Содержание

1 Введение.

Что вообще такое оптимизация? Поиск некоторого оптимального значения. Обычно это задачи на тему максимизации/минимизации значения некоторой функции (и поиска аргумента, при котором оно достигается).
Встречается это дело практически везде. В машинном обучении мы ищем параметр модели, когда она лучшим образом что-то предсказывает. Или мы хотим минимизировать невязку в системе линейных уравнений. Или мы проектируем какое-то устройство, и нам нужно сделать так, чтобы оно работало оптимально. Отсюда есть много слабо связанных методов и разделов, которые созданы для разных задач.
О'кей, вот хотим мы найти

$$\operatorname*{argmin}_{x \in X} f(x)$$

Тут бывают разные X и разные f. Самый большой раздел — когда X это $\mathbb{R}^n$. Тут существует много разных задач (тот же поиск коэффициентов нейросети) и много методов. Методы делятся на 0, 1 и 2 порядка в зависимости от гладкости f.

0. Нулевого порядка (differentiation-free) не опираются на гладкость никак и не используют производную (например, покоординатный спуск, симплекс-метод). При этом тут всё равно обычно полагают непрерывность f, иначе жить совсем грустно.

1. Методы первого порядка используют гардиент. Самый известный метод — градиентный спуск, когда мы идём в направлении антиградиента. И отсюда метод стохастического градиентного спуска (когда мы не вычисляем градиент точно) и некоторые другие вариации. И вдобавок бывают методы сопряжённых градиентов, сопряжённых направлений, сопряжённых невязок. Многие, кстати, требуют от функции ещё каких-то условий.

2. Методы 2 порядка полагают, что функция дифференцируется два раза и либо явно вычисляют вторые производные (тогда это какие-то ньютоновские методы), либо используют просто их наличие, а явно не вычисляют (квазиньютоновские методы).

Вдобавок к этому есть какие-нибудь методы, основанные на случайности (метод Монте-Карло, например). И есть ещё методы, которые в одномерном случае работают (и многие многомерные методы имеют одномерные методы как подзадачу). Что у нас есть в одномерном случае?

- Дихотомия. Если предполагается, что функция имеет один минимум, то можно пилить отрезок пополам и искать. Или не пополам, а в отношении золотого сечения или ещё как-то.

- Полиномиальные методы. Можно считать, что наша функция близка к многочлены и искать минимумы этого многочлена.

Всё вышеперечисленное можно сочетать: сначала построить приближение многочленом, если там какой-то кринж, то воспользоваться другом методом.

Ещё стоит заметить, что у нас есть локальные минимумы, и иногда нам хватит локального, а не глобального. И методы у нас обычно ищут локальный метод, потому что искать глобальный минимум сложно и очень немногие методы могут это сделать (и ещё и от функции чего-то требуют).
Ещё есть отдельный подкласс задач, когда $X = \mathbb{N}$ или $X = \mathbb{Z}$ или большое конечное множество илии вообще какие-то не-числа (обычно дискретной природы). Такого сорта задачи называются задачами целочисленной оптимизации. Конкретно с целыми числами — задачами целочисленного программирования. Тут методы совершенно особые, и мы их в этом курсе затрагивать не будем.
Бывают более сложные варианты, когда у нас $X = \mathbb{R} \times \mathbb{N}$, например. Это задачи смешанного программирования, и с ними вообще всё плохо жить.
И ещё есть задачи, когда X — это некоторое подмножество $\mathbb{R}^n$. Такие задачи — задачи с ограничениями. К ним, кстати, обычно применяется другая нотация:

$$\begin{gathered}\operatorname{argmin} f(x) \\ \scriptstyle x\in\mathbb{R}^n \\ \scriptstyle c_i(x)=0 \\ \scriptstyle c_j(x)\geqslant 0\end{gathered}$$

То есть обычно ограничения — это равенство или неравенство. Это тоже достаточно большой и сложный раздел со своими методами. Иногда задача сводится к задаче без ограничений, иногда приходится решать как есть своими особыми методами. Например, есть линейное программирование, когда у нас

$$f(x) = c_0 + c_1x_1 + c_2x_2 \cdots + c_nx_n \qquad A_1x = b_1, A_2x \geqslant b_2$$

