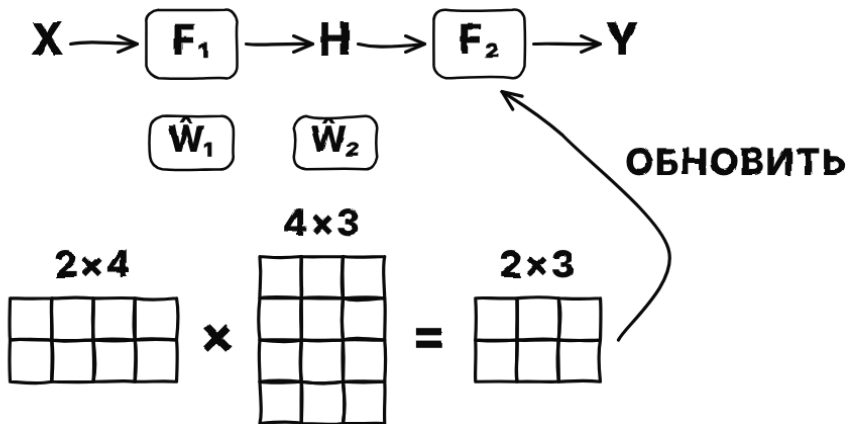
ХРАНИТЬ, БРАТЬ СНОВА

Один и тот же синус дважды

Взгляните на формулу ещё раз, и вы увидите повторы: $\sin(x)$ встречается в функции дважды, $\cos(x)$ — дважды в производной. Разумная программа вычисляет $h = \sin(x)$ один раз, сохраняет и использует повторно, в том числе при вычислении производной, и так же поступает с косинусом. Это похоже на простую уборку, но автор предупреждает, что всё не так тривиально, как кажется: найти наилучший способ вычислить якобиан, без единого лишнего шага, — это задача оптимального накопления якобиана, и она NP-полна, то есть быстрого общего метода для неё не известно. Хорошая новость в том, что у программы есть структура, которой можно воспользоваться. Автоматическое дифференцирование использует её и оказывается гораздо лучше простого символьного подхода; книга добавляет, что на самом

деле оно равносильно символьному подходу, которому разрешено хранить и повторно использовать такие промежуточные куски.

Теперь третий способ, исходящий из цепного правила. Возьмём два примитива подряд: $\mathbf{h} = f_1(\mathbf{x})$, $\mathbf{y} = f_2(\mathbf{h})$. Цепное правило для якобианов гласит: $\partial_{\mathbf{x}}\mathbf{y} = \partial_{\mathbf{h}}\mathbf{y} \cdot \partial_{\mathbf{x}}\mathbf{h}$. Если в $\mathbf{x}$, $\mathbf{h}$ и $\mathbf{y}$ по a , b и c чисел, то это матрица c на b , умноженная на матрицу b на a , и результат — c на a : одна строка на выход, один столбец на вход, как и должно быть. При 3 входах, 4 промежуточных значениях и 2 выходах таблица 2 на 4, умноженная на таблицу 4 на 3, даёт таблицу 2 на 3. Книга читает правило как обновление: если у вас есть f_1 и её якобиан, то применить f_2 — значит умножить то, что у вас есть, на якобиан f_2 , и вы снова в курсе дела.

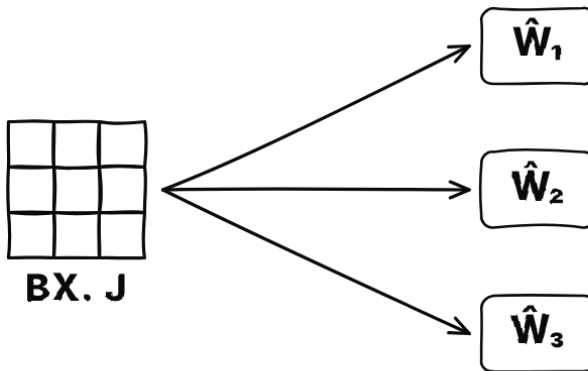


Цепное правило как произведение таблиц

Такое прочтение — уже алгоритм: автоматическое дифференцирование в прямом режиме, сокращённо F-AD. Каждый раз, когда срабатывает примитив, вы заводите новую таблицу для его весов, называемую касательной, и обновляете все касательные, которые у вас уже есть. На первом шаге, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$, ещё ничего не сохранено, поэтому касательная — это просто якобиан по весам: $\hat{W}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$. На втором шаге, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$, старая касательная умножается на входной якобиан f_2 , $\hat{W}_1 \leftarrow [\partial_{\mathbf{h}_1} \mathbf{h}_2] \hat{W}_1$, и заводится новая, $\hat{W}_2 = \partial_{\mathbf{w}_2} \mathbf{h}_2$.

Опыт

Попробуйте это с отдельными числами на калькуляторе. Пусть вход x равен 2, первый шаг умножает на вес w_1 , равный 3, а второй — на вес w_2 , равный 5. После первого шага касательная — это производная 3 на 2 по w_1 , то есть x , значит 2. На втором шаге входной якобиан равен w_2 , то есть 5, так что касательная становится 5 на 2, то есть 10; новая касательная для w_2 — это пришедшее значение, 6. Теперь проверим вручную: выход равен w_2 на w_1 на x , его производная по w_1 — 5 на 2, то есть 10, а по w_2 — 3 на 2, то есть 6.



Прямой режим на калькуляторе

Для i -го примитива в общем виде, $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$, правило то же: $\hat{W}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{W}_j$ для всех $j < i$ и $\hat{W}_i = \partial_{\mathbf{w}_i} \mathbf{h}_i$. Итак, обновление использует входной якобиан шага, а новая касательная — его якобиан по весам. На шаге i выполняется $i - 1$ обновлений, по одному на каждую касательную в памяти, и все они используют один и тот же входной якобиан; на четвёртом шаге три касательные обновляются одной и той же таблицей. Последняя операция программы — сумма, и её градиент даст выход алгоритма.

Запомним

Численное дифференцирование сдвигает каждое число и прогоняет функцию дважды; для проверки это годится, а для миллионов весов безнадёжно. Символьное дифференцирование выписывает формулу производной, но повторяет работу, а найти идеально экономный способ вычислить якобиан — NP-полная задача. Автоматическое дифференцирование вместо этого использует структуру программы, как символьный метод, которому можно хранить и повторно использовать промежуточные значения. Прямой режим читает цепное правило как обновление: на каждом шаге новая касательная начинается с якобиана по весам, а каждая прежняя касательная умножается на входной якобиан шага.

Вопросы

1. Почему книга оставляет численное дифференцирование только для проверки реализации?
 - a. Потому что для большинства производных оно даёт неверный знак
 - b. Потому что каждое число, по которому дифференцируют, требует двух вызовов всей функции, а для миллионов весов это слишком много
 - c. Потому что оно не справляется с функцией синус
 - d. Потому что для работы ему нужна символьная система
2. В чём беда простого символьного выписывания формулы производной?
 - a. Для произведений формула обычно неверна
 - b. Символьные системы не умеют дифференцировать синус
 - c. Оно повторяет работу, а найти идеально экономный способ вычислить якобиан — NP-полная задача
 - d. Ему нужно два вызова функции на каждое число, как численному дифференцированию
3. Что в прямом режиме происходит с касательными, уже хранящимися в памяти, когда срабатывает i -й примитив?
 - a. Каждая умножается на входной якобиан i -го примитива — одну общую для всех таблицу
 - b. Каждая умножается на якобиан i -го примитива по весам
 - c. Их выбрасывают и вычисляют заново с самого начала
 - d. Они не меняются до итоговой суммы

Ответы

1. **б)** В простой реализации каждый скаляр, по которому дифференцируют, стоит двух вызовов функции, так что модели с миллионом весов понадобилось бы два миллиона прогонов на один градиент.
2. **с)** Куски вроде $\sin(x)$ и $\cos(x)$ встречаются по несколько раз; избежать в общем случае всех лишних вычислений — это NP-полная задача оптимального накопления якобиана.
3. **а)** Правило $\hat{W}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{W}_j$ для всех $j < i$ обновляет каждую прежнюю касательную одним и тем же входным якобианом, а новая касательная начинается с якобиана по весам.

Разобранные примеры

Символьный пример книги в одной точке Для функции книги $f(x) = a \sin(x) + b x \sin(x)$ символьная система возвращает $f'(x) = a \cos(x) + b x \cos(x) + b \sin(x)$. Вычислите производную в точке $x = 0$.

При $x = 0$ синус равен 0, а косинус — 1.

Первое слагаемое даёт a на 1, то есть a .

Второе слагаемое даёт b на 0 на 1, то есть 0, из-за множителя x .

Третье слагаемое даёт b на 0, то есть 0.

Ответ: $f'(0) = a$.

Прямой режим через три шага Программа умножает вход $x = 1$ на $w_1 = 2$, затем на $w_2 = 4$, затем на $w_3 = 0,5$. Пронесите касательные вперёд и найдите производную выхода по каждому весу.

Шаг 1: $h_1 = w_1 x = 2$; новая касательная — якобиан по весам, производная $w_1 x$ по w_1 , то есть $x = 1$.

Шаг 2: $h_2 = w_2 h_1 = 8$; входной якобиан — $w_2 = 4$, так что старая касательная становится $4 \times 1 = 4$; новая касательная — $h_1 = 2$.

Шаг 3: $h_3 = w_3 h_2 = 4$; входной якобиан — $w_3 = 0,5$, так что касательные становятся $0,5 \times 4 = 2$ и $0,5 \times 2 = 1$; новая касательная — $h_2 = 8$.

Проверка: $h_3 = w_3 w_2 w_1 x$, и его производные равны $w_3 w_2 x = 2$, $w_3 w_1 x = 1$ и $w_2 w_1 x = 8$.

Ответ: 2 по w_1 , 1 по w_2 , 8 по w_3 .

Задачи

1. У небольшой модели 5000 весов. Сколько вызовов программы нужно простому численному дифференцированию на один полный градиент?

Подсказка: Два вызова на каждое число, по которому дифференцируют.

Ответ: 10 000 вызовов; по два на каждый из 5000 весов.

2. В $\mathbf{h} = f_1(\mathbf{x})$, $y = f_2(\mathbf{h})$ у входа 5 элементов, у $\mathbf{h}$ — 7, у y — 1. Укажите формы двух якобианов в цепном правиле и их произведения.

Подсказка: Для каждого шага — одна строка на выход, один столбец на вход.

