Introduction to Gradient Descent Methods

by 王俊皓(108021121)

I. Introduction

Gradient Descent methods have been an optimization since the introduction of multi-dimensional calculus. It has been outshone by various methods due to the lack of efficiency in many fields, but as a band-aid method, especially in high dimensional problems, Gradient Descent is one of the most reassuring and consistent methods. In recent years, we often see such algorithms appear in machine learning and other optimization problems.

This article is to introduce some basic Gradient Descent methods and their ideas.

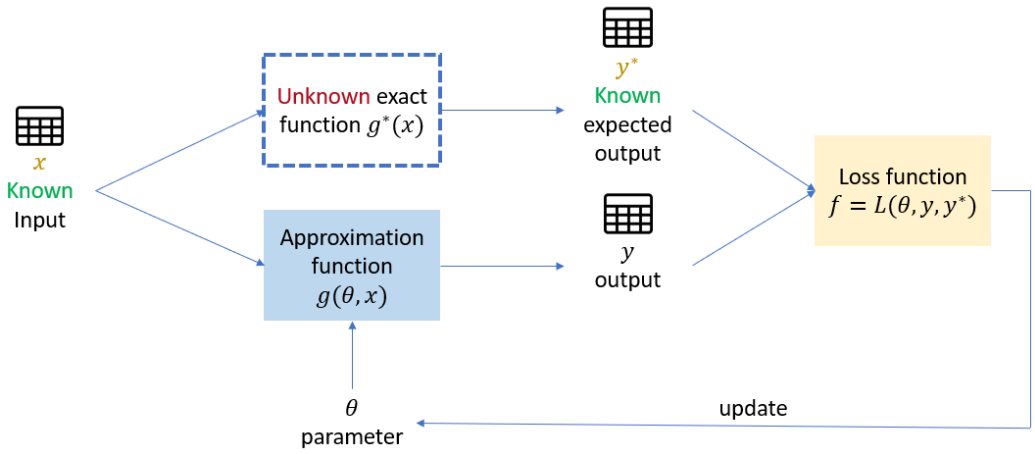
A. The Problem

The problem Gradient Descent methods want to solve is nothing more than an extremum finder. That is to state formally,

$$Find\ \theta\ such\ that\ function\ f(\theta)\ has\ extremum.$$

Although, it can be tricky sometimes to identify what is the tweakable variable and what is the function to find extremum.

Take machine learning (as this article's main example) for instance. The following graph is a simple model of a machine learning process.



We have the experimental data pairs (x, y^*) and we try to find the relation $g^*(x)$ between them. We approximate $g^*(x)$ with a parameter-driven function $g(\theta, x)$. To tell how close g is to g^*, we design a loss function $L(y, y^*)$, which can be thought of as some sort of norm function intuitionally. The goal is then to minimize the loss function L by manipulating θ, which y is a function of.

II. The Original Gradient Descent

The core idea of gradient descent is to utilize a crucial geometrical property of the gradient: it always points to the steepest ascending direction. The intuition then arises to keep going the opposite of the gradient direction for finding minimum. From this point on, we will keep the discussion around minimizing, since maximizing a function is essentially minimizing the negative function.

A. The Algorithm

Let Θ be the parameter space which contains all possible parameter configurations for the function g. We write $L(\theta)$ as a function of θ, not to be confused with other variables like y and y^* first. The goal is then to minimize $L(\theta)$, $\theta \in \Theta$.
Let $C(t)$ be a curve on Θ. We want C to be the ideal path of us tracking down the path using gradient to the minimum optimally and continuously. Formally speaking, we want C to satisfy:

$$\frac{d}{dt}C(t) = -\nabla_\theta L(C) \tag{1}$$

Note that the parameter space Θ is usually high dimensional.
Perform forward difference with step length η on the first term to obtain the original Gradient Descent iteration:

$$\frac{\theta^{(n+1)} - \theta^{(n)}}{\eta} = -\nabla_\theta L\left(\theta^{(n)}\right)$$

consequently

$$\theta^{(n+1)} = \theta^{(n)} - \eta\nabla_\theta L\left(\theta^{(n)}\right) \tag{2}$$

This is the simplest form of Gradient Descent. The intuition is clear: keep going down the negative gradient direction until hitting the minimum.
The step length η is often referred to as "learning rate" in machine learning.

