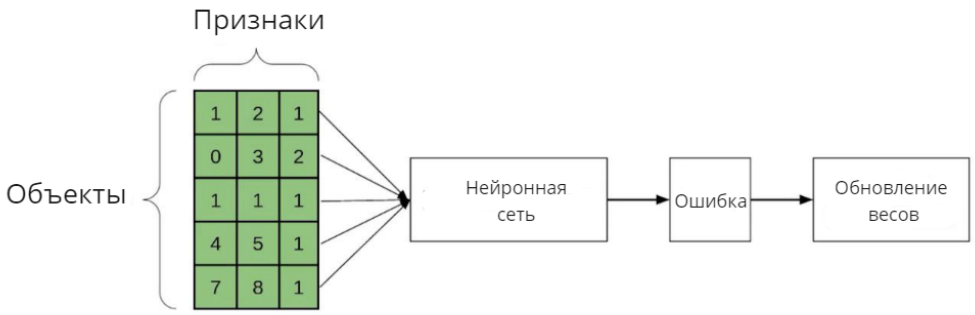
Градиентный спуск: применение и модификации

Batch Gradient Descent

Первая вариация градиентного спуска — **Batch Gradient Descent**. По-русски её называют **пакетным градиентным спуском**, или **ванильным градиентным спуском** (хотя англоязычную вариацию Vanilla Gradient Descent чаще не переводят). По сути, это и есть классический градиентный спуск, который мы рассматривали в предыдущем модуле.

Пакетным его называют по той причине, что он использует всю выборку (весь пакет) на каждом шаге, для того чтобы получить результат.

Слово **"vanilla"** в названии используют для того, чтобы указать, что это самый простой вариант, без «примесей». В английском языке такое название может применяться не только для градиентного спуска, но и для любого метода в целом. В русском языке такое название, как правило, чаще присутствует в дословных переводах англоязычных текстов.



Часто в различных источниках шаг Batch Gradient Descent записывается в следующих обозначениях:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

Примечание. На самом деле совершенно не важно, какими буквами выражать те или иные объекты, но мы будем использовать в этом модуле обозначения, которые часто встречаются в научных статьях и различной литературе.

Здесь θ — вектор с параметрами функции, η — шаг градиента, $\nabla_\theta J(\theta)$ — градиент функции, найденный по её параметрам.

Stochastic Gradient Descent

Представим, что мы реализуем градиентный спуск для набора данных объёмом 10 000 наблюдений и у нас десять переменных. Среднеквадратичную ошибку считаем по всем точкам, то есть для 10 000 наблюдений. Производную необходимо посчитать по каждому параметру, поэтому фактически за каждую итерацию мы будем выполнять не менее 100 000 вычислений. И если, допустим, у нас 1000 итераций, то нам нужно 100000*1000=100000000 вычислений. Это довольно много, поэтому градиентный спуск на сложных моделях и при использовании больших наборов данных работает крайне долго.

Чтобы преодолеть эту проблему, придумали **стохастический градиентный спуск**. Слово «стохастический» можно воспринимать как синоним слова «случайный». Где же при использовании градиентного спуска может возникнуть случайность? При выборе данных. При реализации стохастического спуска вычисляются градиенты не для всей выборки, а только для случайно выбранной единственной точки.

Это значительно сокращает вычислительные затраты.

В виде формулы это можно записать следующим образом:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i)}; y^{(i)}\right)$$

Mini-batch Gradient Descent

Третья вариация градиентного спуска — **Mini-batch Gradient Descent**. Также можно называть его **мини-пакетным градиентным спуском**. По сути, эта модификация сочетает в себе лучшее от классической реализации и стохастического варианта. На данный момент это наиболее популярная реализация градиентного спуска, которая используется в глубоком обучении (т. е. в обучении нейронных сетей).

В ходе обучения модели с помощью мини-пакетного градиентного спуска обучающая выборка разбивается на пакеты (**батчи**), для которых рассчитывается ошибка модели и пересчитываются веса.

То есть, с одной стороны, мы используем все преимущества обычного градиентного спуска, а с другой — уменьшаем сложность вычислений и повышаем их скорость по аналогии со стохастическим спуском. Кроме того, алгоритм работает ещё быстрее за счёт возможности применения векторизованных вычислений.