Это много где применимая задача, к ней много что сводится. А решается она, например, симплекс-методом.
Примером задачи, которая сводится к задаче линейного программирования, — транспортная задача. У нас есть какие-то фабрики и пункты продажи товара. Возникает вопрос с какой фабрики в каком количестве что куда вести. Каждая фабрика производит фиксированное число в единицу времени, у каждого магазина есть спрос (и мы не хотим создавать дефицит) и есть стоимость доставки на единицу товара из каждой фабрики в каждый магазин.
Ещё есть задачи квадратичного программирования, когда

$$f(x) = x^TAx \qquad A > 0$$

Положительная определённость матрицы хороша тем, что минимум у этой функции будет единственным.
Ещё выпуклое программирование бывает, когда f выпуклая функция и следовательно имеет единственный минимум.

2 Решение задачи без ограничений.

Тут есть два класса методов — линейный поиск и доверительные регионы. Что интересно, доверительные регионы показывают себя лучше (но и пишутся сложнее).

- Линейный поиск каким-то берёт точку и исходя из каких-то факторов (тот же антиградиент, например) идём от этой точки в определённом направлении. На какой шаг идём — тоже зависит от метода. Например, это может быть какой-то константный шаг (как в обычном градиентном спуске). Но это может быть плохо. При слишком большом шаге мы сильно проиграем, а если и не так, то, может, нам придётся долго ждать. Кстати, да, обычно стоит цель не только найти минимум, но и сделать это быстро. «Быстро», кстати, измеряется обычно не в секундах, а к количестве раз, сколько мы вычислим функцию и/или её градиент.

- В случае доверительного региона мы строим некоторую модель для нашей функции, ищем минимум модели, уточняет модель, повторяем. Что такое модель? Какая-то функция. Самая простая — квадратичная (в некоторой окрестности локального минимума функция достаточно близка к квадратичной). При этом, чем лучше наша функция соотвествует модели, тем больше мы можем расширить область для следующей модели.

В доверительные регионы также входит задача нахождения гиперпараметров. У нас есть какие-то параметры модели (шаг спуска, например). И тут внутри нашей задачи оптимизации возникает ещё одна задача оптимизации, и тут всё совсем грустно, потому что не хочется очень много раз вычислять функцию. И тут обычно берут несколько комбинаций параметров и либо равномерно раскидывают по пространству параметров, либо случайно (второе лучше). Но ещё есть баесовский метод, когда мы итеративно уточняем модель по схеме, схожей с методом доверительных регионов.
Ну и небольшой офф-топ с просто интересным алгоритмом (генетическим алгоритмом), который выглядит так: берём несколько случайных точек (интерпретируем их как популяцию), выбираем несколько лучше, особым образом скрещиваем их, добавляем немного случайности (мутации) и так создаём новое поколение. И этот алгоритм тоже довольно хорошо работает для довольно многих задач (и, кстати, не требует дифференцирования). Более того, он может работать даже не на числах, а хоть даже на строках.

2.1 Задача неограниченной локальной минимизации

То есть есть некоторая функция $f\colon \mathbb{R}^n \to \mathbb{R}$, и мы хотим найти, на каких аргументах достигается минимум. Минимум мы ищем локальный, потому что глобальный искать очень трудно.
Для начала вспомним вообще, что такое минимум, какие у него есть свойства и т.д. Для начала вспомним формулу Тейлора, а точнее несколько следствий.

$$f(x+p) = f(x) + \nabla f(x+tp)^T p \qquad t \in (0;1)$$

Это работает для гладкой функции f, и вообще дальше предполагаем, что функция у нас гладкая необходимое количество раз.

$$f(x+p) = f(x) + \nabla f(x)^T p + \frac{1}{2} p^T \nabla^2 f(x+tp) p \qquad t \in (0;1)$$

Здесь и далее ∇ — градиент, а ∇^2 — гессиан (матрица вторых производных).
Так вот, что же такое у нас минимум. Ну, с определением понятно, а мы хотим свойства и признаки.

- Если x^* — точка минимума f, то $\nabla f(x^*) = 0$.
- Если x^* — точка минимума f, то $\nabla^2 f(x^*) \geqslant 0$.
- Если $\nabla^2 f(x^*) > 0$, то x^* — точка минимума f.