Ответ: $\partial_{\mathbf{h}} y$ имеет форму 1 на 7, $\partial_{\mathbf{x}} \mathbf{h}$ — 7 на 5, а их произведение $\partial_{\mathbf{x}} y$ — 1 на 5.

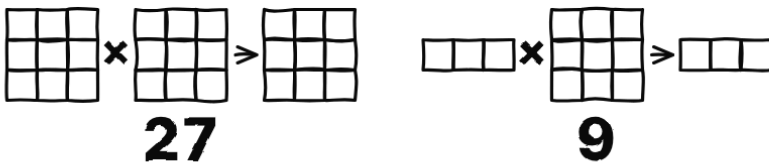
3. Сколько обновлений касательных происходит в прямом режиме на шестом примитиве и какой якобиан у них общий?

Подсказка: Сосчитайте касательные, которые уже в памяти перед шестым шагом.

Ответ: Пять обновлений, по одному на касательные $\mathbf{w}_1$ – $\mathbf{w}_5$, и все используют один и тот же входной якобиан шестого примитива; затем шестая касательная начинается со своего якобиана по весам.

Разворачиваем цепочку

Возьмите карандаш и две маленькие таблицы чисел, каждая 3 на 3, и перемножьте их. Каждому элементу результата нужно 3 умножения, а элементов 9, так что вы делаете 27 умножений. Теперь умножьте одну строку из 3 чисел на одну из этих таблиц. Результат — строка из 3 элементов, по 3 умножения на каждый: всего 9. Строка на таблицу здесь вдвое дешевле, а с таблицами в тысячу строк разрыв становится тысячекратным; в этом разрыве вся суть этой части.



$$\begin{array}{c} \mathbf{1^T} \\ \boxed{1} \quad \boxed{1} \end{array} \times \begin{array}{c} \hat{\mathbf{W}} \\ \boxed{4} \\ \boxed{6} \end{array} = \boxed{10}$$

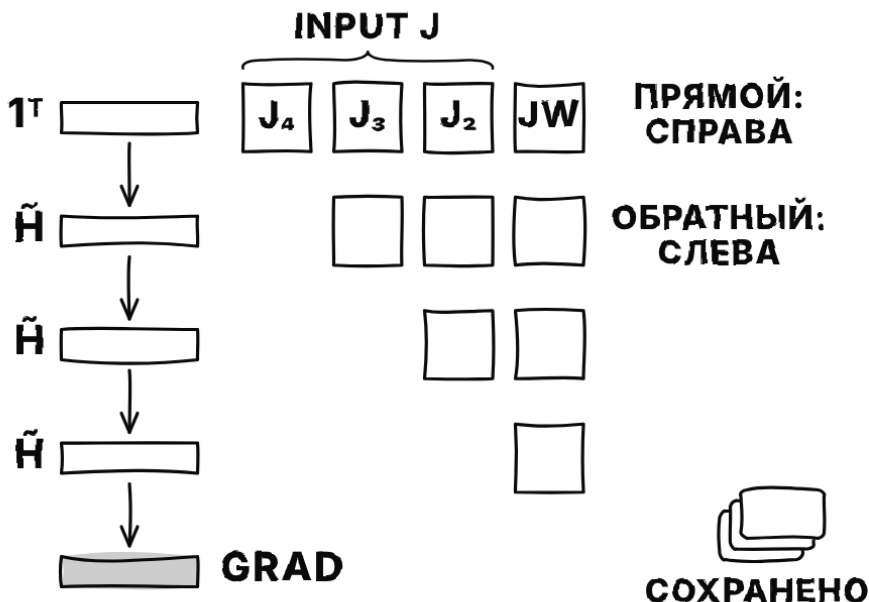
Таблица на таблицу, строка на таблицу

Сначала прямой режим заканчивает свою работу. Программа заканчивается суммой $\mathbf{h}_i$, поэтому градиент для каждого веса получается из последней касательной: $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{\mathbf{W}}_i$ для всех i . Словами: $\mathbf{1}$ — это вектор, заполненный единицами, и умножение на него складывает строки касательной — ровно то, что итоговая сумма делает с выходами. При двух выходах и касательной со строками 4 и 6 градиент равен 4 плюс 6, то есть 10. Мелкое примечание в книге: так получается строка, а не столбец, которым градиент был определён; автор откладывает это до конца главы.

Насколько хорош прямой режим? С хорошей стороны, каждое обновление можно перемежать с исходной программой, шаг за шагом, так что памяти ему нужно примерно столько же, сколько самой программе. С плохой стороны, его главный ход — обновление касательной — перемножает две матрицы форм $(\mathbf{c}'_i, \mathbf{c}_i)$ и $(\mathbf{c}_i, \mathbf{p}_j)$, где p_j — число весов в $\mathbf{w}_j$. Если вход и выход оба имеют форму (n, d) , где n — число примеров в мини-батче, а d — число признаков у каждого, стоимость равна $\mathcal{O}(n^2 d^2 p_j)$: она растёт как квадрат батча и квадрат числа признаков. При 32 фотографиях, 100 признаках и 10 000 весов это 1024 на 10 000 на 10 000 — около ста миллиардов умножений на одно обновление одной касательной. Для изображений, у которых признаков гораздо больше, это быстро становится неосуществимым.

Выход спрятан в последнем шаге прямого режима: это не матрица на матрицу, а вектор на матрицу, потому что выход — одно число. Чтобы им воспользоваться, книга полностью выписывает градиент для одного веса: $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \cdots [\partial_{\mathbf{h}_i} \mathbf{h}_{i+1}] [\partial_{\mathbf{w}_i} \mathbf{h}_i]$. Если отвлечься от обозначений, это просто длинный ряд умножений: вектор единиц, затем цепочка входных

якобианов, которые книга печатает красным, и справа в конце якобиан по весам шага i , зелёным. Красная часть получает собственное имя: $\mathbf{h}_i = \mathbf{1}^T \prod_{j=i+1}^1 \partial_{\mathbf{h}_j} \mathbf{h}_j$.



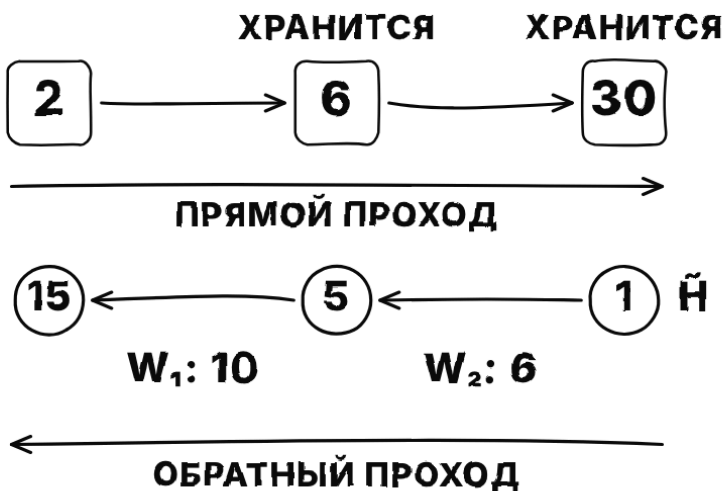
Умножаем слева

Умножение матриц ассоциативно: скобки в произведении можно расставить как угодно, и результат будет тем же. Прямой режим умножал справа налево, в том порядке, в каком работала программа, и всегда держал в руках матрицу. Умножайте вместо этого слева, и произойдут две вещи. Самый левый кусок — вектор на матрицу, это дёшево, и результат снова вектор, так что дешевизна никогда не кончается. И $\mathbf{h}_i$ можно строить рекурсивно, начиная с последнего шага и умножая на входные якобианы в обратном порядке.

Так получается автоматическое дифференцирование в обратном режиме, R-AD, в три шага. Во-первых, прогоните всю программу и сохраните каждый промежуточный выход. Во-вторых, начните с вектора $\tilde{\mathbf{h}} = \mathbf{1}^T$ — самого левого члена. В-третьих, идите назад, i от l до 1, и на каждом шаге делайте две вещи: получите градиент для весов этого шага, $\partial_{\mathbf{w}_i} y = \tilde{\mathbf{h}} [\partial_{\mathbf{w}_i} \mathbf{h}_i]$, а затем передайте вектор на шаг назад, $\tilde{\mathbf{h}} \leftarrow \tilde{\mathbf{h}} [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i]$. Оба действия — вектор на матрицу.

Опыт

Снова возьмите цепочку на калькуляторе из прошлой части: x равен 2, умножается на w_1 , равный 3, затем на w_2 , равный 5. Прогоните её вперёд и запишите, что видите: 6 после первого шага, 30 после второго. Теперь начните обратный путь с $\tilde{h}$, равного 1. На шаге 2 градиент по w_2 равен 1, умноженной на пришедшее значение, 6; затем $\tilde{h}$ становится 1 на w_2 , то есть 5. На шаге 1 градиент по w_1 равен 5 на x , то есть 10; вы получаете те же 10 и 6, что и в прямом режиме, а последнее $\tilde{h}$, 15, — это производная выхода по x .



Назад на калькуляторе

Эти три шага образуют программу, примерно зеркальную исходной, которую книга называет двойственной, или обратной, программой. Векторы $\mathbf{h}$ называются сопряжёнными переменными: по очереди они хранят градиент выхода по каждому $\mathbf{h}_i$, тогда как касательные прямого режима хранили градиенты каждого $\mathbf{h}_i$ по весам. На языке нейронных сетей исходная, или прямая, программа — это прямой проход, что не то же самое, что прямой режим, а обратная программа — это обратный проход. В отличие от прямого режима, весь прямой проход должен закончиться, прежде чем начнётся обратный, и библиотекам нужны особые механизмы, чтобы хранить все промежуточные выходы, — у каждой свои. Обучение

фотоприложения устроено ровно так: фотографии проходят через сеть, вычисляются потери, а затем градиенты текут назад.

Запомним

Прямой режим заканчивается формулой $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{\mathbf{W}}_i$ и использует мало памяти, но каждое обновление касательной перемножает две матрицы, и стоимость порядка $n^2 d^2 p$ взрывается для изображений. Если выписать каждый градиент, это ряд умножений, начинающийся с вектора единиц; поскольку порядок умножений свободен, начиная слева, мы сохраняем каждое произведение произведением вектора на матрицу. Обратный режим сначала прогоняет программу, сохраняя каждое промежуточное значение, затем идёт назад, неся сопряжённую переменную $\mathbf{h}$, и выдаёт по одному градиенту по весам на каждом шаге. Прямой проход должен закончиться до начала обратного — вот почему всё промежуточное приходится хранить.

