GRADIENT DESCENT

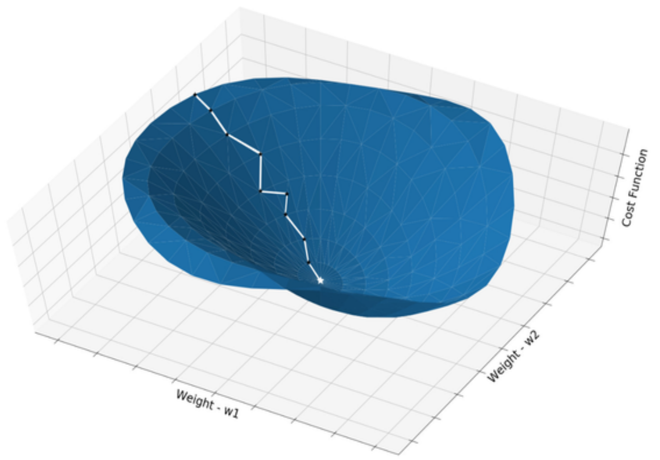
Gradient Descent is the most basic but most used optimization algorithm. It’s used heavily in linear regression and classification algorithms. It is an iterative optimisation algorithm to find the minimum of a function or find the coefficients for which the cost/loss function is minimum

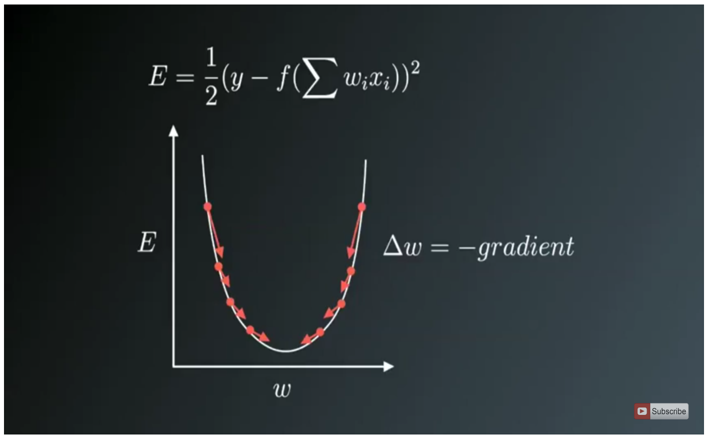
Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost). Parameters refer to coefficients in Linear Regression and weights in neural networks. Gradient Descent is a first-order optimization algorithm used to find local minima. This means it only takes into account the first derivative when performing the updates on the parameters.

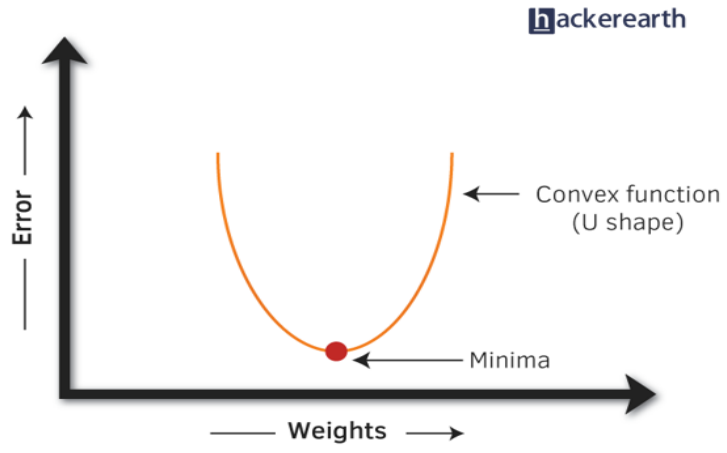
Gradient

A gradient is the direction and magnitude calculated during the training of a neural network it is used to teach the network weights in the right direction by the right amount.

A gradient measure how much the output of a function changes if you change the inputs a little bit. You can also think of a gradient as the slope of a function. The concept of slope is used to measure the rate at which changes are taking place. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. Said it more mathematically, a gradient is a partial derivative with respect to its inputs.







The above curve represents the equation of linear regression with the value of coefficients and bias.

Ex: y=4x + 3 might represent a point on the curve where loss function is huge.

y=3x + 2 might represent a point on the curve where loss function is reduced little bit.

Ex: y=2x+1 might represent the minima and loss function is minized using this coefficient and bias.