B. Calculating Gradient

The calculation of the gradient term $\nabla_\theta L\left(\theta^{(n)}\right)$ isn't the easiest task, however. While for a general Gradient Descent usage, numerical approximation can be feasible, in machine learning or other more complicated loss function L, it needs several evaluations of L, which involves throwing the dataset into g repeatedly and can be tedious.
In the machine learning context, the most common method is to apply chain rule to the loss function:

$$\nabla_\theta L(C) = \nabla_\theta L(y(C), y^*) = \nabla_\theta y(C) \cdot \frac{\partial}{\partial y} L(y, y^*)$$

In more complicated cases like neuron networks, y can be further broken down to

multiple function compositions, in which consecutive chain rule is performed to simplify more.

C. Implementation Example

We now implement this algorithm with a simple linear regression problem.

Example.

Let

$$x_i = \frac{i}{1000}, i = 1, 2, \cdots 1000$$

so x is a partition of $[0,1]$ with even spacing 0.001.

Let

$$y_i = 2x_i + \frac{rand()}{5}$$

where $rand()$ returns a random real number in $[0,1]$.

We approximate this relation using a linear function:

$$y = g(\theta, x) = \theta_1 x + \theta_2$$

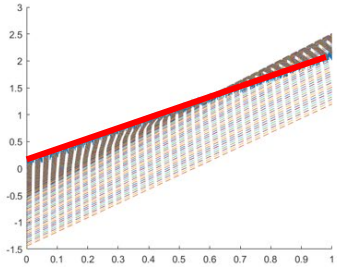
And we use the mean square error as our loss function:

$$L(y, y^*) = \frac{\sqrt{\sum_{i=1}^{n}(y_i - y_i{}^*)^2}}{n}$$

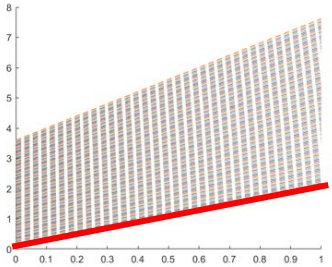
Then we have gradient

$$\nabla_\theta L = \frac{2\sum_{i=1}^{n}(y_i - y_i{}^*)}{2n\sqrt{\sum_{i=1}^{n}(y_i - y_i{}^*)^2}} \cdot \begin{bmatrix} \Sigma x_i \\ 1 \end{bmatrix} \tag{3}$$

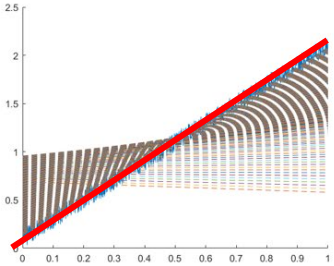
Now everything is set. We can see the results:



$\theta_0 = [2.6395 \quad -1.4435]$



$\theta_0 = [4.0132 \quad 3.6024]$

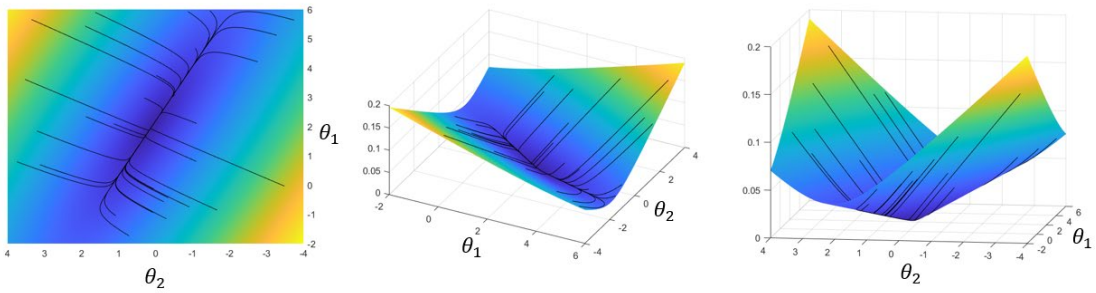


$\theta_0 = [-0.1092 \quad 0.689]$

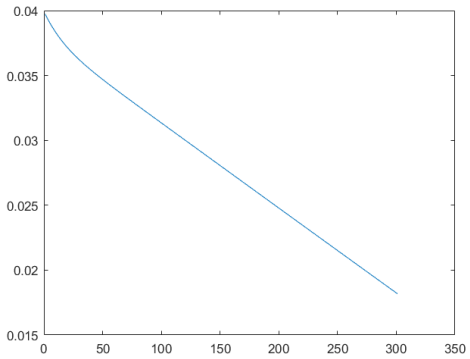
These three graphs show how the method converges from the looser part to denser part. The red line highlights the result.

We start from choosing a random parameter setup $\theta^{(0)}$ in $[-2,6] \times [-4,4]$.

We can also see from the parameter space Θ. Since we only have 2 variables, it can be plotted as a 3D plot, with z axis representing the loss.



Since this is a case that we already know the optimal value, by the help of probability and regression, it turns out to be a simple ravine-shaped graph.
We can see clearly that the flows move along the slope of the surface and towards the minimum as we expected.
For the convergence to the minimum:



The convergence seems to be a steady descent. We can see later that this is not quite a fast convergence.

III. Stochastic Gradient Descent

The first improvement to make is called stochastic gradient descent. This is a feature specifically for Gradient Descent in machine learning since it is done by tweaking the usage of the data.

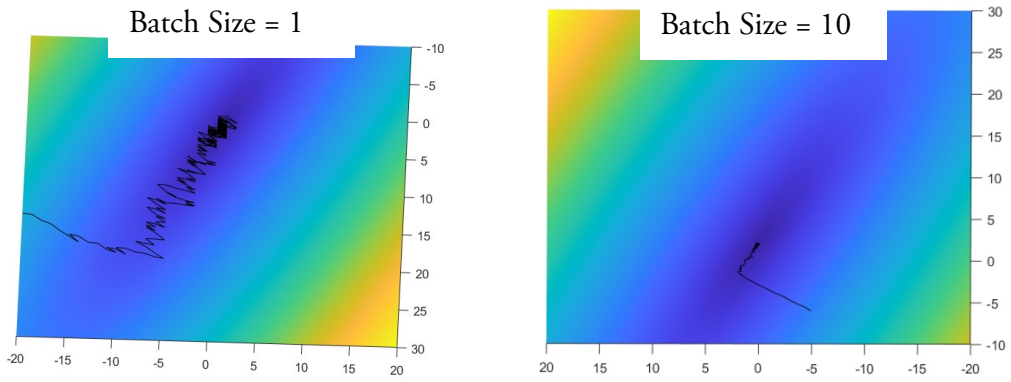
A. Batch Size

In the original Gradient Descent, which we should from now on called Batch Gradient Descent, we used the entire dataset to compute the gradient, as seen in (3). This is time consuming. We can instead choose to use a randomly selected amount of data for each step. This will result in the step we take being slightly off by a bit, but at the end of the day, if we reach the minimum, we don't care that much about staying on the exact path. Rather, we care more about how fast we can reach the minimum.
The random number of samples we take is called the "batch size". A method with a batch size of n, which is the entire data set, is called Batch Gradient Descent. A method with a

batch size of some integer between 2 and n is called Mini-Batch Gradient Descent. A method with a batch size of 1 is called Stochastic Gradient Descent, although this term can in general mean those with non-full batch.