Вопросы

1. Почему прямой режим становится неосуществимым для входов вроде изображений, хотя ему нужно мало памяти?
 - а. Потому что он должен хранить каждый промежуточный выход программы
 - б. Потому что каждое обновление касательной перемножает две матрицы, и стоимость растёт как

$$n^2 d^2 p_j$$

- с. Потому что его нельзя перемежать с исходной программой
 - d. Потому что у изображений нет якобиана по весам
2. Что позволяет вычислять то же произведение якобианов слева, а не справа?
- a. Умножение матриц ассоциативно, поэтому скобки можно расставлять свободно
 - b. Якобианы — всегда квадратные матрицы
 - с. Все входные якобианы равны друг другу
 - d. Якобиан по весам — вектор единиц
3. Почему обратный режим должен прогнать всю программу, прежде чем сможет начаться обратный проход?
- a. Потому что обратному проходу значение потерь нужно только в самом конце
 - b. Потому что обратный проход начинается от выхода и нуждается в сохранённых промежуточных выходах каждого шага
 - с. Потому что веса меняются во время прямого прохода
 - d. Потому что вектор единиц известен только после работы программы
4. Что хранит сопряжённая переменная $\tilde{\mathbf{h}}_i$ по сравнению с касательной прямого режима?
- a. Градиент $\mathbf{h}_i$ по весам, как касательная
 - b. Само значение $\mathbf{h}_i$, сохранённое во время прямого прохода

- с. Градиент выхода по $\mathbf{h}_i$, тогда как касательная хранит градиент $\mathbf{h}_i$ по весам
- д. Якобиан шага i по весам

Ответы

1. **б)** Основное обновление перемножает матрицы форм $(\mathbf{c}'_i, \mathbf{c}_i)$ и $(\mathbf{c}_i, \mathbf{p}_i)$; при входах и выходах формы $(\mathbf{n}, \mathbf{d})$ это стоит порядка $n^2 d^2 p_i$ — квадратично по размеру батча и по числу признаков.
2. **а)** Поскольку умножение матриц ассоциативно, длинное произведение можно вычислять в любом порядке; начиная с вектора единиц, мы сохраняем каждый шаг произведением вектора на матрицу.
3. **б)** Сопряжённая переменная стартует от выхода и проходит назад через каждый шаг, пользуясь якобианами, которые зависят от промежуточных значений, поэтому все они должны быть сначала вычислены и сохранены.
4. **с)** Сопряжённые переменные хранят, шаг за шагом, градиенты выхода по каждому промежуточному значению; касательные шли в другую сторону — от весов к промежуточным значениям.

Разобранные примеры

Обратный проход через три шага Программа умножает $x = 1$ на $w_1 = 2$, затем на $w_2 = 4$, затем на $w_3 = 0,5$, и её выход — результат. С помощью обратного режима найдите градиент по каждому весу.

Прямой проход с сохранением всего: $h_1 = 2$, $h_2 = 8$, $h_3 = 4$.

Начните сопряжённую переменную с $\tilde{h} = 1$.

Шаг 3: градиент по w_3 равен $\tilde{h}$, умноженному на пришедшее значение, $1 \times 8 = 8$; затем $\tilde{h}$ становится $1 \times w_3 = 0,5$.

Шаг 2: градиент по w_2 равен $0,5 \times h_1 = 1$; затем $\tilde{h}$ становится $0,5 \times w_2 = 2$.

Шаг 1: градиент по w_1 равен $2 \times x = 2$; последнее $\tilde{h}$, $2 \times w_1 = 4$, — это производная выхода по x .

Ответ: 8 по w_3 , 1 по w_2 , 2 по w_1 .

Задачи

1. У касательной две строки: (1, 2) и (3, 4). Что даёт 1^T , умноженный на эту касательную?

Подсказка: Умножение на вектор единиц складывает строки.

Ответ: Строку (4, 6): $1 + 3$ и $2 + 4$.

2. Обновление прямого режима стоит порядка $n^2 d^2 p_j$ умножений. Оцените его для $n = 10$ примеров, $d = 20$ признаков и $p_j = 50$ весов, а затем скажите, что будет, если батч удвоится.

Подсказка: Возведите n и d в квадрат, затем умножьте на p_j .

Ответ: $100 \times 400 \times 50 = 2\,000\,000$ умножений. Удвоение батча умножает стоимость на четыре, потому что она растёт как n в квадрате.

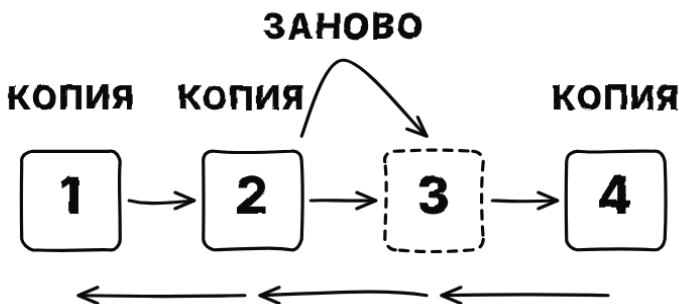
3. Объясните, почему для программы с выходом из одного числа нести вектор слева дешевле, чем нести матрицу справа.

Подсказка: Сравните вектор на таблицу с таблицей на таблицу.

Ответ: Поскольку выход — одно число, самый левый член — вектор, а вектор на матрицу даёт снова вектор; каждый шаг остаётся произведением вектора на матрицу. Если идти справа, каждый шаг перемножает две матрицы, а для таблиц в тысячу строк это примерно в тысячу раз дороже.

Экономим память и просим только произведения

Вспомните длинный уровень видеоигры с точками сохранения. Если бы можно было сохраняться после каждой комнаты, ошибка ничего бы не стоила, но карта памяти быстро бы заполнилась. Если сохраняться только раз в несколько комнат, карта остаётся лёгкой, а после ошибки вы переигрываете несколько комнат от последнего сохранения. Игроки делают такой выбор, не задумываясь. Обратному режиму приходится делать ровно такой же выбор, и решение из книги даже носит игровое имя — контрольные точки.



$$5 / 4 = 1.25$$



ПАМЯТЬ

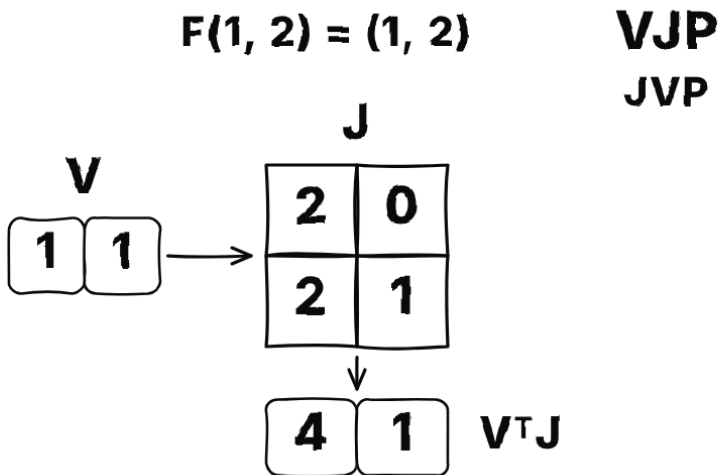
Контрольные точки на четырёх блоках

Сначала хорошая новость, в числах. Обратный режим гораздо эффективнее прямого, потому что оба хода его обратного шага — вектор на матрицу, а такая стоимость растёт лишь линейно по каждому участвующему размеру. Удвойте батч — и работа удвоится, тогда как у прямого режима она вырастает вчетверо. Плата — память: каждое промежуточное значение прямой программы нужно сохранить для обратного прохода, и книга говорит, что хранить его нужно с подходящей стратегией, даже на диске. Для сети, обучаемой на фотографиях высокого разрешения, эти сохранённые значения могут занимать гораздо больше места, чем сами веса.

Контрольные точки градиента улучшают этот компромисс, тратя немного дополнительных вычислений, чтобы сэкономить часть памяти. Сохраняются лишь несколько промежуточных выходов — контрольные точки, — а остальные пересчитываются во время обратного прохода, когда они нужны. Рисунок книги показывает это на четырёх панелях: модель из четырёх блоков, нарисованных в ряд. На первой выполняется прямой проход, но сохраняются только выходы первого, второго и четвёртого блоков. На второй обратный проход, нарисованный красными стрелками справа налево, останавливается на третьем блоке, потому что его значения не были сохранены. На третьей второй прямой проход начинается от ближайшей контрольной точки, второго блока, и снова вычисляет значения третьего блока; на четвёртой обратный проход завершается. Посчитайте работу: четыре блока вперёд плюс один пересчитанный блок — это пять вычислений блоков вместо четырёх, то есть в 1,25 раза больше, — ровно та цифра, которую даёт книга. Чем меньше контрольных точек вы храните, тем дороже обратный проход.

Теперь снова взгляните на третий шаг обратного режима. Единственная операция, которая ему вообще нужна, — вектор-строка $\mathbf{v}$, умноженная на якобиан $\mathbf{f}$, входной или по весам. Эти две операции называются произведениями вектора на якобиан функции $\mathbf{f}$, сокращённо VJP. Книга даёт строгое определение, с транспонированием, чтобы вектор лежал как надо: для $\mathbf{y} = \mathbf{f}(\mathbf{x})$, где $\mathbf{x} \sim (c)$ и $\mathbf{y} \sim (c')$, это $\text{vjp_f}(\mathbf{v}) = \mathbf{v}^T \partial \mathbf{f}(\mathbf{x})$, где в $\mathbf{v}$ c' элементов, по одному на выход. Словами: вы даёте функции вектор размером с её выход, и она возвращает вам вектор размером с её вход — так, как если бы якобиан был использован, — ни разу не показав вам

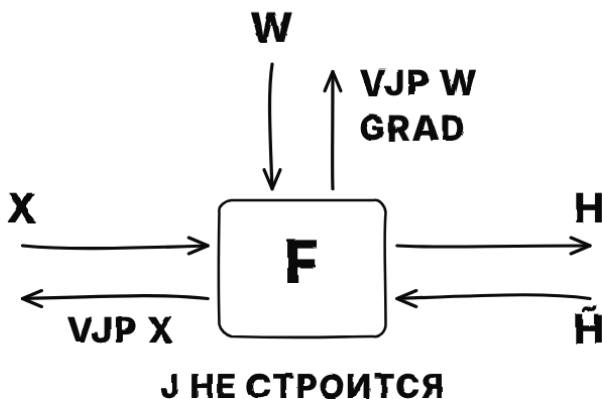
сам якобиан. У функции с несколькими аргументами по одному VJP на каждый аргумент.



