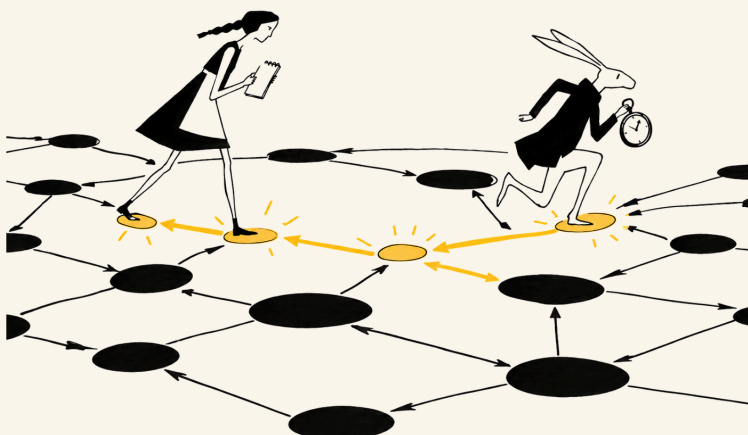


Приключения Алисы в Стране дифференцируемых чудес



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

Автоматическое дифференцирование

Формулировка задачи

Дифференцирование в прямом режиме

Дифференцирование в обратном режиме

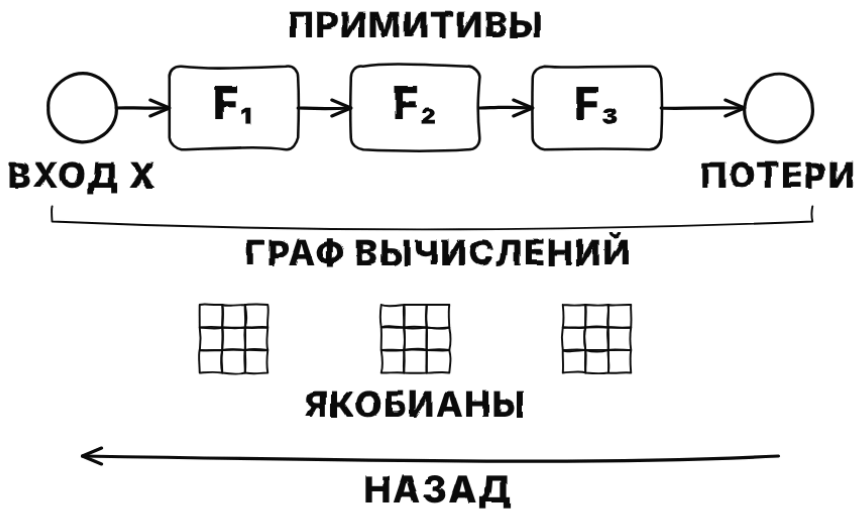
Произведения вектора на якобиан

Реализация системы обратного режима

Выбор функции активации

Автоматическое дифференцирование

В предыдущей главе мы увидели, что нужна эффективная автоматическая процедура для вычисления градиентов любой возможной последовательности операций. В этой главе мы описываем такой метод, который в литературе о нейронных сетях называется обратным распространением ошибки, а в литературе по информатике — автоматическим дифференцированием в обратном режиме. Его разбор даёт несколько важных наблюдений — от выбора модели до требований к памяти при её оптимизации.



Программа как цепочка примитивов

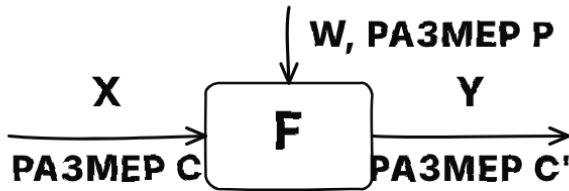
Постановка задачи.

Мы рассматриваем задачу эффективного вычисления градиентов произвольных вычислительных графов, например тех, что возникают при оптимизации скалярной функции потерь полносвязной модели; эта задача называется автоматическим дифференцированием, или AD. Вычислительный граф можно представлять как набор элементарных операций, которые мы называем примитивами и которые получаются, если просто выполнить саму программу. Для краткости мы будем рассматривать последовательные графы, но всё сказанное легко распространяется на ациклические и ещё более общие вычислительные графы.

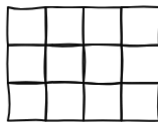
Задача может показаться тривиальной, ведь цепное правило для якобианов из раздела о градиентах и якобианах говорит, что градиент композиции функций — это просто матричное произведение соответствующих матриц Якоби. Однако эффективно реализовать этот процесс — главная трудность этой главы, а получающийся алгоритм, AD в обратном режиме, или обратное распространение ошибки, — краеугольный камень нейронных сетей и дифференцируемого программирования вообще. Понимание этого алгоритма — ключ и к пониманию устройства, и различий, большинства библиотек для построения и обучения таких программ, например TensorFlow, PyTorch или JAX.

Чтобы поставить задачу, предположим, что в нашем распоряжении есть набор примитивов $y = f_i(\mathbf{x}, \mathbf{w}_i)$. Каждый примитив — это операция над входным вектором $\mathbf{x} \sim (c_i)$ с

параметрами в виде вектора $\mathbf{w}_i \sim (p_i)$, например весов линейной проекции, и на выходе она даёт другой вектор $\mathbf{y} \sim (c'_i)$.

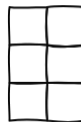


ВХ. ЯКОБ.



$C' \times C$

ЯКОБ. ВЕСОВ



$C' \times P$

Один примитив и два его якобиана

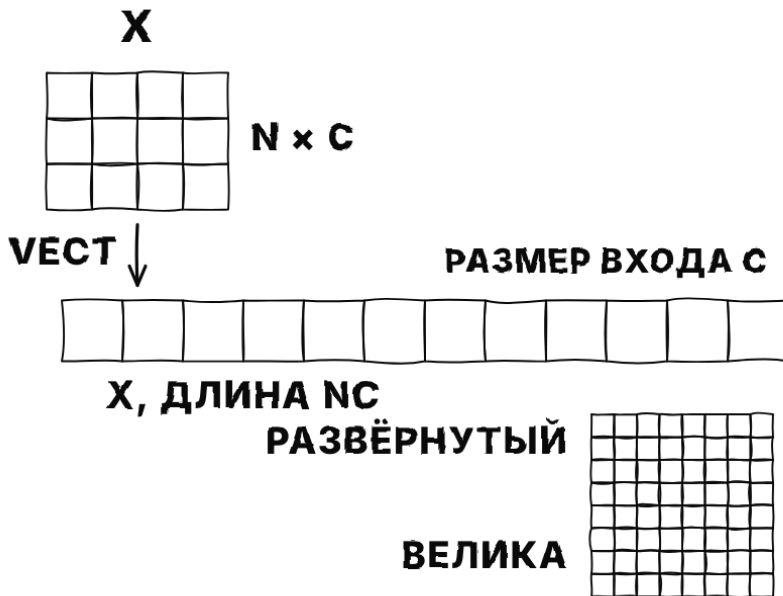
Наше определение примитивов очень гибкое: они могут обозначать базовые операции линейной алгебры, такие как умножение матриц, слои в смысле пятой главы, например полносвязный слой с функцией активации, или даже более крупные блоки и целые модели. Такая рекурсивная составляемость — ключевое свойство программирования, и она переносится на наш случай.

Мы предполагаем лишь одно: для каждого примитива мы умеем вычислять частные производные по двум его входным

аргументам, которые мы называем входным якобианом и якобианом по весам этой операции. Входной якобиан — это $\partial_x[f(\mathbf{x}, \mathbf{w})] \sim (c', c)$. Якобиан по весам — это $\partial_{\mathbf{w}}[f(\mathbf{x}, \mathbf{w})] \sim (c', p)$.

Это разумные допущения, поскольку мы ограничиваем анализ дифференцируемыми моделями. Непрерывные примитивы с одной или несколькими точками недифференцируемости, такие как ReLU, можно уложить в эту схему с помощью субградиентов, как мы увидим в конце главы. Недифференцируемые операции, например выборку или пороговую функцию, тоже можно включить, если найти релаксацию их градиента или эквивалентную оценку. Этот последний случай мы рассмотрим в следующем томе.

О наших обозначениях и якобианах высших порядков.



Как батч разворачивается в вектор

Для удобства чтения мы рассматриваем только векторные величины, так как тогда все получающиеся градиенты — матрицы. На практике у существующих примитивов входы, веса или выходы могут иметь более высокий ранг. Например, рассмотрим обычный полносвязный слой на входе, разбитом на мини-батчи: $f(\mathbf{X}, \mathbf{W}) = \mathbf{XW} + \mathbf{b}$. В этом случае вход $\mathbf{X}$ имеет форму (n, c) , веса — формы (c, c') и (c') , где c' — гиперпараметр, а выход имеет форму (n, c') . Значит, входной якобиан имеет форму (n, c', n, c) , а якобиан по весам — форму (n, c', c, c') , и оба имеют ранг четыре.

В наших обозначениях можно рассматривать эквивалентные развёрнутые векторы $\mathbf{x} = \text{vect}(\mathbf{X})$ и $\mathbf{w} = [\text{vect}(\mathbf{W}); \mathbf{b}]$, и тогда

развёрнутые якобианы имеют формы (nc', nc) и (nc', cc') соответственно. Это очень важно для дальнейшего: каждый раз, когда мы говорим «размер входа s », мы имеем в виду «произведение всех размеров входа», включая размерность мини-батча, если она есть. Отсюда видно и то, что, даже умея вычислять якобианы, мы можем не захотеть целиком хранить их в памяти из-за их огромной размерности.

Наконец, наши обозначения соответствуют тому, как эти примитивы реализованы в функциональной библиотеке, например в JAX. В объектно-ориентированной библиотеке, такой как TensorFlow или PyTorch, слои, как мы видели в предыдущей главе, реализованы как объекты: параметры — это свойство объекта, а вызов функции заменён вызовом метода объекта. Такой стиль упрощает некоторые приёмы, например откладывание инициализации всех параметров до тех пор, пока не станут известны размеры входа, — так называемую ленивую инициализацию, — но добавляет небольшой уровень абстракции, который нужно учитывать, переводя наши обозначения в работающий код. Как мы увидим, эти различия, в свою очередь, отражаются на том, как AD реализовано в библиотеках этих двух видов.