Также есть такое весёлое свойство как выпуклость (там искать минимум вообще очень просто, и локальный минимум будет глобальным), но это редкое свойство.
Методов у нас очень много, нет одного правильного, разные методы лучше в разных случаях,.. Стоп. Что такое «лучше»? Смотрите. Обычно мы решаем всё численным методом, то есть строим последовательность, которая сходится к минимуму. Бесконечно строить эту последовательность мы не можем, поэтому надо остановиться. И мы получим некоторую точность, а для достижения этой точности придётся некоторое количество раз вычислить функцию, её градиент и её гессиан. И вот эти три количества и являются отражением эффективности метода.
Большую роль в этом всём играет размерность пространства параметров. В случае, если $n = 10$, нам хорошо подойдут одни методы, а если $n = 100000$, то другие (например, считать для такой размерности в явном виде гессиан очень трудно; и гессиан надо либо приближённо считать, либо не использовать вообще). В современных нейронных сетях десятки и сотни тысяч параметров, где уже дорого использовать не только гессиан, но и градиент.

Градиентный спуск. Что же такое градиентный спуск? Метод выглядит так. Возьмём x_0 — некое начальное приближение. Как его выбрать — тоже задача непростая, в зависимости от него мы получим разные результаты. Если мы примерно догадываемся, где минимум, можно попытаться выбрать точку близко к нему, например (хотя это не гарантирует нам быструю сходимость). Впрочем сейчас не об

этом.
Так вот, у нас есть начальное условие, и мы двигаем наши параметры по формуле

$$x_{k+1} = x_k + \alpha p_k$$

При этом хороший вопрос — когда остановится. Есть разные соображения о том, когда останавливаться. Можно смотреть на условия минимума и как-то их проверять. Можно смотреть на Гессиан, если нам нужна высокая гарантия, что мы нашли минимум, но это делают достаточно редко, потому что это сложно. Но в любом случае это не делают единственным критерием. Вторым критерием делают изменение x. Если он меняется очень мало, с большой вероятностью мы в тупике.
Ещё нам часто хочется, чтобы функция убывала с ростом k (это, опять же, не что-то универсальное, в стохастическом градиентном спуске это не выполняется, но тут хочется). Для этого надо наложить условия на p_k:

$$p_k^T \nabla f(x_k) < 0$$

Самое очевидное — в качестве p_k взять $-\nabla f(x_k)$. Это, собственно, и есть градиентный спуск. Это не единственный вариант, что можно делать, но один из. Альтернативные пути: можно смотреть на предыдущие шаги каким-то образом (метод сопряжённых градиентов, например) или можно пользоваться Ньютоновскими шагами.
Вопрос: что такое α (в машинном обучении называют коэффициент обучения)? Простейший вариант — какая-то константа. Но также можно менять α в зависимости от номера шага (называется learning rate scheduling). Впрочем, такое не очень имеет смысл использовать в обычном градиентном спуске, помогает только в стохастическом.
Что ещё можно делать? Можно искать минимум вдоль направления. Тут в первую очередь возникают два вопроса: как и насколько это нам надо (типа, насколько это нужно, если это просто один из шагов алгоритма). На второй мы не ответим, а про первый немного поговорим. Для начала мы ищем интервал, где находится минимум. Как? Посмотрим на условие $p_k^T \nabla f(x_k) < 0$. Это даёт направление, в котором функция убывает. Мы смотрим на направление, куда функция убывает, и идём туда с каким-то шагом до тех пор, пока функция убывает. Скорее всего наша функция не убывает бесконечно, а значит когда начнёт возрастать, мы найдём три точки, где центральная меньше боковых, то есть наши отрезок, где лежит минимум (bracketing называется). Дальше можно использовать деление пополам (или какие-то его вариации типа золотого сечения или ещё чего). Можно взять точки, провести многочлен через них и взять минимум (это интерполяция). Можно совместить интерполяцию с дихотомией. Или можно использовать что-то, что использует производные.

Условия Вольфа. Мы уже упоминали, что у нас есть вопрос, с какой точностью нужно искать минимум. Мы можем сколь угодно уточнять минимум, но когда-то надо остановиться. Один из ответов на этот вопрос как раз и дают условия Вольфа (условия достаточного убывания). Пусть $\nabla f_k^T p_k < 0$.