Вектор на якобиан

Небольшой пример, чтобы увидеть числа. Пусть f переводит $x = (1, 2)$ в $(x_1^2, x_1 x_2)$, то есть в $(1, 2)$. Строки её якобиана — $(2x_1, 0)$ и (x_2, x_1) , то есть $(2, 0)$ и $(2, 1)$. При $v = (1, 1)$ VJP равно 1, умноженной на первую строку, плюс 1, умноженная на вторую: $(4, 1)$. Для сравнения книга мимоходом замечает, что прямой режим можно целиком записать через транспонированную операцию — произведение якобиана на вектор, JVP, — которое при одном выходе есть уже встречавшаяся производная по направлению.

У примитива два аргумента, поэтому у него два VJP: $\text{vjp_f, x}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$ и $\text{vjp_f, w}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$. Два хода обратного шага превращаются в два вызова с сопряжённой переменной: $\partial_{\mathbf{w}} y = \text{vjp_f, w}(\mathbf{h})$ и $\mathbf{h} \leftarrow \text{vjp_f, h}(\mathbf{h})$. Так что целую систему автоматического дифференцирования можно построить, выбрав набор примитивов и дав каждому два его VJP, ни разу не материализуя якобиан в памяти. Второй рисунок книги изображает примитив как блок: при прямом проходе вход и веса входят, а выход выходит; при обратном проходе сопряжённая переменная входит справа, и два вызова, по одному на каждое VJP, отправляют один вектор назад ко входу, а другой — к весам. Именно это происходит шаг за шагом, когда при обучении чат-ассистента вызывается обратный проход.



$E_1 \rightarrow (2, 0)$

$E_2 \rightarrow (2, 1)$

Примитив с двумя VJP

Работая так, мы ничего не теряем. Вызовите входное VJP с базисными векторами $\mathbf{e}_1, \dots, \mathbf{e}_n$, у каждого из которых 1 в одном месте и 0 в остальных, и вы получите якобиан обратно по одной строке: $\partial_x f(\mathbf{x}, \mathbf{w}) = [\text{vjp_f}(\mathbf{x}(\mathbf{e}_1)); \dots; \text{vjp_f}(\mathbf{x}(\mathbf{e}_n))]$. В нашем маленьком примере $\mathbf{e}_1 = (1, 0)$ возвращает $(2, 0)$, а $\mathbf{e}_2 = (0, 1)$ возвращает $(2, 1)$: ровно две строки. Чем такая переформулировка ценна? Книга отвечает, рассматривая части полносвязного слоя — линейные проекции и поэлементные нелинейности, — начиная с линейной проекции без смещения.

Запомним

Обратный режим стоит лишь произведений вектора на матрицу, линейных по каждому размеру, но должен хранить каждое промежуточное значение прямого прохода.

Контрольные точки градиента хранят лишь часть значений и пересчитывают остальные во время обратного прохода; в примере книги с четырьмя блоками это стоит в 1,25 раза больше вычислений. Обратному шагу нужны лишь произведения вектора на якобиан, $\text{vjp_f}(\mathbf{v}) = \mathbf{v}^T \partial f(\mathbf{x})$, одно для входа и одно для весов. Поэтому каждому примитиву нужны только его два VJP, и ни один якобиан не строится в памяти; при этом полный якобиан всё равно можно восстановить построчно с помощью базисных векторов.

Вопросы

1. Что на что меняют контрольные точки градиента?

- a. Тратят дополнительные вычисления при обратном проходе, чтобы хранить меньше промежуточных значений
 - b. Тратят дополнительную память, чтобы избежать любого пересчёта
 - c. Заменяют обратный режим прямым для средних блоков
 - d. Отбрасывают часть градиентов ради экономии времени
2. Почему на рисунке книги обратный проход стоит в 1,25 раза больше обычных вычислений?
- a. Потому что все четыре блока дважды проходятся вперёд
 - b. Потому что третий блок не был сохранён и его приходится один раз пересчитать от контрольной точки второго блока
 - c. Потому что обратный проход всегда на четверть дороже прямого
 - d. Потому что значения четвёртого блока потеряны
3. Что даёт произведение вектора на якобиан по сравнению с самим якобианом?
- a. Полный якобиан, вычисленный быстрее
 - b. Только диагональ якобиана
 - c. Произведение $\mathbf{v}^T \partial \mathbf{f}(\mathbf{x})$ — вектор размером со вход, — ни разу не строя якобиана
 - d. Вектор размером с выход
4. Теряется ли что-нибудь, если дать каждому примитиву только два его VJP вместо якобианов?
- a. Да, якобианы больше вообще нельзя получить

- b. Нет, вызов VJP с базисными векторами $\mathbf{e}_1, \dots, \mathbf{e}_n$ возвращает якобиан по одной строке
- c. Да, градиенты по весам становятся приближёнными
- d. Нет, потому что VJP и якобианы всегда одного размера

Ответы

1. a) Хранятся лишь несколько выходов — контрольные точки; остальные пересчитываются от ближайшей контрольной точки во время обратного прохода, что требует больше вычислений, но меньше памяти.
2. b) Четыре блока вперёд плюс один пересчитанный блок — это пять вычислений блоков вместо четырёх, то есть в 1,25 раза больше.
3. c) Вы даёте функции вектор размером с её выход и получаете обратно вектор размером с её вход — так, как если бы якобиан был использован.
4. b) Каждый базисный вектор выбирает одну строку, так что стопка $\text{vjp_f}, \mathbf{x}(\mathbf{e}_1) \dots \text{vjp_f}, \mathbf{x}(\mathbf{e}_n)$ восстанавливает входной якобиан.

Разобранные примеры

Цена контрольных точек в примере книги В модели четыре блока. Прямой проход сохраняет только выходы первого, второго и четвёртого блоков. Сколько вычислений нужно обратному проходу по сравнению с обычным случаем, если считать вычисления блоков?

Прямой проход вычисляет четыре блока по разу: 4 вычисления блоков.

Идя назад, проход доходит до третьего блока и обнаруживает, что его значений нет.

Он заново проходит вперёд от ближайшей контрольной точки, выхода второго блока, через третий блок: ещё 1 вычисление блока.

Всего 5 вместо 4, и $5 / 4 = 1,25$.

Ответ: В 1,25 раза больше вычислений.

Произведение вектора на якобиан вручную f переводит $\mathbf{x} = (3, 4)$ в $(x_1 + x_2, x_1 x_2)$. Вычислите $\text{vjr}_f(\mathbf{v})$ для $\mathbf{v} = (2, 1)$.

В якобиане одна строка на выход: первый выход даёт $(1, 1)$, второй — $(x_2, x_1) = (4, 3)$.

$\mathbf{v}^T$, умноженный на якобиан, — это 2, умноженное на первую строку, плюс 1, умноженная на вторую.

$2 \times (1, 1) = (2, 2)$, и, прибавив $(4, 3)$, получаем $(6, 5)$.

Ответ: $(6, 5)$ — вектор размером со вход.

Задачи

1. В модели 8 блоков. Не сохраняются только выходы блоков 3 и 6, и каждый нужно один раз пересчитать во время обратного прохода. Сколько вычислений нужно обратному проходу по сравнению с обычным случаем?

Подсказка: Сосчитайте прямые вычисления и прибавьте пересчитанные блоки.

Ответ: 8 прямых вычислений плюс 2 пересчитанных блока — 10 вместо 8, то есть в 1,25 раза больше.

2. f переводит $\mathbf{x}$ в $(2x_1 + x_2, x_1 - x_2)$. Вызовите её VJP с $\mathbf{e}_1 = (1, 0)$ и $\mathbf{e}_2 = (0, 1)$ и запишите восстановленный якобиан.

Подсказка: Каждый базисный вектор выбирает одну строку якобиана.

Ответ: $\text{vjr}(\mathbf{e}_1) = (2, 1)$ и $\text{vjr}(\mathbf{e}_2) = (1, -1)$, так что строки якобиана — $(2, 1)$ и $(1, -1)$.

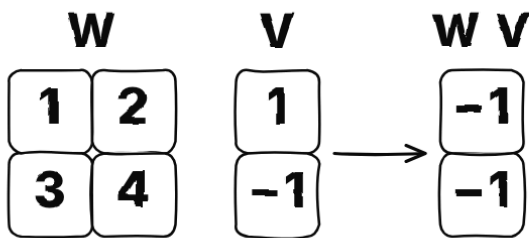
3. Объясните своими словами, почему обратному режиму нужно гораздо больше памяти, чем прямому, хотя вычисляет он гораздо меньше.

Подсказка: Что обратному проходу нужно от прямого прохода и когда?

Ответ: Прямой режим выполняет свои обновления вместе с программой и может забыть каждое значение, как только пошёл дальше. Обратный режим может начать только после того, как вся программа отработала, и каждому обратному шагу нужны значения его шага, поэтому все промежуточные выходы приходится хранить, пока до них не дойдёт обратный проход.

Обратный шаг слоя

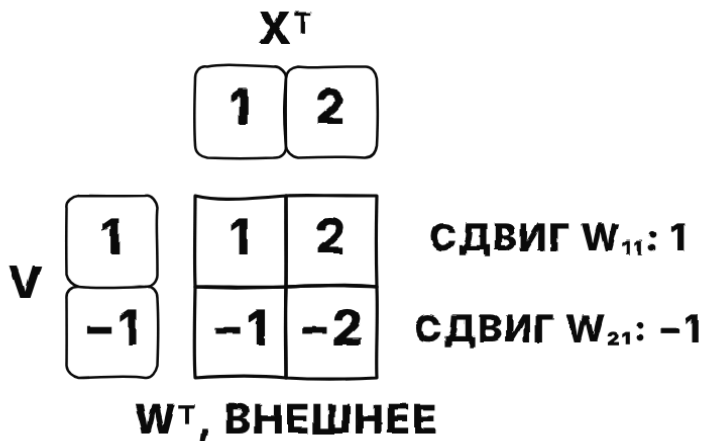
Поставьте небольшой опыт в электронной таблице. Заведите таблицу весов W размером 2 на 2 со строками (1, 2) и (3, 4), вход x , равный (1, 2), и вектор v , равный (1, -1), который взвешивает два выхода. Вычислите W на x — это (5, 11), — а затем одно число: v на это, 1 на 5 минус 1 на 11, то есть -6. Теперь сдвиньте верхний левый вес с 1 на 1,01 и следите за результатом: он сдвигается на 0,01, с -6 до -5,99. Сдвиньте вместо этого нижний левый вес, и результат сдвинется на -0,01; здесь есть закономерность, и книга превращает её в формулу.



