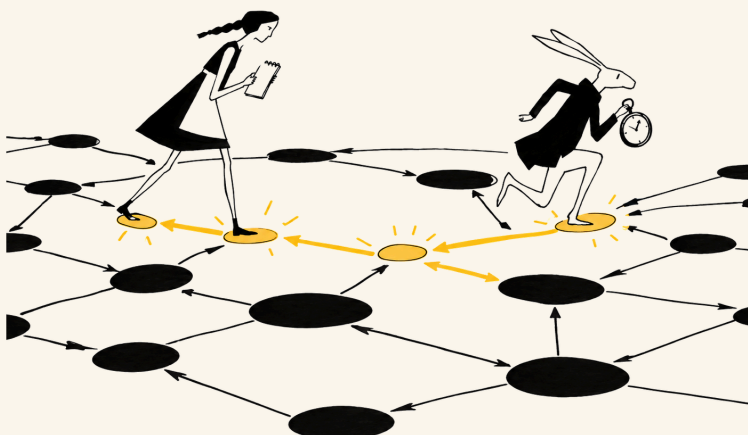


Приключения Алисы в Стране дифференцируемых чудес



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, которая знает свои наклоны

Одно число в конце и медленный путь вперёд

Идём по цепочке назад

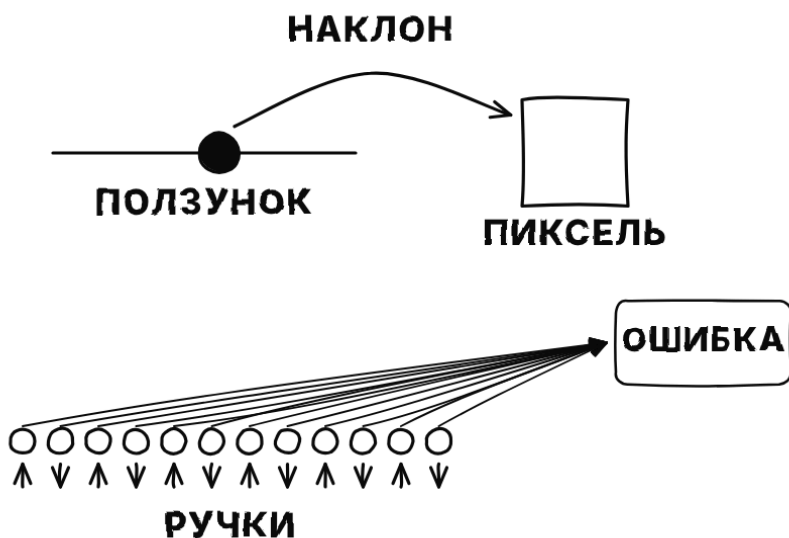
Каждая часть учится отвечать назад

Как это делают библиотеки и почему победила ReLU

Угол пандуса

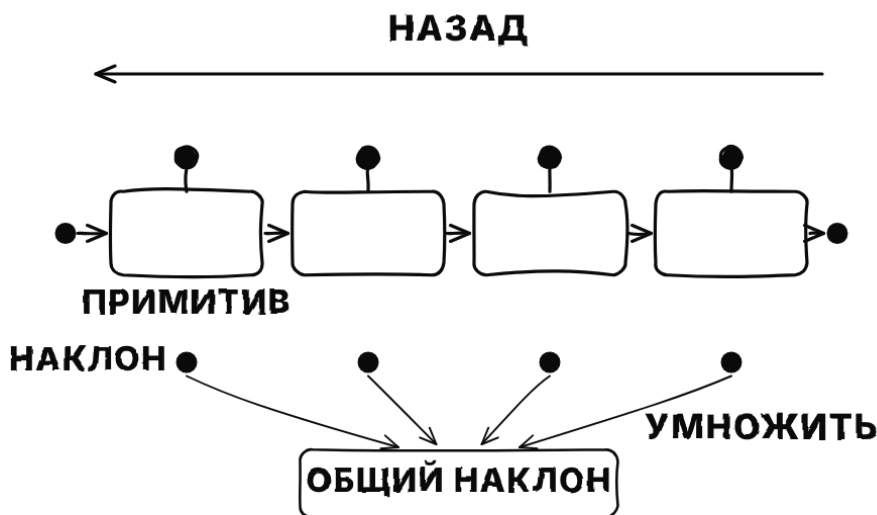
Программа, которая знает свои наклоны

Сфотографируйте что-нибудь телефоном и откройте редактор. Сдвиньте ползунок яркости чуть вправо и следите за одним пикселем неба: он становится чуть светлее, и вы чувствуете, насколько каждое движение ползунка его меняет. Это ощущение — «если чуть повернуть эту ручку, результат сдвинется на столько-то» — математики называют производной. А теперь представьте не один ползунок, а сто миллионов, и все они внутри модели, которая сортирует ваши фотографии или отвечает на вопросы в чате, а в конце стоит одно число, которое говорит, насколько модель ошиблась на пачке примеров. Обучение — это поворот каждой из этих ручек понемногу в правильную сторону, снова и снова, поэтому вопрос этой главы такой: как компьютер быстро узнаёт, в какую сторону поворачивать каждую ручку?



Много ручек, одно число

Ответ книги начинается с того, что модель рассматривается как программа. Когда программа работает, она выполняет цепочку маленьких простых операций, каждая из которых питает следующую; книга называет каждую примитивом, а всю цепочку, как она на самом деле выполнялась, — вычислительным графом. Примитив может быть совсем маленьким, как одно умножение таблицы чисел, большим, как целый слой, или даже целой моделью, потому что части, собранные из частей, — всё равно части. Каждый примитив что-то принимает, хранит собственные настраиваемые числа — веса — и что-то передаёт дальше.

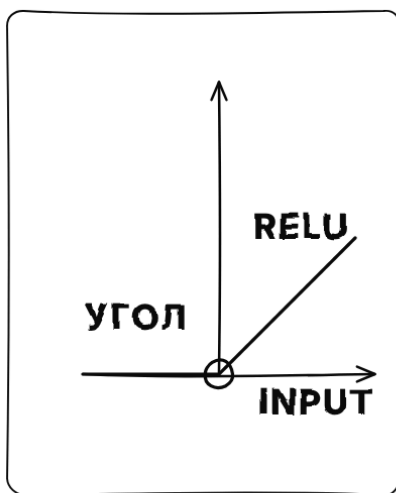


Цепочка примитивов

Всё держится на одном скромном допущении. Для каждого отдельного примитива мы уже знаем, как его выход отзывается на маленькие изменения — и того, что пришло на вход, и его собственных весов; эти две таблицы наклонов книга называет входным якобианом и якобианом по весам. Зная наклоны каждой части, школьное правило для функции внутри функции — цепное правило — в принципе даёт нам наклон всей программы: достаточно перемножить наклоны частей. Так что задача кажется решённой ещё до начала, и вся трудность в том, чтобы выполнить это умножение достаточно хитро, чтобы оно было быстрым. Хитрый способ называется обратным распространением ошибки, или автоматическим

дифференцированием в обратном режиме, и это двигатель каждой системы, которая обучает вашего чат-ассистента.

К этой постановке прилагаются два предупреждения. Таблицы наклонов бывают огромными, ведь каждое входное число может влиять на каждое выходное, а обрабатывается сразу целая пачка фотографий; хороший метод должен пользоваться ими, ни разу не выписывая их целиком. А у некоторых частей есть углы, как у ReLU, которая просто обрезает отрицательные числа до нуля; книга обещает способ справиться с такими углами позже в этой главе.



Два предупреждения

Наконец, книга замечает, что от того, как записана программа, зависит, как находятся наклоны. В библиотеке вроде JAX всё — функция, и веса передаются как любой другой вход, ровно как в обозначениях книги. В PyTorch или TensorFlow слой — это объект, который хранит свои веса внутри себя; это удобно, например чтобы создавать веса только тогда, когда стал известен размер входа, но так веса оказываются на шаг дальше от математики. Здесь опущены размеры таблиц наклонов для пачек данных и то, как книга разворачивает их в простые списки; это вы найдёте в подразделе об обозначениях.

