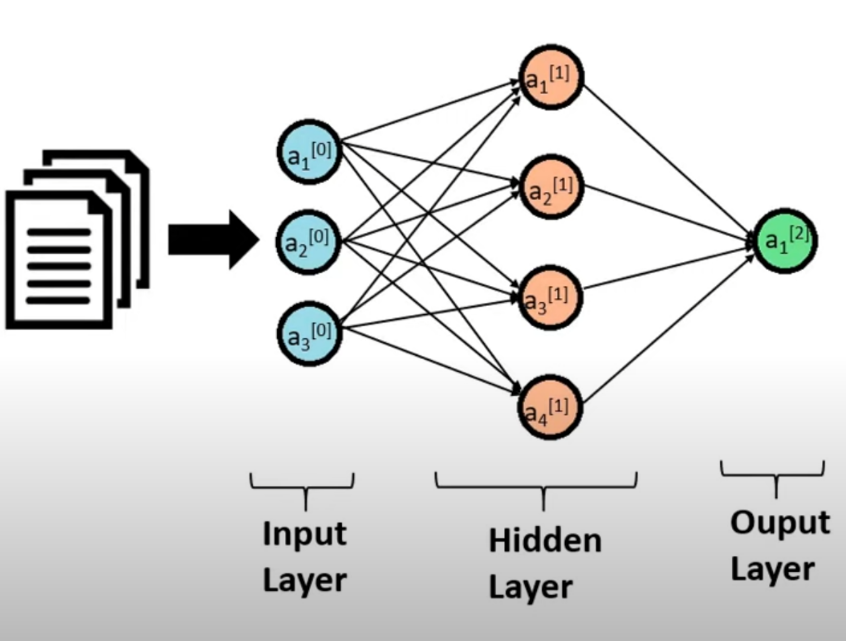
CNN Forward & Backward Propagation

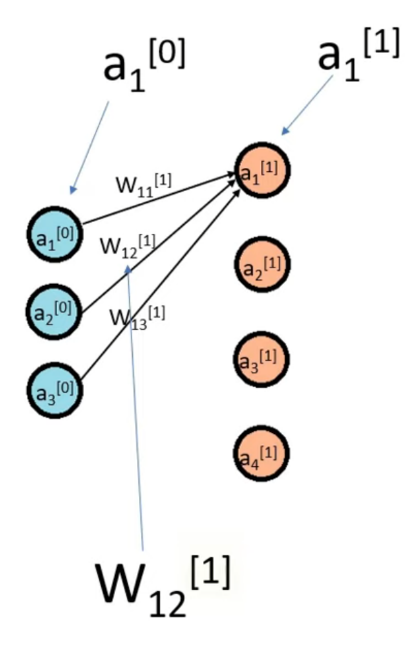
As I have already understood the architecture of CNN so, now I will understand the working of the network and the involvement of mathematical formulation that considered for output prediction.

Considering neural network in which we have,

neuron as **an[0]** at input layer

neuron as **an[1]** at hidden layer

neuron as **an[2]** at output layer

Also, each neuron from input contributes some weight for their next layer which shown as,

**W12[1]** suggest that the weight between **a1[1]** & **a2[0]** and superscript must be of next layer.

With this the value of neuron at hidden layer can be compute with the activation function. For example, if we want to compute neuron value a1[1] it can be done with solving computation between **weight & input neuron.**

