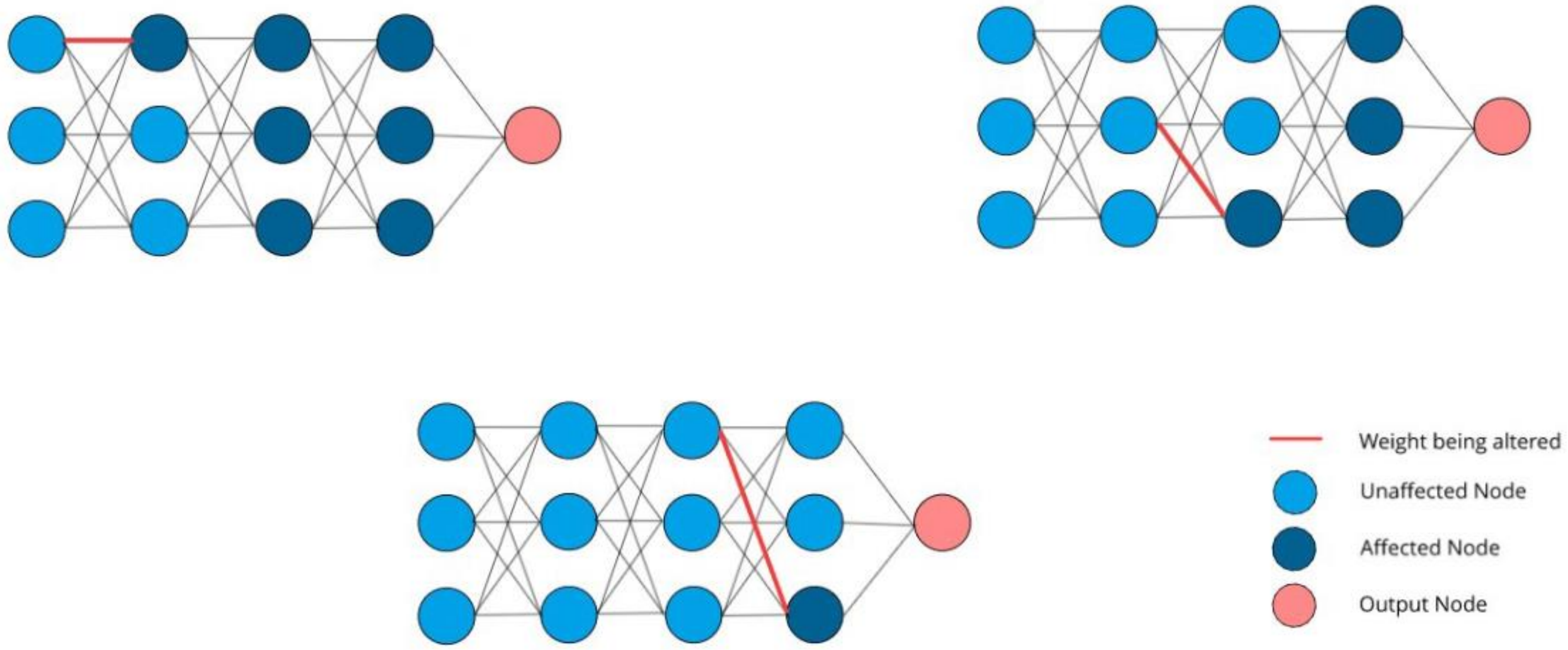
Optimizers

Understanding Optimizers

- In deep learning we have the concept of loss, which tells us how poorly the model is performing at that current instant.
- Now we need to use this loss to train our network such that it performs better.
- Essentially what we need to do is to take the loss and try to minimize it, because a lower loss means our model is going to perform better.
- The process of minimizing (or maximizing) any mathematical expression is called optimization.
- In a neural network, we have many weights in between each layer.
- We have to understand that each and every weight in the network will affect the output of the network in some way, because they are all directly or indirectly connected to the output.