Вопросы

1. Почему глава считает вычисление градиентов настоящей задачей, если цепное правило уже говорит, что градиент композиции — это произведение якобианов?
 - а. Потому что цепное правило не выполняется для матриц

- b. Потому что трудная часть — эффективно вычислить это произведение
 - c. Потому что якобианы примитивов неизвестны
 - d. Потому что цепное правило применимо только к скалярным функциям
2. Что, согласно постановке задачи, мы знаем о каждом примитиве?
- a. Только его выход для каждого возможного входа
 - b. Его частные производные по входу и по весам
 - c. Градиент итоговых потерь по его весам
 - d. Символьную формулу всей программы
3. Полносвязный слой работает с мини-батчем: форма входа (n, c) , форма выхода (n, c') . Почему мы можем не захотеть хранить его входной якобиан в памяти?
- a. Потому что он всегда равен нулю
 - b. Потому что в развёрнутом виде он имеет форму (nc', nc) , а это может быть очень много
 - c. Потому что у него ранг 1 и он не несёт информации
 - d. Потому что объектно-ориентированные библиотеки не умеют хранить матрицы
4. Как примитив вроде ReLU, недифференцируемый в одной точке, всё же вписывается в эту схему?
- a. Его убирают из вычислительного графа
 - b. Его заменяют линейным слоем
 - c. С помощью субградиентов
 - d. Его выход выбирают случайно

Ответы

1. **b)** Цепное правило даёт ответ как матричное произведение якобианов, но эффективная реализация этого процесса — главная трудность главы.
2. **b)** Мы предполагаем лишь, что для каждого примитива умеем вычислять входной якобиан и якобиан по весам — его частные производные по двум аргументам.
3. **b)** Его входной якобиан имеет ранг четыре, а в развёрнутом виде — форму (nc', nc) ; из-за такой большой размерности мы можем не захотеть целиком хранить его в памяти.
4. **c)** Непрерывные примитивы с точками недифференцируемости, такие как ReLU, можно уложить в эту схему с помощью субградиентов.

Разобранные примеры

Размер развёрнутого якобиана Полносвязный слой получает мини-батч из $n = 32$ входов, у каждого $c = 10$ признаков, и выдаёт $c' = 5$ выходов на каждый вход. Сколько чисел в его развёрнутом входном якобиане и сколько в развёрнутом якобиане по весам (без учёта смещения)?

Развёрнутый входной якобиан имеет форму (nc', nc) , потому что каждое выходное число дифференцируется по каждому входному.

Здесь $nc' = 32 \times 5 = 160$ и $nc = 32 \times 10 = 320$, так что форма — $(160, 320)$.

Это даёт $160 \times 320 = 51\,200$ чисел.

Развёрнутый якобиан по весам имеет форму (nc', cc') , где $cc' = 10 \times 5 = 50$ весов в W .

Это даёт $160 \times 50 = 8000$ чисел — гораздо больше, чем самих 50 весов.

Ответ: 51 200 чисел во входном якобиане и 8000 в якобиане по весам

Задачи

1. У слоя форма входа (n, c) , где $n = 2$ и $c = 3$, и форма выхода (n, c') , где $c' = 4$. Запишите форму его входного якобиана ранга 4 и его развёрнутую форму. Одинаково ли число элементов в них?

Подсказка: Форма ранга 4 — (n, c', n, c) ; в развёрнутой форме перемножаются первые два и последние два числа.

Ответ: Форма ранга 4 — $(2, 4, 2, 3)$, развёрнутая форма — $(8, 6)$. В обоих 48 элементов: развёртка лишь переставляет числа.

2. В этой главе «размер входа c » примитива означает произведение всех размеров его входа. Почему это важно, когда модель обучают на мини-батчах?

Подсказка: Подумайте, что происходит с размером входа, когда в него включается размерность мини-батча.

Ответ: Размерность мини-батча входит в размер входа, поэтому c растёт вместе с батчем. Якобианы, формы которых составлены из этих размеров, растут вместе с ним, — вот почему мы можем уметь их вычислять и всё же не хотеть хранить их в памяти.

3. Объясните своими словами, чем различается обращение с параметрами слоя в функциональной библиотеке вроде JAX и в объектно-ориентированной вроде PyTorch и как это связано с обозначением $f(x, w)$.

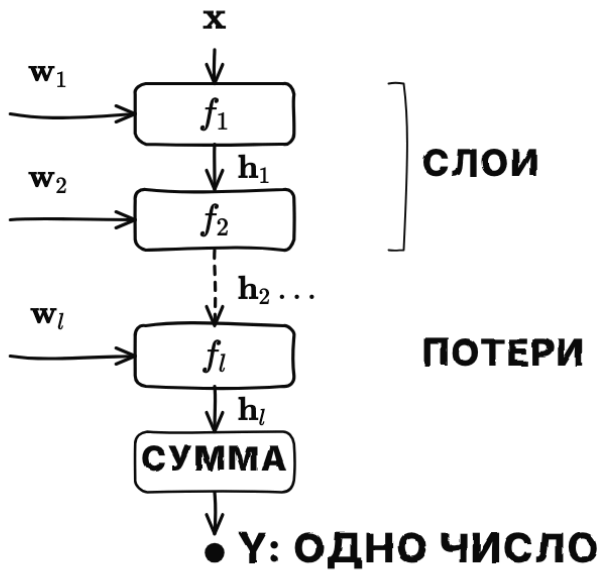
Подсказка: Спросите себя, где в каждом случае живут веса.

Ответ: В обозначении $f(x, w)$ веса — явный второй аргумент функции, и именно так к ним относится функциональная библиотека вроде JAX. В объектно-ориентированной библиотеке

слой — это объект, параметры — свойство этого объекта, а вызов функции — метод; это облегчает, например, ленивую инициализацию, но добавляет слой абстракции между обозначениями и кодом.

Формулировка задачи

Разобравшись со всеми этими подробностями, мы готовы сформулировать задачу AD. Рассмотрим последовательность из l вызовов примитивов, за которой следует итоговое суммирование. Сначала $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$. Затем $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$, и так далее, вплоть до $\mathbf{h}_l = f_l(\mathbf{h}_{l-1}, \mathbf{w}_l)$. Наконец, $y = \Sigma \mathbf{h}_i$.



Трасса вычисления

Это называется трассой вычисления программы. Грубо говоря, первые $l - 1$ операций могут представлять несколько слоёв дифференцируемой модели, операция l может быть функцией

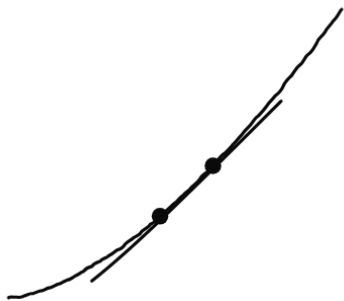
потерь для каждого входа, например перекрёстной энтропией, а последняя операция суммирует потери по мини-батчу. Поэтому выход нашей программы всегда скаляр: он нужен нам для численной оптимизации. Сокращённо обозначим эту программу как $F(\mathbf{x})$.

Определение: автоматическое дифференцирование. Для программы $F(\mathbf{x})$, составленной из последовательности дифференцируемых примитивов, автоматическим дифференцированием, или AD, называется задача одновременно и эффективно вычислить все якобианы программы по весам, зная вычислительный граф и все отдельные входные якобианы и якобианы по весам. В символах $AD(F(\mathbf{x}))$ — это множество градиентов $\partial_{\mathbf{w}_i} y$ для $i = 1, \dots, l$.

Как мы увидим, есть два основных класса алгоритмов AD — прямой режим и обратный режим, — которые соответствуют разному порядку композиции отдельных операций. Мы увидим и то, что обратный режим, который в литературе о нейронных сетях называется обратным распространением ошибки, в нашем случае значительно эффективнее. Хотя мы сосредоточены на упрощённом сценарии, наш вывод сравнительно легко распространить на ациклические графы примитивов, как уже говорилось, а также на случаи, когда параметры общие для нескольких слоёв, — это называется разделением весов. Пример разделения весов мы увидим в главе 13.

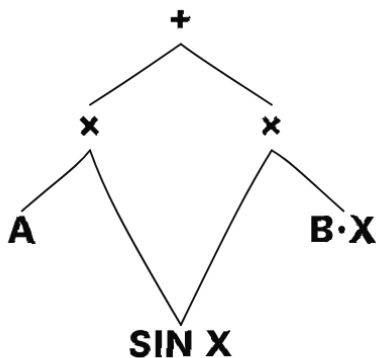
Численное и символьное дифференцирование.

ЧИСЛЕННОЕ



2 ВЫЗОВА

СИМВОЛЬНОЕ



**ОДИН РАЗ
НР-ПОЛНАЯ**

Численное и символьное дифференцирование

Прежде чем перейти к AD в прямом режиме, скажем о разнице между AD и другими классами алгоритмов дифференцирования функций. Во-первых, можно прямо применить определение градиента и получить подходящее численное приближение к нему. Этот процесс называется численным дифференцированием. Однако в простой реализации каждое скалярное значение, по которому дифференцируют, требует двух вызовов функции, и такой подход годится разве что для численной проверки реализации.

Во-вторых, рассмотрим простую функцию $f(x) = a \sin(x) + b x \sin(x)$. Можно попросить символьную систему заранее вычислить полное символьное выражение производной. Это называется

символьным дифференцированием. Во врезке 6.1 книги библиотеке SymPy для Python дают эту функцию и просят найти её производную по x , и она возвращает $a \cos(x) + b x \cos(x) + b \sin(x)$.

В реалистичной реализации промежуточное значение $h = \sin(x)$ вычислялось бы только один раз и хранилось в промежуточной переменной, которую можно повторно использовать и в соответствующем вычислении трассы градиента; аналогично и для члена $\cos(x)$ в производной. Это не так тривиально, как кажется: найти оптимальную реализацию якобиана без единого лишнего вычисления — NP-полная задача, известная как оптимальное накопление якобиана. Однако мы увидим, что структуру нашей программы можно использовать, чтобы построить достаточно эффективную реализацию AD, значительно лучшую, чем символьный подход вроде показанного; на самом деле она равносильна символьному подходу, который допускает общие подпоследовательности.