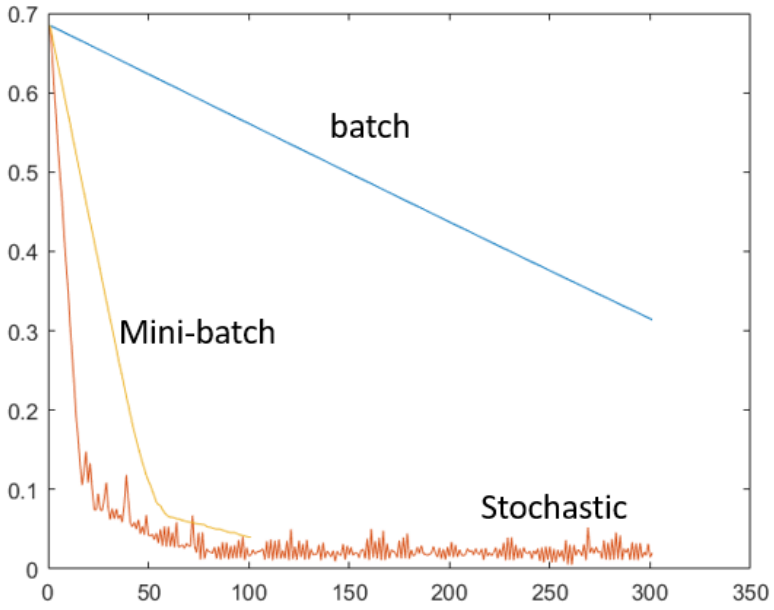
B. Implementation Example

For this example, we expanded the parameter space Θ to give a clearer view about the results.



By decreasing the batch size, the path becomes more oscillatory. This implies that the method might not settle down when reaching the minimum.

For the convergence:



Decreasing the batch size has a significant effect on the convergence speed at the start. The loss function rapidly decreases at the start but start being unstable and has oscillatory results afterward.

Often, the batch size will be set to around 32 for standard problems. A middle ground of fast enough descent and less oscillatory results.

IV.Momentum and Adaptive Methods

Momentum and adaptive methods are other features to add onto the existing algorithm. The improvements this article introduced works like some extension packs. One can choose to include the feature or not and they work independently.

A. Momentum

The intuition of momentum is based on the parameter space and the curve.
Imagine releasing a ball on an arbitrarily shaped hill. the ball starts still, but because it started on a slope, it starts rolling down. At any given moment, the acceleration of the ball would be pointing towards the steepest downward direction.

Momentum copied this intuition and put it on the parameter space. We now use another vector $v^{(n)}$ (in addition to $\theta^{(n)}$) to store the velocity (or in discretized view, "improvement") of the "ball" we are rolling. Continuously speaking, the path we are looking for is now:

$$\frac{d^2}{dt^2}C(t) = -\nabla_\theta L(C)$$

If we let $V(t)$ denote the negative velocity $-\frac{d}{dt}C(t)$, we have the system:

$$\begin{cases} \frac{d}{dt}V(t) = \nabla_\theta L(C) \\ \frac{d}{dt}C(t) = -V(t) \end{cases}$$

Applying forward difference with step length $\tilde{\eta}$,

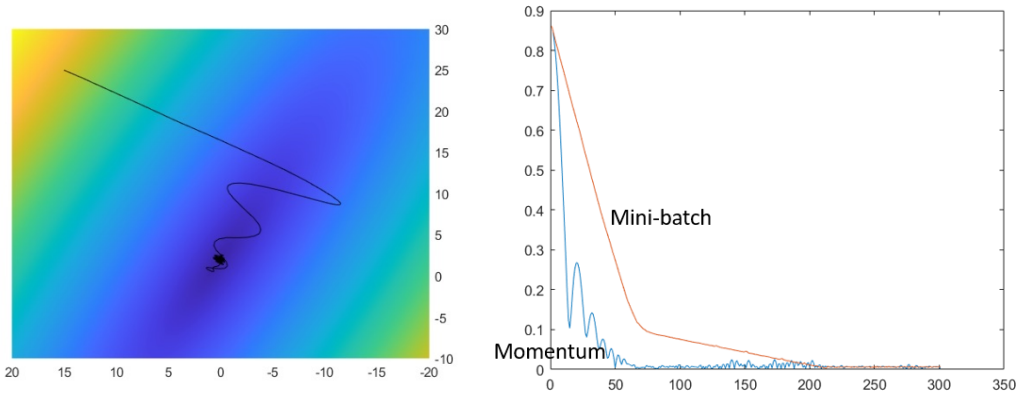
$$\begin{cases} v^{(n+1)*} = \tilde{\gamma}v^{(n)*} + \tilde{\eta}\nabla_\theta L\left(\theta^{(n)}\right) \\ \theta^{(n+1)} = \theta^{(n)} - \tilde{\eta}v^{(n+1)*} \end{cases}$$