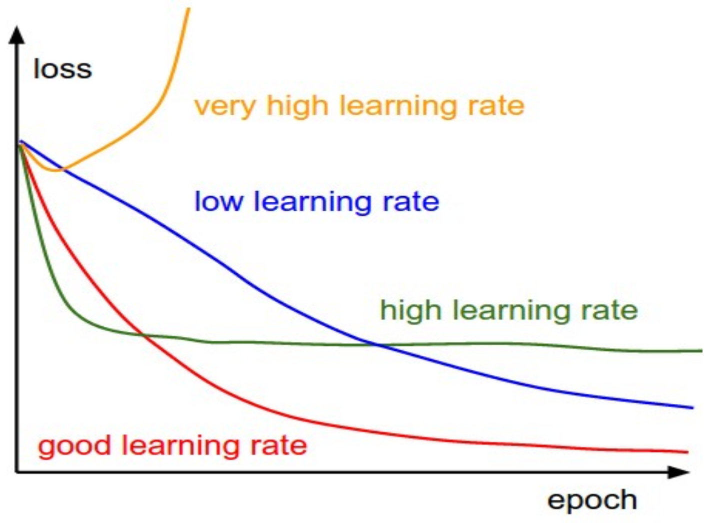
So as we come down the slope we take partial derivate of cost function with respect to all coefficients and bias which gives a partial derivative equation for all coefficients snd bias and then we put the values of the current coefficients and bias into the partial derivate equation for all coefficients and bias which gives a value to us for all coefficients and bias and then subtract these partial derivative values of all coefficients and bias from their current coefficients and bias and we get the updated coefficients and bias. We repeat it until we find the coefficients and bias for which cost function is minimum. Cost function cannot be reduced beyond minima and it starts increasing again and our coefficients and bias also start increasing with increase in cost function.

Minima is the representation of the equation of linear regression with the those values of coefficients and bias where loss function is minimum.

Learning rate

Learning rate determines how fast weights (in case of a neural network) or the coefficients (in case of linear regression or logistic regression) change. The learning rate is one of the most important hyper-parameters to tune for training deep neural networks. The amount that the weights are updated during training is referred to as the step size or the learning rate. A good learning rate could be the difference between a model that doesn't learn anything and a model that presents state-of-the-art results.

Learning rate determines how fast or slow we will move towards the optimal weights. A learning rate of 0.1, a traditionally common default value, would mean that weights in the network are updated 0.1 * (estimated weight error) or 10% of the estimated weight error each time the weights are updated.

