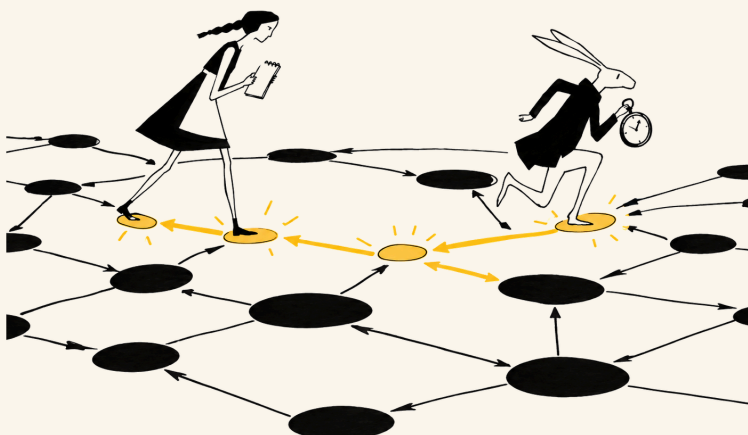


Приключения Алисы в Стране дифференцируемых чудес



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

Рецепт из маленьких шагов

Почему очевидные пути слишком медленны

Путь назад от итоговой оценки

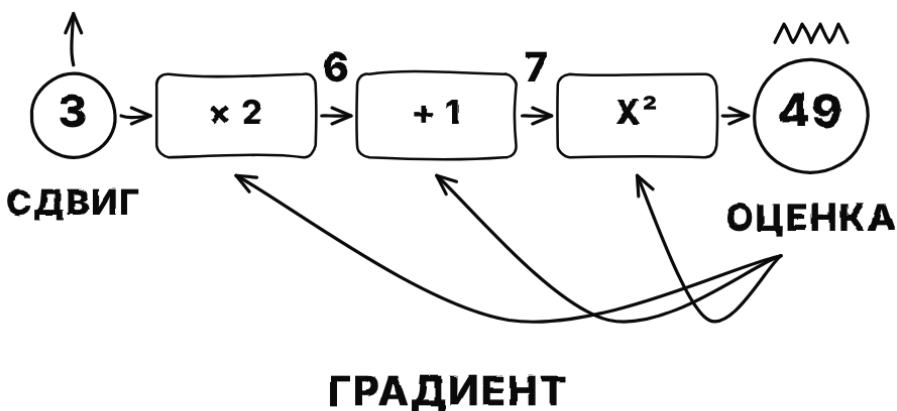
Задать таблице один вопрос

Как библиотеки идут назад и почему победила ReLU

Угол ReLU

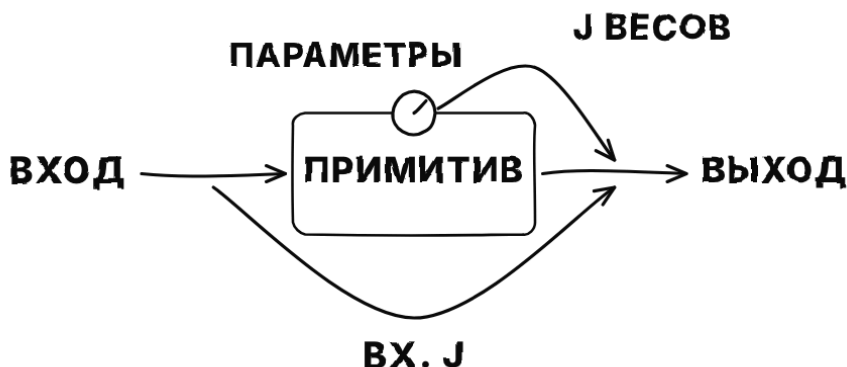
Рецепт из маленьких шагов

Возьмите калькулятор и клочок бумаги и сделайте небольшое вычисление по этапам. Начните с числа три, удвойте его, прибавьте один, а затем возведите результат в квадрат. Получится сорок девять, и каждый этап был крошечным шагом, который вы знаете наизусть. А теперь задайте вопрос потруднее: если чуть-чуть увеличить начальную тройку, насколько сдвинется сорок девять и какой из этапов в какой мере за это отвечает? Этот вопрос, заданный гораздо более длинному рецепту, и есть тема этой главы.



Когда обучают чат-ассистента или фотоприложение в телефоне, программа за ними — именно такой рецепт, только с миллионами этапов и миллионами настраиваемых чисел. В конце рецепта стоит одна оценка, которая говорит, насколько плохо модель справилась с пачкой примеров; книга называет её потерями. Обучение — это поворот каждого настраиваемого числа понемногу в ту сторону, где эта оценка уменьшается, и для этого нужно знать для каждого числа, насколько сильно оценка на него реагирует. Этот набор реакций и есть градиент, а вычислять его машиной для любой программы называется автоматическим дифференцированием. Метод, который строит эта глава, используют все современные библиотеки; специалисты по нейронным сетям называют его обратным распространением ошибки, а специалисты по информатике — автоматическим дифференцированием в обратном режиме.

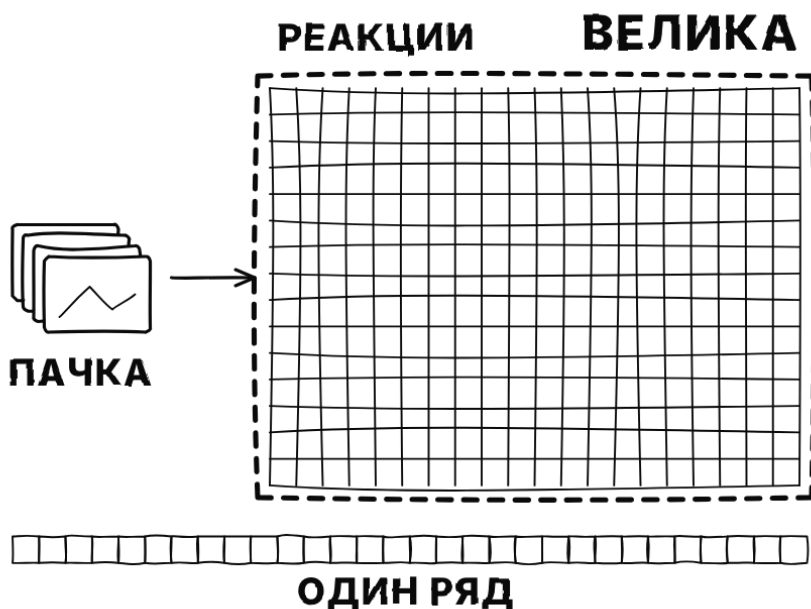
Автор предлагает представить программу как список маленьких операций, которые она действительно выполняет при запуске, одну за другой. Каждую маленькую операцию книга называет примитивом, а всю их цепочку — вычислительным графом. Примитив что-то принимает, пользуется собственными настраиваемыми числами — параметрами — и что-то выдаёт. Что считать одним примитивом — решаем мы: одно умножение таблиц чисел, целый слой сети или даже целую маленькую модель можно считать одним шагом. Это та же привычка, что у программистов повсюду: строить большое из блоков, которые сами построены из блоков.



Один примитив, две реакции

От каждого примитива книга требует только одного. Мы должны знать, как реагирует его выход, если чуть сдвинуть его вход, и как реагирует его выход, если чуть сдвинуть его собственные параметры. Эти две таблицы реакций называются входным якобианом и якобианом по весам, и это единственное местное знание, которое нужно всему методу. На первый взгляд задача на этом решена, ведь в школьном анализе уже есть цепное правило: реакция цепочки шагов — это перемноженные реакции отдельных шагов. Подвох, и настоящая тема главы, в том, чтобы выполнять это умножение достаточно дёшево для модели размером с сеть фотоприложения.

У некоторых шагов есть острый угол, где наклон не определён, например у популярной ReLU, которая пропускает положительные числа и превращает отрицательные в ноль. Книга обещает разобраться с такими углами позже в этой главе, а по-настоящему скачущие операции, вроде случайной выборки, оставлены для её второго тома.



Таблица, которую не удержать

Вот первый намёк на то, почему важна дешевизна. Во время обучения фотоприложение никогда не обрабатывает по одной картинке; оно берёт пачку, скажем несколько десятков картинок, и каждая описана тысячами чисел. Тогда таблица реакций даже одного слоя должна связывать каждое число каждой входящей картинки с каждым числом каждой выходящей. Книга упрощает

дело, выкладывая все эти числа в один длинный ряд, так что «размер входа» всегда означает всё сразу, вместе с пачкой. Если считать так, таблица становится настолько огромной, что мы можем прекрасно знать, как её вычислить, и всё же вовсе не захотеть хранить её в памяти. Запомните эту мысль: в ней ключ к хитрому методу, который появится позже.

Наконец, автор замечает, что библиотеки записывают эти шаги в двух стилях. В JAX всё — функция: шагу передают его вход и его параметры, ровно так, как пишет книга. В PyTorch или TensorFlow слой — это объект, который хранит свои параметры внутри себя, как прибор со встроенными настройками, и вы просто включаете его, подавая вход. Стил «прибора» удобен, например, тем, что слой может подождать с созданием своих параметров, пока впервые не увидит, насколько велик его вход, но он добавляет небольшой слой перевода между записью книги и настоящим кодом. Это различие в стиле снова проявляется в том, как библиотеки двух видов вычисляют градиенты.

Запомним

Обучить модель ИИ — значит сдвигать миллионы настраиваемых чисел, чтобы уменьшить одну итоговую оценку, а автоматическое дифференцирование — машинный способ узнать, как эта оценка реагирует на каждое число.

Программа рассматривается как цепочка маленьких шагов, называемых примитивами, у каждого из которых есть вход, собственные параметры и выход. Для каждого шага нужно знать только, как его выход реагирует на вход и на параметры: его входной якобиан и якобиан по весам.