Запомним

Обучить модель — значит найти для каждого настраиваемого числа, в какую сторону его повернуть, чтобы одна итоговая ошибка стала меньше. Книга видит в модели работающую программу, цепочку простых примитивов, и предполагает, что наклоны каждой части известны. Тогда цепное правило даёт наклон целого, так что настоящая трудность — вычислять его быстро и не храня гигантских таблиц. Библиотеки в стиле функций, как JAX, и в стиле объектов, как PyTorch, по-разному представляют веса, и от этого зависит, как в каждой устроено автоматическое дифференцирование.

Вопросы

1. Если наклоны каждого отдельного примитива уже известны, почему найти наклон всей программы — всё ещё настоящая задача?
 - a. Потому что цепное правило работает только для программ из одного примитива
 - b. Потому что перемножать все эти наклоны нужно хитро, чтобы это было быстро
 - c. Потому что наклоны примитивов меняются при каждом запуске программы
 - d. Потому что итоговую ошибку нельзя вычислить точно
2. Почему хороший метод должен пользоваться таблицами наклонов, ни разу не выписывая их целиком?
 - a. Потому что таблицы бывают огромными: каждое входное число может влиять на каждое выходное, и сразу для целой пачки
 - b. Потому что таблицы — секретные части модели, которые прячет библиотека
 - c. Потому что, если их выписать, изменятся веса модели
 - d. Потому что таблицы существуют только для примитивов без углов
3. Что на самом деле нужно от автоматического дифференцирования для обучения модели, которая сортирует ваши фотографии?
 - a. Итоговая ошибка, вычисленная как можно быстрее
 - b. Список всех примитивов в том порядке, в каком они выполнялись

- c. Для каждого настраиваемого числа — в какую сторону его повернуть, чтобы итоговая ошибка уменьшилась
 - d. Точная форма каждой таблицы наклонов
4. Чем PyTorch отличается от JAX в том, как он представляет веса слоя?
- a. PyTorch передаёт веса как любой другой вход, а JAX хранит их внутри объекта
 - b. В PyTorch нет весов, только входы и выходы
 - c. Оба обращаются с весами точно как в обозначениях книги
 - d. PyTorch хранит веса внутри объекта слоя, а JAX передаёт их как любой другой вход

Ответы

1. **b)** Цепное правило говорит, что наклон целого — это произведение наклонов частей, так что задача кажется решённой; вся трудность в том, чтобы выполнить это умножение достаточно хитро и быстро.
2. **a)** Каждое входное число может влиять на каждое выходное, а обрабатывается сразу целая пачка фотографий, поэтому таблицы наклонов становятся слишком большими, чтобы их хранить.
3. **c)** Обучение — это поворот каждой из множества ручек понемногу в правильную сторону, поэтому компьютер должен быстро узнавать, куда поворачивать каждую.
4. **d)** В JAX всё — функция, и веса передаются как любой другой вход, а в PyTorch слой — это объект, который хранит свои веса внутри себя.

Задачи

1. Вспомните фоторедактор с ползунком яркости. Объясните своими словами, что говорит производная и почему модели с сотней миллионов таких ползунков нужен быстрый способ найти их все.

Подсказка: Начните с того, что происходит с одним пикселем, когда вы чуть сдвигаете один ползунок.

Ответ: Производная говорит, насколько сдвигается результат, если чуть повернуть одну ручку. Обучение должно снова и снова поворачивать каждую ручку модели в правильную сторону, поэтому ему каждый раз нужны все эти наклоны, для всех ста миллионов ручек; если делать это медленно, обучение станет невозможным.

2. Прimitives может быть одно умножение, целый слой или вся модель. Объясните, почему метод этой главы работает, даже когда части так различаются по размеру.

Подсказка: Спросите себя, что методу нужно знать о каждой части, каким бы ни был её размер.

Ответ: Метод предполагает лишь, что для каждой части мы знаем, как её выход отзывается на маленькие изменения того, что пришло на вход, и её собственных весов. Части, собранные из частей, — всё равно части, поэтому со слоем или целой моделью можно обращаться так же, как с одной операцией.

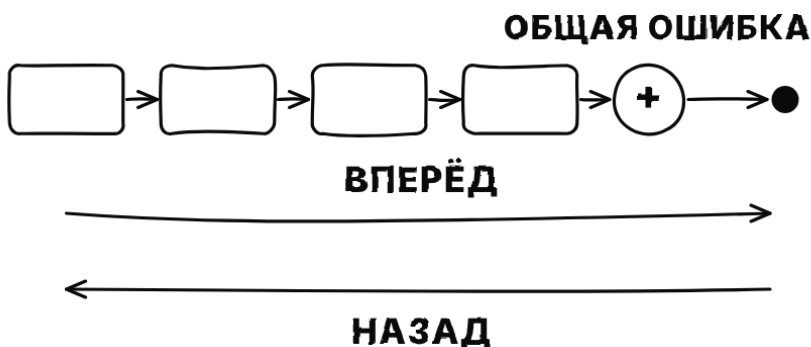
3. У некоторых частей, например у ReLU, есть угол, где они обрезают отрицательные числа до нуля. Почему такой угол может беспокоить метод, построенный на наклонах?

Подсказка: Представьте график ReLU прямо в точке, где он сгибается.

Ответ: В угле у графика нет одного наклона, поэтому там нарушается допущение, что наклон каждой части известен. Книга обещает способ справиться с такими углами позже в этой главе.

Одно число в конце и медленный путь вперёд

Представьте самый простой способ узнать, в какую сторону повернуть ручку. Вы прогоняете модель на пачке фотографий и записываете, насколько она ошиблась; чуть-чуть сдвигаете одну ручку; прогоняете всю модель снова и сравниваете. Разница даёт наклон для этой одной ручки, и это действительно работает. А потом вы вспоминаете, что у модели за вашим фотоприложением многие миллионы ручек и каждая требует своей пары прогонов. Один шаг обучения потребовал бы миллионов полных прогонов, — вот почему так модели никто не обучает.



Сдвинуть, прогнать, одно число

Теперь книга точно ставит задачу. Работающая модель — это цепочка примитивов, выполняемых один за другим, каждый передаёт результат следующему; последнее звено измеряет ошибку на каждом примере, а заключительный шаг складывает эти ошибки по всей пачке. Такой выписанный прогон книга называет трассой вычисления программы. Важно то, что цепочка всегда заканчивается одним числом — общей ошибкой, ведь именно одно число оптимизатор может уменьшать.

Автоматическое дифференцирование — это тогда задача найти, сразу и эффективно, наклон этого одного числа по каждому набору весов в цепочке, пользуясь только самой цепочкой и известными наклонами каждой части.