Возьмём простейшую часть полносвязного слоя — линейную проекцию без смещения: $f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$. Её входной якобиан — просто матрица $\mathbf{W}$, а вот якобиан по весам — тензор ранга 3: одно направление для выходов и два для строк и столбцов $\mathbf{W}$. Это ровно тот вид массива, который книга не хочет строить. VJP обходят его.

У входного VJP нет особой структуры: $\text{vjp_f, x}(\mathbf{v}) = \mathbf{v}^T \mathbf{W}^T = [\mathbf{W}\mathbf{v}]^T$. Словами: возвращающаяся сопряжённая переменная умножается на транспонированные веса. Книга записывает это произведение с транспонированными весами, и результат — снова вектор. В нашей таблице $\mathbf{W}$ на $(1, -1)$ даёт $(1 - 2, 3 - 4)$, то есть $(-1, -1)$, — вектор, который формула книги отправляет назад ко входу.

VJP по весам оказывается внешним произведением, и никакого тензора ранга 3 нигде нет: $\text{vjp_f, w}(\mathbf{v}) = \mathbf{v}\mathbf{x}^T$. Внешнее произведение двух векторов — это таблица, в которой элемент строки i и столбца j равен i -му элементу первого вектора, умноженному на j -й элемент второго. Книга выводит это в короткой врезке: в числе $y = \mathbf{v}^T \mathbf{W}\mathbf{x} = \sum_i \sum_j W_{ij} v_i x_j$ каждый вес встречается ровно один раз, умноженный на v_i и x_j . Значит, $\partial y / \partial W_{ij} = v_i x_j$ — это и есть поэлементное определение внешнего произведения. С нашими числами $\mathbf{v}$ на транспонированное $\mathbf{x}$ даёт строки $(1, 2)$ и $(-1, -2)$: верхний левый элемент, 1, — это те 0,01, что мы видели при сдвиге верхнего левого веса, делённые на 0,01, а нижний левый, -1, совпадает со вторым сдвигом.



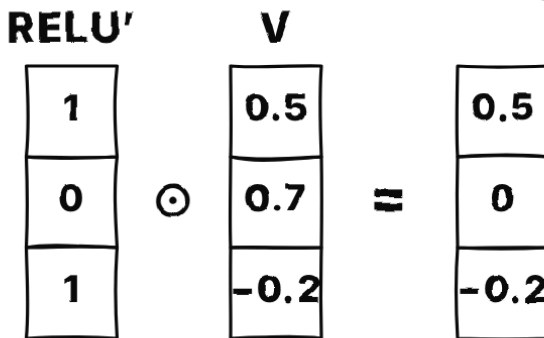
VJP по весам — внешнее произведение

Правило, которое стоит запомнить, короткое. Каждый раз, когда прямой проход применяет линейную проекцию, обратный проход умножает путешествующий градиент на транспонированные веса и вычисляет градиент по W как внешнее произведение. Когда обучают чат-ассистента, большинство его слоёв — такие проекции, и эти две операции составляют значительную часть его обратного прохода.

Теперь поэлементная функция активации без обучаемых параметров, например ReLU: $f(x, \emptyset) = \phi(x)$. Раз весов нет, есть только входное VJP. Её якобиан — диагональная матрица, нулевая всюду, кроме диагонали, где стоят производные ϕ : $[\partial_x \phi(x)]_{ii} = \phi'(x_i)$, потому что каждый выход зависит только от

своего входа. Диагональная матрица на вектор — это просто поэлементное масштабирование, которое называется произведением Адамара: $\text{vjr_x}(f, v) = v \odot \phi'(x)$. Для ReLU при x , равном $(2, -1, 3)$, производные равны $(1, 0, 1)$; путешествующий вектор $(0,5, 0,7, -0,2)$ выходит как $(0,5, 0, -0,2)$. И здесь полная диагональная матрица так и не строится.

$$X = (2, -1, 3)$$



The diagram illustrates the element-wise multiplication of the ReLU derivative vector and the input vector V to produce the output vector. The ReLU derivative vector is shown as a column vector with values 1, 0, and 1. The input vector V is shown as a column vector with values 0.5, 0.7, and -0.2. The result of the multiplication is shown as a column vector with values 0.5, 0, and -0.2. The operation is represented by the symbol $\odot$ and the equals sign.

ReLU'
1
0
1

 $\odot$

V
0.5
0.7
-0.2

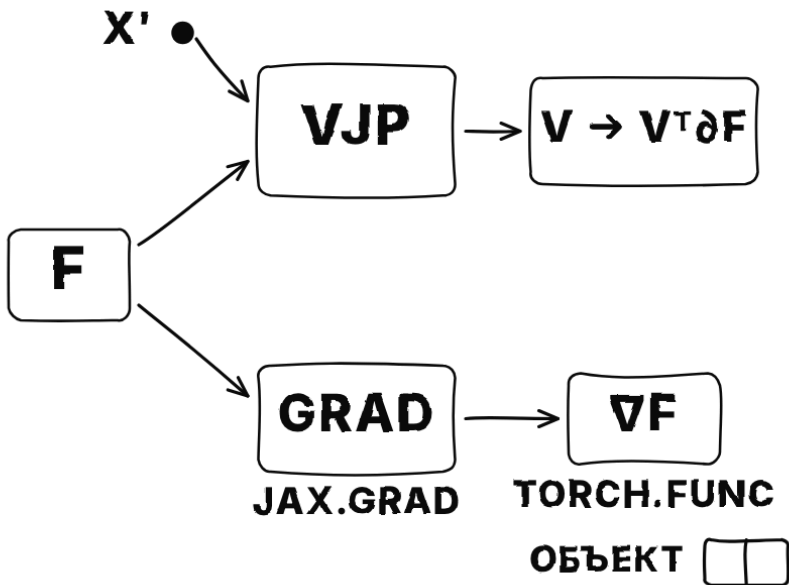
 $=$

0.5
0
-0.2

Обратный шаг ReLU — это маска

Как всё это устроено в настоящих библиотеках? Способов много, говорит книга, — от списков Венгерта, записанного списка выполненных операций, как в TensorFlow, до преобразований исходного кода в исходный код, которые переписывают сам код программы. Запись примитивов как функций от входа и весов подходит функциональным библиотекам вроде JAX, где всё —

функция. Там VJP — функция высшего порядка, то есть функция, которая принимает и возвращает функции, с сигнатурой $(\mathbb{R}^c \rightarrow \mathbb{R}^c) \rightarrow \mathbb{R}^c \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$. Словами: дайте ей функцию f и точку входа, и она вернёт новую функцию, которая берёт вектор v размером с выход и возвращает $v^T \partial f(x')$.



VJP возвращает функцию

Градиент функции с одним выходом — ещё одна функция высшего порядка, с сигнатурой $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$: она берёт f и возвращает функцию, которая вычисляет $\nabla f(x)$. JAX предлагает такие функции — книга называет среди них `jax.grad`, — а PyTorch копирует этот интерфейс в своём модуле `torch.func`. Пример книги берёт сумму квадратов вектора, запрашивает её функцию градиента, применяет её к десяти случайным числам и получает

обратно десять чисел, как и положено: по одному на вход, здесь удвоенный каждый вход. Такую функцию можно ещё и трассировать, например с помощью `torch.compile`, в оптимизированный вычислительный граф, а сочетание VJP и JVP даёт больше, например вторые производные. Но на практике модели строятся как объекты, хранящие параметры внутри себя, и дальше книга переходит к тому, как соединить два стиля.

Запомним

Для линейной проекции $\mathbf{W}\mathbf{x}$ якобиан по весам был бы тензором ранга 3, но VJP его обходят: входное VJP умножает на транспонированное $\mathbf{W}$, а VJP по весам — это внешнее произведение $\mathbf{v}\mathbf{x}^T$. Рассуждение врезки простое: в $\mathbf{v}^T\mathbf{W}\mathbf{x}$ каждый вес W_{ij} встречается один раз, умноженный на v_i и x_j . Для поэлементной активации якобиан диагональный, поэтому её VJP — это просто $\mathbf{v} \odot \phi'(\mathbf{x})$, масштабирование, и снова без построения матрицы. В функциональных библиотеках вроде JAX VJP и градиент — функции высшего порядка, которые принимают функцию и возвращают новую; PyTorch предлагает то же в `torch.func`.

Вопросы

1. Почему с VJP по весам линейной проекции $\mathbf{W}\mathbf{x}$ гораздо проще обращаться, чем с её якобианом по весам?

- a. Потому что якобиан по весам — тензор ранга 3, а VJP по весам — просто внешнее произведение $\mathbf{v}\mathbf{x}^T$
 - b. Потому что VJP по весам не учитывает вход $\mathbf{x}$
 - c. Потому что якобиан по весам — это просто $\mathbf{W}$
 - d. Потому что VJP по весам — всегда диагональная матрица
2. Почему производная $y = \mathbf{v}^T \mathbf{W} \mathbf{x}$ по W_{ij} получается равной $v_i x_j$?
- a. Потому что $\mathbf{W}$ — квадратная матрица
 - b. Потому что в двойной сумме каждый вес W_{ij} встречается ровно один раз, умноженный на v_i и x_j
 - c. Потому что входной якобиан — транспонированная $\mathbf{W}$
 - d. Потому что ReLU обнуляет остальные элементы
3. Что входное VJP поэлементной активации, такой как ReLU, делает с путешествующим вектором $\mathbf{v}$?
- a. Умножает $\mathbf{v}$ на транспонированную матрицу весов
 - b. Берёт внешнее произведение $\mathbf{v}$ и входа
 - c. Масштабирует каждый элемент $\mathbf{v}$ на производную ϕ при соответствующем входе — произведение Адамара $\mathbf{v} \odot \phi'(\mathbf{x})$
 - d. Оставляет $\mathbf{v}$ без изменений, потому что весов нет
4. Что в функциональной библиотеке вроде JAX возвращает VJP функции?
- a. Число: производную в одной точке
 - b. Новую функцию, которая берёт вектор размером с выход и возвращает $\mathbf{v}^T \partial f(\mathbf{x}')$
 - c. Полную матрицу Якоби функции

д. Список операций, выполненных программой

Ответы

1. **а)** У якобиана по весам одно направление для выходов и два для строк и столбцов $\mathbf{W}$; VJP по весам сворачивает его в таблицу с элементами $v_i x_j$ — той же формы, что и сама $\mathbf{W}$.
2. **б)** Если выписать, y — это сумма $W_{ij} v_i x_j$ по всем i и j , так что сдвиг одного веса меняет y на $v_i x_j$, умноженное на сдвиг.
3. **с)** Якобиан диагональный с элементами $\phi'(x_i)$, а диагональная матрица на вектор — это поэлементное масштабирование, так что матрицу никогда не нужно строить.
4. **б)** VJP — функция высшего порядка: по f и точке входа она возвращает функцию, которая превращает вектор размером с выход в $\mathbf{v}^T \partial f(\mathbf{x}')$.