Вопросы

1. Почему программа в трассе вычисления заканчивается суммой, так что её выход y — одно число?
 - a. Потому что суммирование уменьшает якобианы
 - b. Потому что численной оптимизации нужен скалярный выход
 - c. Потому что последний слой модели — всегда сумма

- d. Потому что это нужно дифференцированию в прямом режиме
2. Что именно должно вычислить автоматическое дифференцирование согласно определению?
- a. Все якобианы программы по весам, одновременно и эффективно
 - b. Входной якобиан каждого примитива, по одному
 - c. Символьную формулу выхода программы
 - d. Значение программы для каждого входа
3. Почему численное дифференцирование для модели непригодно, разве что для проверки?
- a. Потому что оно даёт точную производную только для линейных функций
 - b. Потому что ему нужна символьная система
 - c. Потому что в простой реализации каждое скалярное значение, по которому дифференцируют, требует двух вызовов функции
 - d. Потому что оно не справляется со скалярным выходом
4. В чём слабость символьной производной $f(x) = a \sin(x) + b x \sin(x)$, выписанной целиком?
- a. Она математически неверна
 - b. Она повторяет работу, например $\sin(x)$ и $\cos(x)$, которую реалистичная реализация вычислила бы один раз и сохранила
 - c. Её нельзя вычислить на компьютере
 - d. Она работает, только когда a и b равны нулю

Ответы

1. **b)** Последняя операция суммирует потери по мини-батчу, поэтому выход программы всегда скаляр, а он нужен нам для численной оптимизации.
2. **a)** Автоматическое дифференцирование — это задача одновременно и эффективно вычислить все якобианы программы по весам, зная вычислительный граф и отдельные входные якобианы и якобианы по весам.
3. **c)** В простой реализации каждое скалярное значение, по которому дифференцируют, требует двух вызовов функции, поэтому подход годится разве что для численной проверки.
4. **b)** В реалистичной реализации промежуточное значение $h = \sin(x)$ вычислялось бы только один раз и хранилось, и то же верно для члена $\cos(x)$; избежать всех лишних вычислений совсем не просто.

Разобранные примеры

Символьная производная в одной точке Возьмите функцию книги $f(x) = a \sin(x) + b x \sin(x)$, символьная производная которой равна $a \cos(x) + b x \cos(x) + b \sin(x)$. Чему равна производная в точке $x = 0$ при $a = 2$ и $b = 3$?

Запишите производную, которую возвращает символьная система: $a \cos(x) + b x \cos(x) + b \sin(x)$.

При $x = 0$ имеем $\cos(0) = 1$ и $\sin(0) = 0$.

Первое слагаемое равно $a \times 1 = 2$.

Во втором слагаемом есть множитель $x = 0$, поэтому оно равно 0; в третьем есть $\sin(0) = 0$, поэтому и оно равно 0.

Сложите три слагаемых: $2 + 0 + 0 = 2$.

Ответ: 2

Задачи

1. У модели 1 000 000 весов. Сколько вызовов всей программы займёт один градиент при простом численном дифференцировании, которому нужно два вызова функции на каждое скалярное значение?

Подсказка: Каждый вес — одно скалярное значение, по которому дифференцируют.

Ответ: 2 000 000 вызовов: по два на каждый из миллиона весов; вот почему численное дифференцирование применяют лишь для проверки реализации.

2. Запишите трассу вычисления для программы из $l = 3$ примитивов с итоговой суммой. Какая операция может быть функцией потерь для каждого входа?

Подсказка: Каждый примитив берёт предыдущий выход и свои веса.

Ответ: $h_1 = f_1(x, w_1)$, $h_2 = f_2(h_1, w_2)$, $h_3 = f_3(h_2, w_3)$, $y = \text{сумма } h_3$.

Первые две операции могут быть слоями модели, третья, операция l , — функцией потерь для каждого входа, например перекрёстной энтропией, а сумма складывает потери по мини-батчу.

3. Объясните своими словами, что общего у автоматического дифференцирования в прямом и обратном режимах и чем они различаются, насколько об этом говорит эта часть.

Подсказка: Оба вычисляют одно и то же; посмотрите на порядок.

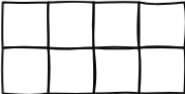
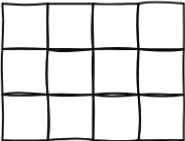
Ответ: Оба вычисляют все якобианы программы по весам из отдельных якобианов её примитивов. Различаются они порядком, в котором составляются отдельные операции, и обратный режим, называемый обратным распространением ошибки, в этих условиях значительно эффективнее.

Дифференцирование в прямом режиме

Начнём с того, что вспомним цепное правило для якобианов.

Рассмотрим сочетание двух примитивных функций: $\mathbf{h} = f_1(\mathbf{x})$ и $\mathbf{y} = f_2(\mathbf{h})$. В терминах их градиентов имеем $\partial_{\mathbf{x}}\mathbf{y} = \partial_{\mathbf{h}}\mathbf{y} \cdot \partial_{\mathbf{x}}\mathbf{h}$.

$$\partial_{\mathbf{x}}\mathbf{y} = \partial_{\mathbf{h}}\mathbf{y} \cdot \partial_{\mathbf{x}}\mathbf{h}$$

$\partial_{\mathbf{h}}\mathbf{y}$		$\partial_{\mathbf{x}}\mathbf{h}$		$\partial_{\mathbf{x}}\mathbf{y}$
	$\times$		$=$	
$\mathbf{C} \times \mathbf{B}$		$\mathbf{B} \times \mathbf{A}$		$\mathbf{C} \times \mathbf{A}$

НОВЫЙ ШАГ

УЖЕ ВЫЧИСЛЕНО

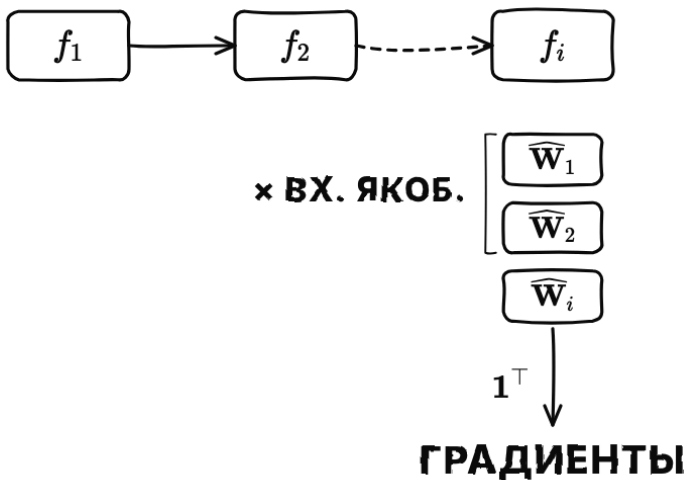
Цепное правило в формах матриц

Если $\mathbf{x}$, $\mathbf{h}$ и $\mathbf{y}$ имеют размерности a , b и c соответственно, то этот якобиан требует умножить матрицу $c \times b$ — якобиан $\mathbf{y}$ по $\mathbf{h}$ — на матрицу $b \times a$ — якобиан $\mathbf{h}$ по $\mathbf{x}$. Правило можно понимать так: если мы уже вычислили f_1 и её якобиан, то, применив f_2 , мы

можем «обновить» градиент, умножив его на соответствующий якобиан.

Это наблюдение сразу даёт работающий алгоритм — автоматическое дифференцирование в прямом режиме, или F-AD. Идея в том, что каждый раз, применяя примитивную функцию, мы заводим соответствующий ей якобиан по весам, который в этом контексте называется касательной, и одновременно обновляем все прежние касательные матрицы. Рассмотрим простой разобранный пример, чтобы показать основной алгоритм.

Возьмём первую инструкцию нашей программы, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$. Поскольку до сих пор ничего не сохранено, мы задаём касательную матрицу для $\mathbf{w}_1$ равной её якобиану по весам: $\hat{W}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$.



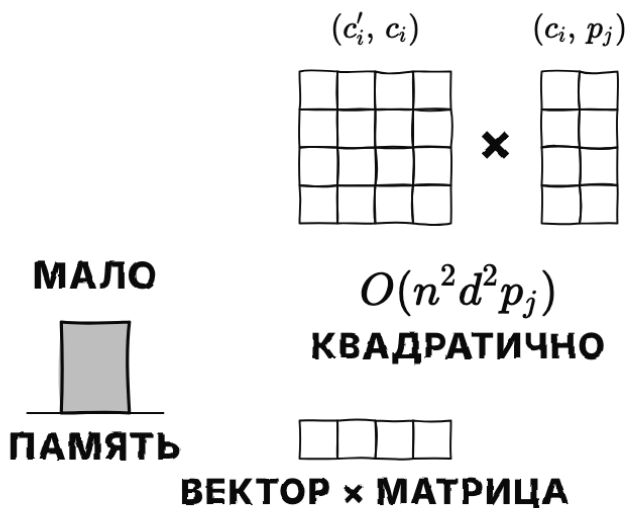
Прямой режим несёт касательные

Теперь переходим ко второй инструкции, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$. Мы обновляем прежнюю касательную матрицу и одновременно заводим вторую. Касательная матрица для $\mathbf{w}_1$ обновляется умножением на входной якобиан f_2 : $\hat{\mathbf{W}}_1 \leftarrow [\partial_{\mathbf{h}_1} \mathbf{h}_2] \hat{\mathbf{W}}_1$. Вторая задаётся как $\hat{\mathbf{W}}_2 = \partial_{\mathbf{w}_2} \mathbf{h}_2$.

Для обновления нужен входной якобиан примитива, а для второго члена — его якобиан по весам. Обобщая, рассмотрим произвольный i -й примитив, заданный как $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$. Мы заводим касательную матрицу для $\mathbf{w}_i$ и одновременно обновляем все прежние матрицы. Для каждого j , меньшего i , $\hat{\mathbf{W}}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{\mathbf{W}}_j$. А новая матрица — это якобиан f_i по весам: $\hat{\mathbf{W}}_i = \partial_{\mathbf{w}_i} \mathbf{h}_i$.

В первой строке $i - 1$ обновлений, по одному на каждую касательную матрицу, уже хранящуюся в памяти, причём входной якобиан i -й операции общий для всех прежних касательных. Последняя операция программы — сумма, и соответствующий градиент даёт выход алгоритма: для каждого i $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{\mathbf{W}}_i$. Строго говоря, в наших обозначениях этот выход — вектор-строка, тогда как градиент мы определили как вектор-столбец; ради простоты мы не будем обращать внимания на эту тонкость, пока дальше в главе не придёт время определять произведение вектора на якобиан.

Готово! Разберём алгоритм подробнее. Во-первых, все перечисленные операции легко перемежать с исходной программой, а значит, требуемая память будет примерно пропорциональна памяти самой дифференцируемой программы.



Мало памяти, много арифметики

С другой стороны, основная операция алгоритма — обновление $\hat{W}_i$ — требует перемножить две матрицы, в общем случае формы (c'_i, c_i) и (c_i, p_j) , где c_i и c'_i — размеры входа и выхода, а p_j — размер w_j . Это чрезвычайно дорогая операция. Например, пусть и вход, и выход имеют форму (n, d) , где n — размерность мини-батча, а d — число входных и выходных признаков. Тогда умножение матриц будет иметь сложность $O(n^2 d^2 p_j)$, квадратичную и по размеру мини-батча, и по размерности признаков. Это легко может стать неосуществимым, особенно для входов высокой размерности, таких как изображения.

Лучшего компромисса можно добиться, заметив, что последняя операция алгоритма — более простое произведение матрицы на

вектор, и это следствие того, что выход скалярный. Подробнее это рассматривается в следующем разделе.