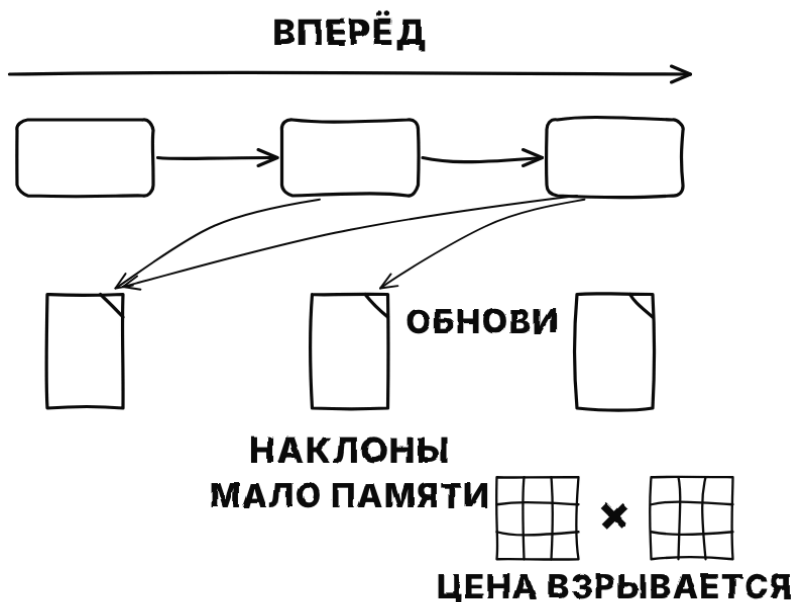
Книга добавляет, что для этой задачи есть два семейства методов — прямой режим и обратный режим — и что различаются они только порядком, в котором сочетаются наклоны частей. Уже здесь она объявляет, что обратный, то есть обратное распространение ошибки в нейронных сетях, для нашей цели гораздо лучше. Те же идеи переносятся на цепочки, которые ветвятся и сливаются, и на модели, использующие одни и те же веса в нескольких местах, — к ним книга вернётся в главе о рекуррентных моделях.

Прежде чем пойти по любому из путей, книга откладывает в сторону два других способа дифференцирования. Первый — тот самый опыт, который вы только что представили: сдвинуть и прогнать заново; он называется численным дифференцированием и, требуя двух прогонов на каждое число, годится лишь для выборочной проверки того, что более хитрый метод даёт правильный ответ. Второй — символьное дифференцирование, то есть школьный анализ в исполнении программы компьютерной алгебры: вы даёте ей формулу, и она возвращает формулу производной. Книга показывает крошечный пример — несколько строк на Python с библиотекой SymPy: короткую формулу с синусом, производную которой программа печатает.

Беда символьного пути — расточительность. Формула производной повторяет куски исходной — тот же синус и тот же косинус, — а разумная программа вычислила бы каждый из них один раз и сохранила. Найти самый лучший способ разделить такую работу, чтобы ничего не вычислялось дважды, оказывается одной из тех задач, для которых не известно быстрого общего метода. Автоматическое дифференцирование

обходит это, используя то, что у нас уже есть, — цепочку шагов, которую программа действительно выполнила, — и потому справляется гораздо лучше простого символьного подхода.

Теперь первый путь — прямой режим. Представьте, что вы идёте вдоль цепочки в ту сторону, куда работает программа, с блокнотом в руках. Каждый раз, когда срабатывает новая часть, вы открываете чистую страницу для её весов и записываете, как её выход от них зависит, а ещё возвращаетесь и обновляете каждую уже исписанную страницу, потому что новая часть передаёт дальше все прежние влияния. Когда вы доходите до конца, сложение итоговых ошибок превращает каждую страницу в нужный вам наклон. Хорошая сторона в том, что эта бухгалтерия идёт прямо рядом с программой, так что ей нужно примерно столько же памяти, сколько самой программе.



Прямой режим: блокнот

Плохая сторона — цена каждого обновления. Обновить страницу — значит перемножить две целые таблицы наклонов, и объём этой работы растёт как квадрат размера пачки и как квадрат числа признаков в каждом примере. Для пачки фотографий, где у каждого примера многие тысячи значений пикселей, стоимость взрывается, и взрывается снова для каждой прежней страницы, которую нужно обновить. Как и опыт со сдвигом ручек, прямой режим верен — и слишком медленен для моделей в вашем телефоне. Здесь опущены точные формулы обновлений и точная оценка стоимости; они в разделе о дифференцировании в прямом режиме.

Запомним

Прогон модели — это цепочка шагов, которая всегда заканчивается одним числом, общей ошибкой на пачке, и автоматическое дифференцирование должно найти наклон этого числа сразу по каждому набору весов. Сдвиг каждой ручки с повторным прогоном работает, но стоит двух прогонов на число и поэтому годится лишь для проверки, а простое символьное дифференцирование тратит силы на повторяющиеся куски. Прямой режим идёт по цепочке в порядке работы программы и на каждом шаге обновляет страницу наклонов для каждого прежнего набора весов. Памяти ему нужно мало, но каждое обновление перемножает целые таблицы, и эта стоимость взрывается для больших пачек и изображений, так что нужен лучший порядок.

Вопросы

1. Почему цепочка шагов при обучении всегда заканчивается одним числом?
 - a. Потому что в конце программы компьютер может хранить только одно число
 - b. Потому что именно одно число — общую ошибку — оптимизатор может уменьшать
 - c. Потому что каждый примитив цепочки выдаёт одно число

- d. Потому что последний шаг модели — всегда перекрёстная проверка
2. Почему сдвиг каждой ручки с повторным прогоном модели годится лишь для выборочной проверки?
- a. Потому что для большинства ручек он даёт неверные наклоны
 - b. Потому что он работает только для моделей без весов
 - c. Потому что ему нужно два прогона на каждое число — миллионы прогонов на шаг обучения
 - d. Потому что его нельзя применить к пачке фотографий
3. В чём расточительность простого символьного дифференцирования?
- a. Формула производной повторяет куски — тот же синус и косинус, — которые следовало бы вычислить один раз
 - b. Оно даёт лишь приближённую производную
 - c. Ему нужен новый прогон модели для каждого веса
 - d. Оно не справляется с формулой, в которой есть синус
4. Прямому режиму нужно мало памяти. Почему же он слишком медленен для модели фотографий?
- a. Потому что ему приходится прогонять всю модель по разу на каждую ручку
 - b. Потому что каждое обновление перемножает целые таблицы, и стоимость растёт как квадрат размера пачки и числа признаков
 - c. Потому что он идёт по цепочке назад и должен хранить каждый шаг

d. Потому что он может обрабатывать только один пример за раз

Ответы

1. **b)** Последнее звено измеряет ошибку на каждом примере, а заключительный шаг их складывает; именно одно число оптимизатор может уменьшать.

2. **c)** Численное дифференцирование стоит два прогона на число, так что при миллионах ручек одному шагу обучения понадобились бы миллионы полных прогонов.

3. **a)** Формула производной повторяет куски исходной, а найти лучший способ разделить такую работу — задача, для которой не известно быстрого общего метода.

4. **b)** Обновить страницу — значит перемножить две целые таблицы наклонов, и эта работа растёт как квадрат размера пачки и числа признаков и повторяется для каждой прежней страницы.

Разобранные примеры

Считаем прогоны при сдвиге ручек Численному дифференцированию нужно два прогона модели на каждое число, наклон которого мы хотим найти. Сколько полных прогонов понадобится на один шаг обучения модели с 1 000 000 весов?

Каждый вес — одно число, наклон которого нужен, так что найти нужно 1 000 000 наклонов.

Каждый наклон стоит двух прогонов всей модели.

Значит, всего $2 \times 1\,000\,000$ прогонов.

Ответ: 2 000 000 полных прогонов на один шаг обучения

Задачи

1. В прямом режиме стоимость каждого обновления растёт как квадрат размера пачки. Во сколько раз вырастет стоимость одного обновления, если удвоить пачку фотографий с 32 до 64?

Подсказка: Квадрат: что происходит с квадратом числа, когда само число удваивается?

Ответ: В 4 раза: стоимость растёт как квадрат размера пачки, а удвоение числа делает его квадрат в четыре раза больше ($64^2 / 32^2 = 4$).