a1[1] = activation\_function(

W11[1] \* a1[0] +

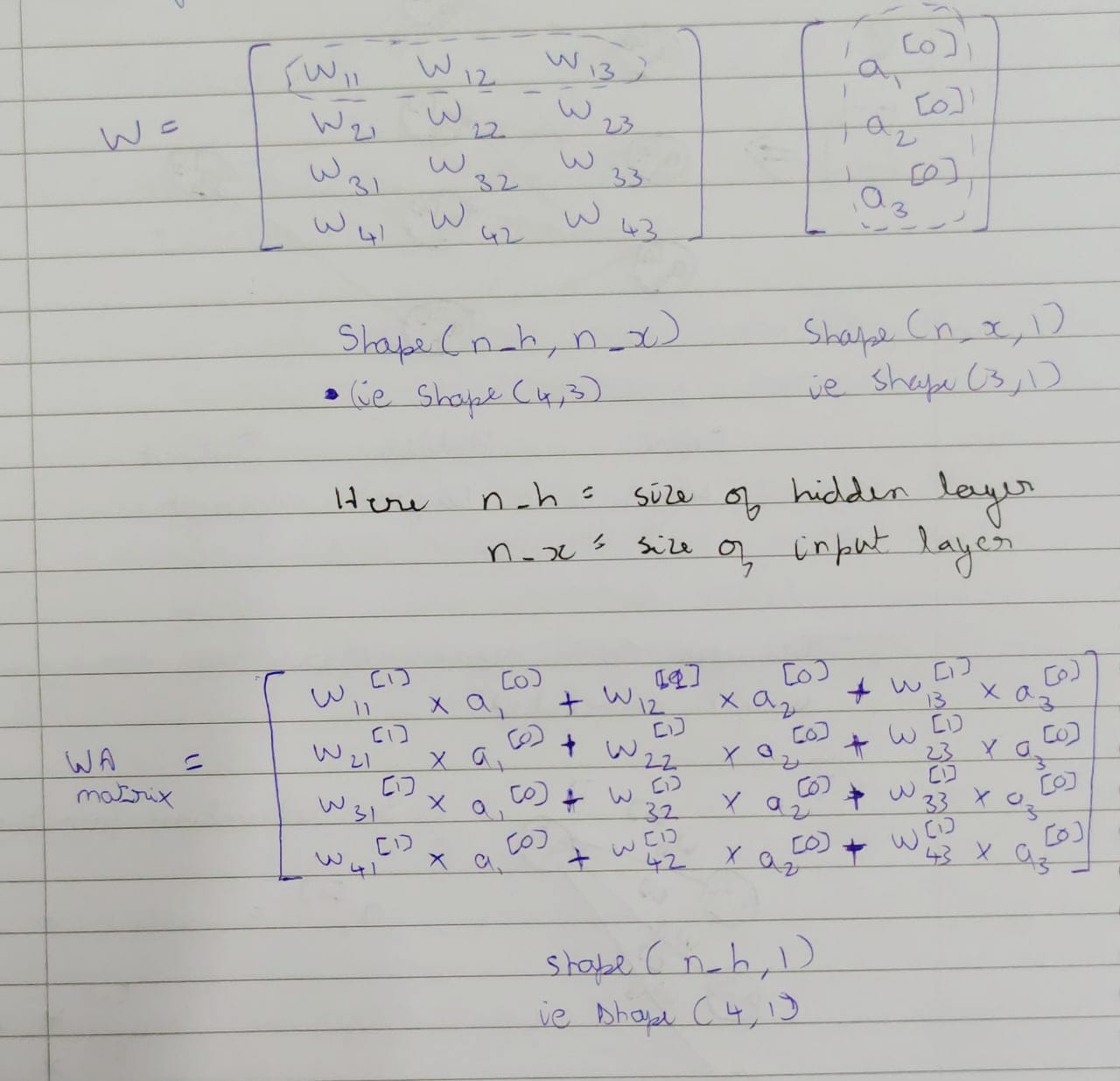
W12[1] \* a2[0] +

W13[1] \* a3[0] +

B1

)

These weights are represented in matrix form. Considering the above neural network, we can have matrix like,