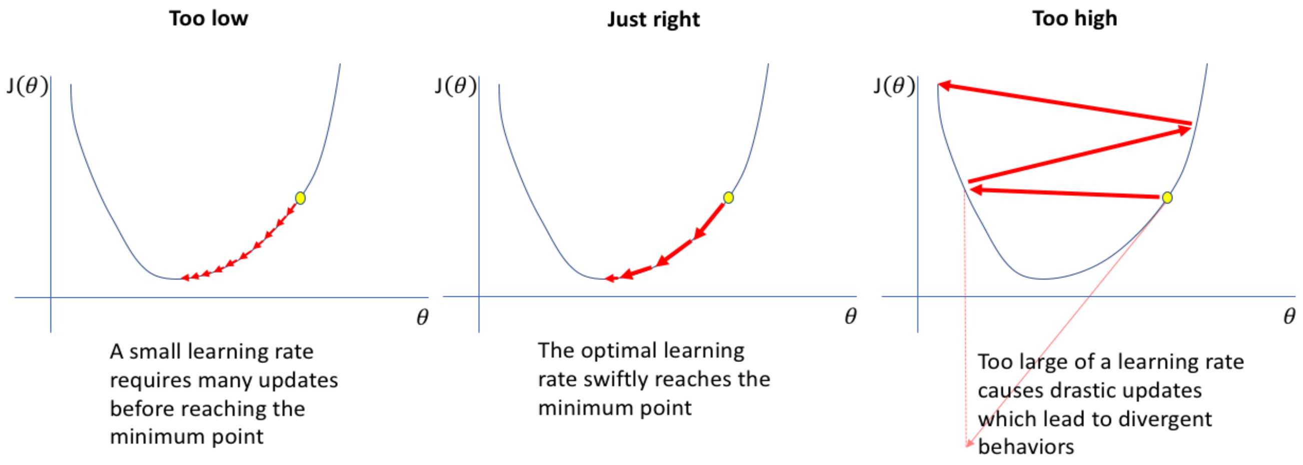


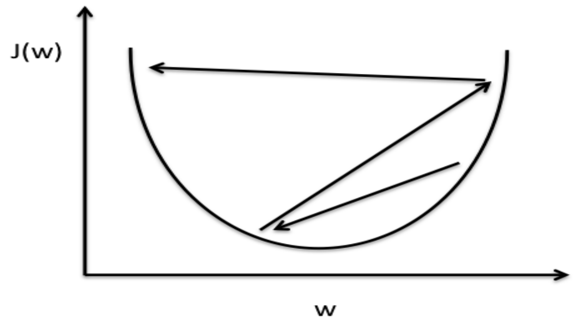
Importance of the Learning Rate

The lower the value, the slower we travel along the downward slope. If Learning rate is small, then smaller will be the steps and easy to converge. If you set the learning rate to a very small value, training will progress very slowly as you are making very tiny updates to the weights in your network and gradient descent will eventually reach the local minimum but it will maybe take too much time. If the learning rate is low, then training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny.

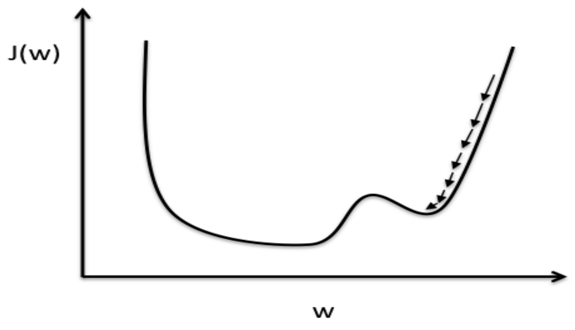
With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing.

If Learning rate is large, then larger will be the steps and model may not reach the local minimum because it just bounces back and forth between the convex function of gradient descent. If the learning rate is set too high, it can cause undesirable divergent behavior in your loss function and algorithm may not converge.


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors



Large learning rate: Overshooting.



Small learning rate: Many iterations until convergence and trapping in local minima.

How it works

To use gradient descent, we need to know the gradient of our cost function, the vector that points in the direction of greatest steepness (we want to repeatedly take steps in the opposite direction of the gradient to eventually arrive at the minimum).

The gradient of a function is the vector whose elements are its partial derivatives with respect to each parameter. For example, if we were trying to minimize a cost function, C (B0, B1), with just two changeable parameters B0 and B1, the gradient would be:

Gradient of C (B0, B1) = [[dC/dB0], [dC/dB1]]

So each element of the gradient tells us how the cost function would change if we applied a small change to that particular parameter, so we know what to tweak and by how much.

Why derivative/differentiation is used?

When updating the curve, to know in which direction and how much to change or update the curve depending upon the slope. That is why we use differentiation in almost every part of Machine Learning and Deep Learning

To summarize, we can march towards the minimum by following these steps:

- Compute the gradient(slope) of our current location (that is first order derivative of the function at the current point) i.e. calculate the gradient using our current parameter values.

- Modify each parameter by an amount proportional to its gradient element and in the opposite direction of its gradient element (Move in the opposite direction of the slope increase from the current point by the computed amount). For example, if the partial derivative of our cost function with respect to B0 is positive but tiny and the partial derivative with respect to B1 is negative and large, then we want to decrease B0 by a tiny amount and increase B1 by a large amount to lower our cost function.

- Recompute the gradient using our new tweaked parameter values and repeat the previous steps until we arrive at the minimum.

$$\boldsymbol{W}^{+} = \boldsymbol{W} - \eta \nabla \boldsymbol{C} \qquad \text{(eq 1)}$$

$$\nabla \boldsymbol{C} = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

$$\begin{bmatrix} w_1^+ \\ w_2^+ \\ \vdots \\ w_n^+ \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

$$\frac{\partial}{\partial w} J(w) = \nabla_w J \tag{2}$$

$$\frac{\partial}{\partial b} J(w) = \nabla_b J \tag{3}$$

The updated equations are:

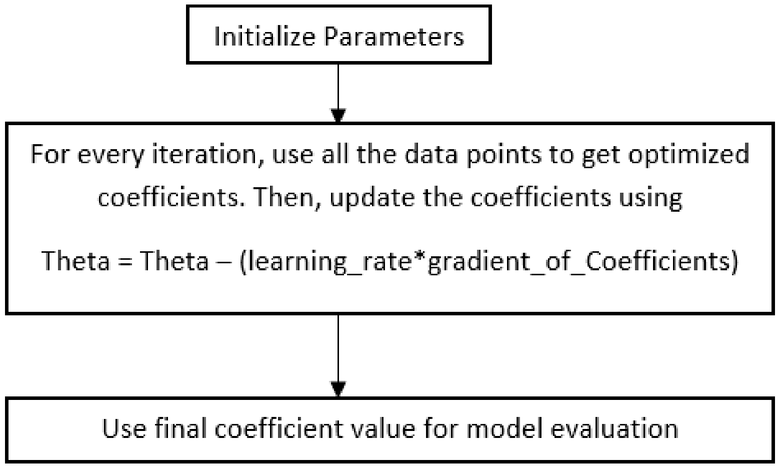
$$w = w - \alpha \nabla_w J \tag{4}$$

$$b = b - \alpha \nabla_b J \tag{5}$$

Types

Batch Gradient Descent

Batch Gradient Descent also called Vanilla gradient descent is the first basic type of gradient descent in which we use the complete dataset available to compute the gradient of cost function. It calculates the error for each example within the training set. After it evaluates all training examples, it updates the model parameters. This process is often referred to as a training epoch. As we need to calculate the gradient on the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. 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. Advantage of batch gradient descent is that it produces a stable error gradient and a stable convergence however it requires that the entire training set resides in memory and is available to the algorithm.


Initialize Parameters
For every iteration, use all the data points to get optimized coefficients. Then, update the coefficients using
Theta = Theta – (learning_rate*gradient_of_Coefficients)
Use final coefficient value for model evaluation

- Easy computation.

- Easy to understand and implement.

- May trap at local minima.

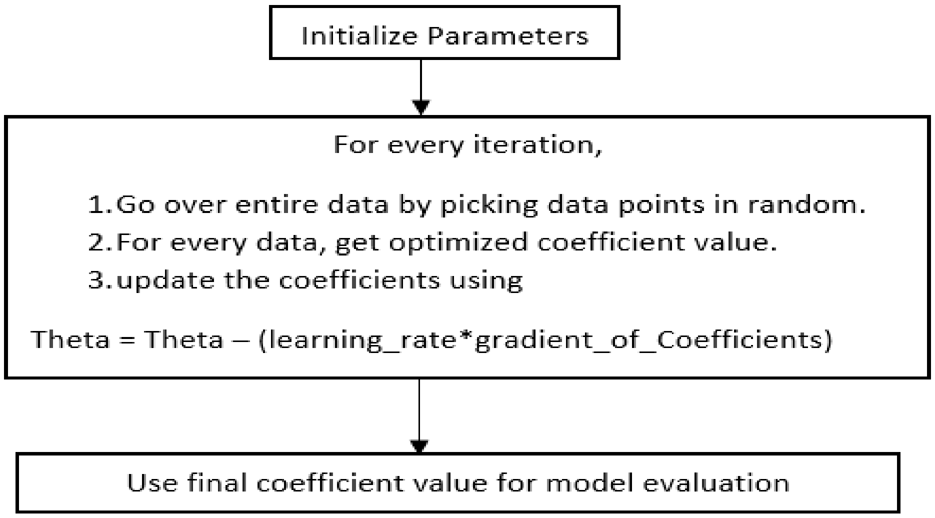
- Weights are changed after calculating gradient on the whole dataset. So, if the dataset is too large than this may a long time to converge to the minima.

- Requires large memory to calculate gradient on the whole dataset

Stochastic Gradient Descent

This is a type of gradient descent which processes 1 training example per iteration. Hence, the parameters are being updated even after one iteration. Hence this is quite faster than batch gradient descent. But again, when the number of training examples is large, even then it processes only one example which can be additional overhead for the system as the number of iterations will be quite large. One advantage is that the frequent updates allow us to have a pretty detailed rate of improvement. In SGD, one might not achieve accuracy, but the computation of results is faster. The frequency of those updates can also result in noisy gradients, which may cause the error rate to jump around(fluctuate) instead of slowly decreasing.

Deep learning models are typically trained by a stochastic gradient descent optimizer. There are many variations of stochastic gradient descent: Adam, RMSProp, Adagrad. All of them let you set the learning rate.