Цепное правило говорит, как сочетать эти реакции, но таблицы бывают так огромны, что настоящая трудность — сочетать их дёшево. JAX записывает шаги как обычные функции, а PyTorch упаковывает каждый слой вместе с его параметрами в объект, и от этого зависит, как каждая из библиотек вычисляет градиенты.

Вопросы

1. Почему для обучения модели нужно знать, как итоговая оценка реагирует на каждое настраиваемое число?
 - а. Потому что каждое число понемногу поворачивают в ту сторону, где оценка уменьшается
 - б. Потому что оценку нужно напечатать для каждого числа отдельно

- c. Потому что числа, не влияющие на оценку, перед обучением удаляют
 - d. Потому что цепное правило работает, только когда известно каждое число
- 2. Что метод требует знать о каждом отдельном шаге программы?
 - a. Только выход шага для нескольких пробных входов
 - b. Как выход шага реагирует на сдвиги его входа и его собственных параметров
 - c. Сколько времени шаг выполняется на телефоне
 - d. Как итоговая оценка реагирует на этот шаг, заранее вычисленное
- 3. Если цепное правило уже говорит, как сочетать реакции шагов, что остаётся решить?
 - a. Найти новое правило, потому что цепное правило не работает для длинных программ
 - b. Решить, какие шаги считать примитивами, ведь правильный выбор только один
 - c. Сочетать реакции дёшево, потому что таблицы реакций бывают огромными
 - d. Убрать шаги с острыми углами, такие как ReLU
- 4. Чем слой в стиле PyTorch отличается от шага, записанного в стиле JAX?
 - a. У слоя PyTorch вообще нет параметров
 - b. Шаг JAX прячет свои параметры внутри объекта

- с. Слой PyTorch хранит параметры внутри себя, а шаг JAX — это функция, которой передают и вход, и параметры
- д. Разницы нет: оба названия описывают один и тот же стиль

Ответы

- 1. а) Обучение понемногу поворачивает каждое настраиваемое число в ту сторону, где оценка уменьшается, и для этого нужна реакция оценки на каждое число — градиент.
- 2. б) Для каждого примитива нужны только его входной якобиан и якобиан по весам — реакции его выхода на вход и на параметры; всё остальное строится из них.
- 3. с) Цепное правило решает, что вычислять; настоящая трудность главы — выполнять это умножение достаточно дешево для модели размером с сеть фотоприложения.
- 4. с) В JAX всё — функция, которой дают вход и параметры, а в PyTorch слой — это объект со встроенными параметрами, как прибор, который включают, подавая вход.

Разобранные примеры

Кто виноват в коротком рецепте Начните с 3, удвойте, прибавьте 1, затем возведите результат в квадрат — получится 49. Если начальную тройку чуть-чуть увеличить, примерно во сколько раз сильнее сдвинется 49?

Посмотрите на каждый шаг отдельно: удвоение делает любой сдвиг в 2 раза больше.

Прибавление 1 пропускает сдвиг без изменений, то есть умножает его на 1.

Возведение в квадрат около 7 делает сдвиг примерно в 14 раз больше, потому что 7 в квадрате растёт примерно на удвоенное 7 на каждый маленький шаг.

По цепному правилу реакции шагов перемножаются: 2 на 1 на 14.

Ответ: Примерно в 28 раз; если начать с 3,01, получится около 49,28.

Задачи

1. Фотоприложение обучается на пачках по 32 картинки, и один слой принимает по 1000 чисел на картинку и выдаёт по 1000 чисел на картинку. Если выложить всё в один длинный ряд, как делает книга, сколько элементов в таблице, связывающей каждое входящее число с каждым выходящим?

Подсказка: Сначала сосчитайте все входящие числа для всей пачки, затем все выходящие; в таблице по одному элементу на каждую пару.

Ответ: 32 на 1000 — это 32 000 чисел на входе и 32 000 на выходе, так что в таблице 32 000 на 32 000, около миллиарда элементов. Вот почему можно знать, как вычислить такую таблицу, и всё же не хотеть её хранить.

2. Объясните своими словами, почему книга позволяет считать одним примитивом целый слой или даже целую маленькую модель.

Подсказка: Подумайте о том, как программы строятся из блоков, которые сами построены из блоков.

Ответ: Методу нужно знать для каждого шага только, как его выход реагирует на вход и на параметры. Любой блок, для которого это известно, можно считать одним шагом — будь то одно умножение,

слой или целая модель, — так же как программисты строят большое из блоков поменьше.

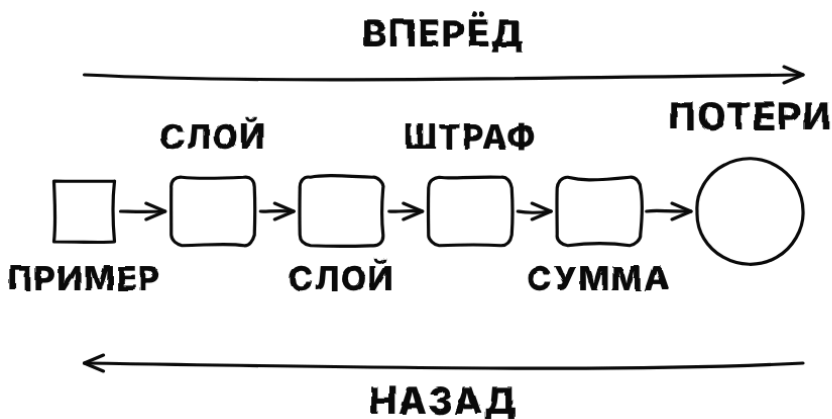
3. Приятель говорит, что ReLU нельзя использовать в этом методе, потому что у неё острый угол, где наклон не определён. Что говорит об этих шагах эта часть?

Подсказка: Отделите шаги с одним острым углом от настоящего скачущих операций.

Ответ: Шаги вроде ReLU, непрерывные, но с углом, можно включить в метод; глава обещает показать как позже. Настоящему скачущим операциям, например случайной выборке, нужны другие приёмы, и они оставлены для второго тома книги.

Почему очевидные пути слишком медленны

Откройте калькулятор в телефоне и попробуйте самый прямой способ узнать, как результат реагирует на маленький сдвиг. Пройдите прежний рецепт — три удвоить, прибавить один, возвести в квадрат — и запишите сорок девять. Затем пройдите его снова, начав с трёх целых одной сотой, и запишите новый результат — чуть больше сорока девяти целых двух десятых. Разность, делённая на величину сдвига, даёт реакцию — около двадцати восьми. Это работает, и понадобилось два прогона всего рецепта, чтобы узнать об одном-единственном начальном числе.



Конвейер заканчивается одним числом

Прежде чем судить об этом методе, книга точно описывает, как выглядит программа обучения. Представьте ряд станций на конвейере. Сырой пример входит на первой станции, каждая станция — один примитив со своими параметрами, и каждая передаёт свой результат следующей. Первые станции — это слои модели; последняя настоящая станция сравнивает ответ модели для каждого примера с правильным и назначает каждому примеру штраф, например перекрёстную энтропию, с которой вы можете встретиться, когда обучают чат-модели; а последний шаг просто складывает штрафы всей пачки. Поэтому конвейер всегда заканчивается одним числом — потерями, ведь именно одно число оптимизатор может толкать вниз по склону. Такую

выписанную последовательность станций книга называет трассой вычисления программы.

С этой картиной задача получает точное название.

Автоматическое дифференцирование — это вычисление, сразу и эффективно, того, как итоговое число реагирует на параметры каждой станции, по одному лишь тому, что мы знаем о каждой станции в отдельности: её входному якобиану и якобиану по весам. Для этого есть два больших семейства методов, и различаются они только порядком, в котором сочетаются реакции станций. Один идёт вдоль конвейера в ту сторону, куда он движется, — прямой режим; другой идёт назад от конца — обратный режим, который специалисты по нейронным сетям называют обратным распространением ошибки. Книга объявляет сейчас и скоро покажет, что для программ вроде наших обратный режим гораздо дешевле. Настоящая модель не всегда прямая линия, а некоторые модели повторно используют одни и те же параметры на нескольких станциях, как рекуррентные модели из одной из следующих глав; то же рассуждение растягивается и на эти случаи.

А теперь вернёмся к калькулятору. То, что вы сделали, называется численным дифференцированием: сдвинуть одно число, прогнать всё заново, сравнить. Для одного числа это два прогона; для чат-ассистента с миллиардами параметров — миллиарды прогонов всей модели на один шаг обучения. Как метод обучения это безнадёжно, и книга оставляет ему только одну скромную работу — проверять, что более хитрый метод даёт правильные ответы.

$$\begin{array}{ccc}
 3 & \xrightarrow{\text{РЕЦЕПТ}} & 49 \\
 3.01 & \xrightarrow{\quad\quad\quad} & 49.28
 \end{array}
 \left. \vphantom{\begin{array}{ccc} 3 & \xrightarrow{\text{РЕЦЕПТ}} & 49 \\ 3.01 & \xrightarrow{\quad\quad\quad} & 49.28 \end{array}} \right] \text{РАЗНИЦА } 0,28$$