1. Условие достаточного убывания:

$$f(x_k + \alpha p_k) \leqslant f(x_k) + c_1 \alpha \nabla f_k^T p_k \qquad c_1 \in (0; 1)$$

2. Условие кривизны

$$\nabla f(x_k + \alpha p_k)^T p_k \geqslant c_2 \nabla f_k^T p_k \qquad c_2 \in (c_1; 1)$$

Тут берут $c_1 \approx 10^{-4}$, а c_2 в зависимости от метода. 0.9, если используется метод высокого порядка (типа Ньютоновского), или 0.1, если какие-нибудь сопряжённые градиенты.
Теперь что это условия значат? Первое условие задаёт некоторую прямую (которая убывает по α), ниже которой мы должны оказаться. То есть если мы делаем большой шаг, то функция должна достаточно уменьшиться. Что значит второе условие? Оно говорит, что градиент в той точке, куда мы шагнём, должен быть в некоторое количество раз больше, чем текущий.
Очень круто. А существуют ли вообще точки, которые этому условию удовлетворяют вообще? Оказывается, что да, существование решения можно доказать для ограниченной снизу функции.
Итак, возьмём функцию $f\colon \mathbb{R}^n \to \mathbb{R}$, $\nabla f(x_k)^T p_k < 0$, которая ограничена снизу. Пусть $\phi(\alpha) = f(x_k +$

$\alpha p_k)$, $l(\alpha) = f(x_k) + \alpha c_1 \nabla f(x_n)^T p_k$. Поскольку l убывает бесконечно, а ϕ — нет, они пересекутся где-то. Возьмём первую же точку пересечения и обозначим её α'.

$$f(x_k + \alpha' p_k) = f(x_k) + \alpha' c_1 \nabla f(x_k) p_k$$

Очевидно, первое условие Вольфа будет выполняться для всех $\alpha < \alpha'$. А для каких будет выполняться второе? Вспомним теорему о среднем:

$$f(x + p) = f(x) + \nabla f(x + \xi p)^T p$$

Применяя эту теорему, можно записать следующее соотношение:

$$f(x + \alpha p_k) - f(x_k) = \alpha' \nabla f(x_k + \alpha'' p_k) p_k^T$$

Тогда

$$\nabla f(x_k + \alpha'' p_k)^T p_k = c_1 \nabla f(x_k)^T p_k > c_2 \nabla f(x_k)^T p_k$$

Круть. Помимо условий Вольфа есть также сильные условия Вольфа. Почему и зачем? Потому что у нас могут быть области, где градиент изменил знак, и которые подходят. Поэтому можно поменять второе условие так:

$$|\nabla f(x_k + \alpha p_k)^T p_k| \leqslant c_2 |\nabla f(x_k)^T p_k|$$

На практике условиями Вольфа пользуются не всегда по понятным причинам: надо считать градиент, а при большом количестве параметров это дорого.
Так, хорошо, что нам эти условия дают? Давайте вот на что посмотрим. Пусть

$$\cos \phi_k = \frac{-\nabla f(x_k)^T p_k}{\|\nabla f(x_k)\| \|p_k\|} \qquad x_{k+1} = x_k + \alpha_k p_k$$

Пусть для α_k выполняются условия Вольфа, пусть p_k — направление убывания и пусть функция ограничена, непрерывно дифференцируема. Также нам потребуется дополнительное условие — градиент должен принадлежать классу Липшица. Тогда докажем, что

$$\sum_{k \geqslant 0} \cos^2 \phi_k \|\nabla f(x_k)\|^2 \text{ сходится}$$

Итак, доказательство. Мы знаем, что

$$\nabla f(x_k + \alpha p_k)^T p_k \geqslant c_2 \nabla f(x_k)^T p_k$$

Вычтем из обеих частей $\nabla f(x_k)^T p_k$:

$$(\nabla f(x_{k+1}) - \nabla f(x_k))^T p_k \geqslant (c_2 - 1) \nabla f(x_k)^T p_k$$

Также из липшицевости следует

$$(\nabla f(x_{k+1}) - \nabla f(x_k))^T p_k \leqslant \alpha_k L p_k^T p_k = \alpha_k L \|p_k\|^2$$