Вопросы

1. Что в дифференцировании в прямом режиме происходит с уже хранящимися касательными матрицами, когда применяется новый примитив?
 - a. Их выбрасывают и вычисляют заново в конце
 - b. Все они обновляются умножением на входной якобиан нового примитива
 - c. Их прибавляют к якобиану нового примитива по весам
 - d. Они остаются без изменений до итоговой суммы
2. Почему дифференцированию в прямом режиме нужно немного памяти?
 - a. Потому что оно хранит только итоговый градиент
 - b. Потому что оно никогда не перемножает матрицы
 - c. Потому что его операции можно перемежать с исходной программой, так что требуемая память примерно такая же, как у программы
 - d. Потому что вместо якобианов оно использует вектор единиц
3. Что делает дифференцирование в прямом режиме дорогим для больших входов, таких как изображения?
 - a. Его основное обновление перемножает две матрицы, и стоимость квадратична и по размеру мини-батча, и по

числу признаков

- b. Ему нужно прогнать программу дважды
- c. Ему нужна символьная система для каждого примитива
- d. Его итоговая сумма — произведение двух матриц

Ответы

- 1. **b)** Каждый раз, когда применяется примитив, заводится его касательная, а все прежние касательные матрицы обновляются умножением на его входной якобиан, общий для всех них.
- 2. **c)** Все операции легко перемежать с исходной программой, поэтому требуемая память примерно пропорциональна памяти дифференцируемой программы.
- 3. **a)** Обновление каждой касательной — это умножение двух матриц со сложностью $O(n^2 d^2 p_j)$, квадратичной по размеру мини-батча и по размерности признаков.

Разобранные примеры

Считаем обновления касательных В программе $l = 4$ примитива. В прямом режиме i -й примитив обновляет $i - 1$ уже хранящихся касательных матриц. Сколько всего делается обновлений касательных и сколько касательных заводится?

Первый примитив ничего хранящегося не находит: 0 обновлений, и он заводит свою касательную.

Второй обновляет одну хранящуюся касательную: 1 обновление.

Третий обновляет две, четвёртый — три.

Всего $0 + 1 + 2 + 3 = 6$ обновлений, и каждое — умножение двух матриц.

Каждый из четырёх примитивов заводит по одной касательной, так что заводится 4 касательные.

Ответ: 6 обновлений и 4 заведённые касательные

Задачи

1. В цепном правиле $\partial_x y = \partial_h y \cdot \partial_x h$ пусть x имеет размерность $a = 3$, $h - b = 4$, а $y - c = 2$. Каковы формы двух множителей и результата?

Подсказка: Книга говорит, что произведение умножает матрицу $c \times b$ на матрицу $b \times a$.

Ответ: $\partial_h y$ имеет форму 2×4 , а $\partial_x h - 4 \times 3$, так что их произведение $\partial_x y - 2 \times 3$: c на a , как и положено якобиану y по x .

2. Обновление в прямом режиме стоит $O(n^2 d^2 p_j)$. Во сколько раз вырастет стоимость, если размер мини-батча n удвоится, а d и p_j останутся прежними? А если удвоятся и n , и d ?

Подсказка: Посмотрите на степени n и d .

Ответ: Удвоение n умножает стоимость на $2^2 = 4$. Удвоение и n , и d умножает её на $4 \times 4 = 16$ — вот почему метод быстро становится неосуществимым для больших входов.

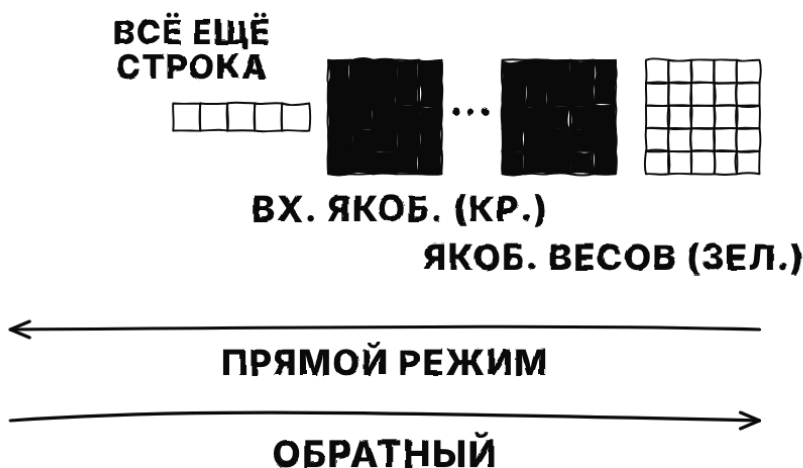
3. Объясните своими словами, почему книга надеется на лучший компромисс благодаря последней операции алгоритма.

Подсказка: Что особенного в выходе программы?

Ответ: Поскольку программа заканчивается суммой и её выход — скаляр, последний шаг — более простое произведение матрицы на вектор: транспонированный вектор единиц, умноженный на касательную. Произведение с вектором гораздо дешевле произведения двух матриц, и следующий раздел этим пользуется.

Дифференцирование в обратном режиме

Чтобы двигаться дальше, развернём вычисление одного члена градиента — того, что соответствует i -й матрице весов. Он имеет вид $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \cdots [\partial_{\mathbf{h}_i} \mathbf{h}_{i+1}] [\partial_{\mathbf{w}_i} \mathbf{h}_i]$.



Один градиент как цепочка произведений

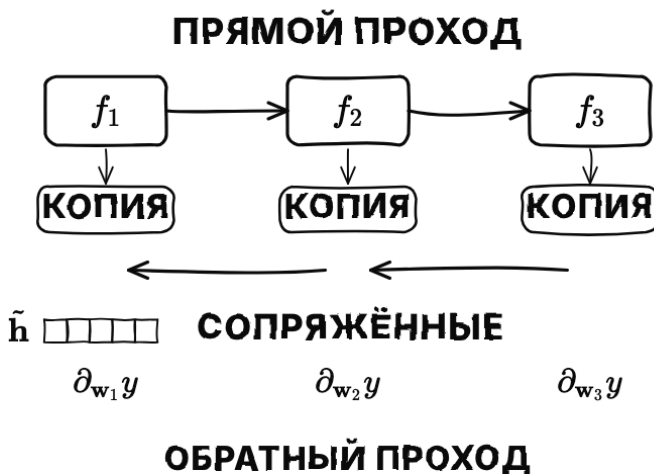
Помните, что, если отвлечься от обозначений, это просто, возможно, длинная цепочка умножений матриц. В ней участвуют постоянный член — вектор $\mathbf{1}$ из единиц; ряд входных якобианов, которые книга печатает красным; и якобиан по весам

соответствующей матрицы весов, напечатанный зелёным.

Введём сокращение для красного члена: $\tilde{\mathbf{h}}_i = \mathbf{1}^T \prod_{j=i+1}^l \partial_{\mathbf{h}_j} \mathbf{h}_{j-1}$

Поскольку умножение матриц ассоциативно, вычисления в этом произведении можно выполнять в любом порядке. В F-AD мы шли справа налево, потому что это соответствует порядку, в котором выполнялись примитивные функции. Однако можно сделать лучше, если заметить две интересные вещи. Во-первых, самый левый член произведения — это произведение вектора на матрицу, что следует из скалярности выхода, а с вычислительной точки зрения это лучше, чем произведение двух матриц. И его результат — снова вектор. Во-вторых, сокращённый член — произведение всех входных якобианов от слоя i до слоя l — можно вычислять рекурсивно, начиная с последнего члена и последовательно умножая на входные якобианы в обратном порядке.

Соединив эти наблюдения, получаем второй подход к автоматическому дифференцированию, который мы называем автоматическим дифференцированием в обратном режиме, или R-AD; он описан ниже.



Прямой проход, затем обратный

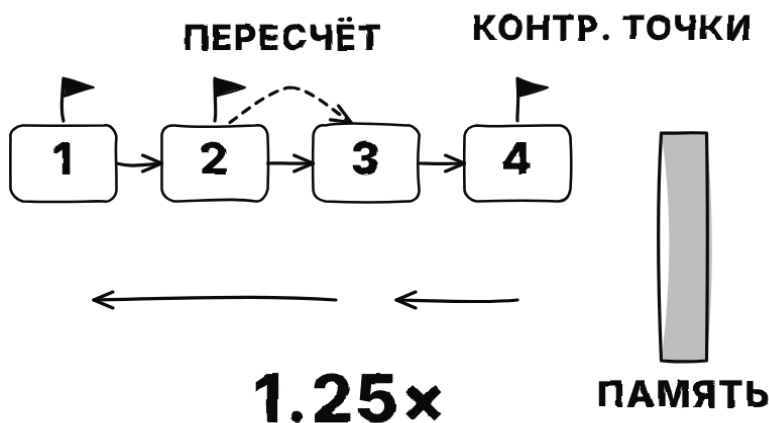
Шаг первый: в отличие от F-AD, мы начинаем с того, что выполняем всю дифференцируемую программу целиком, сохраняя все промежуточные выходы. Шаг второй: заводим вектор $\tilde{\mathbf{h}} = \mathbf{1}^T$, соответствующий самому левому члену произведения. Шаг третий: двигаясь в обратном порядке, то есть для индекса i , пробегающего значения $l, l-1, l-2$ и так далее до 1, мы сначала вычисляем градиент по i -й матрице весов как $\partial_{\mathbf{w}_i} y = \tilde{\mathbf{h}} [\partial_{\mathbf{w}_i} \mathbf{h}_i]$ — это и есть i -й нужный нам градиент. Затем обновляем наш «обратно распространяемый» входной якобиан: $\tilde{\mathbf{h}} \leftarrow \tilde{\mathbf{h}} [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i]$.

Шаги с первого по третий описывают программу, примерно симметричную исходной; мы называем её двойственной, или

обратной, программой. Члены $\tilde{h}$ называются сопряжёнными переменными, и они по очереди хранят все градиенты выхода по переменным $h_1, h_2, \dots, h_l$ нашей программы. Сравните с F-AD, где касательные, наоборот, представляли градиенты переменных h_i по весам.

В терминологии нейронных сетей исходную, или прямую, программу иногда называют прямым проходом — не путать с прямым режимом, — а обратную программу — обратным проходом. В отличие от F-AD, в R-AD вся прямая программа должна выполняться прежде, чем можно будет запустить обратную, и нужны особые механизмы, чтобы сохранить все промежуточные выходы и «развернуть» вычислительный граф. Разные библиотеки реализуют это по-разному, как описано ниже.