ПРОХОДЫ: $0,28 \div 0,01 = 28$

$$\frac{a \sin(x) + b x \sin(x)}{\text{ПОВТОР}}$$

Сдвинуть и прогнать заново

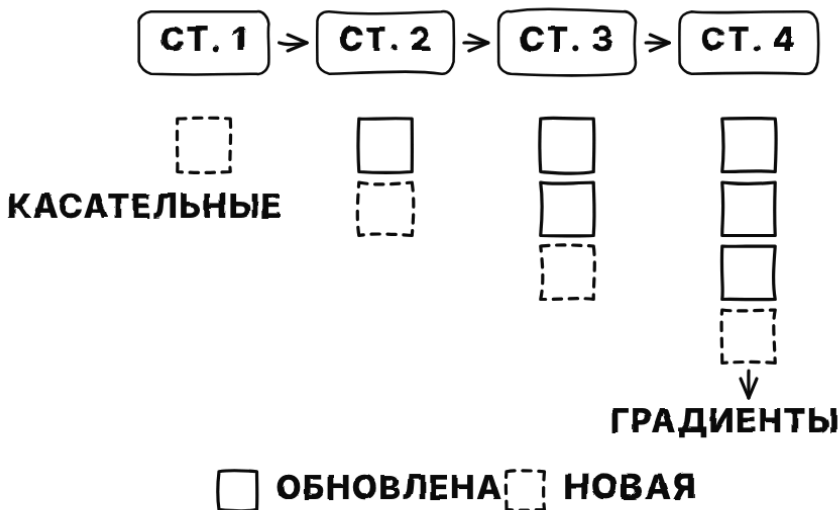
Вторая идея — поручить компьютеру школьную алгебру. Дайте программе символьной математики формулу из синуса и нескольких умножений, и она вернёт формулу её производной — так же, как сделал бы ученик на бумаге. В книге приведён короткий листинг на Python, который делает именно это с помощью библиотеки SymPy и печатает производную в виде новой формулы. Беда в том, что выписанная формула повторяется: один и тот же синус встречается в нескольких местах, и разумная программа вычислила бы его один раз и сохранила значение. Найти самый лучший способ разделить все такие повторяющиеся куски оказывается одной из тех задач, для которых не известно быстрого общего метода. Автоматическое дифференцирование обходит эти поиски, следуя структуре,

которая у программы уже есть, станция за станцией, и книга добавляет, что в итоге оно равносильно символьному методу, которому разрешено хранить и повторно использовать промежуточные результаты.

Опыт

Запишите рецепт — начать с трёх, удвоить, прибавить один, возвести в квадрат — и прогоните его дважды на калькуляторе телефона: один раз с трёх, другой раз с трёх целых одной сотой. Вы увидите, что результат сдвинется примерно на ноль целых двадцать восемь сотых — в двадцать восемь раз больше вашего сдвига. А теперь представьте, что эти два прогона нужно повторить для каждого из миллиардов чисел модели, и потребность в чём-то лучшем станет очевидной.

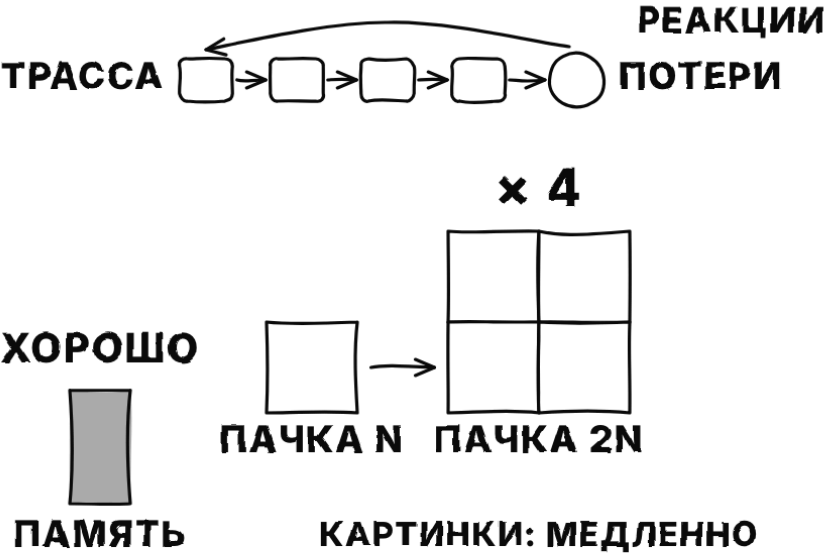
Поэтому книга переходит к первому настоящему методу — прямому режиму, — и он опирается на цепное правило. Если одна станция питает другую, реакция пары равна реакции второй, умноженной на реакцию первой. Понимайте это как бухгалтерию вдоль конвейера: если вы знаете, как выход первой станции реагирует на её параметры, и срабатывает вторая станция, вы обновляете это знание, умножая его на входной якобиан второй станции.



Прямой режим хранит таблицы

Прямой режим держит по одной текущей таблице для каждого встреченного до сих пор набора параметров; такая таблица называется касательной. На первой станции он заводит таблицу для первых параметров — это просто якобиан по весам этой станции. На второй станции он делает две вещи: обновляет первую таблицу входным якобианом второй станции и заводит новую таблицу для вторых параметров. На каждой следующей станции всё повторяется, так что десятая станция должна обновить девять хранимых таблиц и завести десятую, и все они обновляются одним и тем же входным якобианом этой станции. Когда доходит до последнего шага сложения, каждая таблица сворачивается в градиент для своих параметров, и работа сделана.

Хорошая новость — память: вся эта бухгалтерия шагает вместе с программой, поэтому ей нужно примерно столько же памяти, сколько самой программе. Плохая новость — время. Каждое обновление перемножает две большие таблицы реакций, и когда слой обрабатывает пачку примеров, стоимость растёт как квадрат размера пачки и как квадрат числа значений на пример. Удвойте пачку фотографий, на которой обучается фотоприложение, — и эта часть работы вырастет вчетверо; удвойте размер описания каждой картинке — и она вырастет вчетверо ещё раз. Для входов высокой размерности, таких как изображения, прямой режим быстро становится невозможным, и именно эту задачу решит обратный проход.



С памятью хорошо, время в квадрате

Запомним

Программа обучения — это ряд станций, трасса вычисления, которая заканчивается сложением штрафов пачки в одно число — потери. Автоматическое дифференцирование — это нахождение того, как это одно число реагирует на параметры каждой станции, эффективно и сразу, по собственным реакциям каждой станции. Сдвиг каждого числа с повторным прогоном модели стоит два прогона на число, а выписывание формул производных повторяет работу, так что ни то ни другое для обучения не годится. Прямой режим идёт вдоль конвейера, храня и обновляя таблицу для каждого встреченного набора параметров; он бережлив к памяти, но каждое обновление перемножает огромные таблицы. Эта стоимость растёт как квадрат размера пачки и числа значений на пример, что исключает прямой режим для больших входов вроде изображений.

Вопросы

1. Почему программа обучения всегда заканчивается одним числом?
 - а. Потому что модель может отвечать только на один пример за раз
 - б. Потому что штрафы пачки складываются, а именно одно число оптимизатор может толкать вниз по склону

- c. Потому что первая станция конвейера сводит каждый пример к одному числу
 - d. Потому что цепное правило не справляется больше чем с одним выходом
2. Почему сдвиг каждого числа с повторным прогоном модели бесполезен для обучения большого чат-ассистента?
- a. Потому что ответы, которые он даёт, всегда неверны
 - b. Потому что он работает только для чисел меньше единицы
 - c. Потому что ему нужно примерно два прогона всей модели на каждый параметр — миллиарды прогонов на шаг обучения
 - d. Потому что сначала нужно выписать формулу производной
3. Что в прямом режиме должна сделать десятая станция конвейера?
- a. Обновить девять таблиц, хранящихся до сих пор, и завести десятую
 - b. Дождаться итогового числа, а затем пойти назад
 - c. Только завести свою таблицу и не трогать прежние
 - d. Заново прогнать первые девять станций со сдвинутыми числами
4. Если фотоприложение удвоит размер обучающей пачки, что станет со стоимостью обновлений таблиц в прямом режиме?
- a. Она не изменится, потому что растёт память

- б. Она удвоится, как пачка
- с. Она вырастет примерно вчетверо, потому что растёт как квадрат размера пачки

Ответы

1. б) Последняя настоящая станция назначает каждому примеру штраф, а заключительный шаг их складывает; именно одно число оптимизатор может толкать вниз по склону.
2. с) Численное дифференцирование стоит два прогона на число, так что модели с миллиардами параметров понадобились бы миллиарды прогонов на шаг; его оставляют лишь для проверки других методов.
3. а) На каждой станции прямой режим обновляет все хранимые таблицы входным якобианом этой станции и заводит новую таблицу для её собственных параметров.
4. с) Каждое обновление перемножает две большие таблицы, и стоимость растёт как квадрат размера пачки и числа значений на пример.

Разобранные примеры

Измеряем реакцию сдвигом Рецепт такой: начать с 3, удвоить, прибавить 1, возвести в квадрат. Прогоните его с 3 и с 3,01 и по двум результатам оцените, насколько сильно результат реагирует на начальное число.

С 3: удвоенное — 6, плюс 1 — 7, в квадрате — 49.

С 3,01: удвоенное — 6,02, плюс 1 — 7,02, в квадрате — 49,2804.

Результат сдвинулся на 0,2804, а начальное число — на 0,01.

Деление сдвига результата на сдвиг начального числа даёт реакцию: 0,2804, делённое на 0,01.

Ответ: Около 28; чтобы узнать реакцию на одно число, понадобилось два полных прогона.

Задачи

1. У небольшой модели 1000 параметров. Сколько прогонов модели нужно на один шаг обучения при сдвиге с повторным прогоном в простейшем виде — по два прогона на число?

Подсказка: Каждому числу, по которому дифференцируют, нужна своя пара прогонов.

Ответ: 2000 прогонов на один шаг вместо нескольких; вот почему книга оставляет этот метод только для проверки более хитрого.

2. Объясните своими словами, почему прямой режим бережлив к памяти, но слишком медленен для моделей фотографий.

Подсказка: Спросите себя, что ему приходится хранить по ходу дела и что перемножать на каждой станции.

Ответ: Его бухгалтерия шагает вместе с программой, поэтому ему нужно примерно столько же памяти, сколько самой программе. Но на каждой станции он перемножает большие таблицы реакций для каждого встреченного набора параметров, и эта стоимость растёт как квадрат размера пачки и числа значений на пример, что для изображений невозможно.