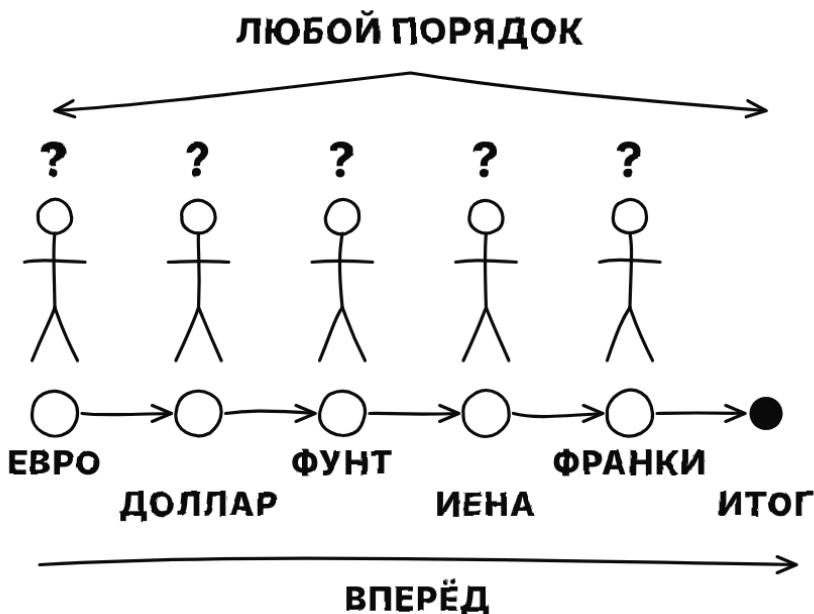
2. Объясните своими словами картину прямого режима с блокнотом: что записывается на новой странице и почему на каждом шаге нужно обновлять прежние страницы?

Подсказка: Подумайте, что каждый новый шаг делает с влиянием всех прежних весов на результат.

Ответ: Когда срабатывает новая часть, на чистой странице записывается, как её выход зависит от её собственных весов. Каждую прежнюю страницу тоже нужно обновить, потому что новая часть передаёт дальше влияние всех прежних весов, и их воздействие на результат меняется на каждом шаге.

Идём по цепочке назад

Выстройте пятерых друзей в ряд и сыграйте в игру с обменными курсами. Первый меняет ваши евро на доллары, следующий — доллары на фунты, затем на иены, затем на франки, а последний говорит, во сколько франков в итоге превращается один евро. Пять курсов можно перемножить в любом порядке и получить тот же ответ, потому что умножению порядок безразличен. Но допустим, вы хотите узнать для каждого друга в ряду, насколько изменение его курса сдвинет итоговое число; тогда порядок работы вдруг становится очень важен.

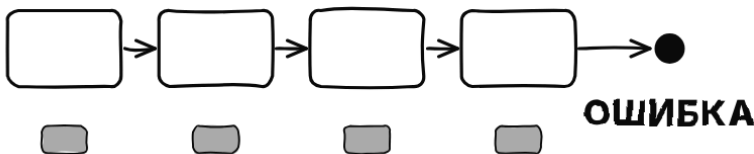


Пять друзей, пять курсов

Именно такой поворот делает книга в разделе об обратном режиме. Если выписать наклон итоговой ошибки по одному набору весов, получится длинный ряд перемноженных таблиц наклонов: таблицы всех последующих шагов, а на дальнем конце — таблица весов этого одного шага. Поскольку умножение можно группировать в любом порядке, выбор за нами. Прямой режим, как мы видели, работал в том порядке, в каком шла программа, и расплачивался за это, снова и снова перемножая целые таблицы.

Главное наблюдение книги: цепочка заканчивается одним числом — общей ошибкой. Поэтому, если начать умножение с этого конца, вы несёте никогда не целую таблицу, а лишь одну строку чисел, а строка на таблицу — это дёшево и даёт снова строку. Более того, строка для одного шага — это просто строка следующего шага, умноженная ещё раз; так что все их можно построить, идя по цепочке назад, шаг за шагом. Это и есть автоматическое дифференцирование в обратном режиме — обратное распространение ошибки. Сначала вся программа работает вперёд и сохраняет каждый промежуточный результат; затем зеркальная программа идёт назад от ошибки к первому шагу и на каждом шаге выдаёт наклон для весов этого шага и передаёт строку дальше назад. Строки, которые она несёт, книга называет сопряжёнными переменными: каждая говорит, насколько сильно итоговая ошибка отзывается на один промежуточный результат.

СТРОКА



НАКЛОН

ПРЯМОЙ ПРОХОД



ОБРАТНЫЙ ПРОХОД

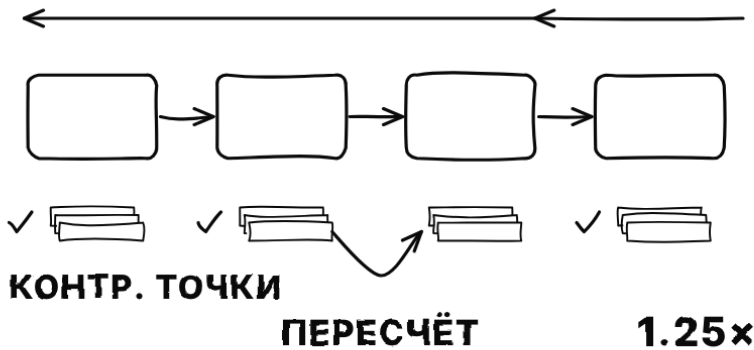
Несём одну строку назад

На языке нейронных сетей обычный прогон — это прямой проход, не путать с прямым режимом, а путь назад — обратный проход. Каждая операция обратного прохода — строка на таблицу, и её стоимость растёт лишь пропорционально участвующим размерам, поэтому обратный режим гораздо быстрее прямого. Когда обучали ассистента в вашем телефоне, каждый из его бесчисленных шагов обучения состоял из одного прямого и одного такого обратного прохода.

Плата — память. Обратному проходу нужен каждый промежуточный результат прямого прохода, поэтому все они должны храниться, пока путь назад до них не дойдёт.

Контрольные точки градиента облегчают это: хранят лишь

несколько результатов — контрольные точки, — а остальное пересчитывают, когда оно понадобится. На рисунке книги цепочка из четырёх блоков: сохраняются только выходы первого, второго и четвёртого; обратный проход останавливается на третьем, снова идёт вперёд от ближайшей контрольной точки, чтобы восстановить недостающее, а затем продолжает путь — в этом примере ценой на четверть большего объёма вычислений. Чем меньше контрольных точек вы храните, тем больше пересчитываете: память меняется на время. Здесь опущены формулы обратной программы и пошаговый алгоритм; они в разделе о дифференцировании в обратном режиме.



Контрольные точки: память за время

Запомним

Наклон ошибки по любым весам — это длинное произведение таблиц, и порядок этого произведения выбираем мы. Поскольку цепочка заканчивается одним числом, начать с конца — значит нести только строку, что дёшево, и строить каждую строку из следующей, идя назад. Это и есть обратное распространение ошибки: прямой проход, который сохраняет каждый промежуточный результат, затем обратный проход, который выдаёт каждый наклон. Оно гораздо быстрее прямого режима, но требует много памяти, и контрольные точки уменьшают этот расход, пересчитывая часть результатов вместо того, чтобы их хранить.

Вопросы

1. Почему начало умножения с конца цепочки делает обратный режим дешёвым?
 - a. Потому что последние таблицы цепочки всегда самые маленькие
 - b. Потому что цепочка заканчивается одним числом, так что нести нужно лишь строку, а строка на таблицу — это дёшево
 - c. Потому что обратный проход пропускает шаги без весов
 - d. Потому что умножение с конца даёт более точный ответ

2. Что должно произойти, прежде чем сможет начаться обратный проход?
 - a. Вся программа должна отработать вперёд и сохранить каждый промежуточный результат
 - b. Каждый вес нужно один раз сдвинуть, чтобы проверить его наклон
 - c. Таблицы наклонов всех шагов нужно перемножить между собой
 - d. Пользователь должен выбрать контрольные точки для каждого шага
3. Что на что меняют контрольные точки градиента?
 - a. Точность наклонов на скорость прямого прохода
 - b. Скорость обратного прохода на меньшую итоговую ошибку
 - c. Память на дополнительные вычисления: меньше хранится — больше пересчитывается
 - d. Прямой проход на второй обратный проход
4. Почему в игре с обменными курсами порядок работы важен, если умножению порядок безразличен?
 - a. Потому что курсы меняются, пока вы умножаете
 - b. Потому что на итоговое число влияет только курс последнего друга
 - c. Потому что итоговое число зависит от порядка
 - d. Потому что вам нужно влияние курса каждого друга, а при одних порядках работу можно использовать повторно гораздо лучше, чем при других

Ответы

1. **b)** Цепочка заканчивается одним числом — общей ошибкой, — поэтому, начав с этого конца, вы несёте лишь одну строку, а строка на таблицу — это дешево и даёт снова строку.
2. **a)** Сначала вся программа работает вперёд и сохраняет каждый промежуточный результат; только потом зеркальная программа идёт назад от ошибки.
3. **c)** При контрольных точках хранится лишь несколько результатов, а остальное пересчитывается по мере надобности; чем меньше контрольных точек, тем больше пересчёт — память меняется на время.
4. **d)** Ответ одинаков при любом порядке, но когда нужно влияние каждого шага, от выбора порядка зависит, сколько работы повторится.