С вычислительной точки зрения R-AD значительно эффективнее F-AD. В частности, обе операции третьего шага R-AD — произведения вектора на матрицу, и их стоимость растёт лишь линейно по всем размерам. Платой за это служит то, что выполнение R-AD требует много памяти, поскольку все промежуточные значения прямой программы нужно хранить, придерживаясь подходящей стратегии. Особые приёмы, например контрольные точки градиента, позволяют улучшить этот компромисс, увеличив объём вычислений и частично снизив требования к памяти. Для этого сохраняют лишь несколько промежуточных выходов, называемых контрольными точками, а остальные значения пересчитывают во время обратного прохода.



МЕНЬШЕ ТОЧЕК — БОЛЬШЕ РАБОТЫ

Контрольные точки градиента

Рисунок 6.1 показывает пример контрольных точек градиента в четырёх картинках. На первой мы выполняем прямой проход, но сохраняем только выходы первого, второго и четвёртого блоков — контрольные точки. На второй обратный проход, нарисованный красными стрелками, останавливается на третьем блоке, активации которого недоступны. На третьей мы запускаем второй прямой проход, начиная с ближайшей контрольной точки, чтобы снова материализовать активации. На четвёртой мы завершаем проход. По сравнению с обычным обратным проходом это требует в 1,25 раза больше вычислений. В общем случае чем меньше контрольных точек хранится, тем выше вычислительная стоимость обратного прохода.

Вопросы

1. Почему длинное произведение для градиента можно вычислять слева направо, а не справа налево?
 - a. Потому что умножение матриц коммутативно
 - b. Потому что умножение матриц ассоциативно
 - c. Потому что все якобианы квадратные
 - d. Потому что вектором единиц можно пренебречь
2. Что делает движение слева, начиная с вектора единиц, более дешёвым?
 - a. Тогда каждый шаг — произведение вектора на матрицу, и его результат снова вектор
 - b. Тогда больше не нужны якобианы по весам
 - c. Тогда программу больше не нужно прогонять
 - d. Тогда якобианы становятся диагональными
3. Какова плата за эффективность обратного режима?
 - a. Он даёт лишь приближённые градиенты
 - b. Ему нужно много памяти, потому что все промежуточные значения прямой программы нужно хранить
 - c. Он справляется только с программами из одного слоя
 - d. Его стоимость квадратична по размеру мини-батча
4. Что происходит при контрольных точках градиента, если хранить меньше контрольных точек?

- a. Используется меньше памяти, а обратный проход требует больше вычислений
- b. Используется больше памяти, а обратный проход быстрее
- c. Ничего не меняется, потому что всё равно хранится каждое значение
- d. Градиенты становятся менее точными

Ответы

- 1. **b)** Поскольку умножение матриц ассоциативно, вычисления в произведении можно выполнять в любом порядке.
- 2. **a)** Самый левый член — произведение вектора на матрицу, следствие скалярного выхода; с вычислительной точки зрения это лучше, чем произведение двух матриц, и результат — снова вектор.
- 3. **b)** Сначала должна выполняться вся прямая программа, и все её промежуточные значения нужно хранить, поэтому обратному режиму требуется много памяти.
- 4. **a)** Сохраняется лишь несколько выходов, а остальные пересчитываются во время обратного прохода; чем меньше контрольных точек хранится, тем выше вычислительная стоимость обратного прохода.

Разобранные примеры

Цена контрольных точек На рисунке книги контрольные точки заставляют обратный проход выполнять в 1,25 раза больше вычислений, чем обычный обратный проход. Если обычный обратный проход для некоторой модели занимает 8 секунд, сколько он займёт с такими контрольными точками?

При контрольных точках сохраняется лишь часть выходов, поэтому недостающие активации вычисляются заново во время обратного прохода.

В примере книги это делает объём вычислений в 1,25 раза больше, чем у обычного обратного прохода.

Значит, время становится $1,25 \times 8$ секунд.

Это 10 секунд: на 2 секунды больше — плата за то, что хранится меньше промежуточных выходов.

Ответ: 10 секунд

Задачи

1. В программе $l = 3$ примитива. Перечислите по порядку шаги дифференцирования в обратном режиме: какой градиент вычисляется первым и как меняется сопряжённая переменная $\tilde{h}$ после каждого.

Подсказка: Начните с прогона всей программы, затем положите $\tilde{h} = 1^T$ и двигайтесь назад от $i = 3$.

Ответ: Прогоните программу и сохраните h_1, h_2, h_3 . Положите $\tilde{h} = 1^T$. Для $i = 3$: градиент по w_3 равен $\tilde{h}$, умноженному на якобиан h_3 по w_3 , затем $\tilde{h}$ умножается на якобиан h_3 по h_2 . Для $i = 2$: градиент по w_2 , затем $\tilde{h}$ умножается на якобиан h_2 по h_1 . Для $i = 1$: градиент по w_1 . Градиент последнего веса получается первым.

2. Умножение вектора-строки длины b на матрицу $b \times a$ требует $b \cdot a$ умножений; умножение матрицы $c \times b$ на матрицу $b \times a$ требует $c \cdot b \cdot a$. Сравните их при $a = b = c = 100$.

Подсказка: Вычислите оба произведения размерностей.

Ответ: Произведение вектора на матрицу требует $100 \times 100 = 10\,000$ умножений, произведение двух матриц — $100 \times 100 \times 100 = 1\,000\,000$

ooo, в сто раз больше. Вот почему начинать с вектора единиц, как делает обратный режим, выгодно.

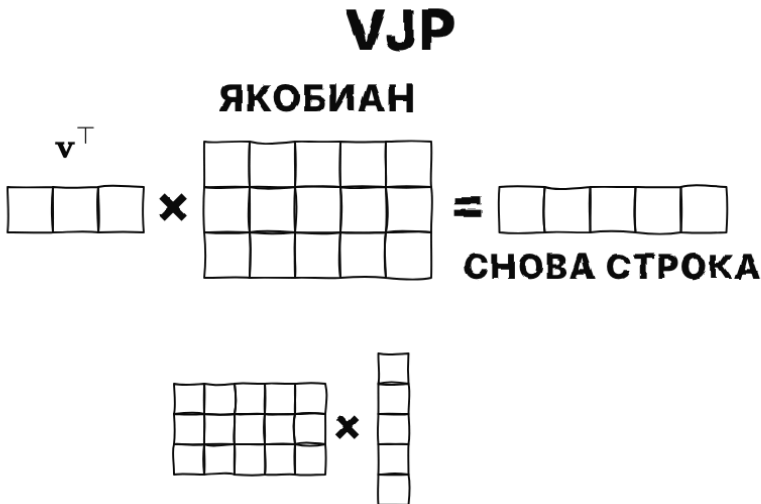
3. Объясните своими словами разницу между тем, что хранят касательные прямого режима, и тем, что хранят сопряжённые переменные обратного режима.

Подсказка: Градиенты чего и по чему?

Ответ: Касательные прямого режима — это градиенты промежуточных переменных h по весам. Сопряжённые переменные обратного режима — это градиенты выхода y по промежуточным переменным h_1, h_2 и так далее, которые сохраняются по очереди по ходу работы обратной программы.

Произведения вектора на якобиан

Практические соображения.



Строка на якобиан

Глядя на третий шаг алгоритма R-AD, можно сделать интересное наблюдение: единственная нужная нам операция — произведение вектора-строки v на якобиан f , входной или по весам. Эти две операции мы называем произведениями вектора на якобиан, или VJP, функции f . Напротив, F-AD можно целиком сформулировать через транспонированное VJP, называемое произведением якобиана на вектор, или JVP; при одномерном

выходе JVP — это производная по направлению. По аналогии VJP представляет собой применение линейного отображения, связанного с бесконечно малыми изменениями выхода функции. В следующем определении мы восстанавливаем согласованность размерностей, добавляя к вектору транспонирование.

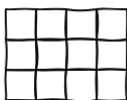
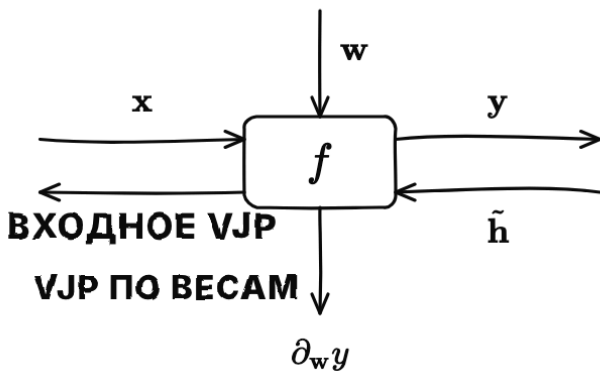
Определение: произведение вектора на якобиан. Для функции $y = f(\mathbf{x})$, где $\mathbf{x} \sim (c)$ и $y \sim (c')$, её VJP — это другая функция, определённая как $\text{vjp}_f(\mathbf{v}) = \mathbf{v}^T \partial f(\mathbf{x})$, где $\mathbf{v} \sim (c')$. Если $y = f$ несколько параметров, $f(\mathbf{x}_1, \dots, \mathbf{x}_n)$, можно определить n отдельных VJP, по одному для каждого из $\mathbf{x}_1, \dots, \mathbf{x}_n$.

В частности, в нашем случае можно определить два вида VJP, соответствующих частным производным примитива по входному аргументу и по аргументу весов. Входное VJP: $\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$. VJP по весам: $\text{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$.

Теперь две операции третьего шага алгоритма R-AD можно переписать как два вызова VJP примитивной функции с сопряжёнными значениями, опуская для удобства индекс i : сопряжённая переменная, поданная в VJP по весам, и сопряжённая переменная, поданная во входное VJP. Первый даёт градиент: $\partial_{\mathbf{w}} y = \text{vjp}_{f,\mathbf{w}}(\tilde{\mathbf{h}})$. Второй обновляет сопряжённую переменную: $\tilde{\mathbf{h}} \leftarrow \text{vjp}_{f,\mathbf{x}}(\tilde{\mathbf{h}})$.

Значит, целую систему автоматического дифференцирования можно реализовать, сначала выбрав набор примитивных операций, а затем снабдив их соответствующими VJP, ни разу не материализуя якобианы в памяти. Схематически это показано на рисунке 6.2: для R-AD примитивы нужно снабдить двумя операциями VJP — входным VJP и VJP по весам, — и одного

вызова каждой достаточно, чтобы выполнить обратный проход через примитив.