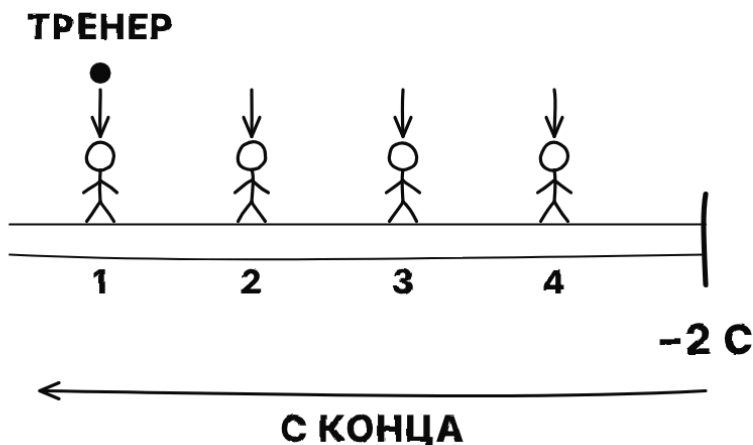
3. Программа символьной математики выписывает производную формулы, в которой один и тот же синус встречается несколько раз. Почему это ещё не хороший способ дифференцировать модель?

Подсказка: Подумайте, что выписанная формула делает с повторяющимися кусками.

Ответ: Выписанная формула повторяет одни и те же куски, тогда как разумная программа вычислила бы каждый один раз и использовала повторно; найти лучший способ разделить все повторяющиеся куски — задача, для которой не известно быстрого общего метода. Автоматическое дифференцирование обходит эти поиски, следуя структуре, которая у программы уже есть.

Путь назад от итоговой оценки

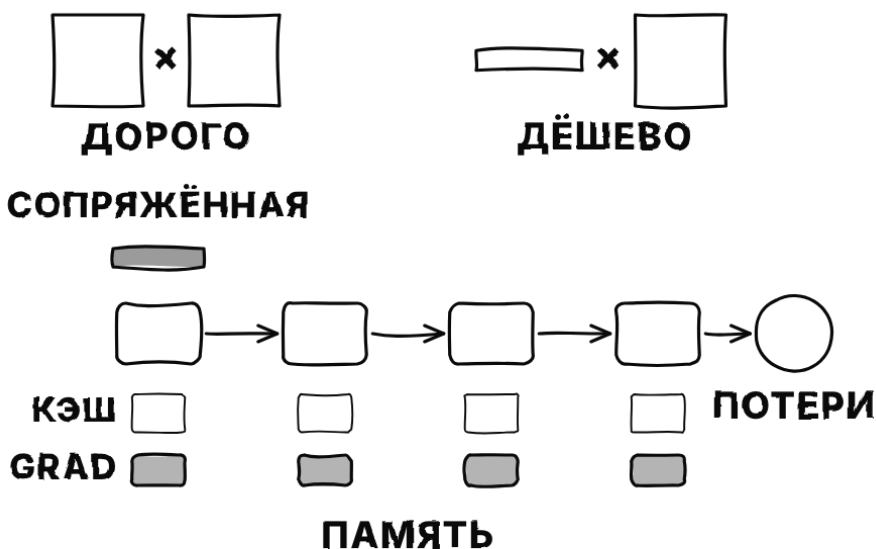
Представьте эстафету, которую ваша команда только что проиграла на две секунды. Тренер не просит каждого бегуна заново пробежать дистанцию в чуть других кроссовках. Она начинает от финишной черты, где стоит одно-единственное число — две секунды, — и идёт назад вдоль дорожки, по очереди спрашивая каждого бегуна, какая часть этого числа прошла через его этап. Каждый ответ короткий, и каждый готовит следующий вопрос для бегуна перед ним. Этот путь назад от одного итогового числа — вся идея этой части.



Назад от финиша

Прямой режим, как мы видели, нёс большую таблицу для каждого набора параметров вдоль всего конвейера и перемножал огромные таблицы на каждой станции. Теперь книга смотрит на цепочку реакций для одного набора параметров, выписанную целиком. Это длинный ряд умножений: на дальнем его конце стоит заключительный шаг сложения, затем входные якобианы всех последующих станций, один за другим, и наконец якобиан по весам той станции, которая нас интересует. Перемножать такие таблицы можно в любом порядке, и результат будет тем же. Прямой режим просто проходил ряд в том порядке, в каком работала программа; весь фокус в том, чтобы начать с другого конца.

У начала с конца два подарка. Первый — конец крошечный: поскольку программа заканчивается одним числом, последний член — не большая таблица, а простая строка значений, а умножать строку на таблицу гораздо дешевле, чем две таблицы, и в результате снова получается строка. Второй — длинный отрезок входных якобианов от конца до любой станции можно наращивать шаг за шагом, каждый раз добавляя ещё одну станцию с левой стороны пути. Так одна маленькая строка значений путешествует назад, от станции к станции, и на каждой остановке она используется снова, а не вычисляется заново.



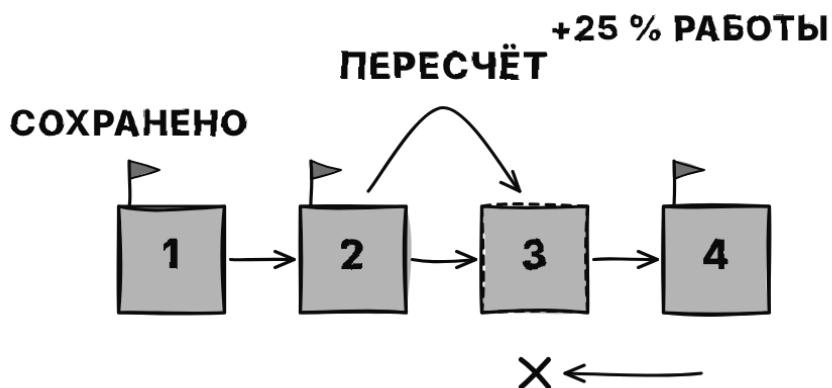
Маленькая строка идёт назад

Вместе это и есть автоматическое дифференцирование в обратном режиме. Сначала вся программа работает как обычно, и каждый промежуточный результат сохраняется. Затем на финише заводится строка единиц, которая обозначает заключительное сложение. На обратном пути на каждой станции происходят две дешёвые вещи: путешествующая строка сочетается с якобианом этой станции по весам, что даёт ровно градиент для её параметров, а затем — с входным якобианом станции, чтобы быть готовой для предыдущей. Путешествующие строки называются сопряжёнными переменными; каждая говорит, как итоговое число реагирует на выход одной станции. Обратный путь зеркально повторяет исходную программу, и

практики называют две половины прямым и обратным проходом — не путать с прямым режимом.

Именно это происходит каждый раз, когда чат-ассистент учится на пачке: один прямой проход, чтобы получить потери, и один обратный, чтобы получить все градиенты. Каждый шаг пути стоит лишь пропорционально участвующим размерам, никогда не их квадратам, поэтому обратный режим разительно быстрее прямого. Плата — память, потому что обратному пути нужны промежуточные результаты каждой станции, и все они должны храниться, пока он до них не дойдёт.

Книга показывает способ вернуть часть этой памяти ценой времени — он называется контрольными точками градиента. Представьте четыре блока в ряд. При прямом проходе сохраняются только выходы первого, второго и четвёртого блоков; это и есть контрольные точки. Обратный путь идёт гладко, пока не доходит до третьего блока, результаты которого были выброшены, и тут он останавливается. Тогда программа заново выполняет прямой проход от ближайшей контрольной точки, второго блока, чтобы восстановить недостающее, и путь продолжается. В этом примере вся работа стоит на четверть больше вычислений, чем обычный обратный проход, и в общем случае чем меньше контрольных точек вы храните, тем больше платите пересчётом.



Контрольные точки экономят память

Запомним

Цепочка реакций для любого набора параметров — это длинный ряд умножений, которые можно выполнять в любом порядке. Поскольку программа заканчивается одним числом, начать с конца — значит нести назад маленькую строку значений вместо перемножения больших таблиц, и каждый шаг использует предыдущий. Обратный режим один раз прогоняет программу, сохраняет каждый промежуточный результат, а затем идёт назад, получая каждый градиент за умеренную цену; это прямой и обратный проходы любой обучаемой модели. Его слабость — память, ведь все промежуточные результаты должны дожидаться обратного пути. Контрольные точки градиента сохраняют лишь часть результатов, а остальные пересчитывают на обратном пути, меняя дополнительные вычисления на меньший расход памяти.

Вопросы

1. Почему проходить цепочку реакций дешевле, начиная с итогового числа?
 - а. Потому что итоговое число маленькое, и таблицы ближе к концу пустые
 - б. Потому что программа заканчивается одним числом, так что конец — это маленькая строка, а строка на таблицу гораздо дешевле таблицы на таблицу

- c. Потому что у станций ближе к концу нет параметров
 - d. Потому что пути назад не нужны промежуточные результаты
2. Что происходит на каждой станции на обратном пути?
- a. Станцию прогоняют заново со сдвинутыми параметрами
 - b. Путешествующая строка сочетается с якобианом по весам и даёт градиент этой станции, а затем — с входным якобианом для предыдущей станции
 - c. Для каждой прежней станции заводится новая таблица
 - d. Промежуточный результат станции выбрасывается, и путь продолжается
3. Какова главная плата обратного режима по сравнению с прямым?
- a. Ему нужно гораздо больше времени на каждый шаг обучения
 - b. Он даёт лишь приближённые градиенты
 - c. Ему нужно много памяти, потому что каждый промежуточный результат приходится хранить, пока до него не дойдёт обратный путь
 - d. Его можно применять только к моделям из четырёх блоков
4. Что будет, если при обучении хранить меньше контрольных точек?
- a. Памяти уйдёт меньше, но больше придётся пересчитывать во время обратного прохода
 - b. Памяти уйдёт больше, а вычислений меньше

- c. Ничего не изменится, ведь контрольные точки влияют только на прямой проход
- d. Градиенты станут менее точными

Ответы

- 1. **b)** Программа заканчивается одним числом, поэтому последний член — простая строка значений; умножать строку на таблицу гораздо дешевле, чем две таблицы, и в результате снова получается строка.
- 2. **b)** На каждой станции путешествующая строка через якобиан по весам даёт градиент для её параметров, а затем передаётся дальше через входной якобиан, готовая для предыдущей станции.
- 3. **c)** Обратный режим гораздо быстрее, но обратному пути нужны промежуточные результаты каждой станции, поэтому все они должны храниться, пока он до них не дойдёт.
- 4. **a)** Контрольные точки меняют память на время: чем меньше результатов хранится, тем больше приходится пересчитывать от ближайшей контрольной точки на обратном пути.