Разобранные примеры

Цена одного пропуска контрольной точки В цепочке четыре блока. При контрольных точках сохраняются только выходы первого, второго и четвёртого блоков. Во время обратного прохода выход третьего блока отсутствует, и его нужно восстановить, снова пройдя вперёд от ближайшей контрольной точки. Насколько больше вычислений нужно по сравнению с хранением всего?

Обычный прямой проход один раз выполняет все четыре блока.

Обратный проход останавливается на третьем блоке, результат которого не сохранён.

Пройти вперёд заново от ближайшей контрольной точки, выхода второго блока, — значит выполнить третий блок ещё раз.

Значит, сверх четырёх блоков добавляется работа ещё одного — пять блоков вместо четырёх.

Ответ: В 1,25 раза больше вычислений, на четверть больше, — в обмен на то, что результат одного блока не хранится

Задачи

1. Что произошло бы с памятью и вычислениями во время обратного прохода, если бы вы не хранили вообще никаких контрольных точек, кроме входа?

Подсказка: Спросите себя, что приходится восстанавливать каждый раз, когда обратному проходу нужен недостающий результат.

Ответ: Памяти понадобилось бы совсем мало, но каждый недостающий результат пришлось бы восстанавливать, проходя вперёд с самого начала, так что вычислений стало бы гораздо больше. Чем меньше контрольных точек вы храните, тем выше стоимость обратного прохода.

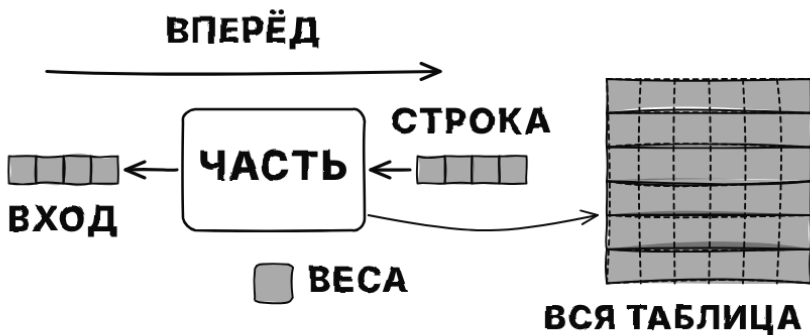
2. Объясните своими словами, что такое сопряжённая переменная и почему каждую можно построить из следующей по цепочке.

Подсказка: Подумайте, что измеряет строка, которую несут назад, и что к ней добавляет ещё один шаг назад.

Ответ: Сопряжённая переменная говорит, насколько сильно итоговая ошибка отзывается на один промежуточный результат. Шаг дальше назад означает лишь ещё одно умножение на таблицу наклонов этого шага, поэтому каждая сопряжённая переменная получается из следующей, и, идя назад, мы строим их все.

Каждая часть учится отвечать назад

Спросите продавщицу, как изменится цена торта, если мука подорожает на один цент. Ей не нужно показывать вам всю таблицу расчёта себестоимости пекарни; хороший продавец просто ответит: «примерно на полцента», — потому что знает, что идёт в торт. В этом и дух этого раздела: на обратном пути никому не нужно видеть полные таблицы наклонов — только ответ, который даёт каждая часть, когда ей передают строку, идущую назад.



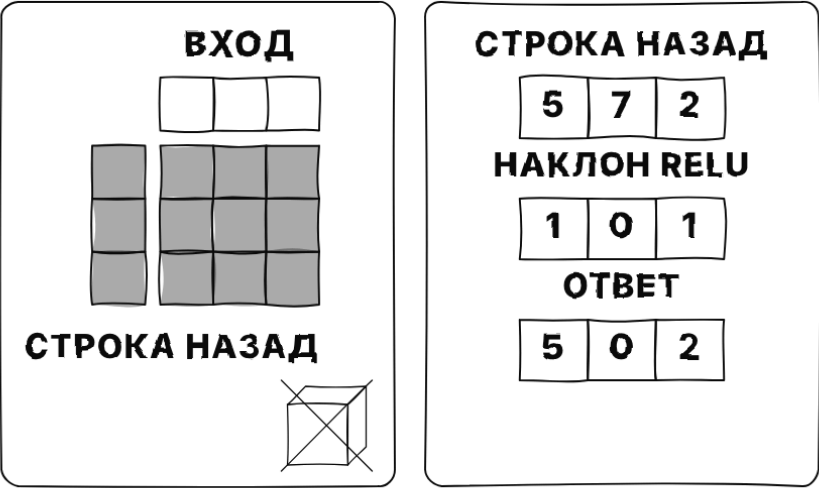
Два ответа назад

Взгляните ещё раз на то, что на самом деле делает обратный проход на каждом шаге. Он берёт строку, которую несёт, — сопряжённую переменную — и умножает её на одну таблицу текущей части: один раз на таблицу для весов части, чтобы получить их наклон, и один раз на таблицу для входа части, чтобы получить строку для следующего шага назад. Строка на таблицу — вот и всё. Книга даёт этой операции имя — произведение вектора на якобиан, — и у каждой части их два: одно для входа, другое для весов.

Отсюда следует главный практический результат. Чтобы построить целую систему автоматического дифференцирования, вы выбираете набор примитивов и учите каждый, помимо его обычной прямой работы, отвечать на эти два обратных вопроса. Полные таблицы наклонов ни в какой момент не нужно выписывать в память. Рисунок книги изображает примитив как блок: его прямая стрелка идёт в одну сторону, а два маленьких обратных ответа выходят в другую — один для входа, другой для весов. И ничего не теряется: если вам всё-таки понадобится полная таблица, её можно восстановить по одной строке, задавая обратный вопрос с простыми пробными строками.

Два примера показывают, почему ответы настолько дешевле таблиц. Для самой распространённой части сети — умножения входа на таблицу весов — полная таблица наклонов по весам была бы огромным трёхмерным блоком чисел. И всё же обратные ответы просты: строка, передаваемая назад, — это просто пришедшая строка, пропущенная через те же веса, повернутые набок, а наклон для весов — обычная сетка из двух списков, пришедшей строки и входа, где каждый элемент — произведение одной пары. Для частей-переключателей между

слоями, функций активации вроде ReLU, которые действуют на каждое число отдельно, полная таблица почти целиком состояла бы из нулей; обратный ответ — просто масштабировать каждое число пришедшей строки на собственный наклон этой части в этой точке. Каждый раз, когда обучалась модель фотографий в вашем телефоне, её обратный проход был длинной последовательностью именно таких маленьких ответов. Здесь опущены сами формулы и короткий вывод ответа для весов; они в подразделе о произведениях вектора на якобиан.



