**What is Gradient Descent?**

Gradient Descent is an iterative process that finds the minima of a function. This is an optimisation algorithm that finds the parameters or coefficients of a function where the function has a minimum value. Although this function does not always guarantee to find a global minimum and can get stuck at a local minimum.



To understand the difference between local minima and global minima, take a look at the figure above. The global minimum is the least value of a function while a local minimum is the least value of a function in a certain neighbourhood.

To get an idea of how Gradient Descent works, let us take an example. Suppose you are at the top of a mountain and want to reach the base camp which is all the way down at the lowest point of the mountain. Also, due to the bad weather, the visibility is really low and you cannot see the path at all. How would you reach the base camp?

One of the ways is to use your feet to know where the land tends to descend. This will give an idea in what direction, the steep is low and you should take your first step. If you follow the descending path until you encounter a plain area or an ascending path, it is very likely you would reach the base camp.

But what if there is a slight rise in the ground when you are going downhill? You would immediately stop assuming that you reached the base camp (global minima), but in reality, you are still stuck at the mountain at global a local minima. At the end of this article, we ‘ll see how to solve this problem.

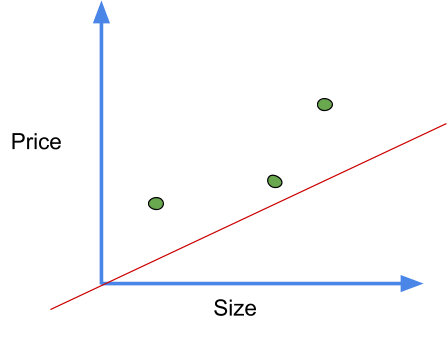
**Gradient Descent in Machine Learning**

Optimisation is an important part of [machine learning](https://www.mygreatlearning.com/blog/what-is-machine-learning/) and [deep learning](https://www.mygreatlearning.com/blog/what-is-deep-learning/). Almost every machine learning algorithm has an optimisation algorithm at its core that wants to minimize its cost function. When we fit a line with a [Linear Regression](https://www.mygreatlearning.com/blog/linear-regression-in-machine-learning/), we optimise the intercept and the slope. When we use [Logistic Regression](https://www.mygreatlearning.com/blog/logistic-regression-with-examples-in-python-and-r/) for classification, we optimise a squiggle and when we use the t-SNE algorithm we optimise clusters. Note that the working of Gradient Descent remains the same for all the above scenarios.

Now let us see in detail how gradient descent is used to optimise a linear regression problem. Take an example of a data-set where we are given prices of various houses depending upon their area. For simplicity, we ‘ll only consider a few examples from the dataset with the following price and area.

|  |  |
| --- | --- |
| Area (Acre sq) | Price(in millions) |
| 0.5 | 1.4 |
| 2.3 | 1.9 |
| 2.9 | 3.2 |

Here is a representation of this data on the graph. To fit the best fit line we have to optimise the slope of the line and the intercept of the line. For simplicity, we take a constant slope of 0.64, so that we can understand how gradient descent would optimise the intercept. In the next section, we implement gradient descent on the slope and intercept simultaneously.



First, we calculate the residual errors for each. Follow the below steps to calculate it

The gradient descent is provided with a random guess for the value of the intercept.

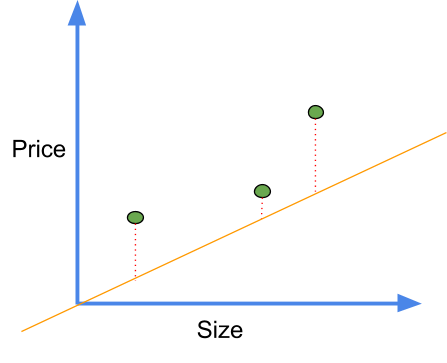
In our case, we take a random guess of zero, so the equation becomes

Predicted value = intercept + slope \* x ( If you are not familiar with this formula refer to Linear Regression)

The predicted values for the above can be calculated like this.