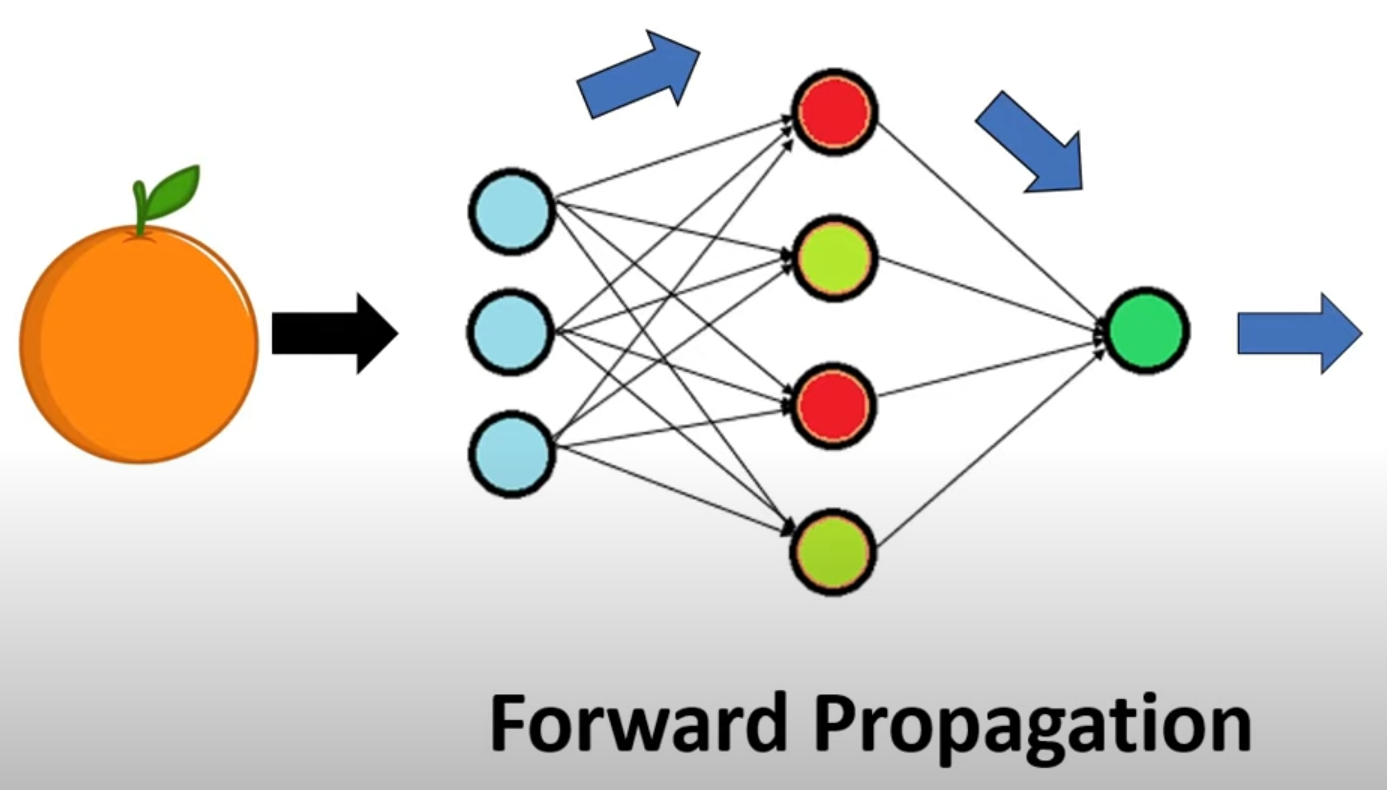
This WAmatrix value is added with some bias value and pass as parameter in activation function to get A1­n matrix i.e. hidden layer neurons.

So final equation looks like,

A1 = f( W1 \* A0 + B1 ) { Hidden layer neuron }

A2 = f( W2 \* A1 + B2 ) { Output Prediction }

**Note:** Activation in the hidden layer is dependent on input we pass so for different input different neuron will be activated. And these activated neuron affects the neuron in the next layer and generate the output.

**Forward Propagation**

So, we get the final output by propagating the information from input layer to the final output layer and as we moving forward here, we called this step as forward propagation.

Where we found the value of neurons in the next layer as we propagate forward.