Дешёвые ответы, два примера

Запомним

Каждый шаг обратного прохода — один и тот же ход: строка, которую несут назад, на одну таблицу текущей части. Такие ходы называются произведениями вектора на якобиан, и каждой части нужно два — одно для входа и одно для весов. Так что система автоматического дифференцирования — это набор частей, каждая из которых умеет отвечать на эти два обратных вопроса, а полные таблицы наклонов никогда не хранятся. Для умножения на веса ответы — это повернутые набок веса и сетка произведений; для активации — простое масштабирование каждого числа.

Вопросы

1. Почему система автоматического дифференцирования может работать, ни разу не выписывая полные таблицы наклонов?
 - a. Потому что в таблицах одни нули, кроме первой строки
 - b. Потому что каждой части нужно лишь ответить, чему равна пришедшая строка, умноженная на её таблицу, а этот ответ можно вычислить напрямую
 - c. Потому что обратному проходу нужен только наклон последней части
 - d. Потому что таблицы вычисляются один раз и выбрасываются до начала обучения

2. Сколько обратных ответов должна давать часть с весами и для чего они?
 - a. Один — наклон итоговой ошибки
 - b. Три — для входа, для весов и для выхода
 - c. Два: один для входа, дающий строку для следующего шага назад, и один для весов, дающий их наклон
 - d. Ни одного: библиотека вычисляет их по одному лишь прямому проходу
3. Каков обратный ответ функции активации вроде ReLU?
 - a. Масштабировать каждое число пришедшей строки на собственный наклон части в этой точке
 - b. Умножить пришедшую строку на огромную таблицу весов
 - c. Повернуть пришедшую строку набок и передать дальше без изменений
 - d. Составить сетку произведений из пришедшей строки и входа

Ответы

1. **b)** Каждый шаг обратного прохода — строка, которую несут назад, на одну таблицу, и каждая часть может ответить на это напрямую, поэтому полные таблицы никогда не хранятся.
2. **c)** Обратный проход умножает строку один раз на таблицу весов — ради наклона весов — и один раз на таблицу входа — ради строки, которая идёт дальше назад.
3. **a)** Активация действует на каждое число отдельно, поэтому её полная таблица почти целиком из нулей, а ответ — простое масштабирование каждого числа.

Разобранные примеры

Сетка произведений для весов Часть умножает свой вход на таблицу весов. Вход — список (1, 4), а пришедшая назад строка — (2, 3). Книга показывает, что наклон для веса в строке i и столбце j — это i -е число пришедшей строки, умноженное на j -е число входа. Постройте сетку наклонов для весов.

Каждый элемент — одно произведение пары: число пришедшей строки для его строки и число входа для его столбца.

Первая строка, пришедшее число 2: $2 \times 1 = 2$ и $2 \times 4 = 8$.

Вторая строка, пришедшее число 3: $3 \times 1 = 3$ и $3 \times 4 = 12$.

Никакой трёхмерный блок не понадобился — только эти четыре произведения.

Ответ: В сетке 2, 8 в первой строке и 3, 12 во второй

Задачи

1. Часть ReLU получила входы (−1, 2, 5). Её наклон равен 0 там, где вход был отрицательным, и 1 там, где положительным. Назад приходит строка (4, 7, 2). Какую строку ReLU передаёт дальше назад?

Подсказка: Масштабируйте каждое число пришедшей строки на наклон ReLU при соответствующем входе.

Ответ: (0, 7, 2): первый вход был отрицательным, поэтому его наклон 0, и 4 превращается в 0; два других наклона равны 1, так что 7 и 2 проходят без изменений.

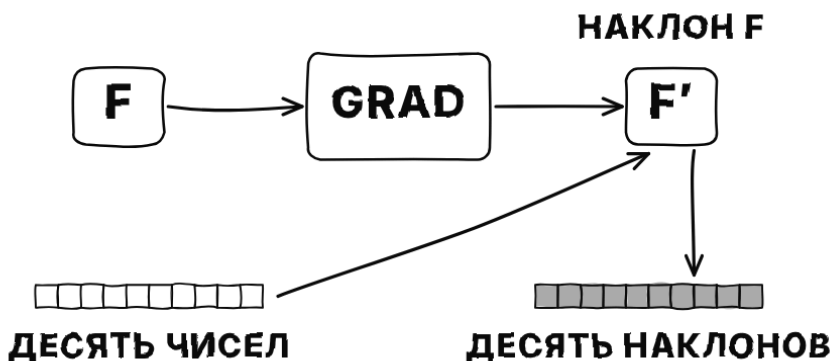
2. Если кому-то всё же нужна полная таблица наклонов части, как получить её, пользуясь только её обратным ответом?

Подсказка: Подумайте о том, чтобы задавать обратный вопрос с очень простыми пробными строками.

Ответ: Задайте обратный вопрос по разу для каждой простой пробной строки, в которой одна единица, а остальное нули; каждый ответ — одна строка полной таблицы, так что таблица восстанавливается по одной строке.

Как это делают библиотеки и почему победила ReLU

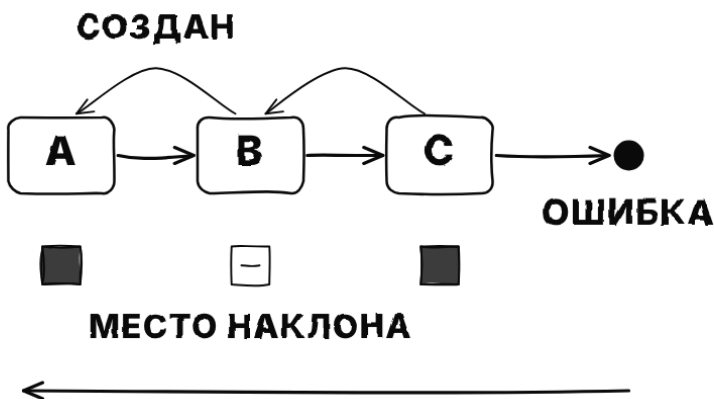
Откройте калькулятор в телефоне и наберите одну четверть, 0,25. Умножьте её на 0,25, и ещё раз, и ещё, всего десять раз. Вы ожидали маленькое число; получается примерно одна миллионная, а после двадцати таких шагов — примерно одна миллион-миллионная. Запомните это обрушение: оно решает, какими функциями активации может пользоваться глубокая сеть, и им заканчивается эта часть.



Четверть, а затем изготовители функций

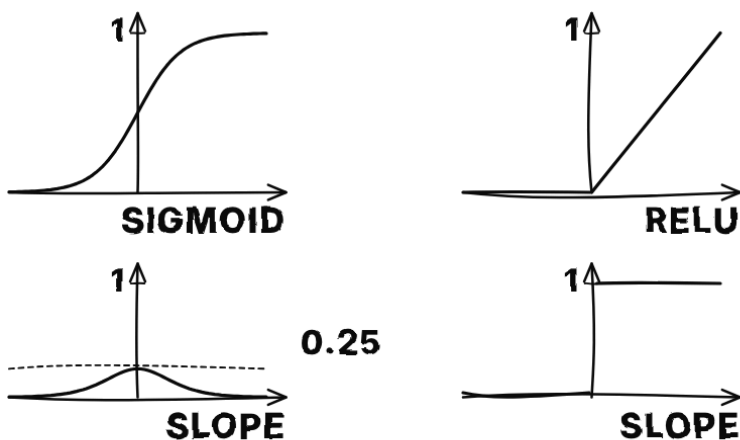
Сначала книга смотрит, как настоящие библиотеки устраивают обратный проход. Построить его можно несколькими способами — от ведения списка каждой операции, как делает TensorFlow, до переписывания собственного кода программы в обратную версию. В библиотеке в стиле функций, такой как JAX, идея из предыдущей части становится совсем буквальной: обратный ответ — это функция, которая берёт функцию и точку и возвращает новую функцию, ждущую пришедшую строку. Точно так же и сам градиент — функция, которая берёт вашу функцию и возвращает другую функцию, вычисляющую наклон в любой точке, которую вы ей дадите. Маленький пример книги делает именно это для суммы квадратов десяти чисел: вы запрашиваете её градиент и получаете функцию, ответ которой для любых десяти чисел — снова список из десяти наклонов. PyTorch предлагает тот же стиль в отдельном модуле, а сочетая эти функции-изготовители, можно вычислять и более сложные вещи, например наклоны наклонов.