Initialize Parameters
For every iteration,
1. Go over entire data by picking data points in random.
2. For every data, get optimized coefficient value.
3. update the coefficients using
Theta = Theta – (learning_rate*gradient_of_Coefficients)
Use final coefficient value for model evaluation

- Frequent updates of model parameters hence, converges in less time.

- Requires less memory as no need to store values of loss functions.

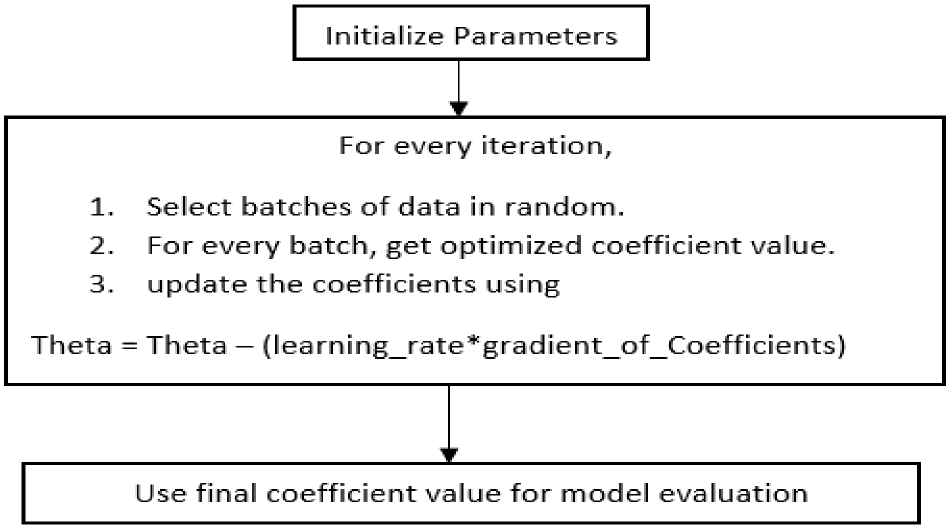
- High variance in model parameters.

- May shoot even after achieving global minima.

- To get the same convergence as gradient descent needs to slowly reduce the value of learning rate

Mini Batch Gradient Descent

Mini batch algorithm is the most favorable and widely used algorithm that makes precise and faster results using a batch of ‘m’ training examples. In mini batch algorithm rather than using the complete data set, in every iteration we use a set of ‘m’ training examples called batch to compute the gradient of the cost function. It simply splits the training dataset into small batches and performs an update for each of these batches. In this way, algorithm reduces the variance of the parameter updates, which can lead to more stable convergence. Therefore, it creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.



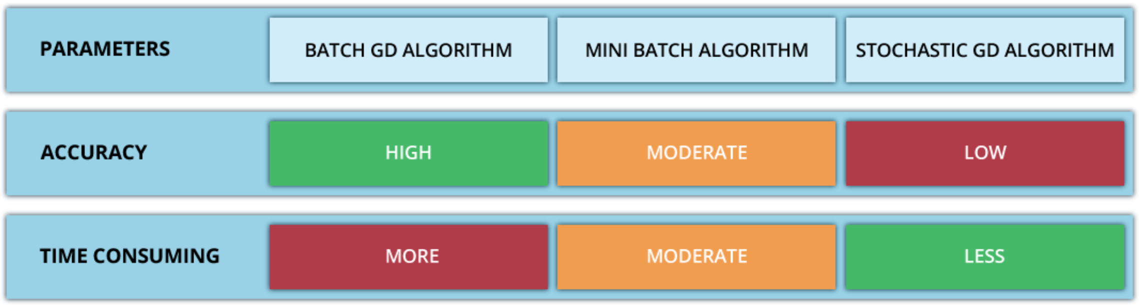
Initialize Parameters

For every iteration,

1. Select batches of data in random.
2. For every batch, get optimized coefficient value.
3. update the coefficients using

Theta = Theta – (learning_rate*gradient_of_Coefficients)

Use final coefficient value for model evaluation



PARAMETERS	BATCH GD ALGORITHM	MINI BATCH ALGORITHM	STOCHASTIC GD ALGORITHM
ACCURACY	HIGH	MODERATE	LOW
TIME CONSUMING	MORE	MODERATE	LESS

- Frequently updates the model parameters and also has less variance

- Requires medium amount of memory

- May get trapped at local minima

If the dataset is small (less than 2000 examples) use batch gradient descent**.** For larger datasets, typical mini-batch sizes are 64, 128, 256 or 512.