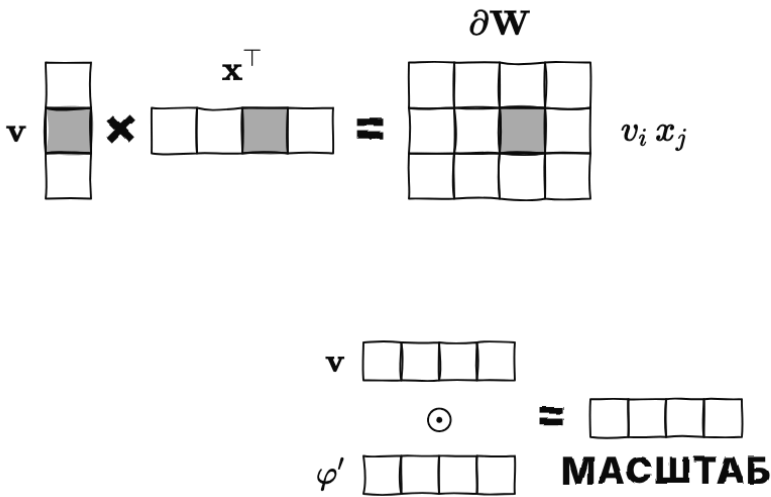
one row per e_i

Примитив с двумя VJP

На самом деле якобианы можно восстановить, многократно вызывая VJP с базисными векторами $e_1, \dots, e_n$ и получая их по одной строке за раз. Для входного якобиана, например, строки $\partial_x f(x, w)$ — это $\text{vjp_f, x}(e_1)$, $\text{vjp_f, x}(e_2)$ и так далее до $\text{vjp_f, x}(e_n)$.

Чтобы понять, чем такая переформулировка удобна, посмотрим на VJP полносвязного слоя, состоящего из линейных проекций и поэлементных нелинейностей. Сначала рассмотрим простую линейную проекцию без смещения: $f(x, W) = Wx$. Входной якобиан здесь просто W , а вот якобиан по весам — тензор ранга 3.

Для сравнения, у входного VJP нет особой структуры: $\text{vjp}_{f,x}(v) = v^T W^T = [Wv]^T$. А VJP по весам оказывается простым внешним произведением, которое полностью обходится без тензоров ранга 3: $\text{vjp}_{f,w}(v) = vx^T$.



Внешнее произведение и масштабирование

Чтобы вывести это VJP, можно записать $y = v^T W x = \sum_i \sum_j W_{ij} v_i x_j$. Отсюда сразу получаем $\partial y / \partial W_{ij} = v_i x_j$ — это и есть поэлементное определение внешнего произведения.

Значит, каждый раз, применяя в прямом проходе линейную проекцию, мы преобразуем обратно распространяемые градиенты транспонированной матрицей её весов и выполняем внешнее произведение, чтобы вычислить градиент W .

Рассмотрим теперь поэлементную функцию активации без обучаемых параметров, например ReLU: $f(\mathbf{x}) = \varphi(\mathbf{x})$ с пустым набором весов. Поскольку обучаемых параметров нет, достаточно рассмотреть входное VJP. Градиент — диагональная матрица, элементы которой — производные φ : её i -й диагональный элемент равен $\varphi'(x_i)$. Входное VJP — это умножение диагональной матрицы на вектор, что равносильно произведению Адамара, то есть масштабированию: $\text{vjp_x}(f, \mathbf{v}) = \mathbf{v} \odot \varphi'(\mathbf{x})$. Любопытно, что и в этом случае VJP можно вычислить, не материализуя всю диагональную матрицу.

Вопросы

1. Почему от примитива дифференцированию в обратном режиме нужны только произведения вектора на якобиан?
 - a. Потому что третий шаг алгоритма только и делает, что умножает вектор-строку — сопряжённую переменную — на входной якобиан или якобиан по весам
 - b. Потому что якобианы примитивов всегда векторы
 - c. Потому что прямой проход всё равно вычисляет якобианы
 - d. Потому что VJP дешевле хранить, чем вход
2. Что даёт снабжение каждого примитива двумя его VJP вместо якобианов?
 - a. Градиенты становятся точными, а не приближёнными
 - b. Больше не нужно выполнять прямой проход
 - c. Якобианы никогда не нужно материализовать в памяти

- d. Примитивам больше не нужны веса
3. Чем удобно VJP по весам для линейной проекции $f(x, W) = Wx$?
- a. Оно равно нулю
 - b. Это простое внешнее произведение $v x^T$, которое полностью обходит якобиан по весам ранга 3
 - c. Оно равно входному якобиану
 - d. Ему нужна диагональная матрица производных
4. К чему сводится входное VJP поэлементной активации, такой как ReLU?
- a. К умножению на транспонированные веса
 - b. К внешнему произведению
 - c. К поэлементному масштабированию v на производные $\phi'(x)$
 - d. К сумме всех элементов v

Ответы

1. a) Единственная операция, нужная третьему шагу, — произведение вектора-строки на якобиан f , входной или по весам; это и есть два VJP функции f .
2. c) Целую систему автоматического дифференцирования можно построить из примитивов, снабжённых своими VJP, ни разу не материализуя якобианы в памяти.
3. b) Якобиан по весам — тензор ранга 3, но VJP по весам оказывается внешним произведением $v x^T$, которое полностью обходится без тензоров ранга 3.

4. c) Её градиент — диагональная матрица производных ϕ , а умножение вектора на неё — произведение Адамара $v \odot \phi'(x)$, масштабирование, которому диагональная матрица в памяти не нужна.

Разобранные примеры

VJP по весам как внешнее произведение Линейная проекция $f(x, W) = Wx$ переводит $x = (3, 4, 5)$ в два выхода, так что W имеет размер 2×3 . Сверху приходит сопряжённая переменная $v = (1, 2)$. Вычислите VJP по весам, то есть градиент по W .

Книга показывает, что $\partial y / \partial W_{ij} = v_i x_j$, так что VJP по весам — внешнее произведение $v x^T$.

Строка 1 использует $v_1 = 1$: $(1 \times 3, 1 \times 4, 1 \times 5) = (3, 4, 5)$.

Строка 2 использует $v_2 = 2$: $(2 \times 3, 2 \times 4, 2 \times 5) = (6, 8, 10)$.

Результат имеет ту же форму 2×3 , что и W , и ни одного тензора ранга 3 так и не было построено.

Ответ: Матрица 2×3 со строками $(3, 4, 5)$ и $(6, 8, 10)$

Задачи

1. ReLU получает $x = (1, -2, 0, 5)$; её производная равна 1 там, где вход положителен, и 0 там, где отрицателен. Сопряжённая переменная $v = (2, -1, 3)$. Вычислите входное VJP $v \odot \phi'(x)$.

Подсказка: Сначала запишите $\phi'(x)$ поэлементно, затем перемножьте поэлементно.

Ответ: $\phi'(x) = (1, 0, 1)$, так что $v \odot \phi'(x) = (2, 0, 3)$. Средняя компонента выключена, потому что её вход был отрицательным.

2. У примитива 4 выхода. Сколько вызовов его входного VJP нужно, чтобы восстановить полный входной якобиан, и с какими

векторами?

Подсказка: Каждый вызов с базисным вектором даёт одну строку.

Ответ: Четыре вызова, с базисными векторами e_1 , e_2 , e_3 и e_4 ; каждый вызов возвращает одну строку входного якобиана.

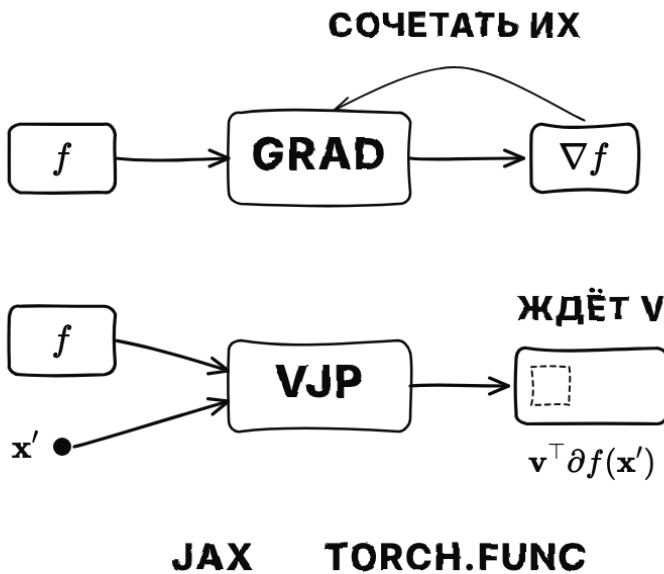
3. Объясните своими словами, что происходит с обратным распространением градиента и что вычисляется для весов всякий раз, когда в прямом проходе есть линейная проекция.

Подсказка: Есть два VJP: одно для входа, другое для весов.

Ответ: Сопряжённая переменная, идущая назад, умножается на транспонированные веса слоя — через входное VJP. Одновременно градиент W — это внешнее произведение сопряжённой переменной и входа слоя — через VJP по весам.

Реализация системы обратного режима

Систему R-AD можно реализовать многими способами — от списков Венгерта, как в TensorFlow, до преобразований исходного кода в исходный код. Здесь мы кратко рассмотрим некоторые распространённые реализации в существующих библиотеках.



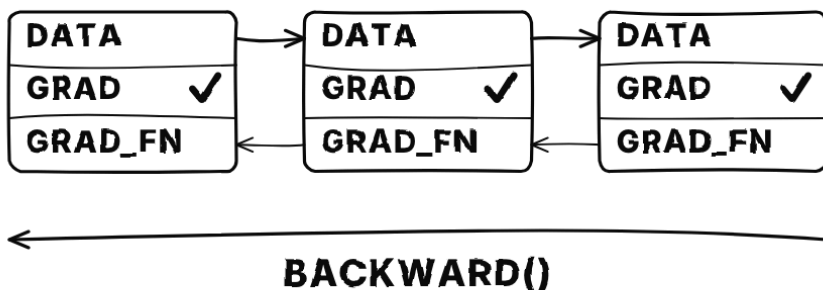
Функции, которые делают функции

Во-первых, описание примитивов как функций двух аргументов, $f(\mathbf{x}, \mathbf{w})$, соответствует функциональным библиотекам вроде JAX,

где vsf — функция. Рассмотрим функцию $f(\mathbf{x})$ с s -мерным входом и s' -мерным выходом. С этой точки зрения VJP можно реализовать как функцию высшего порядка с сигнатурой $(\mathbb{R}^c \rightarrow \mathbb{R}^{c'}) \rightarrow \mathbb{R}^c \rightarrow (\mathbb{R}^{c'} \rightarrow \mathbb{R}^c)$. То есть по функции f и входу $\mathbf{x}'$ VJP возвращает другую функцию, которую можно применить к s' -мерному вектору $\mathbf{v}$ и получить $\mathbf{v}^T \partial f(\mathbf{x}')$.