Формализовать это можно следующим образом:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right)$$

Batch Gradient Descent	Stochastic Gradient Descent	Mini-batch Gradient Descent
Рассматриваются все обучающие данные	Рассматривает только один объект	Рассматривается подвыборка

Затрачивает много времени на работу	Работает быстрее пакетного	Работает быстрее двух других
Плавное обновление параметров модели	Сильные колебания в обновлении параметров модели	Колебания зависят от размера подвыборки (увеличиваются с уменьшением её объёма)

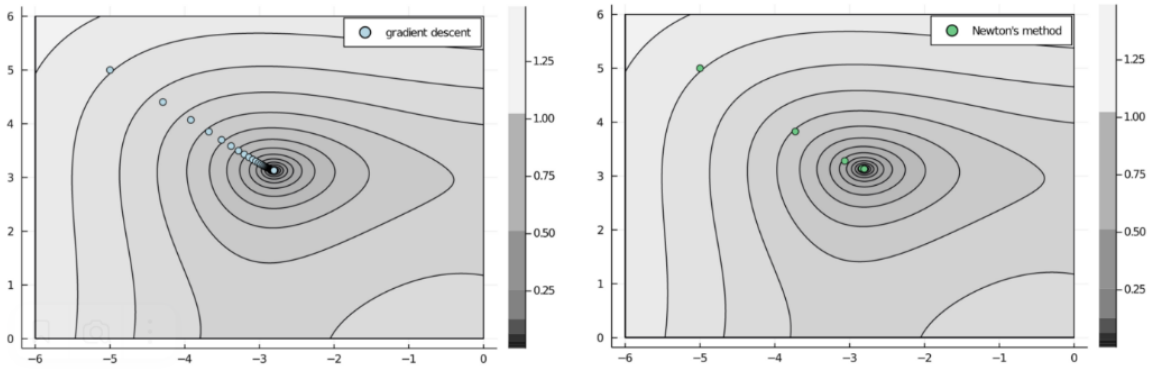
Метод Ньютона

Метод Ньютона используется во многих алгоритмах машинного обучения. Часто в литературе его сравнивают с градиентным спуском, так как два этих алгоритма очень популярны.

Для многомерного случая формула выглядит следующим образом:

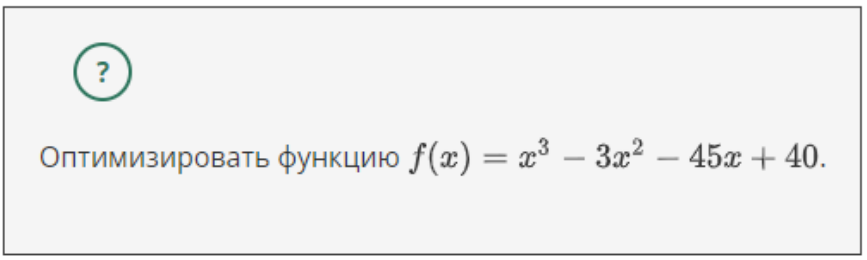
$$x^{(n+1)} = x^{(n)} - \left[Hf(x^{(n)})\right]^{-1} \nabla f(x^{(n)})$$

Можно заметить, что эта формула совпадает с формулой для градиентного спуска, но вместо умножения на learning rate (темп обучения) используется умножение на обратную матрицу к гессиану. Благодаря этому функция может сходиться за меньшее количество итераций, так как мы учитываем информацию о выпуклости функции через гессиан. Можно увидеть это на иллюстрации работы двух методов, где метод Ньютона явно сходится быстрее:



Метод Ньютона, если считать в количестве итераций, в многомерном случае (с гессианом) работает быстрее градиентного спуска.

Рассмотрим пример оптимизации функции с помощью метода Ньютона.



Оптимизировать функцию $f(x) = x^3 - 3x^2 - 45x + 40$.