Из последних двух утверждений

$$(c_2 - 1) \nabla f(x_k)^T p_k \leqslant \alpha_k L \|p_k\| \Leftrightarrow \alpha_k \geqslant \frac{c_2 - 1}{L} \frac{\nabla f(x_k)^T p_k}{\|p_k\|^2}$$

Применив первое условие Вольфа, получим

$$f(x_{k+1}) \leqslant f(x_k) - \underbrace{c_1 \frac{1 - c_2}{L}}_{c} \frac{(\nabla f(x_k)^T p_k)^2}{\|p_k\|^2}$$

Давайте теперь вспомним, как мы определили косинус и подставим всё в формулу

$$f(x_{k+1}) \leqslant f(x_k) - c\cos^2\phi_k\|\nabla f(x_k)\|^2$$

Хм-м-м, кажется это значит, что $f(x_{k+1}) \leqslant f(x_0) - c\sum\limits_{i=0}^{k}\cos^2\phi_i\|\nabla f(x_i)\|^2$. Поскольку наша функция ограничена снизу, мы получаем, что всё круто.
Что отсюда следует? А то, что если взять метод градиентного спуска, то $\cos^2\phi_k = 1$, а значит $\|\nabla f(x_k)\|$ сходится к нулю. Это не значит, что x_k стремится к именно минимуму, но уж к какой-то стационарной точке мы точно сойдёмся.
Есть также некоторые другие варианты хороших условий, которые дают нам сходимость. А также условия Вольфа не всегда выгодно проверять. Или можно применять только одно из двух условий.
Для конкретно градиентного спуска есть ещё одна более интересная теорема. Пусть $f \in C^{(2)}$. Пусть градиентный спуск сходится к x^* и $\nabla^2 f(x^*) > 0$. Тогда для любой

$r \in \left(\dfrac{\lambda_n - \lambda_1}{\lambda_n + \lambda_1}; 1\right)$ λ_1, λ_n — соотвественно миимальное и максимальное собственные числа гессиана

$f(x_k) - f(x^*) \leqslant r^2|f(x_k) - f(x^*)|$. Без доказательства.

Офф-топ про численное дифференцирование. Мы знаем, что $\frac{f(x+h)-f(x)}{h}$ при устремлении h к нулю стремится к f'. Этим можно воспользоваться для того, чтобы примерно посчитать производную (такая формула называется «правой разностью»). Можно подоказывать примерные оценки точности этой формулы, мы сейчас этого делать не будем. Мы зададимся вопросом, какое взять h? Казалось бы, чем меньше, тем лучше, но нет, потому что у нас числа с плавающей точкой обладают конечной точностью. На практике берут числа в диапазоне $(10^{-6}; 10^{-3})$, но для особых функций бывает полезно брать и другие. Ровно как и другие формулы можно брать, например левую разность $\frac{f(x)-f(x-h)}{h}$ или центральную разность $\frac{f(x+h)-f(x-h)}{2h}$. Центральная разность имеет более высокий порядок точности, но требует больше вычислений функции.
Но можно и дифференцировать не численно. В ML, например, его не используют из-за большого количества вычислений (на большом количестве параметров считать неприятно). На маленьком же количестве параметров численно дифференцировать норм.

Методы, родственные градиентному спуску. Градиентный спуск — один из самых простых и базовых методов, и преимуществ у него немного. В какую сторону можно пойти, чтобы попытаться его улучшить? Есть два ~~стула~~ пути: можно либо пытаться ускорить сходимость, либо ускорить каждую итерацию.

Стохастический градиентный спуск. Это метод, пошедший по второму пути, он используется в нейросетях. Прежде чем рассмотреть сам метод, рассмотрим класс функций, для которых его вообще используют. Они вот такие:

$$f(W) = \sum_i f_i(W) \qquad f_i \text{ — какие-то функции, которые вылезли из задачи}$$

Для машинного обучения у нас есть какие-то обучающие образцы, а минимизировать мы хотим ошибку на каждом образце. И вот обычно f_i — это ошибка на каждом образце.
Стандартный пример — линейная регрессия: у нас есть точки (например, на плоскости), и мы хотим найти прямую, которая наиболее близко через них проходит. Под «близко» имеется ввиду маленькая сумма квадратов разностей между абсциссой точки и абсциссы точки на прямой с той же ординатой:

$$\operatorname*{argmin}_{a,b}\sum\sum_i(ax_i + b - y_i)^2$$

В более общем случае у нас может быть не прямая, а полином.