Аналогично градиент одномерной функции можно реализовать как ещё одну функцию высшего порядка с сигнатурой $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$. Она принимает на вход функцию $f(\mathbf{x})$ и возвращает другую функцию, которая вычисляет $\nabla f(\mathbf{x})$. В JAX эти идеи реализованы в функциях вроде `jax.grad` и `jax.vjp`, а в PyTorch они повторены в модуле `torch.func`. Например, во врезке 6.2 книги определяется функция суммы квадратов, `func.grad` превращает её в функцию градиента, и, применённая к случайному вектору из десяти чисел, она возвращает градиент размера десять. Интерфейс `torch.func` повторяет интерфейс JAX, и на практике функцию можно трассировать, например с помощью `torch.compile`, чтобы получить оптимизированный вычислительный граф. Многие операции, например вычисление гессиана, можно выполнить, умело сочетая JVP и VJP в соответствии с их сигнатурами.

Как мы уже говорили, на практике наши модели реализованы как композиции объектов, параметры которых инкапсулированы в виде свойств. Одна возможность — «очистить» объект, превратив его в чистую функцию. В PyTorch, например, параметры модели извлекают в словарь, а затем вызывают прямую функцию модели через `torch.func.functional_call`, передавая модель, параметры и вход.



MICROGRAD

Тензоры, которые помнят своё происхождение

В более общем случае библиотеки вроде PyTorch снабжены приёмами, которые справляются с этой ситуацией напрямую, без промежуточных операций. В PyTorch, например, объекты-тензоры дополнены сведениями об операции, которая их породила. Всякий раз, когда для скалярного значения вызывается `backward()`, эти свойства используются, чтобы обойти вычислительный граф в обратном порядке, сохраняя соответствующие градиенты внутри тензоров, которым они нужны.

Рисунок 6.3 показывает это в двух половинах. Слева тензор PyTorch несёт сведения о своём градиенте, пустом при инициализации, и об операции, которая его создала. Справа во

время обратного прохода свойство `grad_fn` используется, чтобы обойти вычислительный граф в обратном порядке, а градиенты сохраняются внутри свойства `grad` тензора всякий раз, когда `requires_grad` явно установлено в `true`, — чтобы не расходовать память зря.

Это лишь общий обзор того, как такие системы реализованы на практике, и мы опускаем многие детали, за которыми отсылаем к официальной документации. Автор горячо советует попробовать реализовать систему обратного режима с нуля; в сети можно найти много учебных реализаций, например `micrograd` Андрея Карпаты.

Вопросы

1. Что принимает и что возвращает операция градиента в функциональной библиотеке вроде JAX?
 - a. Принимает число и возвращает число
 - b. Принимает функцию f и возвращает другую функцию, которая вычисляет градиент f в любой точке входа
 - c. Принимает тензор и сохраняет градиент внутри него
 - d. Принимает список операций и возвращает их якобианы
2. Что возвращает VJP, если дать ему функцию f и вход x' ?
 - a. Полный якобиан f в точке x'
 - b. Значение $f(x')$
 - c. Другую функцию, которая, применённая к вектору v , возвращает v^T , умноженный на якобиан f в точке x'

d. Список параметров f

3. Откуда PyTorch знает, как пройти назад по вычислительному графу, когда вызывается `backward()`?
- a. Каждый тензор несёт сведения об операции, которая его создала, и по ним идут в обратном порядке
 - b. Он символично дифференцирует исходный код
 - c. Он заново выполняет прямой проход с маленькими изменениями каждого входа
 - d. Он просит пользователя перечислить операции

Ответы

1. **b)** Градиент — функция высшего порядка: она принимает на вход функцию $f(x)$ и возвращает другую функцию, которая вычисляет $\nabla f(x)$.
2. **c)** По f и x' VJP возвращает другую функцию, которую можно применить к c' -мерному вектору v и получить $v^T \partial f(x')$.
3. **a)** Объекты-тензоры дополнены сведениями об операции, которая их породила, и при вызове `backward()` эти свойства используются, чтобы обойти граф в обратном порядке.

Разобранные примеры

Градиент суммы квадратов Во врезке 6.2 книги функция — $f(x)$ = сумма квадратов элементов x . Её градиент равен $2x$. Что вернёт функция градиента для $x = (1, -2, 3)$ и какова её форма по сравнению с x ?

Каждый элемент x_k входит в f только как x_k^2 , производная которого равна $2x_k$.

Значит, градиент равен $(2 \times 1, 2 \times (-2), 2 \times 3) = (2, -4, 6)$.

У градиента по одному элементу на каждый элемент входа, поэтому его форма — как у x , точно так же как во врезке вектор размера 10 даёт градиент размера 10.

Ответ: (2, -4, 6), той же формы, что и x

Задачи

1. Прочитайте своими словами сигнатуру градиента, $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$. Почему на вход ему нужна функция с одномерным выходом?

Подсказка: Посмотрите, что входит слева и что выходит справа.

Ответ: Он принимает функцию из c чисел в одно число и возвращает функцию из c чисел в c чисел. Градиент определён для скалярного выхода: для каждого из c входов он даёт одну частную производную, так что в результате c элементов.

2. Объясните, почему модель, записанную как объект с параметрами, нужно «очистить», прежде чем пользоваться функциональными инструментами вроде `func.grad`.

Подсказка: Функциональные инструменты дифференцируют по аргументам функции.

Ответ: Функциональные инструменты считают всё функцией и дифференцируют по её аргументам, как в $f(x, w)$. В объекте параметры спрятаны внутри как свойства, поэтому их сначала извлекают, а затем явно передают через функциональный вызов, и модель становится чистой функцией своих параметров и входа.

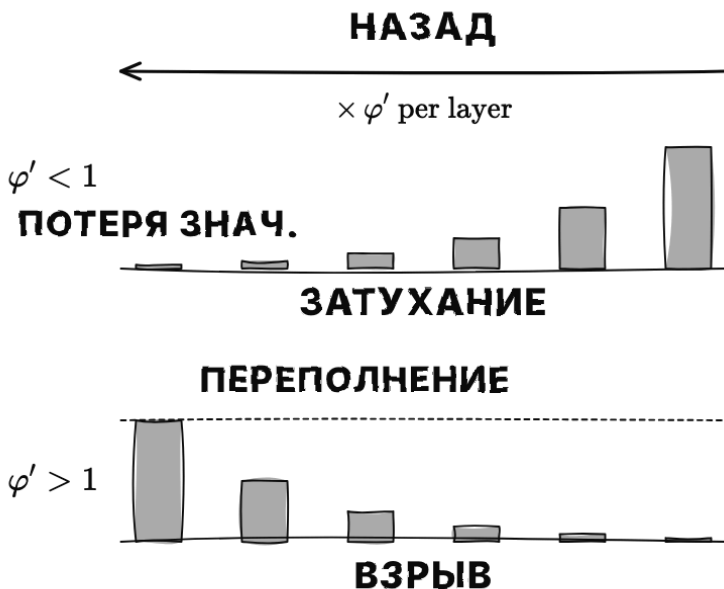
3. Почему в PyTorch градиенты хранятся только в тех тензорах, у которых `requires_grad` установлено в `true`?

Подсказка: Подумайте о памяти.

Ответ: Хранить градиент в каждом тензоре значило бы зря расходовать память; PyTorch хранит градиенты только там, где они явно нужны, — обычно в обучаемых параметрах.

Выбор функции активации

Попутно мы теперь можем объяснить, почему ReLU — хороший выбор функции активации. Внимательный взгляд на входное VJP функции активации говорит нам, что всякий раз, когда мы добавляем в модель функцию активации, сопряжённые переменные при обратном проходе масштабируются на множитель $\phi'(x)$. В моделях со многими слоями это может привести к двум патологическим явлениям. Если $\phi' < 1$ всюду, есть риск, что градиент будет сжиматься к нулю экспоненциально быстро с ростом числа слоёв. Это называется проблемой затухающего градиента. И наоборот, если $\phi' > 1$ всюду, возникает противоположная проблема: градиенты экспоненциально быстро растут к бесконечности с ростом числа слоёв. Это называется проблемой взрывающегося градиента.

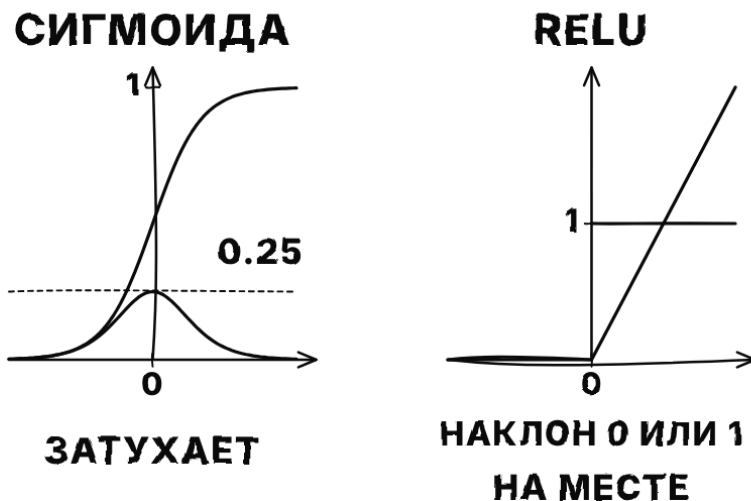


Затухающие и взрывающиеся градиенты

На практике это серьёзные проблемы, потому что библиотеки представляют числа с плавающей точкой с ограниченной точностью, обычно 32 бита или меньше, а значит, с ростом числа слоёв быстро могут возникнуть потеря значимости или переполнение.

Удивительно, но стопка линейных слоёв, реализованная в арифметике с плавающей точкой, не полностью линейна из-за мелких разрывов на уровне машинной точности! Обычно это не проблема, но этим можно воспользоваться, чтобы обучать полностью линейные глубокие нейронные сети.

В качестве примера того, как может возникнуть затухание градиентов, рассмотрим сигмоиду $\sigma(s)$. Мы уже упоминали, что в прошлом это была распространённая функция активации, потому что она — мягкое приближение ступенчатой функции. Мы также знаем, что $\sigma'(s) = \sigma(s)(1 - \sigma(s))$. Вместе с тем, что $\sigma(s)$ лежит между 0 и 1, получаем, что $\sigma'(s)$ лежит между 0 и 0,25. Поэтому сигмоида — первый кандидат на проблемы с затуханием градиентов. Рисунок 6.4 показывает слева сигмоиду и её производную, а справа — ReLU и её производную.