Находим производную функции:

$$f' = 3x^2 - 6x - 45$$

Находим вторую производную:

$$f'' = 6x - 6$$

Теперь необходимо взять какую-нибудь изначальную точку. Например, пусть это будет точка $x = 42$. Также нам необходима точность — её возьмем равной 0.0001. На каждом шаге будем переходить в следующую точку по уже упомянутой выше формуле:

$$x^{(n+1)} = x^{(n)} - \frac{f'\left(x^{(n)}\right)}{f''\left(x^{(n)}\right)}$$

Например, в нашем случае следующая после 42 точка будет рассчитываться следующим образом:

$$x_2 = 42 - \frac{f'(x_1)}{f''(x_2)}$$

$$f'(x_1) = 3x^2 - 6x - 45\,|_{x_1=42} = 3 \cdot 42^2 - 6 \cdot 42 - 45 = 4995$$

$$f''(x_1) = 6x - 6\,|_{x_1=42} = 6.42 - 6 = 246$$

$$x_2 = 42 - \frac{4995}{246} \approx 42 - 20.305 = 21.695$$

Третья точка будет вычисляться по аналогичному принципу:

$$x_3 = 21.695 - \frac{f'(21.695)}{f''(21.695)}$$

Достоинства метода Ньютона:

✅ Если мы минимизируем квадратичную функцию, то с помощью метода Ньютона можно попасть в минимум целевой функции за один шаг.

✅ Также этот алгоритм сходится за один шаг, если в качестве минимизируемой функции выступает функция из класса поверхностей вращения (т. е. такая, у которой есть симметрия).

✅ Для несимметричной функции метод не может обеспечить сходимость, однако скорость сходимости всё равно превышает скорость методов, основанных на градиентном спуске.

Недостатки метода Ньютона:

⛔ Этот метод очень чувствителен к изначальным условиям.

При использовании градиентного спуска мы всегда гарантированно движемся по антиградиенту в сторону минимума. В методе Ньютона происходит подгонка параболоида к локальной кривизне, и затем алгоритм движется к неподвижной точке данного параболоида. Из-за этого мы можем попасть в максимум или седловую точку. Особенно ярко это видно на невыпуклых функциях с большим количеством переменных, так как у таких функций седловые точки встречаются намного чаще экстремумов.

Поэтому здесь необходимо обозначить ограничение: метод Ньютона стоит применять только для задач, в которых целевая функция выпуклая.

Впрочем, это не является проблемой. В линейной регрессии или при решении задачи классификации с помощью метода опорных векторов или логистической регрессии мы как раз ищем минимум у выпуклой целевой функции, то есть данный алгоритм подходит нам во многих случаях.

⛔ Также метод Ньютона может быть затратным с точки зрения вычислительной сложности, так как требует вычисления не только градиента, но и гессиана и обратного гессиана (при делении на матрицу необходимо искать обратную матрицу).

Если у задачи много параметров, то расходы на память и время вычислений становятся астрономическими. Например, при наличии 50 параметров нужно вычислять более 1000 значений на каждом шаге, а затем предстоит ещё более 500 операций нахождения обратной матрицы. Однако метод всё равно используют, так как выгода от быстрой сходимости перевешивает затраты на вычисления.

Квазиньютоновские методы

В **квазиньютоновских методах** вместо вычисления гессиана мы просто аппроксимируем его матрицей, которая обновляется от итерации к итерации с использованием информации, вычисленной на предыдущих шагах. Так как вместо вычисления большого количества новых величин мы использует найденные ранее значения, квазиньютоновский алгоритм тратит гораздо меньше времени и вычислительных ресурсов.

Формально это описывается следующим образом:

$$x_{k+1} = x_k - H_k \nabla f(x_k)$$

В данном случае вместо обратного гессиана появляется матрица H_k, которая строится таким образом, чтобы максимально точно аппроксимировать настоящий обратный гессиан.

Математически это записывается так:

$$H_k - \left[\nabla^2 f(x_k)\right]^{-1} \to 0 \text{ при } k \to \infty$$

Здесь имеется в виду, что разница между матрицей в квазиньютоновском методе и обратным гессианом стремится к нулю.

Эта матрица обновляется на каждом шаге, и для этого существуют разные способы. Для каждого из способов есть своя модификация квазиньютоновского метода. Эти способы объединены ограничением: процесс обновления матрицы должен быть достаточно эффективным и не должен требовать вычислений гессиана. То есть, по сути, на каждом шаге мы должны получать информацию о гессиане, не находя непосредственно сам гессиан.

Если вас интересует математическая сторона обновления и аппроксимации матрицы, прочитайте эту статью. В силу того, что понимание этой части метода требует очень серьёзной математической подготовки, мы опустим её. Однако можем заверить вас, что для успешного использования алгоритма и его понимания знание всех математических выводов не требуется.