Разобранные примеры

Контрольные точки в четырёх блоках В модели четыре блока одинаковой стоимости. При прямом проходе сохраняются только выходы первого, второго и четвёртого блоков. Сколько дополнительных вычислений нужно обратному проходу по сравнению с хранением всего?

Обратный путь проходит четвёртый блок и доходит до третьего, результаты которого не сохранены.

Он заново выполняет прямой проход от ближайшей контрольной точки, второго блока, чтобы восстановить результаты третьего.

Этот повторный прогон стоит работы одного блока сверх четырёх блоков обычного прямого прохода.

Один лишний блок из четырёх — это на четверть больше вычислений.

Ответ: Примерно в 1,25 раза больше вычислений, на четверть больше, — в обмен на то, что результаты третьего блока не хранятся.

Задачи

1. Объясните своими словами, почему порядок умножений в цепочке реакций вообще можно менять.

Подсказка: Подумайте, из каких операций состоит цепочка.

Ответ: Цепочка — это просто длинный ряд умножений таблиц, а такие умножения дают тот же результат, какие бы пары ни перемножались первыми. Поэтому мы вольны начать с итогового числа, а не с первой станции, и именно этот выбор делает обратный режим дешёвым.

2. Модель фотографий для телефона слишком велика, чтобы при обучении хранить в памяти все промежуточные результаты. Что могут сделать её инженеры и чего это им будет стоить?

Подсказка: Подумайте о том, чтобы хранить лишь часть результатов.

Ответ: Они могут использовать контрольные точки градиента: хранить лишь несколько промежуточных результатов — контрольные точки — и пересчитывать остальные от ближайшей контрольной точки во время обратного прохода. Нужная память уменьшается, а платой служат дополнительные вычисления, которых тем больше, чем меньше контрольных точек хранится.

3. Почему обратный режим должен прогнать всю программу, прежде чем сможет вычислить хоть один градиент, тогда как

прямой режим мог работать вместе с программой?

Подсказка: Где начинается обратный путь?

Ответ: Обратный путь начинается от итогового числа в конце программы, поэтому это число и каждый промежуточный результат, который понадобится в пути, должны уже существовать. Прямой режим следует собственному порядку программы, поэтому его бухгалтерия может шагать вместе с ней.

Задать таблице один вопрос

Представьте рыночный прилавок с огромной доской цен: для каждого фрукта — сколько стоит каждый сок, в который он идёт. Вы подходите со списком покупок, в котором сказано, сколько бутылок каждого сока вам нужно. Вам не нужно, чтобы продавец переписал для вас всю доску; вам нужен один ответ — во что обходится ваш список по каждому фрукту. Продавец может прикинуть это в уме, строку за строкой, ни разу не выписывая доску. Эта разница — между тем, чтобы отдать всю таблицу, и тем, чтобы ответить на один заданный ей вопрос, — и есть фокус этой части.

VJP



× 2 НА СТАНЦИЮ

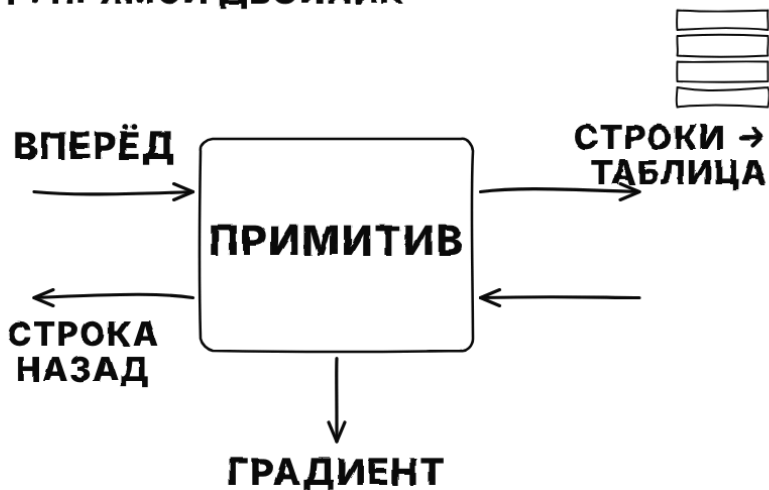
Один вопрос, а не таблица

Взгляните ещё раз на обратный путь. На каждой станции происходила операция только одного вида: маленькая путешествующая строка сочеталась с одной из таблиц станции — входным якобианом или якобианом по весам. Книга даёт этой операции имя — произведение вектора на якобиан, сокращённо VJP: на вход подаётся строка «насколько важен каждый выход», а выходит строка «насколько важен каждый вход». Поэтому каждой станции нужно ровно два таких произведения — одно для входа и одно для параметров, — и одного вызова каждого достаточно, чтобы провести обратный проход через эту станцию.

Вот следствие, и оно большое. Целую систему автоматического дифференцирования можно построить, выбрав набор

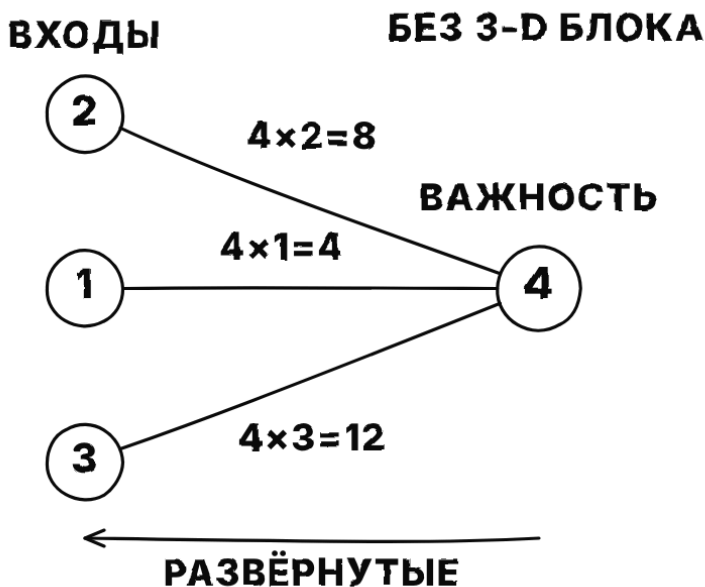
примитивов и дав каждому два его VJP, ни разу не записывая в память полный якобиан. Книга рисует это как блок для каждого примитива, с обычной прямой дверью и двумя дополнительными обратными: одна возвращает путешествующую строку предыдущей станции, другая выдаёт градиент для собственных параметров блока. Ничего не теряется: если вам когда-нибудь понадобится полная таблица, её можно восстановить по одной строке, спрашивая VJP о каждом выходе по отдельности. Для зеркального случая, прямого режима, книга замечает, что есть операция-двойник, идущая в другую сторону, — произведение якобиана на вектор, сокращённо JVP.

JVP: ПРЯМОЙ ДВОЙНИК



Блок с двумя задними дверями

Почему это настолько лучше? Возьмите самую распространённую станцию — слой, который смешивает свои входы с сеткой весов, как это много раз делает каждый слой чат-модели или модели фотографий. Его таблица реакций по весам — даже не плоская таблица, а трёхмерный блок, потому что каждый выход может реагировать на каждый вес сетки. И всё же у вопроса, который задаёт ему обратный путь, поразительно простой ответ. Чтобы передать путешествующую строку назад, слой пропускает её через собственную сетку весов, развёрнутую так, что выходы и входы меняются местами. А градиент для одного веса — это просто два перемноженных числа: насколько важен выход, который он питает, умноженное на входное значение, на которое он умножал на прямом пути.

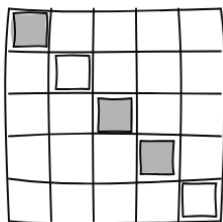


Доля вины веса

Этот последний факт стоит повторить другими словами. Каждый вес соединяет один вход с одним выходом, поэтому его доля вины — это важность его выхода, умноженная на величину его входа, и вся сетка таких произведений заполняется сразу. Трёхмерному блоку никогда не нужно существовать.

Вторая распространённая станция — функция активации, например ReLU, которая обрабатывает каждое значение отдельно и не имеет параметров, так что нужно только одно VJP. Её таблица почти целиком пуста, потому что каждый выход зависит только от своего входа; всё, что важно, стоит на диагонали — наклон функции при каждом значении. Поэтому на обратный вопрос отвечают, масштабируя каждый элемент путешествующей строки на этот наклон, по одному. Для ReLU это значит пропустить значение там, где вход был положительным, и остановить там, где он был отрицательным. И снова полная таблица с её океаном нулей так и не строится.

ТАБЛИЦА RELU



3	0	2	4	0
---	---	---	---	---

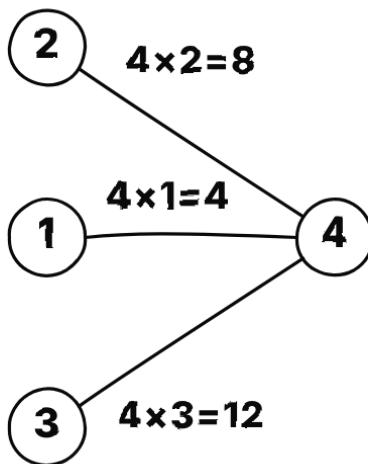


ТАБЛИЦА НЕ НУЖНА

ReLU лишь масштабирует строку

Опыт

Возьмите три числа как входы слоя, скажем два, один и три, и представьте один выход, важность которого равна четырём. Градиент веса, связывающего этот выход с первым входом, равен четырём на два, со вторым — четырём на один, с третьим — четырём на три. Вы заполнили строку градиентов тремя умножениями и без всякой таблицы.