Моделям в стиле объектов — обычный способ писать на PyTorch — нужен ещё один шаг. Один вариант — вынуть веса из объекта модели и вызывать её как обычную функцию, передавая эти веса. Чаще PyTorch делает работу за кулисами: каждый вычисленный им результат помнит, какая операция его создала, и у него есть место для собственного наклона, поначалу пустое. Когда вы просите обратный проход для итоговой ошибки, PyTorch идёт по этим воспоминаниям назад по цепочке и заполняет места у всех величин, которые попросили сохранить наклон, — только у них, чтобы не тратить память зря. Каждый раз, когда вы видите, как модель дообучают на ноутбуке, этот вызов обратного прохода совершает ровно такой путь.



Результаты, которые помнят

Теперь вернёмся к калькулятору. При обратном проходе каждая функция активации умножает идущую назад строку на свой наклон. Если этот наклон всюду меньше единицы, глубокая сеть снова и снова умножает строку на маленькое число, и наклоны первых слоёв сжимаются к нулю — это проблема затухающего градиента. Если он всюду больше единицы, они растут без предела — проблема взрывающегося градиента. Это не только теория: компьютеры хранят числа с ограниченной точностью, так что слишком маленькие или слишком большие числа просто теряются уже через не так уж много слоёв. Любопытное отступление в книге: именно из-за этой ограниченной точности стопка чисто линейных слоёв не идеально линейна, и этим даже пользовались для обучения таких сетей.



Сигмоида сжимает, ReLU пропускает

Сигмоида, гладкая S-образная кривая, популярная в прошлом, показывает беду в худшем виде: её наклон никогда не превышает одной четверти — того самого числа, которое вы только что умножали само на себя. Рисунок книги изображает S-кривую вместе с её наклоном — низким горбом с вершиной в четверть посередине, сходящим на нет по обе стороны. У идеальной активации наклон был бы всюду ровно единица, но это функция, которая вообще ничего не делает и не отделяет один слой от следующего. ReLU — это компромисс: для отрицательных входов её наклон равен нулю, так что строка там просто выключается, а для положительных — единице, так что строка проходит без изменений, не сжимаясь и не разрастаясь. Её график — ровная линия, переходящая в прямой пандус, а наклон — ступенька от

нуля до единицы. Ещё один подарок: поскольку ReLU лишь обнуляет отрицательные числа, её наклон можно прочесть и по выходу, и по входу, так что вход хранить не нужно, и библиотеки могут затираять его на месте, экономя память. Здесь опущены точные обратные формулы и код этих библиотек; их даёт подраздел о реализации обратного режима.

Запомним

Библиотеки строят обратный проход по-разному: JAX превращает градиенты в функции, возвращающие функции, а в PyTorch каждый результат помнит создавшую его операцию и во время обратного прохода заполняет своё место для наклона. При обратном проходе каждая активация умножает строку на свой наклон, так что наклон, всюду меньший единицы, заставляет градиенты в глубоких сетях затухать, а больший единицы — взрываться. Наклон сигмоиды не больше одной четверти — поэтому она и не справляется в глубоких сетях. ReLU пропускает строку либо без изменений, либо совсем не пропускает, что сохраняет градиенты здоровыми, и даже позволяет библиотекам экономить память, затирая её вход.

Вопросы

1. Почему глубокие сети с сигмной страдают от затухающих градиентов?

- a. Потому что наклон сигмоиды не больше одной четверти, и строка, идущая назад, сжимается на каждом слое
 - b. Потому что у сигмоиды совсем нет наклона при отрицательных входах
 - c. Потому что наклон сигмоиды больше единицы, и числа переполняются
 - d. Потому что сигмоиду нельзя вычислить с ограниченной точностью
2. Почему бы не взять функцию активации с наклоном, всюду равным ровно единице?
- a. Потому что компьютер не может хранить единицу точно
 - b. Потому что такая функция заставила бы градиенты взрываться
 - c. Потому что единственная такая функция ничего не делает и не отделяет один слой от другого
 - d. Потому что её обратному ответу понадобилась бы полная таблица наклонов
3. Откуда PyTorch знает, куда идти при обратном проходе модели в объектном стиле?
- a. Он снова запускает модель, проходя строки кода в обратном порядке
 - b. Каждый результат помнит операцию, которая его создала, и обратный проход идёт назад по этим воспоминаниям
 - c. Он просит пользователя перечислить операции модели
 - d. Он превращает модель в список весов и дифференцирует каждый, слегка сдвигая его

4. Почему библиотека может затереть вход ReLU, не сломав обратный проход?
- a. Потому что у ReLU нет обратного ответа
 - b. Потому что наклон ReLU всегда равен единице
 - c. Потому что обратный проход никогда не заходит в функции активации
 - d. Потому что ReLU лишь обнуляет отрицательные значения, так что её наклон можно прочитать и по входу

Ответы

1. **a)** Каждая функция активации умножает идущую назад строку на свой наклон, а наклон сигмоиды никогда не превышает одной четверти, так что множество слоёв сжимает наклоны ранних слоёв почти до нуля.
2. **c)** Наклон, всюду равный единице, означает функцию, которая вообще ничего не делает и не разделяет слои; ReLU — это компромисс.
3. **b)** Каждый результат, который вычисляет PyTorch, помнит, какая операция его создала, и обратный проход идёт по этим воспоминаниям назад по цепочке, заполняя ячейки для наклонов.
4. **d)** ReLU оставляет положительные значения без изменений и обнуляет отрицательные, поэтому её наклон одинаков, читать ли его по входу или по выходу, и вход хранить не нужно.

Разобранные примеры

Десять сигмоидных слоёв в лучшем случае Наклон сигмоиды никогда не превышает одной четверти. При обратном проходе каждый сигмоидный слой умножает идущую назад

строку на свой наклон. Во сколько раз, даже в лучшем случае, уменьшится наклон, пройдя назад через десять сигмоидных слоёв?

Лучший случай — наибольший наклон, одна четверть, на каждом слое.

Каждый слой умножает на 0,25, так что десять слоёв умножают на 0,25 десять раз подряд.

0,25 в десятой степени — примерно 0,00000095, около одной миллионной.

Ответ: Не меньше чем примерно в миллион раз, даже в лучшем случае

Задачи

1. Строка, идущая назад, проходит через пять слоёв ReLU, и на каждом слое вход того числа, за которым мы следим, был положительным. На что умножилось это число?

Подсказка: Каков наклон ReLU при положительном входе?

Ответ: Ровно на единицу: наклон ReLU при положительных входах равен единице, а единица, умноженная на себя пять раз, остаётся единицей, так что число проходит назад без изменений — не сжимаясь и не вырастая.

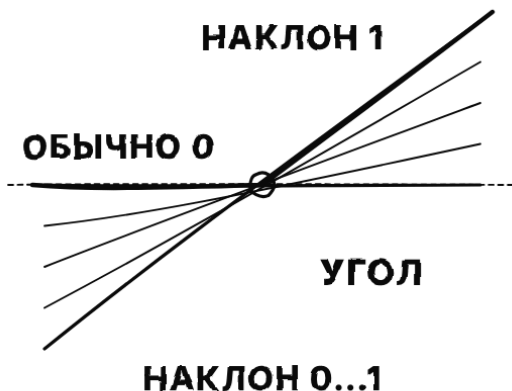
2. Пример книги просит градиент суммы квадратов десяти чисел. Сколько чисел вернёт функция-градиент, если вызвать её на каких-то десяти числах, и почему?

Подсказка: Градиент содержит по одному наклону на каждое число, которое функция принимает на вход.

Ответ: Десять: градиент даёт по одному наклону на каждое из десяти входных чисел, так что ответ — снова список из десяти чисел.