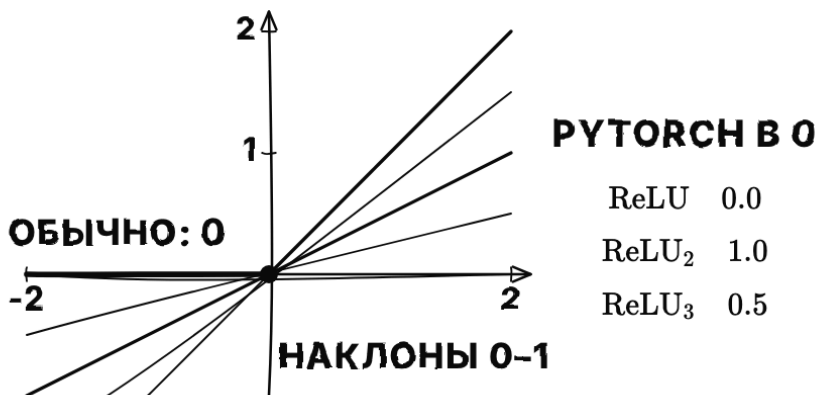
Наклоны сигмоиды и ReLU

Построить функцию активации, у которой никогда не бывает затухающих или взрывающихся градиентов, непросто, поскольку единственная функция с $\phi'(s) = 1$ всюду — тождественная

функция. Значит, нам нужна функция, «достаточно линейная», чтобы избежать проблем с градиентом, но достаточно «нелинейная», чтобы разделять линейные слои. ReLU оказывается хорошим кандидатом, поскольку её производная равна 0 при x меньше 0 и 1 при x больше 0. Градиент либо обнуляется, что делает вычисление разреженным, либо умножается на 1, что избавляет от проблем масштаба; это показано справа на рисунке 6.4.

Заметим попутно, что градиент ReLU одинаков, вычисляем ли мы его по входу слоя ReLU или по его выходу, поскольку мы лишь маскируем отрицательные значения, оставляя положительные нетронутыми. Поэтому ещё одно преимущество ReLU как функции активации в том, что при R-AD можно немного сэкономить память, затирая вход слоя при прямом проходе без ущерба для правильности процедуры AD. В PyTorch, например, это делается установкой параметра работы на месте.

Субдифференцируемость и автоматическое дифференцирование.



Много наклонов в углу

Есть одна мелочь, которую мы до сих пор обходили: ReLU недифференцируема в 0, из-за чего вся сеть оказывается негладкой. Что происходит в этом случае? «Прагматичный» ответ таков: при минимизации стохастическим градиентным спуском из случайной ненулевой инициализации вероятность оказаться точно в $s = 0$ практически нулевая, а градиент определён в $\text{ReLU}(\epsilon)$ для любого ϵ , отличного от нуля.

Для более технического ответа можно ввести понятие субградиента функции. Определение: для выпуклой функции $f(x)$ субградиентом в точке x называется такое число z , что для всех y выполняется $f(y) \geq f(x) + z(y - x)$.

Обратите внимание на сходство с определением выпуклости: субградиент — это наклон «касательной» к $f(x)$ прямой, такой, что вся функция f лежит выше неё. Если f дифференцируема в x , такая прямая только одна, и её наклон — производная f в x . В негладкой точке существует много субградиентов, и они образуют множество, называемое субдифференциалом f в x : множество всех z , являющихся субградиентами $f(x)$.

С этим определением можно завершить анализ градиента ReLU, заменив градиент в 0 её субдифференциалом. При s меньше 0 субдифференциал содержит только 0; при s больше 0 — только 1; а при $s = 0$ это весь отрезок от 0 до 1. Значит, любое значение от 0 до 1 — допустимый субградиент в 0, а большинство реализаций на практике выбирают $\text{ReLU}'(0) = 0$. Выбор субградиентов на каждом шаге итеративной процедуры спуска называется субградиентным спуском.

На самом деле ситуация ещё сложнее, потому что для невыпуклых функций субградиент может быть не определён. В этом случае можно прибегнуть к обобщениям, которые ослабляют прежнее определение до локальной окрестности x , таким как субдифференциал Кларка. Субдифференцируемость может создавать проблемы и в AD, где разные реализации одной и той же функции могут давать разные, а возможно, и недопустимые субградиенты, и для строгого доказательства приходится рассматривать более тонкие варианты цепного правила. Рассмотрим такой пример: определим две функции, $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ и $\text{ReLU}_3(s) = 0,5(\text{ReLU}(s) + \text{ReLU}_2(s))$. Обе равносильны ReLU, но в PyTorch обратный проход в 0 возвращает 0,0 для ReLU, 1,0 для ReLU_2 и 0,5 для ReLU_3 .

От теории к практике.

Если вы выполняли упражнения главы 5, вы уже видели применение R-AD и в PyTorch, и в JAX, а эта глава, особенно раздел о реализации системы обратного режима, должна была прояснить, как они устроены.

Хорошая идея — попробовать заново реализовать простую систему R-AD, похожую на систему PyTorch. Например, если ограничиться скалярными величинами, репозиторий `micrograd` — очень хорошая учебная реализация. Единственная деталь, которую мы не рассматриваем: при переходе к общему ациклическому графу перед обратным проходом важно упорядочить переменные вычислительного графа, чтобы не создать неверных путей обратного распространения. В `micrograd` это достигается недорогой топологической сортировкой переменных.

Интересно также попробовать реализовать в PyTorch новый примитив в том смысле, в каком это слово используется в этой главе, — для этого нужно задать его прямой проход вместе с его VJP. Примером может служить одна из обучаемых функций активации из главы 5. Это учебное упражнение в том смысле, что то же самое можно реализовать, унаследовав класс от `nn.Module` и предоставив механизму AD в PyTorch самому вывести обратный проход.

Все эти шаги можно повторить и в JAX. Реализуйте учебную версию JAX с помощью `autodidax`. Напишите новый примитив, реализовав соответствующее ему VJP. И прочитайте JAX Autodiff Cookbook, чтобы узнать о продвинутых применениях механизма

автоматического дифференцирования, таких как производные высших порядков, гессианы и многое другое.

Вопросы

1. Почему функция активации, производная которой всюду меньше 1, может сорвать обучение глубокой модели?
 - a. Потому что на каждом слое сопряжённые переменные умножаются на ϕ' , и градиент может экспоненциально быстро сжиматься к нулю с ростом числа слоёв
 - b. Потому что прямой проход становится линейным
 - c. Потому что веса растут до бесконечности
 - d. Потому что субградиент перестаёт быть определённым
2. Почему сигмоида — первый кандидат на затухание градиента?
 - a. Потому что её производная может быть больше 1
 - b. Потому что её производная лежит между 0 и 0,25
 - c. Потому что она недифференцируема в 0
 - d. Потому что это тождественная функция
3. Почему вход ReLU можно при прямом проходе затереть её выходом, не сломав обратный режим?
 - a. Потому что у ReLU нет обратного прохода
 - b. Потому что её градиент одинаков, считать ли его по входу или по выходу: маскируются только отрицательные значения

- c. Потому что PyTorch всё равно пересчитывает все входы
 - d. Потому что выход ReLU всегда равен нулю
4. Чему равен субдифференциал ReLU в нуле?
- a. Только значению 0
 - b. Только значению 1
 - c. Всему отрезку от 0 до 1
 - d. Он пуст

Ответы

- 1. a)** Каждая функция активации при обратном проходе умножает сопряжённые переменные на $\phi'(x)$; если $\phi' < 1$ всюду, градиент рискует экспоненциально быстро сжаться до нуля с ростом числа слоёв — это проблема затухающего градиента.
- 2. b)** Поскольку $\sigma'(s) = \sigma(s)(1 - \sigma(s))$, а $\sigma(s)$ лежит между 0 и 1, производная лежит между 0 и 0,25, так что каждый сигмоидный слой уменьшает градиент.
- 3. b)** ReLU только маскирует отрицательные значения и не трогает положительные, поэтому её градиент одинаков, берётся ли он по входу или по выходу; это экономит немного памяти.
- 4. c)** При $s = 0$ любой наклон от 0 до 1 даёт прямую, лежащую под ReLU, поэтому субдифференциал — отрезок от 0 до 1; большинство реализаций выбирают 0.

Разобранные примеры

Десять сигмоидных слоёв В модели с 10 сигмоидными слоями каждый слой умножает сопряжённую переменную не более чем на 0,25 — наибольшее значение σ' . Каков наибольший

возможный множитель, во сколько раз уменьшится градиент, пройдя назад через все 10 слоёв?

Книга показывает, что $\sigma'(s)$ лежит между 0 и 0,25.

Каждый слой умножает распространяемую назад сопряжённую переменную на σ' , то есть не более чем на 0,25.

Десять слоёв умножают не более чем на $0,25^{10}$.

$0,25^{10} = 1 / 4^{10} = 1 / 1\,048\,576$, примерно 0,00000095.

Даже в лучшем случае градиент становится примерно миллионной долей своей величины: градиент затухает.

Ответ: Не более примерно 0,00000095, одна 1 048 576-я часть

Задачи

1. При некотором входе сигмоида даёт $\sigma(s) = 0,9$. Вычислите $\sigma'(s)$ по формуле $\sigma'(s) = \sigma(s)(1 - \sigma(s))$.

Подсказка: Умножьте $\sigma(s)$ на единицу минус $\sigma(s)$.

Ответ: $\sigma'(s) = 0,9 \times 0,1 = 0,09$ — гораздо меньше максимума 0,25.

2. По определению $f(y) \geq f(x) + z(y - x)$ для всех y проверьте, являются ли $z = 0,3$ и $z = 1,2$ субградиентами ReLU в точке $x = 0$.

Подсказка: При $x = 0$ условие принимает вид $\text{ReLU}(y) \geq z \cdot y$; попробуйте положительное и отрицательное y .

Ответ: Для $z = 0,3$: если $y > 0$, то $y \geq 0,3y$ верно; если $y < 0$, то $0 \geq 0,3y$ верно; значит, 0,3 — субградиент. Для $z = 1,2$: при $y = 1$ неравенство $1 \geq 1,2$ ложно, значит, 1,2 — не субградиент. Это согласуется с тем, что субдифференциал — отрезок от 0 до 1.

3. Функция $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ из книги равна ReLU. Считая, что PyTorch берёт производную ReLU в 0 равной 0, объясните, почему обратный проход в 0 возвращает 1,0 для ReLU_2 и 0,5 для $\text{ReLU}_3(s) = 0,5(\text{ReLU}(s) + \text{ReLU}_2(s))$.

Подсказка: Продифференцируйте каждое слагаемое так, как это делает движок автоматического дифференцирования, с $\text{ReLU}'(0) = 0$.

Ответ: Для ReLU_2 движок получает $-\text{ReLU}'(0) + 1 = 0 + 1 = 1$. Для ReLU_3 он получает $0,5 \times (\text{ReLU}'(0) + \text{ReLU}_2'(0)) = 0,5 \times (0 + 1) = 0,5$. Все три функции равны, а движок возвращает три разных субградиента.