$$r(W) = W_0 + W_1x + W_2x^2 + \cdots + W_nx^n$$

А вообще слагаемые могут быть какие-то произвольные, не обязательно такие, какие указано тут. Но вообще $r(W) = W_0 t_0(x) + \cdots + W_n t_n(x)$ можно также представить в векторном виде $r(W) = t^T(x)W$. У задачи линейной регрессии, вообще говоря, есть аналитическое решение:

$$f(W) = \sum_i (t^T(x_i)W - y_i)^2 = \|TW - y\|^2 \qquad W = (T^T T)^{-1} T^T y$$

Зачем нам методы оптимизации, если есть аналитическое решение? Затем что перемножать две здоровые матрицы, а затем искать обратную — дело небыстрое. Да и ещё один момент — если мы немного модифицируем задачу, аналитическое решение может сразу пропасть.
Ну, хорошо, а как это решать? Понятно, можно градиентным спуском, но мы сейчас не об этом. Сейчас о стохастическом градиентном спуске. Как выглядит обычный спуск?

$$W_{i+1} = W_i - \alpha \nabla f(W_i)$$

А что делаем в стохастическом спуске? Пусть $g_i = \nabla f_j(W_i)$, где j мы выбираем случайным образом. Тогда делаем

$$W_{i+1} = W_i - \alpha \nabla f_j(W_i)$$

Понятно, что мы ускорили работу нашего метода. Но теперь мы идём не обязательно в направлении минимума, а чёрт пойми куда. Эмпирическим путём выяснено, что в за много шагов стохастический градиентный спуск в среднем идёт в сторону минимума. И эффективность в итоге получается выше, чем у обычного градиентного спуска.
Есть важное обобщение данного метода, которое тоже называют стохастическим градиентным спуском, хотя это не совсем верно. Это mini-batch вариант. Давайте считать не одно f_j, но и не все. А какое-то (обычно, фиксированное) количество k разных компонент. Ещё для общего развития следует знать такой термин как «эпоха» — время, когда мы проходимся по всем компонентам. В такой вариации наши прыжки стохастического спуска будут менее хаотическими. Обычно размер batch'а выбирают так, чтобы хорошо задействовать векторизированные инструкции.
Теперь давайте вот на что посмотрим. Стохастический градиентный спуск вряд ли приведёт нас к самому минимуму. Он может привести нас близко, но вряд ли очень быстро. С жтим можно сделать следующее:

1. Изменить learning rate. По мере сходимости с минимуму имеет смысл уменьшать скорость обучения, чтобы наши «прыжки» не были слишком большими. Как конкретно его уменьшать, мы сейчас говорить не будем.

2. Экспоненциальное скользящее среднее. Пусть у нас есть последовательность g_i. Запишем новую последовательность v_i, заданную рекуррентным соотношением:

   $$v_{i+1} = \gamma v_i + (1 - \gamma) g_i \qquad \gamma \in (0; 1)$$

   Если у нас была очень осциллирующая последовательность, то в зависимости от γ мы получим менее осциллирующую последовательность. Но тут есть проблема: у нас получается некое смещение за счёт того, что $v_0 = 0$.
   Поэтому сюда ещё нужно вставить коррекцию этого смещения:

   $$\widetilde{v}_i = \frac{v_i}{1 - \gamma^{i-1}}$$

   Теперь как это применять это в стохастическом градиентном спуске? Ну, понятно, как (и, самое главное, когда). Когда у нас функция — узкий жёлоб, там градиент осциллирует, а значит его можно сгладить. Получится стохастический градиентный спуск с инерцией.

   $$v_{i+1} = \gamma v_i + (1 - \gamma) g_i \qquad g_i \text{ — } \nabla f_j \text{ или batch}$$

   $$w_{i+1} = w_i - \alpha v_{i+1}$$

3. Метод Нестерова. Эта штука сначала была создана для обычного градиентного спуска, но можно применять и для стохастического. План такой: давайте считать градиент не там, где мы находимся, а там, куда хотим шагнуть (точнее, в некотором её приближении):

$$v_{i+1} = \gamma v_i + (1-\gamma) g(w_i - \alpha\gamma v_i)$$

$$w_{i+1} = w_i - \alpha v_{i+1}$$

За счёт этого тоже неплохо ускоряется сходимость, причём не только у стохастического спуска, но и у обычного.