Угол пандуса

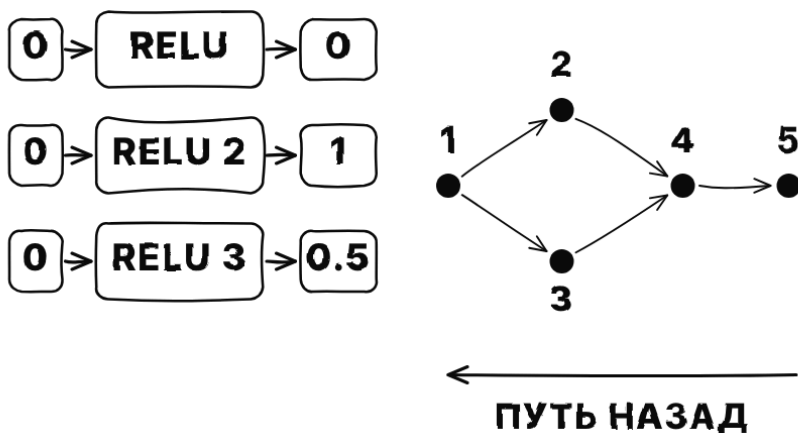
Положите линейку на пол у основания пандуса, там, где ровный пол встречается со склоном. Линейку можно наклонить так, чтобы она лежала ровно вдоль пола, или поднять вдоль пандуса, или поставить где-то посередине, — и в каждом положении она всё равно касается угла, оставаясь целиком под полом и пандусом. На гладком полу или на гладком пандусе у линейки только один способ лечь; в углу их целый веер. Ровно в этом трудность с ReLU, чей график — ровный пол, а затем прямой пандус, и это последний вопрос главы: какой наклон брать в углу?



Веер линеек в углу

Сначала практический ответ. Обучение начинается со случайных чисел и движется маленькими шумными шагами, так что шанс попасть ровно в угол — на вход, равный ровно нулю, — практически нулевой; везде в других местах наклон совершенно определён. Технический ответ — субградиент: любой наклон прямой, которая касается кривой в точке и остаётся ниже всей кривой, как та линейка. Там, где кривая гладкая, такая прямая всего одна — обычная производная; в углу их много, и вместе они образуют субдифференциал. Для ReLU в нуле подойдёт любой наклон от нуля до единицы, и большинство библиотек просто выбирают ноль. Обучение, которое на каждом шаге выбирает такой наклон, называется субградиентным спуском.

Книга добавляет предупреждение. Для кривых, изгибающихся в обе стороны, субградиентов может и не быть, и математики используют вместо них более свободную, локальную версию. Хуже того, две программы, вычисляющие ровно одну и ту же функцию, могут сообщать в углу разные наклоны: книга приводит случай, когда три способа записать ReLU дают в PyTorch в одной и той же точке ноль, единицу и одну вторую.



Одна функция — разные наклоны

Глава заканчивается практикой. Книга советует построить крошечную систему обратного режима самому и указывает на micrograd, небольшую учебную версию; как только цепочки начинают ветвиться и сливаться, шаги нужно сначала расставить в правильном порядке, а уже потом идти назад. Ещё она

предлагает написать новый примитив с собственными обратными ответами, в PyTorch или в JAX, и прочитать руководство JAX по автоматическому дифференцированию — о наклонах наклонов и многом другом. Здесь опущены строгие определения и подробности этих упражнений; они — в подразделе о субдифференцируемости и в заключительном разделе. На этом всё устройство, которое стоит за обучением каждого чат-ассистента и каждого фотоприложения, на месте: одна ошибка в конце и быстрый обратный путь, который подсказывает каждой ручке, куда повернуться.

Запомним

У ReLU в нуле угол, где обычного наклона нет. На практике обучение почти никогда не попадает точно туда, а в теории можно использовать любой наклон прямой, касающейся угла снизу, — субградиент; для ReLU это любое значение от нуля до единицы, обычно ноль. Углы всё же могут заставлять разные, но равносильные программы сообщать разные наклоны. Лучший способ всё это понять — построить небольшую систему обратного режима самому.

Вопросы

1. Почему угол ReLU на практике редко мешает обучению?
 - а. Потому что библиотеки убирают угол ещё до начала обучения

- b. Потому что обучение начинается со случайных чисел и движется шумными шагами, так что попасть ровно в ноль практически невозможно
 - c. Потому что наклон ReLU в нуле всегда равен единице
 - d. Потому что обратный проход пропускает слои ReLU
2. Сколько субградиентов в гладкой точке кривой?
- a. Ровно один — обычная производная
 - b. Ни одного: субградиенты есть только в углах
 - c. Два — по одному с каждой стороны
 - d. Целый диапазон от нуля до единицы
3. Какие наклоны — законные субградиенты ReLU в нуле?
- a. Только ноль
 - b. Только единица
 - c. Любое значение от нуля до единицы, причём библиотеки обычно выбирают ноль
 - d. Вообще любое значение, раз наклон там не определён
4. Что показывает пример книги с тремя способами записать ReLU?
- a. Что два из трёх способов вычисляют другую функцию
 - b. Что PyTorch вообще не умеет дифференцировать ReLU
 - c. Что субградиентный спуск никогда не работает для ReLU
 - d. Что программы, вычисляющие одну и ту же функцию, могут сообщать в углу разные наклоны

Ответы

1. **b)** Обучение начинается со случайных чисел и движется маленькими шумными шагами, так что шанс получить вход, равный ровно нулю, практически нулевой, а везде в других местах наклон определён.
2. **a)** Там, где кривая гладкая, есть всего одна прямая, касающаяся её снизу, — обычная производная; много их только в углу.
3. **c)** Для ReLU в нуле любой наклон от нуля до единицы даёт прямую, которая касается угла и остаётся под кривой, и большинство библиотек выбирают ноль.
4. **d)** Три способа записать одну и ту же ReLU дают в PyTorch в одной и той же точке ноль, единицу и одну вторую: из-за углов равносильные программы могут сообщать разные наклоны.

Разобранные примеры

Одна функция — разные наклоны Вторая ReLU записана так: ReLU от минус s плюс s . Третья — половина суммы обычной ReLU и второй. PyTorch берёт для обычной ReLU в нуле наклон 0. Какой наклон в нуле сообщит обратный проход для второй и для третьей?

Вторая функция всюду равна ReLU: при положительных s она даёт $0 + s = s$, при отрицательных s — минус s плюс $s = 0$.

Её обратный проход в нуле складывает наклон части с ReLU, взятый равным 0, и наклон слагаемого плюс s , равный 1: итого 1.

Третья — это половина от (ReLU + вторая ReLU), так что её сообщаемый наклон — половина от $(0 + 1)$.

Значит, три равные функции сообщают в одной и той же точке 0, 1 и 0,5.

Ответ: 1 для второй, 0,5 для третьей — против 0 для обычной ReLU

Задачи

1. Законный ли субградиент ReLU в нуле наклон 0,3? А наклон 1,5?

Подсказка: Представьте прямую с таким наклоном через угол: остаётся ли она ниже и пола, и пандуса?

Ответ: 0,3 годится: он лежит между 0 и 1, и его прямая остаётся под полом и под пандусом. 1,5 — нет: при положительных входах его прямая растёт быстрее пандуса и уходит выше кривой.

2. Объясните своими словами, почему самодельная система обратного режима должна расставить шаги в правильном порядке перед обратным путём, когда цепочка ветвится и сливается.

Подсказка: Подумайте о величине, которую используют два более поздних шага: когда её обратная строка становится полной?

Ответ: Когда цепочки ветвятся и сливаются, одна величина может питать несколько более поздних шагов, и её обратная строка полна только тогда, когда все они вернули свой вклад. Если сначала упорядочить шаги, обратный путь посещает каждую величину лишь после всего, что от неё зависит, и не идёт по неверной дорожке.