Запомним

Обратный путь только и делает, что умножает маленькую строку на таблицу станции, и эта операция называется произведением вектора на якобиан. Каждому примитиву нужно два таких произведения — одно для входа и одно для параметров, — и одних их достаточно, чтобы построить полную систему автоматического дифференцирования, не храня ни одного якобиана. Для слоя весов строка передаётся назад через развёрнутые веса, а градиент каждого веса — это важность его выхода, умноженная на вход, на который он умножал. Для активации вроде ReLU строка просто масштабируется, значение за значением, на наклон функции. В обоих случаях огромные таблицы реакций никогда не записываются — именно это и делает возможным обучение больших моделей.

Вопросы

1. Почему система автоматического дифференцирования может работать, ни разу не храня полный якобиан?
 - a. Потому что якобианы всегда достаточно малы, чтобы ими пренебречь
 - b. Потому что обратному пути нужна только строка на таблицу, и каждый примитив может ответить на это прямо через два своих VJP

- c. Потому что полный якобиан вычисляется один раз, а затем выбрасывается
 - d. Потому что якобианы нужны только прямому режиму
2. Чему равен градиент одного веса в слое, который смешивает свои входы с сеткой весов?
- a. Важности выхода, который он питает, умноженной на входное значение, на которое он умножал
 - b. Собственному значению веса, умноженному на размер сетки
 - c. Сумме всех входов слоя
 - d. Наклону функции активации у этого веса
3. Почему станции ReLU нужно только одно VJP вместо двух?
- a. Потому что у неё нет параметров, так что вычислять градиент весов не нужно
 - b. Потому что её наклон всегда равен единице
 - c. Потому что она всегда последняя станция модели
 - d. Потому что ею вместо этого занимается прямой режим
4. Что можно сделать, если вам всё-таки нужна полная таблица станций?
- a. Ничего: раз используются VJP, таблица потеряна навсегда
 - b. Восстановить её по одной строке, спрашивая VJP о каждом выходе по отдельности
 - c. Прогнать прямой проход дважды и сравнить
 - d. Перемножить VJP всех станций

Ответы

1. **b)** Единственная операция, нужная обратному пути, — произведение вектора на якобиан, поэтому достаточно дать каждому примитиву два его VJP, и ни один якобиан не записывается в память.
2. **a)** Каждый вес соединяет один вход с одним выходом, поэтому его доля вины — важность его выхода, умноженная на величину его входа.
3. **a)** Активация вроде ReLU обрабатывает каждое значение отдельно и не имеет параметров, так что нужно только VJP для её входа.
4. **b)** Ничего не теряется: если спрашивать VJP о каждом выходе по отдельности, получится полная таблица — по одной строке.

Разобранные примеры

Сетка градиентов весов Маленький слой принимает три входа — 2, 1 и 3 — и имеет два выхода. Путешествующая строка говорит, что важность первого выхода равна 4, а второго — 1. Каковы градиенты шести весов?

Каждый вес связывает один вход с одним выходом, поэтому его градиент — важность этого выхода, умноженная на этот вход.

Для первого выхода с важностью 4: 4 на 2, 4 на 1, 4 на 3 — получается 8, 4 и 12.

Для второго выхода с важностью 1: 1 на 2, 1 на 1, 1 на 3 — получается 2, 1 и 3.

Вместе они заполняют всю сетку градиентов весов, и никакой трёхмерный блок строить не нужно.

Ответ: Веса первого выхода: 8, 4, 12; веса второго выхода: 2, 1, 3.

Задачи

1. При прямом проходе ReLU получила входы 1,5, минус 0,7 и 2. На обратном пути к ней приходит строка 5, минус 2 и 3. Какую строку она передаёт назад?

Подсказка: Наклон ReLU равен единице там, где вход был положительным, и нулю там, где отрицательным.

Ответ: 5, 0 и 3: первое и третье значения проходят, потому что их входы были положительными, а второе останавливается, потому что его вход был отрицательным.

2. Объясните своими словами, почему градиент веса должен быть важностью его выхода, умноженной на вход, на который он умножал.

Подсказка: Подумайте, что делает один вес при прямом проходе.

Ответ: При прямом проходе вес умножает один вход и добавляет результат в один выход. Поэтому, если увеличить вес, этот выход сдвинется пропорционально входу, а влияние выхода на итоговое число — это его важность; значит, влияние веса — это их произведение.

3. Почему таблицу ReLU называют океаном нулей и почему это важно?

Подсказка: От чего зависит каждый выход ReLU?

Ответ: Каждый выход зависит только от своего входа, поэтому значения в таблице есть только на диагонали, а везде остальное — нули. Строить её значило бы тратить память на нули, тогда как VJP просто масштабирует каждое значение на наклон, и таблица для этого вовсе не нужна.

Как библиотеки идут назад и почему победила ReLU

Положите страницу на копировальный аппарат, поставьте уменьшение до четверти размера и сделайте копию копии, а потом копию этой копии. После двух кругов текст превращается в пятно; после десяти страница — точка, которую можно прикрыть песчинкой. Каждый круг уменьшал лишь в постоянное число раз, но эти множители перемножались. Запомните этот копировальный аппарат: сейчас он объяснит, почему одна маленькая функция захватила современный ИИ. Но сначала книга смотрит, как настоящие библиотеки выполняют обратный путь.



PYTORCH: TORCH.FUNC

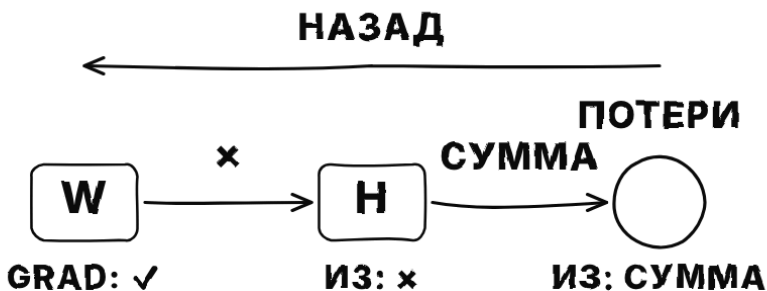


Инструмент, который возвращает функцию

Построить обратный режим в программе можно многими способами — от ведения списка каждой выполненной операции, как делает TensorFlow, до переписывания кода программы в её обратного двойника. В JAX, где всё — функция, и сами инструменты — функции, которые принимают функции. Вы даёте JAX функцию и получаете обратно новую функцию, которая вычисляет её градиент. В том же духе инструмент VJP берёт функцию и точку и возвращает маленькую функцию, которая отвечает на обратный вопрос в этой точке. Короткий пример книги определяет функцию, которая возводит в квадрат десять чисел и складывает их, запрашивает её функцию градиента и подаёт ей десять случайных чисел; на выходе — десять чисел, по одной реакции на каждый вход. PyTorch

предлагает тот же стиль в отдельном модуле, а умело сочетая эти инструменты, можно получить и больше, например вторые производные.

Однако большинство моделей записаны как объекты — слои, которые хранят свои параметры внутри себя, как мы видели в начале. Один способ перевести их в стиль функций — «очистить» их: вынести параметры в отдельный набор и вызывать модель как обычную функцию от входа и этих параметров. У PyTorch есть и свой встроенный способ, которому такой шаг не нужен. Каждый вычисляемый им блок чисел несёт две дополнительные пометки: пустое место для будущего градиента и запись об операции, которая его произвела. Рисунок книги показывает это: слева числа с их двумя пометками; справа обратный путь, который следует по записям от итоговых потерь назад по графу и по дороге раскладывает градиенты по местам. Чтобы экономить память, места заполняются только у тех чисел, которые попросили градиент, — обычно у обучаемых параметров. Именно это буквально и происходит при обучении модели фотоприложения, когда программа вызывает обратный шаг для потерь, и автор горячо советует собрать крошечную версию такой системы самому, как делают многие учебные проекты в сети.

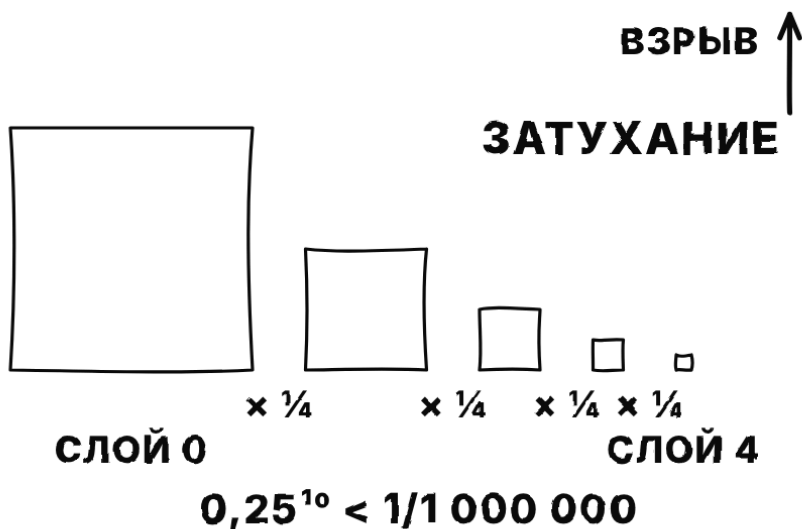


ТОЛЬКО ПО ЗАПРОСУ

Пометки, по которым идут назад

Теперь вернёмся к копировальному аппарату. Помните: когда обратный путь проходит через функцию активации, путешествующая строка масштабируется на наклон функции. В глубокой модели много таких станций, так что строка масштабируется снова и снова, и множители перемножаются точно так же, как уменьшения копира. Если наклоны всюду меньше единицы, градиент сжимается до нуля с каждым слоем всё быстрее — это называется проблемой затухающего градиента. Если всюду больше единицы, он растёт без предела — проблема взрывающегося градиента. Обе опасности реальны, потому что компьютеры хранят числа с ограниченной точностью, обычно в тридцать два двоичных разряда или меньше, так что слишком

маленькое число становится нулём, а слишком большое переполняется.