Visualizing which parts of a network altering particular weights will affect

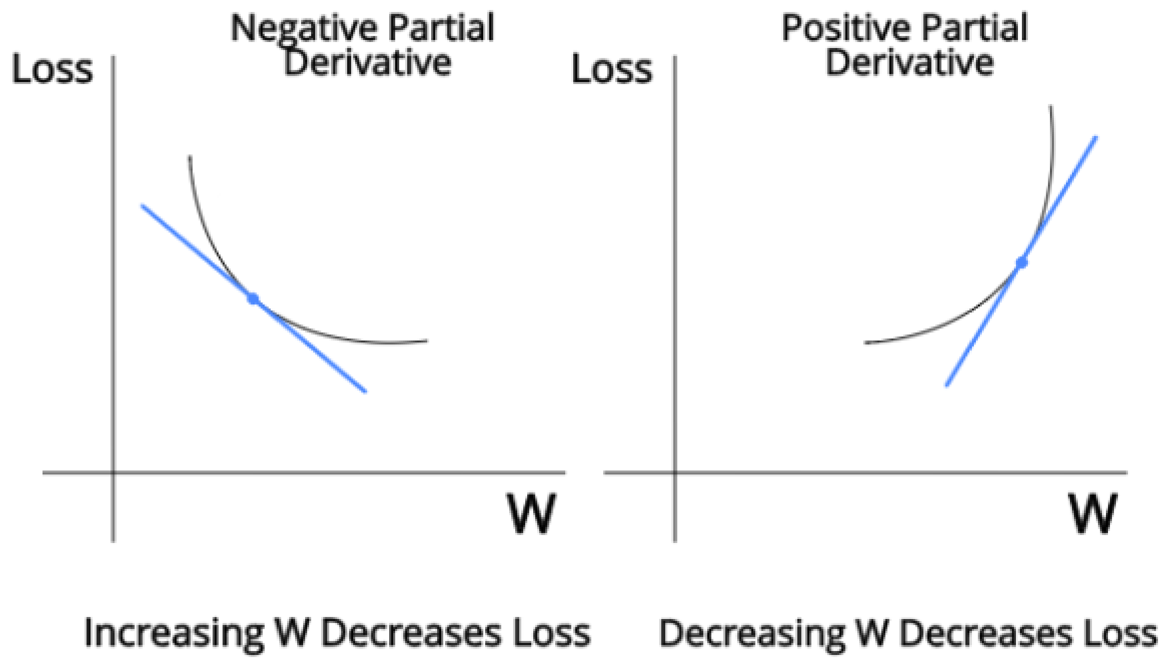


Relationship of weights with loss

- Now that we understand how to change the output of the network by changing the weights, let’s go ahead to understand how we can minimize the loss.
- Changing the weights changes the output.
- Changing the output changes the loss, since loss is a function of the predicted value (Y_pred), which is basically the output of the network.
- Hence we can say that changing the weights will ultimately change the loss.

- Changing can mean increase or decrease, but we need to decrease the loss.
- So now we need to see how to change the weights in such a way that the loss decreases.
- This process is called optimization.

Partial Derivative for optimizing loss visual


Loss
Negative Partial Derivative
W
Increasing W Decreases Loss
Loss
Positive Partial Derivative
W
Decreasing W Decreases Loss

At a particular instant, if the weight's partial derivative is positive, then we will decrease that weight in order to decrease the loss.
If the partial derivative is negative, then we will increase that weight in order to decrease the loss.

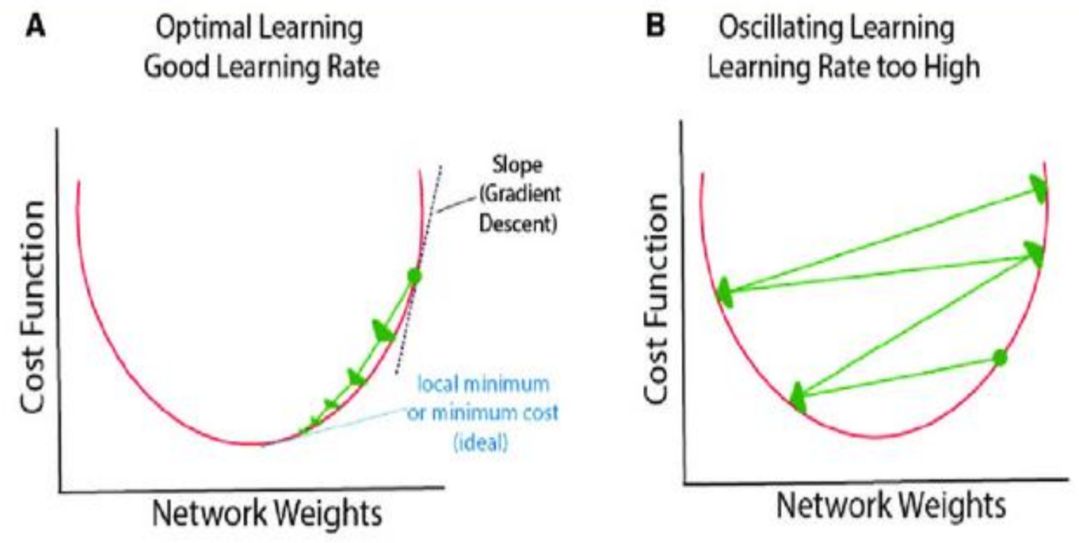
- This algorithm is called Gradient Descent.
- And this is the most basic method of optimizing neural networks.
- This happens as an iterative process and hence we will update the value of each weight multiple times before the Loss converges at a suitable value.