We could have more then one hidden layer but procedure will remain same,

Z1 = W1 \* A0 + B1

A1 = f( Z1 ) { Hidden layer 1 neuron }

Z2 = W2 \* A1 + B2

A2 = f( Z2 ) { Hidden layer 2 neuron }

Z3 = W3 \* A2 + B3

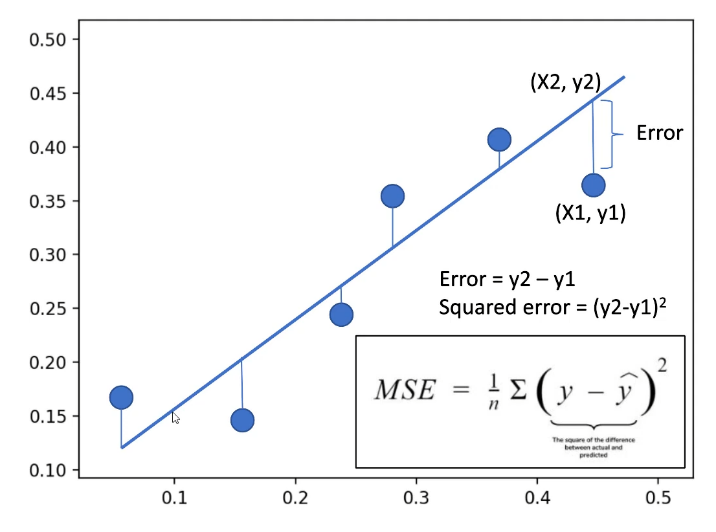
A3 = f( Z3 ) { Output Prediction }

**Backward Propagation**

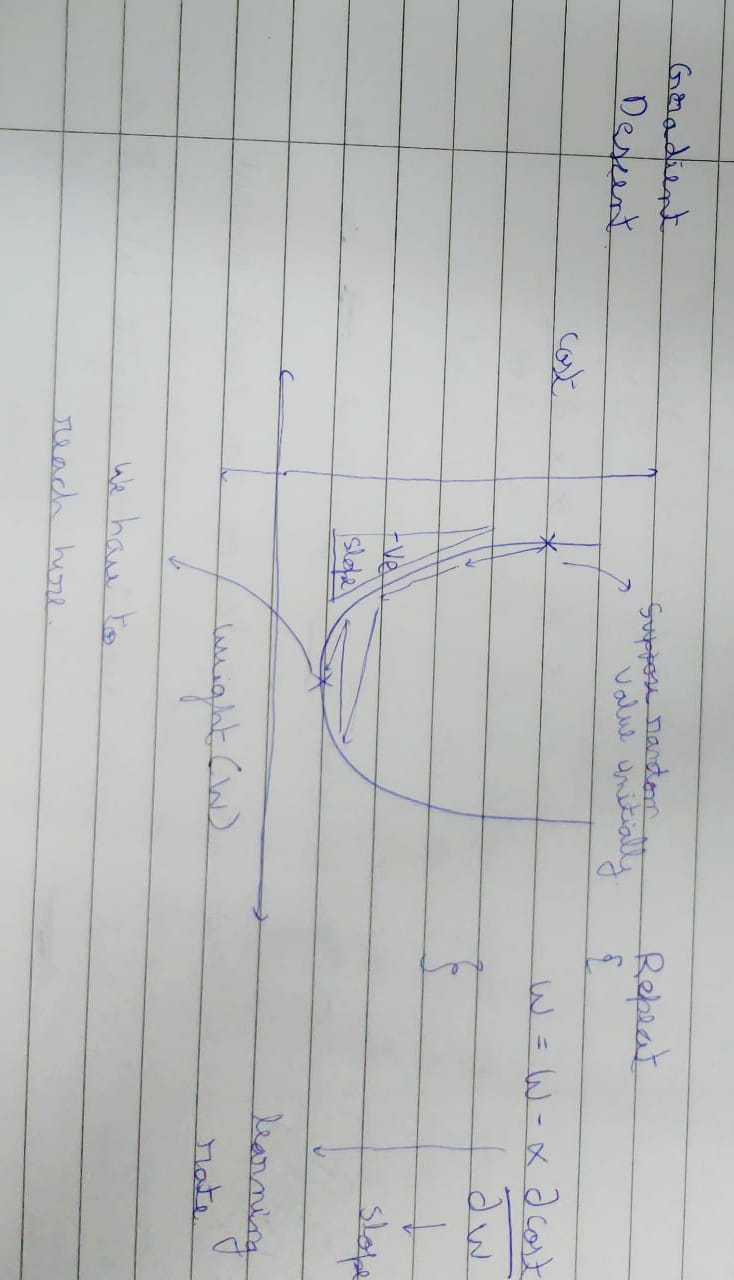
It is difficult to assign weight manually to each neuron connection as there could be so many weights.