Три самые популярные схемы аппроксимации:

- симметричная коррекция ранга 1 (SR1);
- схема Дэвидона — Флетчера — Пауэлла (DFP);
- схема Бройдена — Флетчера — Гольдфарба — Шанно (BFGS).

Последняя схема (**BFGS**) самая известная, стабильная и считается наиболее эффективной. На ней мы и остановимся. Своё название она получила из первых букв фамилий создателей и исследователей данной схемы: Чарли Джорджа Бройдена, Роджера Флетчера, Дональда Гольдфарба и Дэвида Шанно.

У этой схемы есть две известных вариации:

- L-BFGS;
- L-BFGS-B.

Обе этих вариации необходимы в случае большого количества переменных для экономии памяти (так как во время их реализации хранится ограниченное количество информации). По сути, они работают одинаково, и L-BFGS-B является лишь улучшенной версией L-BFGS для работы с ограничениями.

Метод BFGS очень устойчив и на данный момент считается одним из наиболее эффективных. Поэтому, если, например, применить функцию `optimize` без указания метода в библиотеке SciPy, то по умолчанию будет использоваться именно BFGS либо одна из его модификаций, указанных выше. Также данный метод используется в библиотеке sklearn при решении задачи логистической регрессии.

Линейное программирование

Задача линейного программирования — это задача оптимизации, в которой целевая функция и функции-ограничения линейны, а все переменные неотрицательны.

Целочисленным линейным программированием (ЦЛП) называется вариация задачи линейного программирования, когда все переменные — целые числа.

?

Пример № 1

Фабрика игрушек производит игрушки-антистресс и игрушки-вертушки.

Для изготовления игрушки-антистресс необходимо потратить 2 доллара и 3 часа, для изготовления игрушки-вертушки — 4 доллара и 2 часа.

На этой неделе в бюджете у фабрики есть 220 долларов, и оплачено 150 трудочасов для производства указанных игрушек.

Если одну игрушку-антистресс можно продать за 6 долларов, а игрушку-вертушку — за 7, то сколько экземпляров каждого товара необходимо произвести на этой неделе, чтобы максимизировать прибыль?

Такая задача идеально подходит для использования методов линейного программирования по следующим причинам:

- все условия являются линейными;
- значения переменных каким-то образом ограничены;
- цель состоит в том, чтобы найти значения переменных, которые максимизируют некоторую величину.

Обратите внимание, что производство каждой детали связано с затратами, как временными, так и финансовыми. Изготовление каждой игрушки-антистресс стоит 2 доллара, а изготовление каждой вертушки — 4 доллара. У фабрики есть только 220 долларов, чтобы потратить их на производство изделий. Отсюда возникает **ограничение на возможное количество изготовленных товаров**.

Обозначим за x количество произведённых игрушек-антистресс, за y — количество произведённых игрушек-вертушек Тогда это ограничение можно записать в виде следующего неравенства:

$$2x + 4y \leq 220$$

Также существует ограничение на то, сколько времени мы можем потратить на производство игрушек. На изготовление каждой игрушки-антистресс уходит 3 часа, а на изготовление каждой игрушки-вертушки — 2 часа. На этой неделе у фабрики есть только 150 рабочих часов, так что **производство ограничено по времени**. Это ограничение можно записать в виде неравенства:

$$3x + 2y \leq 150$$

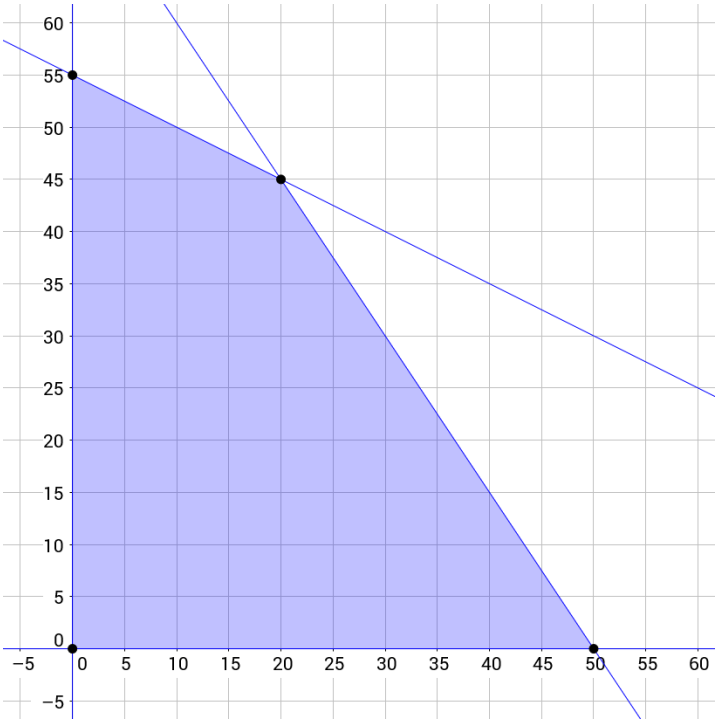
К этим ограничениям мы также можем добавить ограничения из соображений здравого смысла. Невозможно произвести отрицательное количество игрушек, поэтому необходимо обозначить следующие ограничения:

$$x \geq 0$$
$$y \geq 0$$

Итак, мы записали все необходимые ограничения. Они образуют систему неравенств:

$$\begin{cases} 2x + 4y \leq 220 \\ 3x + 2y \leq 150 \\ x \geq 0 \\ y \geq 0 \end{cases}$$

Если представить систему этих неравенств в графическом виде, получим многоугольник:



Закрашенная область является областью допустимых решений этой задачи. **Наша цель** — найти внутри этого многоугольника такую точку, которая даст наилучшее решение нашей задачи (максимизацию или минимизацию целевой функции).

Итак, мы поняли, что нам нужно найти максимальное значение прибыли (целевой функции) для точек внутри области допустимых значений, однако самой целевой функции у нас пока нет. Давайте составим её. На изготовление каждой игрушки-антистресс требуется 2 доллара, а продать её можно за 6. Получается, что чистая прибыль от продажи составляет 4 доллара. Чтобы изготовить игрушку-вертушку, мы потратим 4 доллара, а продадим её за 7. Значит, чистая прибыль для вертушки составляет 3 доллара. Исходя из этого, получаем целевую функцию для суммарной прибыли:

$$p(x, y) = 4x + 3y$$

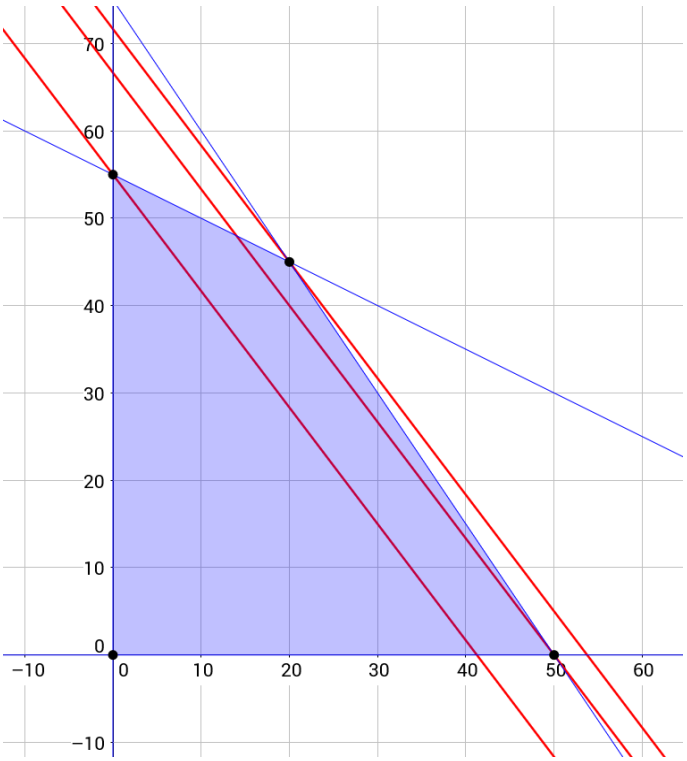
Нам необходимо найти максимально возможное значение этой функции с учётом того, что точка, в которой оно будет достигаться, должна удовлетворять условиям системы, которую мы написали ранее.

Для того чтобы решить задачу, выразим одну переменную через другую (так удобнее строить графики линейной функции в стандартной системе координат):

$$y = -\frac{4}{3}x + \frac{P}{3}$$

На графике ниже наша линия изображена красным цветом. Она может двигаться вверх и вниз в зависимости от значения P (вы можете видеть три прямых — это три разных положения одной и той же прямой).