Разобранные примеры

Градиент по весам как внешнее произведение Следуя врезке книги, возьмите $y = \mathbf{v}^T \mathbf{W} \mathbf{x}$ при $\mathbf{v} = (1, -1)$ и $\mathbf{x} = (1, 2)$.

Запишите градиент y по матрице $\mathbf{W}$ размером 2 на 2.

Запишите y как двойную сумму $W_{ij} v_i x_j$: каждый вес встречается один раз.

Значит, $\partial y / \partial W_{ij} = v_i x_j$, а это внешнее произведение $\mathbf{v} \mathbf{x}^T$.

Первая строка, $v_1 = 1$: $(1 \times 1, 1 \times 2) = (1, 2)$.

Вторая строка, $v_2 = -1$: $(-1 \times 1, -1 \times 2) = (-1, -2)$.

Ответ: Таблица 2 на 2 со строками $(1, 2)$ и $(-1, -2)$.

Задачи

1. Вычислите VJP по весам $\mathbf{v} \mathbf{x}^T$ для $\mathbf{v} = (2, 1)$ и $\mathbf{x} = (3, 0, 1)$.

Подсказка: Строка i — это v_i , умноженное на весь вектор x .

Ответ: Строки $(6, 0, 2)$ и $(3, 0, 1)$: первая — 2, умноженное на x , вторая — 1, умноженная на x .

2. ReLU получает $x = (-2, 0, 5, 4)$, а обратный проход приносит вектор $v = (3, 3, 3)$. Что отправляет назад её входное VJP?

Подсказка: Производная ReLU равна 0 для отрицательных входов и 1 для положительных; масштабируйте поэлементно.

Ответ: $(0, 3, 3)$: производные равны $(0, 1, 1)$, а $v \odot \phi'(x)$ перемножает поэлементно.

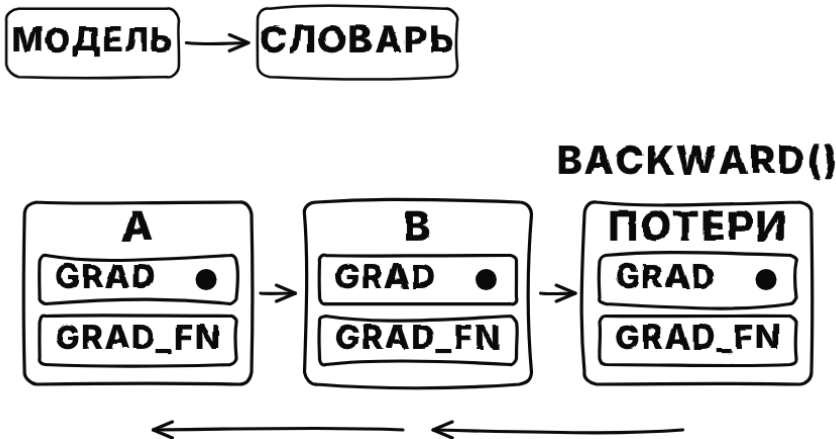
3. Объясните, почему якобиан поэлементной функции активации диагональный.

Подсказка: Спросите себя, от каких входов зависит каждый выход.

Ответ: Каждый выход $\phi(x_i)$ зависит только от своего входа x_i , так что сдвиг любого другого входа его не сдвигает: все элементы вне диагонали равны нулю, а на диагонали стоят $\phi'(x_i)$.

Фреймворки — и почему побеждает ReLU

Перед фреймворками — один небольшой мостик. В PyTorch модель — это объект, который хранит свои параметры внутри, а функциональный стиль хочет функцию от входа и весов. Книга показывает, как «очистить» объект: собрать именованные параметры модели в словарь, а затем вызвать прямое вычисление модели через `torch.func.functional_call`, явно передав модель, этот словарь и вход. Теперь параметры снова стали аргументом, как в обозначениях книги.

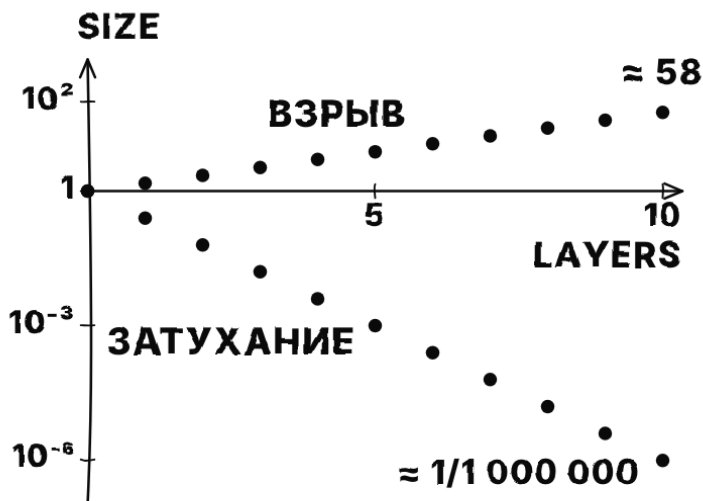


Тензор помнит, кто его создал

В общем же случае PyTorch работает с объектным стилем напрямую, без таких промежуточных шагов. На рисунке книги тензор, массив PyTorch, изображён как коробка с прикреплёнными к ней дополнительными полями. Слева, при создании, у тензора есть ячейка для градиента, сначала пустая, и пометка об операции, которая его породила, — свойство `grad_fn`. Справа, во время обратного прохода, по этим пометкам идут в обратном порядке через вычислительный граф, и каждый градиент сохраняется в поле `grad` своего тензора, но только там, где `requires_grad` явно установлено в `true`, чтобы не тратить память зря. Всё это начинается, когда вы вызываете `backward()` у одного-единственного числа — функции потерь; в сценарии обучения любого фотоприложения это тот самый момент, когда появляются градиенты для каждого веса. Автор подчёркивает, что это лишь обзор, отсылает к официальной документации и горячо советует построить небольшую систему обратного режима самостоятельно — например, разобрав учебную реализацию под названием `micrograd`.

Опыт

Возьмите калькулятор, наберите 1 и десять раз умножьте на 0,25. Вы, возможно, ждёте чего-то маленького, но пригодного. А видите 0,00000095 — примерно одну миллионную. Теперь начните снова с 1 и десять раз умножьте на 1,5: получится около 58, а пятьдесят раз дали бы больше шестисот миллионов.

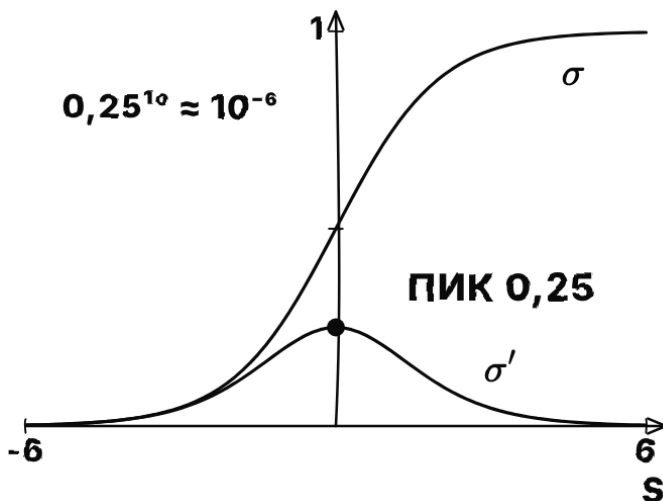


Десять слоёв по 0,25 и по 1,5

Эта игра с калькулятором и есть обратный проход глубокой сети. Из части о ReLU мы знаем, что каждая функция активации на обратном пути умножает сопряжённые переменные на $\phi'(x)$. Если $\phi'(\cdot) < 1$ всюду, градиент экспоненциально быстро сжимается к нулю с ростом числа слоёв: это проблема затухающего градиента. Если $\phi'(\cdot) > 1$ всюду, он экспоненциально растёт к бесконечности: это проблема взрывающегося градиента. На практике это серьёзно, потому что библиотеки хранят числа с ограниченной точностью, обычно 32 бита или меньше, так что нескольких десятков слоёв хватает, чтобы число ушло в ноль или переполнилось.

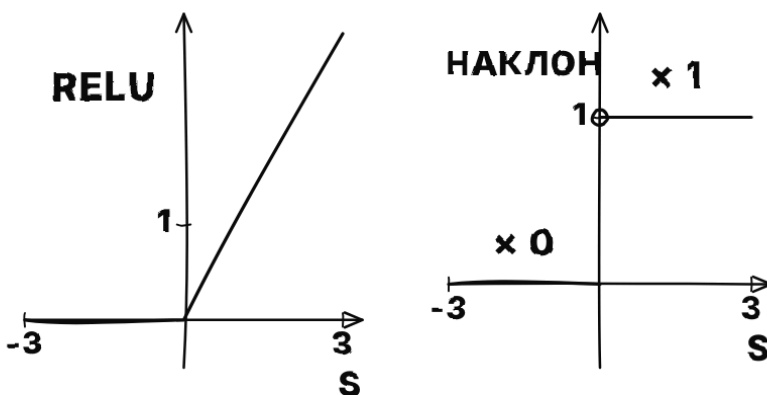
Книга добавляет любопытную врезку. Стопка чисто линейных слоёв, вычисленная в числах с плавающей точкой, не вполне линейна — из-за крошечных скачков на пределе машинной точности. Обычно это не мешает, но автор отмечает, что этим уже пользовались, чтобы обучать глубокие сети, состоящие только из линейных слоёв.