With coefficient $\tilde{\gamma}$ act as a frictional multiplier. If there is no friction, the rolling ball will not stop and keep going up and down. Since the friction is proportional to the velocity, we simulate this by multiplying the term $v^{(n)*}$ in the first equation by $\tilde{\gamma}$, a number smaller than 1.
For v to represent the improvement on each iteration, we distribute γ into the first equation: (this part is to be consistent with the reference)

$$\begin{cases} v^{(n+1)} = \gamma v^{(n)} + \eta\nabla_\theta L\left(\theta^{(n)}\right) \\ \theta^{(n+1)} = \theta^{(n)} - v^{(n+1)} \end{cases} \tag{4}$$

where $\gamma = \tilde{\gamma}\tilde{\eta} < 1$, $\eta = \tilde{\eta}^2$ and $v^{(n)*} = \tilde{\eta}v^{(n)}$. Once again η is called the learning rate.
This is the iteration of using momentum to do Gradient Descent. The initial velocity is set to 0.

The γ is often called the "momentum", despite it works more like a frictional coefficient. Often this is set to a value around 0.9.
On the parameter space, it would act as if it were a ball rolling down a hill.



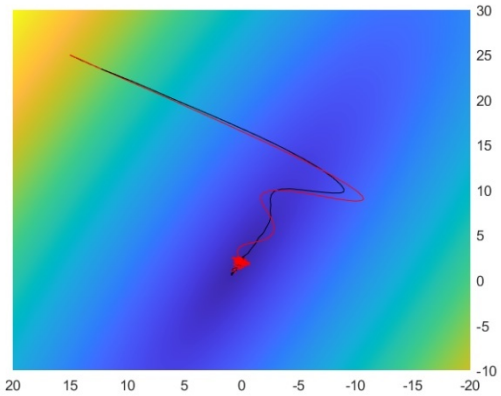
The biggest difference here is that it now overshoots instead of just steadily follow a smooth path. It may seem like it is taking the long path down from the graph. But if one graph the loss function, it becomes clear that the accumulated velocity does help descending faster.

B. Nesterov Accelerated Descent (NAD)

NAD is a modification of the momentum method. Instead of calculating the gradient on the current point, NAD tries to evaluate at the estimated next point.

$$\begin{cases} v^{(n+1)} = \gamma v^{(n)} + \eta \nabla_\theta L\left(\theta^{(n)} - \gamma v^{(n)}\right) \\ \theta^{(n+1)} = \theta^{(n)} - v^{(n+1)} \end{cases} \tag{5}$$

This evaluates the gradient one step prior to the current point, which can decrease the effect of going too high on the slope.



black is using NAD, red is not.

C. Adaptive Methods

Adaptive Methods are methods that dynamically set some of the parameters we introduced, primarily in this article, learning rate. Not only did η change with time, but

it also acts differently on every parameter.

1. Adaptive Gradient (Adagrad and Adadelta)

Parameter tuning is often an annoying yet important thing to do in performing Gradient Descent. Adaptive gradient method utilizes the fact that at the start of the iteration, since the initial value are expected to be nowhere near the minimum, we can risk taking bigger steps to start up. As time progresses, we then slow down to make it converge steadily.

A popular way of doing so is to add up past gradients. The bigger the last gradient, the bigger step taken on the last iteration, therefore the next step should be much smaller.