So, we can initialize weight randomly and train the model such that after training the values of these weight changed in such a way that it can generate proper output prediction.

And to do so this required to understand **Cost Function or Loss Function** concept

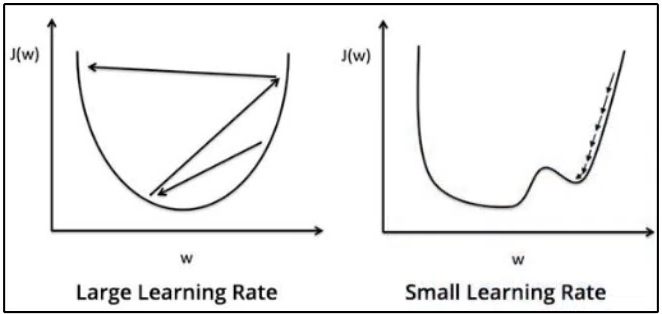
For Example, we can say that cost function quantifies the error between output of the algorithm (**Blue Line**) and given target value (**Data Points**) shown in the figure.

Goal is to minimize the Mean Square Error (MSE) by placing the line equidistance from all the data points.

The random weight is can be made perfect with the help of **optimizer**. **Optimizer assist in minimizing loss function.**

The easiest way to know how does optimizer works is by looking at gradient descent,

Considering a Cost-Weight parabolic graph in which let say initially we start from random weight whose LOSS or COST is pretty high. For next iteration we could move in any direction and again check for its corresponding LOSS if its high we change our direction else move in same direction, these steps are called learning rate in general.

If learning rate is large then we never reach to true minimum.

If learning rate is too small then it takes the first minima as its true minimum.

we must reach to local minima from initial random weight for perfect weight between two neurons. This can be done with the help of formula which shown in the graph.

Repeat {

W = W – α \*

}

Where, α = learning rate

= Slope

The above plotted gradient descent equation is only for single weight, but earlier we saw there are more parameter W for layers. So, we need to update all the parameters of neural network.

Final gradient descent for one hidden layer will be,

Repeat {

W2 = W2 – α \*

B2 = B2 – α \*

W1 = W1 – α \*

B1 = B1 – α \*

}

Now to find we need to use cost depend on our final output production A2 and A2 is depend on weight W2 and neuron layer A1 (i.e. hidden layer). Similarly A1 is depend on weight W1 and input A0. As we need to move backward from A2 output layer to the input layer this process is called backward propagation.

**Summary**

We first pass input data which move forward changing the value of neuron in the next layer which move forward to generate final output prediction

But as this prediction are not way accurate, they are performing poorly so we tell our model what actually the thing is. So change those weight which are associated to recognize it as desire subject so some weight change while some not change much.

Neural Network Training,

Step-1: Initialize random weight.

Step-2: Forward Propagation,

Z1 = W1 \* A0 + B1

A1 = f( Z1 ) { Hidden layer 1 neuron }

Z2 = W2 \* A1 + B2

A2 = f( Z2 ) { Output Prediction }

Step-3: Find value of cost function to check how poor our model performing.