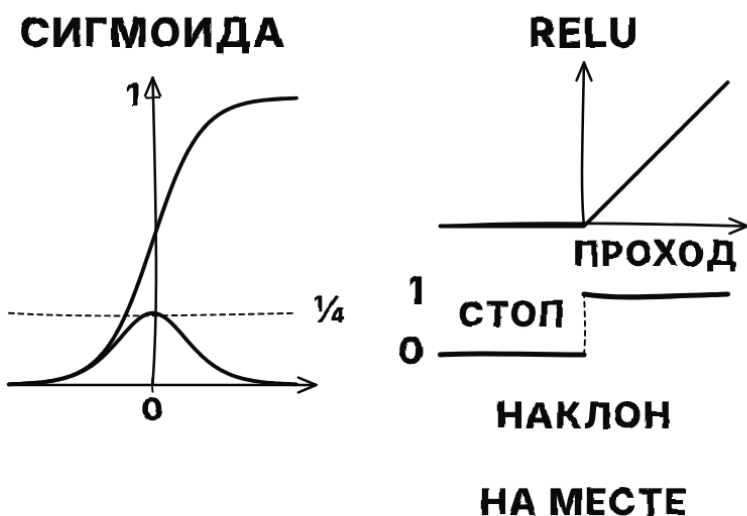


Множители, которые перемножаются

Опыт

На калькуляторе телефона наберите ноль целых двадцать пять сотых и умножьте на ноль целых двадцать пять сотых десять раз. У вас должно получиться меньше одной миллионной. Примерно это — в лучшем случае — могут сделать с градиентом десять слоёв старой любимой активации, а в настоящих сетях слоёв гораздо больше.

Эта старая любимица — сигмоида, гладкая S-образная кривая, которая была популярна, потому что это мягкая версия выключателя. Её наклон — низкий горб, который никогда не поднимается выше четверти: он наибольший в середине буквы S и почти плоский на обоих концах. Поэтому каждый слой сигмоиды сжимает обратную строку до четверти или меньше — копир на самом сильном уменьшении.



Наклон сигмоиды и наклон ReLU

Идеалом была бы функция с наклоном, всюду равным единице, но такая функция только одна — та, что ничего не меняет, а стопка слоёв, которые ничего не меняют, сделала бы всю модель не мощнее одного слоя. Значит, нужна функция, достаточно прямая, чтобы градиенты оставались здоровыми, и всё же

достаточно изогнутая, чтобы слои не сливались. ReLU подходит: она ровно ноль для отрицательных входов и прямая линия для положительных, так что её наклон либо ноль, либо ровно единица. Поэтому на обратном пути каждое значение либо останавливается, что делает вычисление разреженным, либо проходит нетронутым — без сжатия и без раздувания. Рисунок книги показывает ReLU в форме шарнира рядом с её наклоном — ступенькой, которая прыгает от нуля к единице.

Во врезке автор добавляет любопытный факт: стопка чисто линейных слоёв, запущенная на настоящем компьютере, не совсем линейна из-за крошечных шероховатостей на пределе машинной точности, и этим даже пользовались, чтобы обучать полностью линейные глубокие сети.

У ReLU есть ещё один маленький подарок. Её наклон можно прочитать по выходу так же хорошо, как по входу, потому что она лишь обнуляет отрицательные значения, а положительные оставляет как есть. Поэтому библиотека может во время прямого прохода затереть вход ReLU её выходом и всё равно правильно пройти обратный путь, сэкономя немного драгоценной памяти; в PyTorch для этого есть параметр работы «на месте».

Запомним

JAX считает градиенты и VJP инструментами, которые принимают функцию и возвращают новую функцию, а PyTorch помечает каждый вычисленный блок чисел операцией, которая его сделала, и местом для его градиента, а затем проходит эти записи назад от потерь. Каждая активация масштабирует обратную строку на свой наклон, и через много слоёв эти множители перемножаются. Наклоны меньше единицы заставляют градиенты затухать, а больше единицы — взрываться, и ограниченная точность компьютера превращает это в настоящие сбои. Наклон сигмоиды не больше четверти, поэтому глубокие стопки из неё морят градиент голодом; наклон ReLU — либо ноль, либо единица, так что значения либо останавливаются, либо проходят нетронутыми. Поскольку наклон ReLU можно прочитать по её выходу, её вход можно затереть на месте, чтобы сэкономить память.

Вопросы

1. Когда PyTorch выполняет обратный шаг для потерь, как он находит путь назад через модель?
 - a. Он заново прогоняет каждый слой со сдвинутыми числами
 - b. Он следует по записи, которую каждый блок чисел хранит об операции, что его произвела, и по дороге

раскладывает градиенты по местам

с. Он просит JAX вычислить функцию градиента

d. Он сначала выписывает формулу производной всей модели

2. Почему глубокие стопки слоёв сигмоиды морят градиент голодом?

a. Потому что у сигмоиды вообще нет наклона

b. Потому что её наклон никогда не выше четверти, а наклоны всех слоёв на обратном пути перемножаются

с. Потому что сигмоида затирает свой вход на месте

d. Потому что у слоёв сигмоиды слишком много параметров

3. Почему бы просто не взять активацию с наклоном, всюду равным единице?

a. Потому что такая функция ничего не меняет, и стопка слоёв была бы не мощнее одного

b. Потому что наклон, равный единице, заставляет градиенты взрываться

с. Потому что компьютеры не умеют хранить число один

d. Потому что ей понадобилось бы место для градиента у каждого значения

4. Почему библиотека может во время прямого прохода затереть вход ReLU её выходом?

a. Потому что наклон ReLU можно прочитать по выходу так же хорошо, как по входу

- b. Потому что ReLU никогда не встречается на обратном пути
- c. Потому что вход больше не нужен ни одному последующему слою
- d. Потому что затираание увеличивает градиент

Ответы

1. **b)** Каждый блок чисел несёт запись об операции, которая его сделала, и место для своего градиента; обратный путь идёт по этим записям назад от потерь и заполняет места.
2. **b)** Каждая активация масштабирует обратную строку на свой наклон; наклон сигмоиды не больше четверти, так что через много слоёв градиент сжимается почти в ничто.
3. **a)** Единственная функция с наклоном, всюду равным единице, — та, что ничего не меняет; активация должна быть достаточно изогнутой, чтобы слои не сливались.
4. **a)** ReLU лишь обнуляет отрицательные значения и оставляет положительные без изменений, поэтому её наклон можно прочесть по выходу, а вход — затереть ради экономии памяти.

Разобранные примеры

Градиент суммы квадратов Как в примере книги, функция возводит свои входы в квадрат и складывает. Что вернёт её функция градиента для трёх входов 1, минус 2 и 3?

В градиенте по одному элементу на каждый вход: насколько сумма реагирует на сдвиг этого входа.

Каждый вход встречается только в собственном квадрате, так что для него важен только этот квадрат.

Квадрат реагирует на маленький сдвиг с силой, равной удвоенному возводимому значению.

Значит, каждый элемент — удвоенный вход: 2 на 1, 2 на минус 2, 2 на 3.

Ответ: 2, минус 4 и 6: столько же чисел, сколько было входов, — так же как десять входов из книги дали десять чисел.

Задачи

1. В модели пять слоёв сигмоиды, и в лучшем случае каждый масштабирует обратную строку на четверть. Во сколько раз, самое большее, уменьшится градиент, пройдя все пять?

Подсказка: Множители перемножаются: четверть на четверть, пять раз.

Ответ: Четверть в пятой степени — это $1 \text{ к } 1024$, так что градиент в лучшем случае уменьшится примерно в тысячу раз.

2. Теперь замените пять сигмоид пятью ReLU и допустите, что интересующее нас значение на каждом слое положительно. Что с ним происходит на обратном пути?

Подсказка: Каков наклон ReLU для положительных входов?

Ответ: Наклон на каждом слое ровно единица, так что значение передаётся назад нетронутым — без сжатия и без раздувания.

Окажись оно на каком-то слое отрицательным, там бы оно и остановилось.

3. Объясните своими словами, почему затухающие и взрывающиеся градиенты — настоящая проблема на компьютере, а не только на бумаге.

Подсказка: Подумайте, сколько разрядов компьютер хранит для каждого числа.

Ответ: Компьютеры хранят числа с ограниченной точностью, обычно 32 двоичных разряда или меньше. Градиент, умноженный на много маленьких наклонов, в конце концов становится слишком малым, чтобы его хранить, и превращается в ноль, а умноженный на много больших — слишком большим и переполняется, так что обучение ломается.

Угол ReLU

Сложите открытку неглубокой буквой V и поставьте её на стол остриём вниз — как график ReLU, у которого угол внизу. Теперь возьмите линейку и попробуйте приложить её к углу снизу так, чтобы она касалась угла и оставалась под всей открыткой. У гладкой кривой был бы ровно один способ это сделать — касательная. В углу же линейку можно наклонить как угодно между наклонами двух сторон, и годится любой из этих наклонов. Это маленькое наблюдение — то, чем книга закрывает последний незавершённый вопрос главы.