predicted value = 0 + 0.64 \* 0.5=0.32

The rest can be calculated in similar manner

Residual errors represented by red dotted lines

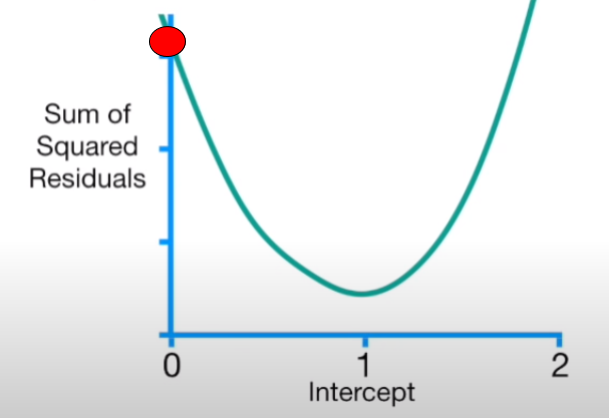
Next, we calculate the squared residual error for each point

Squared Residual error= (actual error - predicted)^2

For the first point, squared residual error = (1.4-0.32)^2 = (1.1)^2

Thus the sum of squared error = (1.1)^2 + (0.4)^2 + (1.3)^2 =3.1

Now we plot this point in a graph with the value of intercept as X-axis and value of a sum of squared error as Y-axis. In a similar manner, we plot points for many values of intercept. The plot represents the cost functions and looks like this.



The primary task of Gradient Descent is to find the minimum of this cost function. To find the minimum point, we find its derivatives with respect to intercept. So the equation of this cost function is given by

f(intercept) = (1.4-(intercept+ 0.64 \* 0.5))^2 +

(1.9-(intercept+0.64 \* 2.3))^2 +

(3.2-(intercept+0.64 \* 2.9))^2

The derivative of this function with respect to intercept is given by

Derivative= d/d(intercept)(1.4-(intercept+ 0.64 \* 0.5))^2

+ d/d(intercept) (1.9-(intercept+0.64 \* 2.3))^2

+ d/d(intercept)(3.2-(intercept+0.64 \* 2.9))^2

Applying chain rule, we find derivative of each term individually and add them up. Note that here slope is taken constant so its derivative is zero.

Derivative of (1.4-(intercept+0.64 \* 0.5))^2 = - 2 (1.4-(intercept+0.64 \* 0.5))

In a similar way we find derivatives of next two terms and the value we get is

Derivative= - 2 (1.4-(intercept+0.64 \* 0.5))+

-2 (1.9-(intercept+0.64 \* 2.3))+

-2 (3.2-(intercept+0.64 \* 2.9))

Let us put the value of intercept=0 to find the value of the next intercept

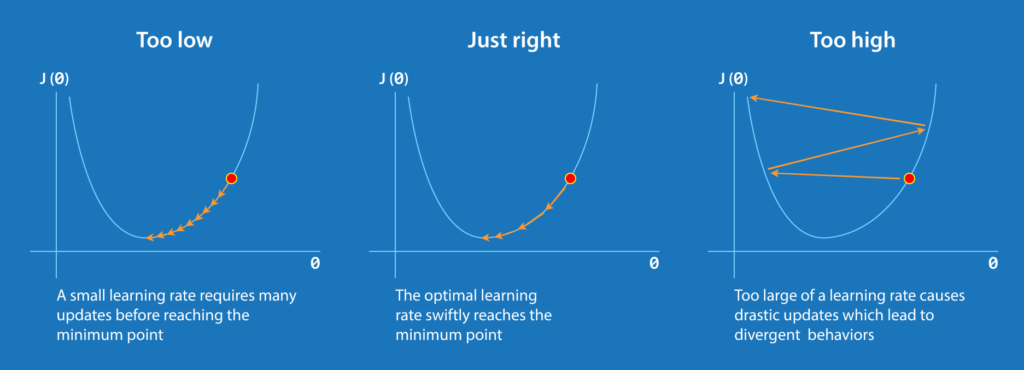
Derivative= - 2 (1.4-(0+0.64 \* 0.5))+

-2 (1.9-(0+0.64 \* 2.3))+

-2 (3.2-(0+0.64 \* 2.9))

= -5.7

Gradient descent subtracts the step size from the current value of intercept to get the new value of intercept. This step size is calculated by multiplying the derivative which is -5.7 here to a small number called the learning rate. Usually, we take the value of the learning rate to be 0.1, 0.01 or 0.001. The value of the step should not be too big as it can skip the minimum point and thus the optimisation can fail. It is a hyper-parameter and you need to experiment with its values.