Попробуем найти такую точку, для которой значение y будет наибольшим. Нас это интересует, так как мы хотим максимизировать значение P, а при максимально возможном P прямая поднимется настолько высоко, насколько это возможно, и пересечение с точкой ординат тоже будет находиться максимально высоко.



Линия, которая максимизирует точку пересечения с осью ординат, проходит через точку (20;45) — это точка пересечения первых двух ограничений. Все остальные прямые, которые проходят выше, не проходят через область допустимых решений. Все остальные «нижние» прямые проходят более чем

через одну точку в допустимой области и не максимизируют пересечение прямой с осью ординат, так как находятся ниже.

Таким образом, получаем, что завод должен произвести 20 игрушек-антистресс и 45 игрушек-вертушек. Это даст прибыль в размере 215 долларов:

$$20 * 4 + 45 * 3 = 215$$

Дополнительные методы оптимизации

Метод имитации отжига (simulated annealing)

Данный метод можно использовать, когда нужно оптимизировать достаточно сложную и «неудобную» функцию, то есть, такую, для которой не получится применить градиентные или другие оптимизационные методы. Метод имитации отжига удобен для применения, так как это метод нулевого порядка, а значит у него намного меньше ограничений.

Метод отжига используется также в ускорении обучения нейронных сетей.

Идея для алгоритма имитации отжига взята из реальной жизни. Во время отжига материал (обычно это металл) нагревают до определённой температуры, а затем медленно охлаждают. Когда материал горячий, у него снижается твёрдость и его проще обрабатывать. Когда он остывает, то становится более твёрдым и менее восприимчивым к изменениям.

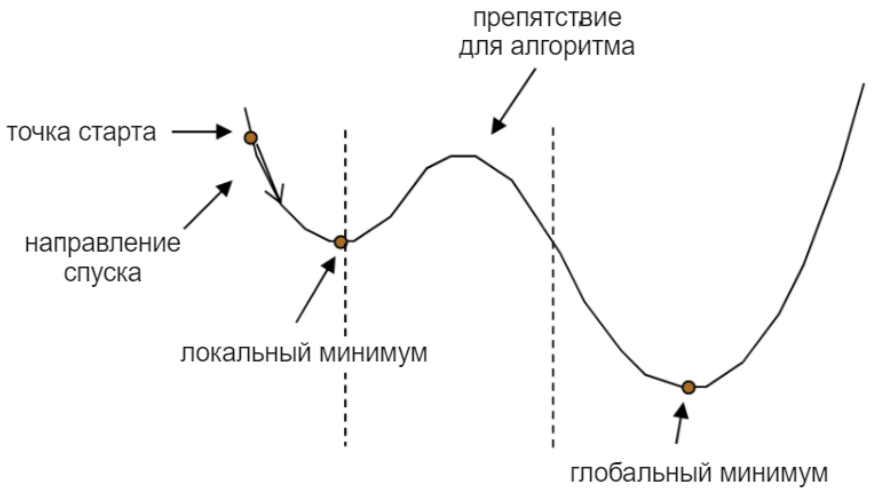
Наша задача — получить максимально холодный твёрдый материал. Для этого нужна крепкая кристаллическая решётка, в которой все атомы находятся на местах с минимальной энергией. Смысл в том, что, когда металл сильно нагревается, атомы его кристаллической решётки вынужденно покидают свои положения. Когда начинается охлаждение, они стремятся вернуться в состояние, где будет минимальный уровень определённой энергии. То есть нам было нужно, чтобы металл стал холодным и твёрдым, но мы сначала нагрели его (то есть «ухудшили» ситуацию относительно необходимой), чтобы в итоге (после охлаждения) получить намного более крепкую кристаллическую решётку.

Идея метода заключается в том, чтобы иногда позволять значению функции в точке «ухудшаться» (то есть удаляться от минимума) по аналогии с отжигом,

где мы «ухудшали» состояние металла (т. е. нагревали его), чтобы в итоге достичь наилучшего результата.

Зачем это нужно?

Представим, что мы ищем минимум для такой функции:



Метод отжига реализуется по следующему алгоритму:

1. Выбор изначальной точки (в терминах отжига её ещё называют изначальной температурой). Обычно начальная точка выбирается случайным образом.
2. Оценка решения. Оцениваем значение нашей функции (уровень «энергии»).
3. Основой алгоритм:
    a. Случайно изменяем точку.
    b. Оцениваем, переходим ли в эту новую точку.
4. Уменьшение температуры и, если температура больше некоторого порога, возвращение к основному алгоритму. Температура регулируется так, чтобы с каждым её уменьшением вероятность выбора менее оптимального решения также уменьшалась.