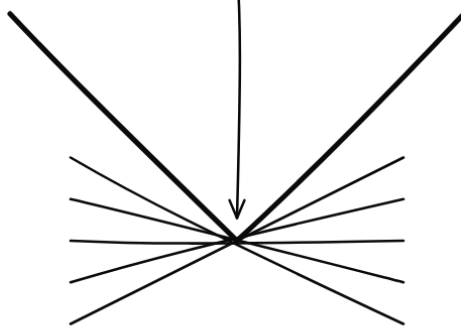
ГЛАДКАЯ



ReLU в 0



ОТКРЫТКА



УГОЛ: МНОГО

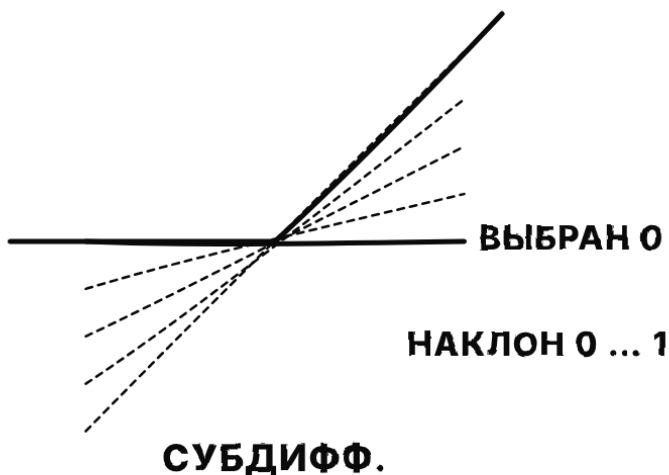
Линейка под сложенной открыткой

У ReLU, ровной для отрицательных и прямой линией для положительных, в нуле угол, где наклон не определён. Что там делает обратный путь? Практический ответ: это почти неважно — обучение начинается со случайных чисел и движется маленькими случайными шагами, так что шанс попасть ровно в ноль практически нулевой, а везде в других местах наклон совершенно ясен.

Строгий ответ использует идею линейки. Для функции, изогнутой вверх, как чаша, любая прямая, которая касается её в точке и остаётся ниже всей функции, называется в этой точке субградиентом, и её наклон — законная замена отсутствующему наклону. Там, где функция гладкая, такая прямая только одна — обычная касательная; в углу их много, и вместе они образуют субдифференциал. Для ReLU в нуле подходит любой наклон от нуля до единицы, и большинство фреймворков просто выбирают ноль. Обучение, которое на каждом шаге выбирает такой наклон, называется субградиентным спуском.

RELU $\rightarrow 0$

RELU₂ $\rightarrow 1$ RELU₃ $\rightarrow 0,5$

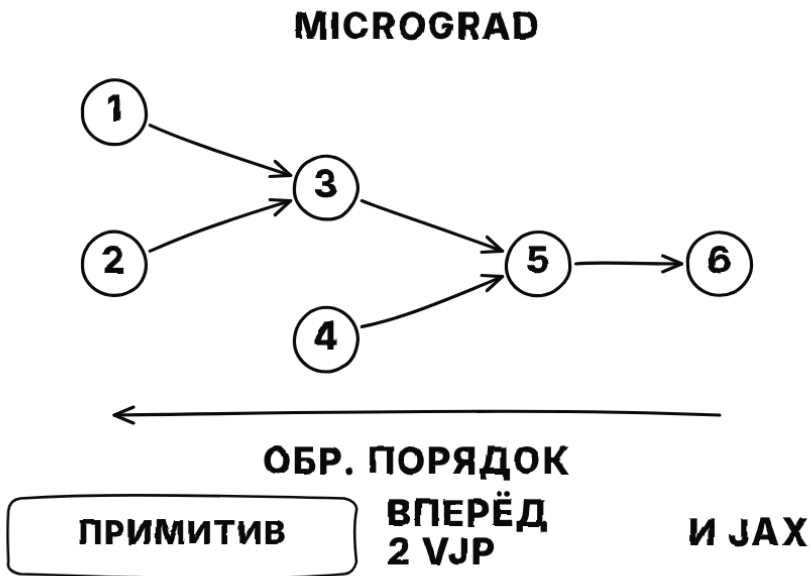


Любой наклон от 0 до 1

Книга признаёт, что на деле всё запутаннее. Для функций, не похожих на чашу, определение приходится ослабить, чтобы оно выполнялось лишь вблизи точки, а системы автоматического дифференцирования могут давать разные ответы в зависимости от того, как записана функция. Пример автора поразителен: три способа записать одну и ту же ReLU заставляют PyTorch сообщить в нуле наклон ноль, единица и одна вторая.

В заключение книга обращается к практике. Её самый тёплый совет — построить небольшую систему обратного режима самому, как это делает для отдельных чисел учебный проект `micrograd`. Единственная дополнительная забота, когда модель ветвится, — расставить станции в правильном порядке перед обратным

путём, чтобы ни один градиент не пошёл по неверной дорожке; `micrograd` делает это дешёвой сортировкой. Второе хорошее упражнение — написать в `PyTorch` новый примитив, задав ему прямой проход и его VJP, например одну из обучаемых функций активации из предыдущей главы, хотя `PyTorch` мог бы вычислить обратный проход и сам. То же можно сделать в `JAX`, где есть учебная версия самой библиотеки, способ писать собственные VJP и сборник рецептов для продвинутых задач вроде вторых производных. И теперь обратный путь, который обучает каждого чат-ассистента и каждое фотоприложение, которыми вы пользуетесь, — уже не загадка, а метод, который вы могли бы написать сами.



Постройте свой обратный путь

Запомним

У ReLU в нуле угол, но обучение почти никогда не попадает точно туда, так что на практике её наклон всегда определён. Формально субградиент — это наклон прямой, которая касается чашеобразной функции в точке и остаётся ниже неё везде; в углу таких наклонов много, и они образуют субдифференциал. Для ReLU в нуле годится любой наклон от нуля до единицы, и фреймворки обычно берут ноль. Разные записи одной и той же функции могут давать в углу разные наклоны, как показывают ноль, единица и одна вторая у PyTorch. Лучший способ сделать главу своей — построить крошечную систему обратного режима или собственный примитив с его VJP.

Вопросы

1. Почему неопределённый наклон ReLU в нуле почти не важен при обычном обучении?
 - a. Потому что фреймворки убирают из данных все нули
 - b. Потому что обучение начинается со случайных чисел и движется маленькими случайными шагами, так что попасть ровно в ноль практически невозможно
 - c. Потому что наклон ReLU в нуле всегда равен единице
 - d. Потому что обратный путь пропускает все станции ReLU

2. Почему в углу чашеобразной функции субградиентов много, а не один?
- a. Потому что много прямых могут касаться угла, оставаясь ниже всей функции
 - b. Потому что у функции много углов рядом друг с другом
 - c. Потому что в углу функция не чашеобразная
 - d. Потому что любая прямая через угол — субградиент, каков бы ни был её наклон
3. Какие наклоны — законные субградиенты ReLU в нуле?
- a. Только ноль
 - b. Только единица
 - c. Любой наклон от нуля до единицы
 - d. Вообще любой наклон
4. Что показывает пример книги с PyTorch об углах?
- a. Что PyTorch отказывается вычислять градиенты в углах
 - b. Что три способа записать одну и ту же ReLU дают в нуле три разных наклона
 - c. Что ReLU на самом деле не равна своим переписанным версиям
 - d. Что только JAX правильно обращается с углами

Ответы

1. **b)** Шанс попасть ровно в ноль практически нулевой, а везде в других местах наклон ReLU совершенно ясен.
2. **a)** Субградиент — это наклон прямой, которая касается функции в точке и остаётся ниже неё; в гладкой точке так может только

касательная, а в углу — целый диапазон наклонов.

3. c) Для ReLU в нуле подходит любой наклон от нуля до единицы, и большинство фреймворков просто выбирают ноль.

4. b) Три способа записать одну и ту же ReLU заставляют PyTorch сообщить в нуле наклон ноль, единица и одна вторая.

Разобранные примеры

Три ReLU — три наклона Вторая версия ReLU записана так: ReLU от минус входа плюс сам вход. Третья берёт половину исходной ReLU плюс половину второй. Проверьте, что обе равны ReLU, и найдите наклон, который PyTorch сообщает в нуле для каждой, если собственный наклон ReLU в нуле он считает равным 0.

Для положительного входа ReLU от минус входа равна нулю, так что вторая версия даёт сам вход, как и ReLU; для отрицательного входа она даёт минус вход плюс вход, то есть ноль, — снова как ReLU.

Третья версия усредняет две функции, каждая из которых равна ReLU, поэтому она тоже равна ReLU.

В нуле наклон второй версии берётся как минус наклон ReLU, то есть 0, плюс 1 от прибавленного входа, итого 1.

Наклон третьей версии — половина от 0 плюс половина от 1, то есть 0,5.

Ответ: Все три — одна и та же функция, но PyTorch сообщает в нуле наклоны 0, 1 и 0,5.

Задачи

1. Прямая проходит через угол ReLU в нуле с наклоном одна вторая. Законный ли это субградиент? А прямая с наклоном полтора?

Подсказка: Проверьте каждую прямую при входе 2 и минус 2: она должна везде быть ниже ReLU или касаться её.

Ответ: С наклоном одна вторая прямая равна 1, когда ReLU равна 2, и минус 1, когда ReLU равна 0, — значит, она ниже: годится. С наклоном полтора она равна 3, когда ReLU равна 2, — выше функции, так что это не субградиент.

2. Почему, когда модель ветвится, станции нужно расставить в правильном порядке перед обратным путём?

Подсказка: Подумайте о станции, выход которой питает две более поздние станции.

Ответ: Станция может передать свой градиент дальше только после того, как всё, что от неё зависит, вернуло ей свою долю. Без правильного порядка путь мог бы передать неполный градиент и пойти по неверной дорожке; `micrograd` предотвращает это дешёвой сортировкой.

3. Опишите своими словами, что нужно дать новому примитиву, чтобы фреймворк мог обучать через него.

Подсказка: Вспомните две обратные двери каждого примитива.

Ответ: Его прямой проход, который вычисляет выход, и его VJP: одно передаёт бегущую строку назад к предыдущей станции, а другое даёт градиент его собственных параметров, если они есть.