Теперь сигмоида — S-образная кривая $\sigma(s)$, популярная в прошлом как сглаженная версия ступенчатой функции. Её производная — $\sigma'(s) = \sigma(s)(1 - \sigma(s))$. Словами: наклон — это значение, умноженное на то, чего не хватает до 1. Поскольку σ всегда лежит между 0 и 1, это произведение наибольшее, когда σ равна половине, и тогда получается половина на половину: значит, $\sigma'(s) \in [0, 0,25]$. При s , равном 0, σ равна 0,5, а наклон — ровно 0,25; это и есть множитель нашего калькулятора, лучший случай, и уже десять сигмоидных слоёв уменьшают градиент в миллион раз. График в книге это показывает: красная сигмоида поднимается от 0 до 1, а её зелёная производная — невысокий горб с вершиной в четверть, сходящий на нет по обе стороны.



Наклон сигмоиды достигает максимума в четверть

Может ли функция активации избежать обеих проблем? Только тождественная функция, которая ничего не меняет, имеет $\phi'(s) = 1$ всюду, а с ней слои схлопнулись бы в одно линейное отображение. Значит, нужна функция, достаточно линейная, чтобы градиенты оставались здоровыми, и при этом достаточно нелинейная, чтобы разделять слои. ReLU подходит: $\partial_s \text{ReLU}(s) = 0$ при $s < 0$ и 1 при $s > 0$. Градиент либо обнуляется, что делает вычисление разреженным, либо умножается ровно на 1, без всякого масштабирования; на втором графике книги красная ReLU сначала идёт ровно, а потом поднимается по диагонали, а зелёная производная — ступенька от 0 до 1.



ЗАТЕРЕТЬ ВХОД

Наклон ReLU — 0 или 1

И ещё один подарок ReLU. Её градиент одинаков, считать ли его по входу слоя или по выходу, потому что ReLU только маскирует отрицательные значения и не трогает положительные: положительный выход означает положительный вход. Поэтому в обратном режиме вход слоя можно при прямом проходе затереть его выходом и сэкономить немного памяти, не нарушив правильности автоматического дифференцирования; в PyTorch для этого есть параметр работы на месте.

Запомним

В PyTorch каждый тензор помнит операцию, которая его создала; вызов `backward` у функции потерь проходит по этим пометкам в обратном порядке и сохраняет градиенты внутри тех тензоров, которые их просят. Каждая функция активации умножает градиент на обратном пути на свою производную, поэтому множители меньше 1 заставляют градиенты затухать, а больше 1 — взрываться, экспоненциально с ростом числа слоёв, а ограниченная точность делает это заметным очень быстро. Производная сигмоиды $\sigma(s)(1 - \sigma(s))$ не больше 0,25 — главная причина затухающих градиентов. Производная ReLU равна 0 или ровно 1, поэтому она ни сжимает, ни раздувает то, что через неё проходит, и даже позволяет прямому проходу затирать её вход ради экономии памяти.

Вопросы

1. Что в PyTorch позволяет вызову `backward()` найти дорогу назад через вычисление?
 - a. Каждый тензор несёт пометку об операции, которая его создала, и по этим пометкам идут в обратном порядке
 - b. PyTorch заново выполняет всю программу символично
 - c. Градиенты вычисляются численно, по два вызова на каждый вес

- d. Каждый тензор хранит полный якобиан шага, который его создал
2. Глубокая сеть использует функцию активации, производная которой всюду меньше 1. Что происходит с градиентами, доходящими до первых слоёв?
- a. Они экспоненциально растут с числом слоёв
 - b. Они не меняются, потому что веса это компенсируют
 - c. Они экспоненциально быстро сжимаются к нулю с ростом числа слоёв
 - d. Они становятся ровно равными 1
3. Почему сигмоида — первый кандидат на затухание градиента?
- a. Потому что её производная может быть больше 1
 - b. Потому что её производная $\sigma(s)(1 - \sigma(s))$ никогда не превышает 0,25
 - c. Потому что она недифференцируема в 0
 - d. Потому что это линейная функция
4. Почему ReLU может при прямом проходе затереть свой вход выходом, не сломав обратный режим?
- a. Потому что у ReLU нет производной, которую нужно вычислять
 - b. Потому что её градиент так же хорошо читается по выходу: положительные выходы получаются из положительных входов, а отрицательные маскируются
 - c. Потому что PyTorch всё равно хранит вторую копию

d. Потому что обратный проход никогда не заходит в слой активации

Ответы

1. **a)** К тензорам прикреплена создавшая их операция (`grad_fn`); `backward()` проходит по ним в обратном порядке и сохраняет градиенты внутри тех тензоров, которым они нужны.
2. **c)** Каждая функция активации на обратном пути умножает сопряжённые переменные на ϕ' , так что множители меньше 1 перемножаются в затухающий градиент.
3. **b)** Так как σ лежит между 0 и 1, произведение $\sigma(1 - \sigma)$ не больше половины на половину, так что каждый сигмоидный слой уменьшает градиент по меньшей мере вчетверо.
4. **b)** ReLU только маскирует отрицательные значения и не трогает положительные, поэтому её градиент одинаков, вычислять ли его по входу или по выходу.

Разобранные примеры

Наибольший наклон сигмоиды Используя $\sigma'(s) = \sigma(s)(1 - \sigma(s))$ и то, что $\sigma(s)$ лежит между 0 и 1, найдите наибольшее возможное значение σ' и множитель, на который десять сигмоидных слоёв в лучшем случае уменьшают градиент.

Запишите производную как произведение двух чисел, дающих в сумме 1: σ и $1 - \sigma$.

Такое произведение наибольшее, когда числа равны, при $\sigma = 0,5$, что даёт $0,5 \times 0,5 = 0,25$.

Значит, $\sigma'(s)$ лежит между 0 и 0,25, и каждый сигмоидный слой умножает градиент на обратном пути не более чем на 0,25.

За десять слоёв: 0,25 в десятой степени, примерно 0,00000095.

Ответ: Не более 0,25 на слой; десять слоёв в лучшем случае уменьшают градиент примерно до одной миллионной.

Задачи

1. При некотором входе значение сигмоиды $\sigma = 0,8$. Каков там её наклон?

Подсказка: Умножьте значение на то, чего не хватает до 1.

Ответ: $0,8 \times 0,2 = 0,16$ — меньше максимума 0,25.

2. Пусть каждый слой десятислойной сети умножает градиент на обратном пути на 2. Во сколько раз изменится градиент в первом слое? Как называется эта проблема?

Подсказка: Возведите множитель в степень, равную числу слоёв.

Ответ: 2 в десятой степени, 1024: градиент взрывается — это проблема взрывающегося градиента.

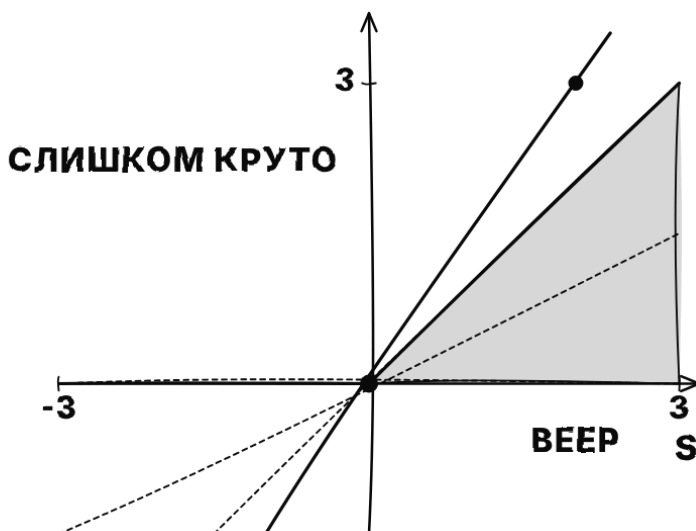
3. Объясните, почему ReLU избегает и затухания, и взрыва градиента для тех входов, где она активна.

Подсказка: Посмотрите на два значения, которые может принимать её производная.

Ответ: Её производная равна 0 для отрицательных входов и ровно 1 для положительных. Там, где она активна, градиент умножается ровно на 1, так что он не сжимается и не раздувается; в остальных местах он обнуляется, что лишь делает вычисление разреженным.

Угол ReLU

Нарисуйте ReLU на листе бумаги: ровно вдоль нуля для отрицательных чисел, а затем вверх по диагонали для положительных, с острым углом в начале координат. Теперь положите через этот угол линейку. Положите её горизонтально — и весь график лежит на ней или над ней; поднимите до диагонали — и график всё равно лежит на ней или над ней. Годится и любой наклон посередине, но наклоните её круче диагонали — и график справа опустится под неё. В углу нет одного наклона, а есть целый их веер, и этим веером книга решает вопрос, который до сих пор оставляла открытым.



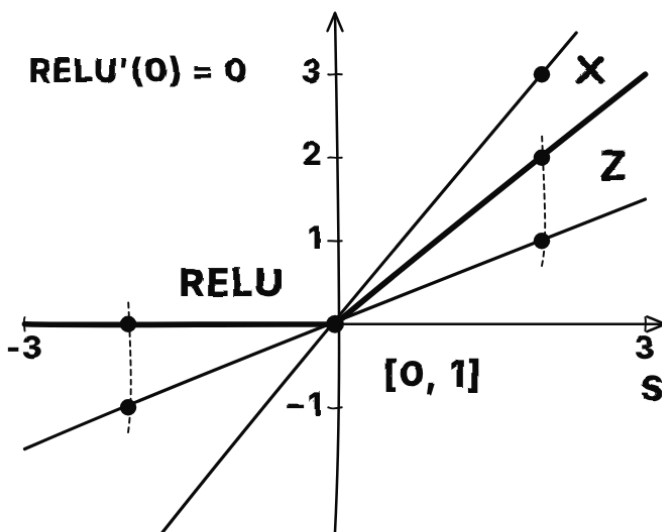
Вeer линеек в углу

У ReLU нет производной в 0, поэтому сеть, построенная на ней, не гладкая. И что тогда? Сначала книга даёт прагматичный ответ: при обучении стохастическим градиентным спуском из случайной ненулевой начальной точки шанс попасть ровно в s , равное 0, практически нулевой, а градиент существует в $\text{ReLU}(\epsilon)$ для любого ϵ , отличного от 0, как бы мало оно ни было. В обучении чат-ассистента, через которое проходят миллиарды чисел, точный ноль — редкая случайность.

Технический ответ — субградиент. Для выпуклой функции f , чей график всюду изгибается вверх, субградиент в точке x — это такое число z , что для любого y выполняется $f(y) \geq f(x) + z(y - x)$.