Математически и в более строгом виде это можно записать следующим образом:

1. Выбираем начальную точку $x(0)$ и определяем счётчик итераций k.
2. На каждом этапе случайно выбираем x^* из окрестности $x(k)$.
3. Если мы видим, что значение функции в новой точке отличается от текущего значения на величину, меньше заданной нами, то переходим в эту точку: $(x^*) - f\left(x^{(k)}\right) < c_k : x^{(k+1)} = x^*; k = k + 1$.
4. Повторяем шаги 2 и 3 до тех пор, как алгоритм не сойдётся.

За c_k здесь обозначен параметр, определяющий, на сколько значение функции может ухудшаться.

Если мы применяем этот метод, то нам предстоит ответить на два вопроса:

- ➔ Как выбирать изначальную точку x^*?
- ➔ Как выбирать параметр c_k?

Как правило, оба значения выбираются случайно. **Главное правило:** c_k должен сходиться к нулю.

У метода отжига есть ряд плюсов и минусов.

✅ Не использует градиент. Это значит, что его можно использовать для функций, которые не являются непрерывно дифференцируемыми.

✅ Можно использовать даже для дискретных функций. Причём именно для дискретных функций метод подходит очень хорошо.

✅ Прост в реализации и использовании.

⛔ Сложно настраивать под задачу из-за множества параметров.

⛔ Нет гарантий сходимости.

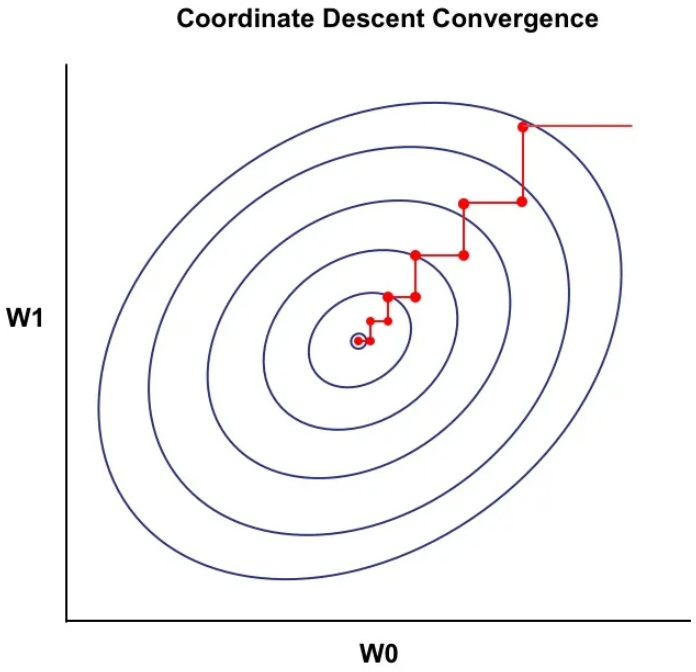
⛔ Выполнение может занимать много времени.

Метод координатного спуска

Метод координатного спуска можно считать одним из простейших методов оптимизации , который в целом достаточно эффективно ищет локальные минимумы для гладких функций.

При поиске минимума с помощью этого метода мы всегда изменяем положение точки в направлении осей координат, т. е. всегда изменяется только одна координата, и благодаря этому задача становится одномерной.

Если визуализировать работу алгоритма, то мы увидим, что за счёт того, что каждый шаг происходит параллельно одной или другой координатной оси, ход алгоритма становится похож на «лесенку»:



По сути, мы просто выбираем некоторую точку и правило изменения координаты и движемся в соответствии с ним до локального минимума.

Посмотрим на **основные отличия координатного и градиентного спусков**:

Координатный спуск	Градиентный спуск
При минимизации меняет одну	Меняет значения сразу всех

координату, фиксируя другие.	координат.
Используется для сильно выпуклой функции. Применимо для функций, у которых нет решений в замкнутой форме (например, Lasso-регрессия).	Применяется для функций, у которых есть решение в замкнутой форме (например, метод наименьших квадратов).
Не требует настройки гиперпараметра, определяющего темп обучения.	Требует настройки гиперпараметра, определяющего темп обучения.