All the operations in the following iteration are all element-wise operations, for the ease of expression in notations. The learning rate η is now a vector of equal length with θ.

$$\begin{cases} \theta^{(n+1)} = \theta^{(n)} - \dfrac{\eta_0}{\sqrt{\delta^{(n)}}} \nabla_\theta L\left(\theta^{(n)}\right) \\ \delta^{(n+1)} = \delta^{(n)} + \left(\nabla_\theta L\left(\theta^{(n)}\right)\right)^2 \end{cases}$$

The iteration is called Adaptive Gradient or Adagrad.
This would rise a critical problem. If the quantity $\delta^{(n)}$ keeps increasing, the dynamic learning rate $\frac{\eta_0}{\sqrt{\delta^{(n)}}}$ would be too small for the method to have any progress.

We can propose not to sum all the gradients. The more recent ones should matter more and vice versa. This brings us to the Adadelta iteration.

$$\begin{cases} \theta^{(n+1)} = \theta^{(n)} - \dfrac{\eta_0}{\sqrt{\delta^{(n)}}} \nabla_\theta L\left(\theta^{(n)}\right) \\ \delta^{(n+1)} = \lambda\delta^{(n)} + (1-\lambda)\left(\nabla_\theta L\left(\theta^{(n)}\right)\right)^2 \end{cases} \tag{6}$$

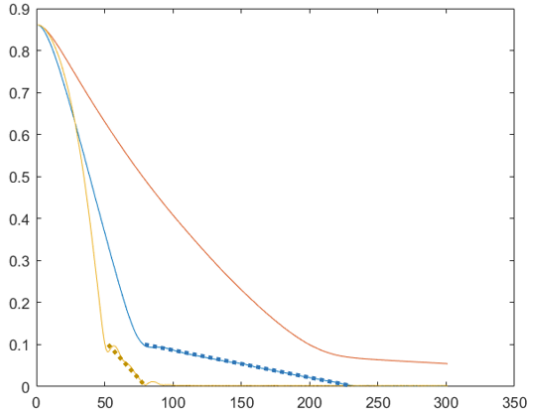
If we introduce a weighted average on $\delta^{(n)}$, the past terms would be exponentially less impactful, therefore not letting the iteration stop mid-way through.

2. Adaptive Moment (Adam)

Adam is the ultimate method of gradient descent in this article. It combines everything from all the previous features into one. It is still used by a lot of gradient descent algorithms to this date.

$$\begin{cases} v^{(n+1)} = \gamma v^{(n)} + \dfrac{\eta_0}{\sqrt{\delta^{(n)}}} \nabla_\theta L\left(\theta^{(n)} - \gamma v^{(n)}\right) \\ \theta^{(n+1)} = \theta^{(n)} - v^{(n+1)} \\ \delta^{(n+1)} = \lambda\delta^{(n)} + (1-\lambda)\left(\nabla_\theta L\left(\theta^{(n)} - \gamma v^{(n)}\right)\right)^2 \end{cases} \tag{7}$$

We can see momentum, NAD, and adaptive learning rate all in one iteration.



The convergence Adam > NAD > Adagrad

V. Conclusion

Gradient Descent algorithms are highly flexible with many modifications and parameter tunings. Adam being the most popular with all the modifications introduced in this article, is also much more efficient in sparse cases.

After the research and readings, there are still a lot of modifications to be done. One can propose higher order approximation to derive a multi-step update. Or for the momentum method, directly use the second difference formula for second derivative to derive a two-step method for a different momentum iteration.

VI.References

Main reference of this article: https://arxiv.org/pdf/1609.04747

Convergence of Gradient Descent: https://www.stat.cmu.edu/~ryantibs/convexopt-F13/scribes/lec6.pdf

Basic and intuitional Gradient Descent guide: https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e

Example source code (no addons needed): https://drive.google.com/drive/folders/1iTEp6xHtmpmsWXYWqIWuyoKYbiGPR6E8?usp=sharing