Словами: прямая через точку $(x, f(x))$ с наклоном z лежит на всём

графике или под ним, как наша линейка. Это перекликается с определением выпуклости, а там, где у f есть производная, такая прямая только одна — касательная, так что субградиент — это просто производная. В углу подходит много наклонов, и вместе они образуют субдифференциал: $\partial_x f(x)$ — множество всех z , которые являются субградиентами f в точке x .



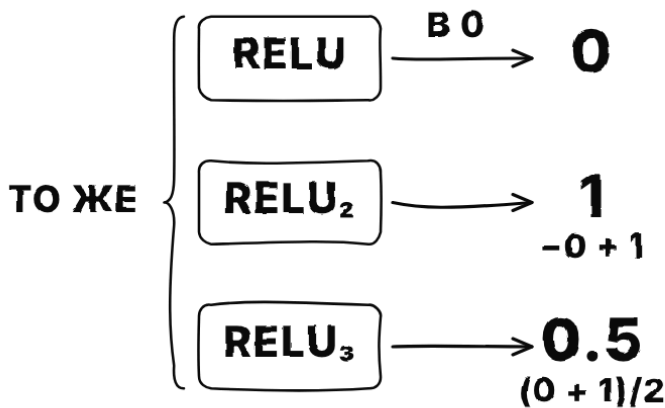
Проверка наклонов 0,5 и 1,5

Теперь ReLU можно достроить, заменив её градиент в 0 субдифференциалом: $\partial_s \text{ReLU}(s)$ — это 0 при $s < 0$, 1 при $s > 0$ и весь отрезок $[0, 1]$ при $s = 0$. Проверим одно значение на числах: возьмём в углу z , равное 0,5. При u , равном 2, прямая даёт 1, а ReLU даёт 2; при u , равном -2 , прямая даёт -1 , а ReLU даёт 0; прямая остаётся ниже, как и должно быть. Возьмите z , равное 1,5,

— и при u , равном 2, прямая даёт 3, что выше двойки ReLU: это не субградиент, как и показала крутая линейка. Значит, в углу годится любое значение от 0 до 1, и большинство реализаций выбирают $\text{ReLU}'(0) = 0$; выбор субградиента на каждом шаге спуска называется субградиентным спуском.

Полная история сложнее. Для невыпуклых функций субградиента может и не быть, и тогда обращаются к обобщениям, которые ослабляют определение до малой окрестности x , — например, к субдифференциалу Кларка. Автоматическое дифференцирование добавляет свою ловушку: разные реализации одной и той же функции могут возвращать разные субградиенты, иногда даже недопустимые, и для строгого доказательства нужны более тонкие правила цепочки.

Примечание книги даёт пример. Определим $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ и $\text{ReLU}_3(s) = 0,5(\text{ReLU}(s) + \text{ReLU}_2(s))$. Попробуйте несколько чисел — обе дают ровно ReLU, и всё же обратный проход в PyTorch в 0 возвращает 0 для ReLU, 1 для ReLU_2 и 0,5 для ReLU_3 . Причина — правило цепочки в действии: производная ReLU_2 равна минус $\text{ReLU}'(0)$, то есть 0, плюс 1, а ReLU_3 усредняет 0 и 1.



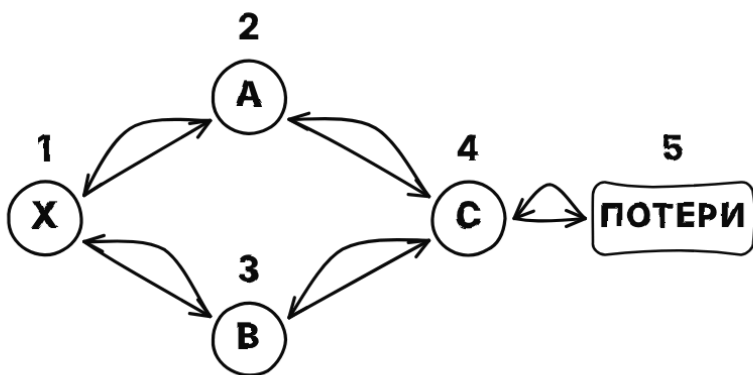
Одна функция — три ответа

Отсюда книга переходит к практике. Если вы делали упражнения предыдущей главы, вы уже пользовались обратным режимом в PyTorch и JAX, а эта глава объясняет, что происходило внутри. Автор советует заново построить небольшую систему обратного режима самому, для скалярных значений, взяв за прекрасный учебный образец `micrograd`. Одна деталь, которую глава пропустила: на общем графе без циклов переменные нужно упорядочить перед обратным проходом, иначе градиенты пойдут по неверным путям; `micrograd` делает это дешёвой топологической сортировкой — порядком, в котором каждое значение идёт после всего, от чего оно зависит. Ещё одно упражнение — написать в PyTorch новый примитив, задав его прямой проход и его VJP, например обучаемую функцию

активации из предыдущей главы, хотя PyTorch мог бы вычислить такой обратный проход и сам. В JAX есть те же пути: учебная версия JAX под названием autodidax, написание примитива с собственным VJP и сборник JAX Autodiff Cookbook — о старших производных, гессианах и многом другом.

MICROGRAD

AUTODIDAX



Упорядочьте граф, потом идите назад

На этом глава закрывается: от цепочки в электронной таблице до движка, который обучает каждую модель в вашем телефоне, — и всё построено на одной идее: умножать в обратном порядке.

Запомним

У ReLU нет производной в 0, но обучение почти никогда не попадает точно туда, а везде в других местах градиент существует. Субградиент в точке x — это наклон z , чья прямая через $(x, f(x))$ остаётся ниже всего графика; в гладкой точке это производная, а в углу — целое множество, субдифференциал. Для ReLU в 0 годится любое значение от 0 до 1, реализации обычно берут 0, а разные, но равносильные формулы могут давать во фреймворке разные ответы, как показывают ReLU₂ и ReLU₃. Книга заканчивается приглашением построить обратный режим самостоятельно — с micrograd или autodidax — и писать примитивы с собственными VJP.

Вопросы

1. Каков прагматичный ответ книги на то, что у ReLU нет производной в 0?
 - a. Обучение начинается со случайной ненулевой точки, так что попасть ровно в 0 практически невозможно, а везде в других местах градиент существует
 - b. Вблизи 0 ReLU заменяют сигмойдой
 - c. Перед обучением сеть сглаживают
 - d. Фреймворки отказываются дифференцировать в 0 и останавливают обучение

2. Сколько субградиентов у выпуклой функции в точке, где у неё есть производная?
- a. Ни одного
 - b. Ровно один — сама производная
 - c. Целый отрезок
 - d. Два — по одному с каждой стороны
3. Почему наклон 1,5 — не субградиент ReLU в 0?
- a. Потому что субградиенты должны быть отрицательными
 - b. Потому что прямая с наклоном 1,5 через угол поднимается над ReLU при положительных входах, например 3 против 2 при $y = 2$
 - c. Потому что в углу законный субградиент только 0
 - d. Потому что ReLU не выпукла
4. $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ всюду равна ReLU. Почему обратный проход PyTorch в 0 возвращает для неё 1, а для ReLU — 0?
- a. Потому что ReLU_2 — другая функция, чем ReLU
 - b. Потому что для ReLU_2 PyTorch использует численное дифференцирование
 - c. Потому что правило цепочки соединяет выбор реализации $\text{ReLU}'(0) = 0$ с производной прибавленного s , давая $0 + 1$
 - d. Потому что ReLU_2 не непрерывна в 0

Ответы

1. a) При стохастическом градиентном спуске из случайной ненулевой инициализации вероятность попасть ровно в $s = 0$ практически

нулевая, а градиент определён при любом ненулевом входе.

2. b) Там, где f дифференцируема, только одна прямая через точку остаётся ниже всего графика — касательная, так что единственный субградиент — это производная.

3. b) Прямая субградиента должна лежать на всём графике или под ним; при $y = 2$ крутая прямая даёт 3, а ReLU даёт 2.

4. c) Обратный проход следует формуле в том виде, как она записана: минус $\text{ReLU}'(o)$, взятое равным 0, плюс 1 от прибавленного s , — так что равносильные формулы могут возвращать разные субградиенты.

Разобранные примеры

ReLU₂ и ReLU₃ из книги в нуле При выборе реализации $\text{ReLU}'(o) = 0$ пройдите по правилу цепочки и найдите, что возвращает обратный проход в $s = 0$ для $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ и $\text{ReLU}_3(s) = 0,5(\text{ReLU}(s) + \text{ReLU}_2(s))$.

Для ReLU₂ первое слагаемое $\text{ReLU}(-s)$ даёт минус $\text{ReLU}'(o)$, то есть 0, из-за внутреннего знака минус.

Второе слагаемое, s , даёт 1, так что ReLU₂ возвращает $0 + 1 = 1$.

Для ReLU₃ обратный проход усредняет две части: 0,5 на $(0 + 1)$.

Получается 0,5, тогда как обычная ReLU даёт 0.

Ответ: 1 для ReLU₂ и 0,5 для ReLU₃ — против 0 для ReLU.

Задачи

1. Проверьте, является ли $z = 0,3$ субградиентом ReLU в o , подставив $y = 4$ и $y = -4$ в $f(y) \geq f(o) + z(y - o)$.

Подсказка: Вычислите значение прямой 0,3у и сравните с $\text{ReLU}(y)$.

Ответ: При $y = 4$ прямая даёт 1,2, а ReLU — 4; при $y = -4$ прямая даёт -1,2, а ReLU — 0. В обоих случаях прямая остаётся ниже, и

действительно любое z между 0 и 1 — субградиент в 0.

2. Покажите, что $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ равна ReLU , попробовав $s = 3$ и $s = -3$.

Подсказка: Сначала вычислите $\text{ReLU}(-s)$, затем прибавьте s .

Ответ: Для $s = 3$: $\text{ReLU}(-3) = 0$, плюс 3 даёт $3 = \text{ReLU}(3)$. Для $s = -3$: $\text{ReLU}(3) = 3$, плюс -3 даёт $0 = \text{ReLU}(-3)$.

3. Объясните, почему система обратного режима на общем графе без циклов должна упорядочить свои переменные перед обратным проходом.

Подсказка: Подумайте о значении, которое питает два более поздних шага.

Ответ: Сопряжённая переменная значения полна только после того, как каждый использующий его шаг вернул свой вклад. Если обрабатывать переменные в топологическом порядке, каждую после всего, от чего она зависит, и проходить этот порядок в обратную сторону, ни один градиент не будет передан дальше, пока он не полон; иначе градиенты пошли бы по неверным путям. `micrograd` делает это дешёвой топологической сортировкой.