4. Адаптивный градиентный спуск.
Есть learning rate и есть формула $W_{i+1} = W_i - \alpha g_j$. Так может оказаться, что по некоторым компонентам у нас большие скачки, а по другим — маленькие. Это нам не нравится. Давайте введём матрицу

$$G = \sum_{j=1}^{i} g_j g_j^T$$

И поделим на неё наш шаг:

$$W_{i+1} = W_i - \frac{\alpha g_j}{\sqrt{G_{i,i}}}$$

В результате то направление, по которому мы делаем большие скачки, даёт нам очень быстро увеличивающуюся сумму G, а значит шаги уменьшаются. Идея норм, но на практике же есть проблемы.
Первое — у нас без нашей воли возник learning rate scheduling (шаги автоматически уменьшаются), и они могут делать это не с той скоростью, с которой нам надо.

5. RMS prop.
Тут происходит штука, схожая с градиентом и инерцией, но применительно к другим вещам.

$$s_{i+1} = \beta s_i + (1-\beta) g_i^2$$

Где g_i — покомпонентное возведение в квадрат.

$$W_{i+1} = W_i - \alpha \frac{g_i}{\sqrt{s_{i+1} + \varepsilon}}$$

ε — маленькое число порядка 10^{-7}, чтобы не было деления на ноль. Деление в формуле опять же покомпонентное.

6. Adaptive moment estimation.
По сути объединение нескольких описанных ранее методик.

$$v_{i+1} = \beta_1 v_i + (1-\beta_1) g_i$$

$$s_{i+1} = \beta_2 s_i + (1-\beta_2) g_i^2$$

$$\widetilde{v}_{i+1} = \frac{v_{i+1}}{1-\beta_1^{i+1}} \qquad \widetilde{s}_{i+1} = \frac{s_{i+1}}{1-\beta_2^{i+1}}$$

$$W_{i+1} = W_i - \alpha \frac{\widetilde{v}_{i+1}}{\sqrt{\widetilde{s}_{i+1} + \varepsilon}}$$

Один их самый популярных методов. β_1 имеет стандартное значение 0.9, β_2 — 0.999, ε — 10^{-8}.

Где про всё это можно почитать? Есть такая штука как PyTorch — один из самых популярных фреймворков для машинного обучения. И там в частности описаны методы, которые ускоряют сходимость стохастического градиентного спуска в разделе про оптимизаторы. Помимо этого там есть методики, как выбирать learning rate с различными планировщиками. Не все из них будут хорошо работать на каждой функции, но могут и работать.

Переобучение. Давайте рассмотрим пример. У нас есть некоторая неизвестная нам функция и мы хотим по точкам её приблизить. И если мы найдём приближение этих точек, это ещё не значит, что мы решили задачу. Это ещё не значит, что когда нам дадут ещё одну точку, наш результат правильно предскажет значение в ней. Это называется «обобщающей способностью модели». Мы хотим, чтобы она была большой.

Вот что выяснилось на практике. Точный поиск минимума — не совсем то, что нам нужно. Когда мы строим какую-то модель, нам интересует не только глубина минимума, но и ширина минимума (насколько изменится значение функции при изменении аргумента). Как правило, эта величина связана с обобщающей способностью модели. Почему? Если у нас очень узкий минимум, то новые данные (на которые мы пытаемся обобщить модели) могут превратить наш минимум в точку, которая с минимумом имеет ничего общего.

Именно поэтому градиентные спуски эффективны. Поскольку они идут шагами, они не замечают слишком узких минимумов. Чем выше learning rate, тем более узкие минимумы не замечаются. Но если мы используем тот же метод Adam, то он уже делает меньше скачков, за счёт чего он может уменьшить обобщающую способность. В итоге на практике могут использовать методы оптимизации попроще, могут переключаться между методами, но это, впрочем, к курсу имеет мало отношения, поскольку у нас тут методы оптимизации, а не машинное обучение в общем.