Step-4: Backward Propagation,

Repeat {

W2 = W2 – α \*

B2 = B2 – α \*

W1 = W1 – α \*

B1 = B1 – α \*

}

Step-5: Perform Step-2,3,4 again & again until we get perfect prediction.

Classification Problem

In this type of problem our task is to predict the respective probabilities of all classes.

Regression Problem

In this type of problem our task is to predict the continuous value concerning a given set of independent features to the learning algorithm.

Types of Loss Functions

The loss function is a method of evaluating how well your algorithm models your featured data set. In other words, loss functions are a measurement of **how good your model is in terms of predicting** the expected outcome.

The cost function or loss function refer to the same context (i.e. the training process that uses backpropagation to minimize the error between the actual and predicted outcome).

We calculate the cost function as the average of all loss function values

Whereas we calculate the loss function for each sample output compared to its actual value.

**Loss Function in Regression**

1. Mean Squared Error (MSE) Loss

MSE loss function is the average of squared difference between the actual and predicted value.

MSE =

Advantages:

1. It is in the form of quadratic equation,
2. Plot the quadratic equation, we get a gradient descent with only global minima.
3. We don’t get any local minima.
4. The MSE loss penalizes the model for making large error by squaring them.

Disadvantage:

1. It is not robust to outliers.
2. Mean Absolute Error Loss

MAE loss function is the average of absolute difference between the actual and predicted value.

MAE =

The MAE loss function is more robust to outliers compared to the MSE loss function.

Advantage:

1. The MAE is more robust to outliers as compared to MSE.

Disadvantages:

1. Computation of MAE is very difficult.
2. It may have local minima.
3. Huber Loss

It is combination of both MSE and MAE loss function. It approaches MSE when δ ~ 0 and MAE when δ ~ ∞ (large value).

Loss =

Where δ is hyper-parameter

**Loss Function in Classification**

1. Binary Cross-Entropy Loss

It measures the performance of a classification model where predicted output is a probability value between 0 & 1.

Binary cross entropy =

As we know ‘y’ is actual value which means ‘y’ can be 0 or 1 so,

=

can be identify by sigmoid activation function

=

1. Multi-class Cross Entropy Loss

It is considered with more than 2 output class

Where, C = number of output class

Optimizers

Optimizer assist in minimizing loss function.

An optimizer is a function or an algorithm that modifies the attributes of the neural network, such as weights and learning rate. Thus, it helps in reducing the overall loss and improve the accuracy.

Terminologies used in optimization are,

**Epoch** – The number of times the algorithm runs on the whole training dataset.

**Sample** – A single row of a dataset.

**Batch** – It denotes the number of samples to be taken for updating the model parameters.

**Learning rate** – It is a parameter that provides the model a scale of how much model weights should be updated.

**Cost Function/Loss Function** – A cost function is used to calculate the cost that is the difference between the predicted value and the actual value.

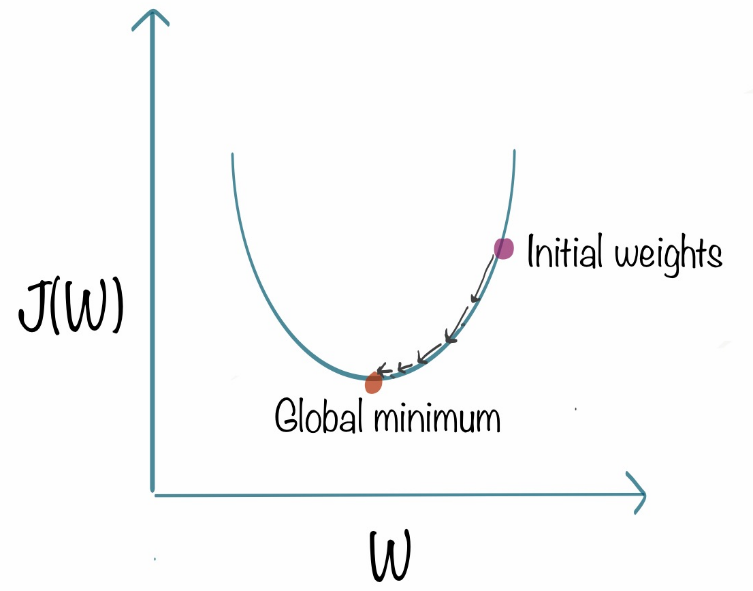
**Weights/Bias** – The learnable parameters in a model that controls the signal between two neurons.