In this case, let us take the learning rate 0.1, then the step size is equal to

Step size=-5.7\*0.1

New intercept = old intercept-step size

= 0-(-0.57)=0.57

Let us now put the new intercept in the derivative function

d sum of squared error /d(intercept)= -2 (1.4-(0.57+0.64 \* 0.5))+

-2 (1.9-(0.57+0.64 \* 2.3))+

-2 (3.2-(0.57+0.64 \* 2.9))

= -2.3

Now calculate the next step size

Step size=-2.3\*0.1

New intercept = old intercept-step size

= 0.57-(-0.23)=0.8

Again let us now put the new intercept in the derivative function

d sum of squared error /d(intercept)= - 2 (1.4-(0.8+0.64 \* 0.5))+

-2 (1.9-(0.8+0.64 \* 2.3))+

-2 (3.2-(0.8+0.64 \* 2.9))

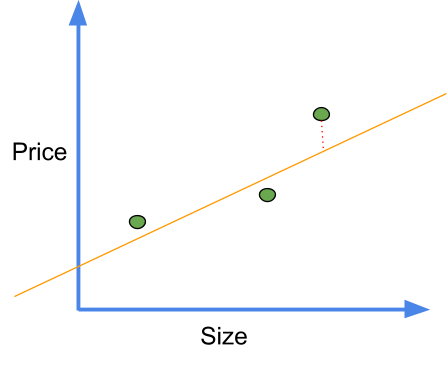
= -0.9

Step size= -0.9\*0.1

New intercept= old intercept-step size

= 0.8-(-0.09)=0.89

You might have noticed that the value of the step is high when the optimal solution is far away and this value is less as we approached an optimal solution. Thus we can say that gradient descent takes a bigger step when away from the solution and takes small steps when nearer to an optimal solution. This is the reason why gradient descent is efficient and fast.



Now as we can see the line with intercept 0.89 is a much better fit. But is this our optimal solution? No, we continue to find new intercept values until the value of step tends to zero(less than 0.001) or even in some cases we predefine the number of steps that are to be taken. In practice, this number can go to 1000 or even greater.

**Optimising Linear Regression**

Now let us come to the real problem and see how gradient descent optimises slope and intercept simultaneously. As before we take the derivatives but this time of this equation

f(intercept) = (1.4-(intercept+ slope \* 0.5))^2+

(1.9-(intercept+slope \* 2.3))^2+

(3.2-(intercept+slope \* 2.9))^2

Here we again use the chain rule, first as before we find the derivative of D with respect to intercept keeping slope as constant

Derivative w.r.t intercept = -2 (1.4-(intercept+slope \* 0.5))+

-2 (1.9-(intercept+slope \* 2.3))+

-2 (3.2-(intercept+slope \* 2.9))

Now we find derivative of D with respect to slope and consider intercept as constant

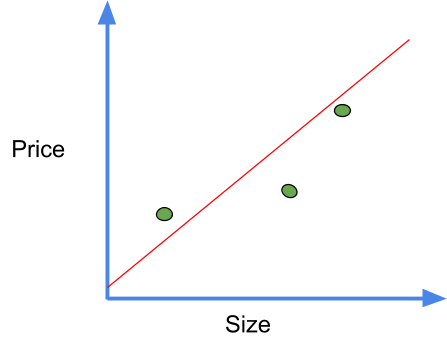
Derivative w.r.t slope= -2(0.5) (1.4-(intercept+slope \* 0.5))+

-2(2.3) (1.9-(intercept+slope \* 2.3))+

-2(2.9)(3.2-(intercept+slope \* 2.9))

When we have two or more derivatives of the same function, they are called gradients. We use these gradients to descend down the cost function. Thus the algorithm is called gradient descent. Note here the cost function we have been using so far is the sum of the square residuals.

As before we initialise intercept and slope randomly as zero and one. Now putting these values in the above gradients.

*the plot when intercept=0 and slope=1*

Derivative w.r.t intercept= -2 (1.4-(0+1 \* 0.5))+

-2 (1.9-(0+1 \* 2.3))+

-2 (3.2-(0+1 \* 2.9))

= -1.6

We take a different learning rate here

Step size= -1.6\*0.01=-0.016

New intercept=0-(-0.016)=0.016

d/d(slope)=- 2(0.5) (1.4-(0+1 \* 0.5))+

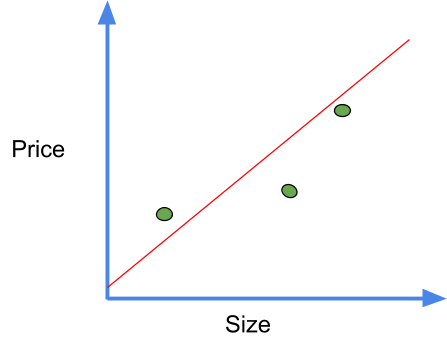
-2(2.3) (1.9-(0+1 \* 2.3))+

-2 (2.9)(3.2-(0+1 \* 2.9))

=-0.8

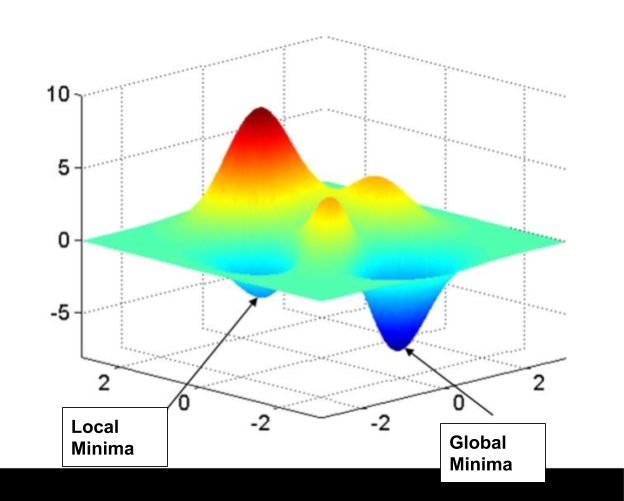
Step size= -0.8\*0.01=-0.008

New slope=1-(-0.008)=1.008



This is definitely a better fit than random initialisation. Repeating this process until we get step size near zero for both slope and intercept gives us an optimal solution and best fit line.

If we have more than one parameter, such as the number of rooms, the process remains the same but the number of derivatives increases. Also here we used the sum of squared residuals as loss function, but we can use any other loss function as well such as least squares.

*Loss function when we have more than one parameters*[(source)](https://www.researchgate.net/figure/A-3D-surface-plot-demonstrating-local-and-global-minima_fig10_305881922)

To briefly summarise the process, here are some points

1. Take the gradient of the loss function or in simpler words, take the derivative of the loss function for each parameter in it.
2. Randomly select the initialisation values.
3. Substitute these parameter values in the gradient
4. Calculate step size by using appropriate learning rate.
5. Calculate new parameters
6. Repeat from step 3 until an optimal solution is obtained.

**Variants of Gradient descent:**

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

**Stochastic Gradient Descent:**

Stochastic gradient descent (SGD) computes the gradient using a single sample. In this case, the noisier gradient calculated using the reduced number of samples tends SGD to perform frequent updates with a high variance. This causes the objective function to fluctuate heavily.

One benefit of SGD is that it’s computationally a whole lot faster. Large datasets often can’t be held in RAM, which makes vectorization much less efficient. Rather, each sample or batch of samples must be loaded, worked with, the results stored, and so on.

**Batch Gradient Descen**t:

In Batch Gradient Descent we consider all the examples for every step of Gradient Descent which means we compute derivatives of all the training examples to get a new parameter. Thus unlike SGD, we get a smoother objective function.

But if the number of training examples is large, then batch gradient descent is computationally very expensive. Hence if the number of training examples is large, then batch gradient descent is not preferred. Instead, we prefer to use stochastic gradient descent or mini-batch gradient descent which is discussed next.

**Mini Batch gradient descent:**

This is a type of gradient descent which works faster than both batch gradient descent and stochastic gradient descent. Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch.

Doing this helps us achieve the advantages of both the former variants we saw. Although Mini-batch requires the configuration of an additional “mini-batch size” hyperparameter for the learning algorithm.

This brings us to the end of this article where we have learned about working of Gradient Descent and its variants.