* **Gradient Descent**

In simple terms, consider a ball resting at the top of a bowl. When we lose the ball, it goes along the steepest direction and eventually settles at the bottom of the bowl. A Gradient provides the ball in the steepest direction to reach the local minimum that is the bottom of the bowl.

Where, = learning rate means how far to move against each gradient with each iteration

**Working:**

1. It starts with some coefficient, sees their cost and search for cost value lesser than what it is now.
2. It moves towards the lower weight and updates the value of the coefficients.
3. The process repeats until the global minimum is reached.

**Usage:**

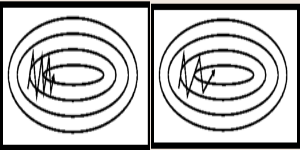
It is expensive to calculate the gradients if the size of the data is huge. Gradient descent works well for convex functions but it doesn’t know how far to travel along the gradient for nonconvex functions.

* **Stochastic Gradient Descent**

For massive data we have stochastic gradient descent. The term stochastic means **randomness** on which the algorithm is based upon. In stochastic gradient descent, instead of taking the whole dataset for each iteration, we randomly select the batches of data. That means we only take few samples from dataset.

* **Stochastic Gradient Descent with Momentum**

As we seen that stochastic gradient descent takes a much noisier path then gradient descent. Due to this reason, it requires a more significant number of iterations to reach the optimal minimum and hence computation time is very slow. To overcome the problem, we use stochastic gradient descent with a momentum algorithm.

What the momentum does is helps in faster convergence of the loss function. Stochastic gradient descent oscillates between either direction of the gradient and updates the weights accordingly. In this algorithm the learning rate should be decreased with a high momentum term.

Left image is showing SGD while right image shows SGD with momentum.

Clearly seen that with using momentum helps reach convergence in less time.

* **AdaGrade (Adaptive Gradient Descent)**

It uses different learning rate for each iteration.

The change in learning rate depends upon the difference in the parameter during training. The more the parameter get changes, the more minor the learning rate changes.

As real-world dataset contains sparse as well as dense features, so it is unfair to have same value of learning rate.

Formula to update the weights,

Where, = learning rate at each iteration

η is a constant

ε is a small positive to avoid division by 0

**Advantage:**

Does not required to modify the learning rate manually.

It is more reliable than gradient descent algorithms and their variants.

It reaches convergence at a higher speed.

**Disadvantage:**

It decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small. This is because the squared gradients in the denominator keep accumulating, and thus the denominator part keeps on increasing. Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.

* **AdaDelta**

It is more robust version of AdaGrade optimizer. It is based upon adaptive learning.

The main problem with AdaGrade is we need to initialize learning rate manually. One other problem is the decaying learning rate which becomes infinitesimally small at some point. Due to which a certain number of iterations later, the model can no longer learn new knowledge.

To deal with these problems, AdaDelta uses two state variables to store the leaky average of the second moment gradient and a leaky average of the second moment of change of parameters in the model.

* **Adam**

Adam is derived from adaptive moment estimation. This optimization algorithm is a further extension of stochastic gradient descent to update network weights during training.

Unlike maintaining a single learning rate through training in SGD, Adam optimizer updates the learning rate for each network weight individually.

In Adam, instead of adapting learning rates based upon the first moment(mean), it also uses the second moment of the gradients.

The Adam optimizer has several benefits, straightforward to implement, has faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.