$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

- The weights are updated when the whole dataset gradient is calculated.

$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

- Here the alpha symbol is the learning rate.
- This will affect the speed of optimization of our neural network.
- If we have a large learning rate, we will reach the minima for Loss faster because we are taking big steps, however as a result we may not reach a very good minimum since we are taking big steps and hence we might overshoot it.
- A smaller learning rate will solve this issue, but it will take a lot of steps for the neural network's loss to decrease to a good value.
- Hence we need to keep a learning rate at an optimal value.
- Usually keeping alpha = 0.01 is a safe value.



- In some cases, problems like Vanishing Gradient or Exploding Gradient may also occur due to incorrect parameter initialization.
- These problems occur due to a very small or very large gradient, which makes it difficult for the algorithm to converge.

Gradient Descent variants

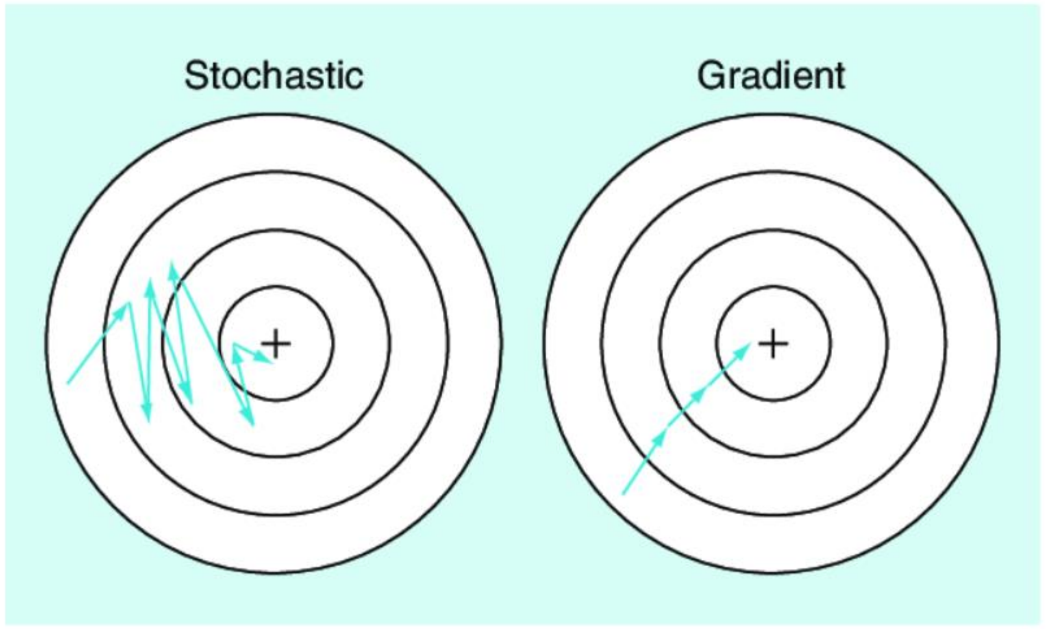
- There are three variants of gradient descent based on the amount of data used to calculate the gradient:
- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent

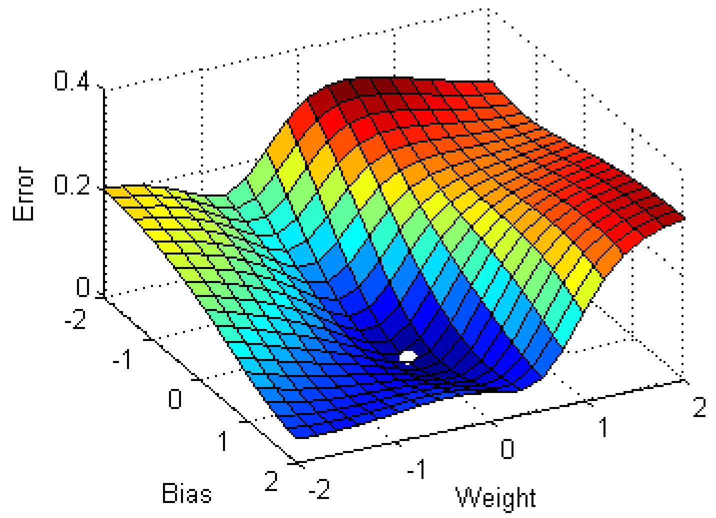
Batch Gradient Descent

- Batch Gradient Descent (Vanilla gradient descent), calculates the error for each observation in the dataset but performs an update only after all observations have been evaluated.
- Batch gradient descent is not often used, because it represents a huge consumption of computational resources, as the entire dataset needs to remain in memory.

Stochastic Gradient Descent

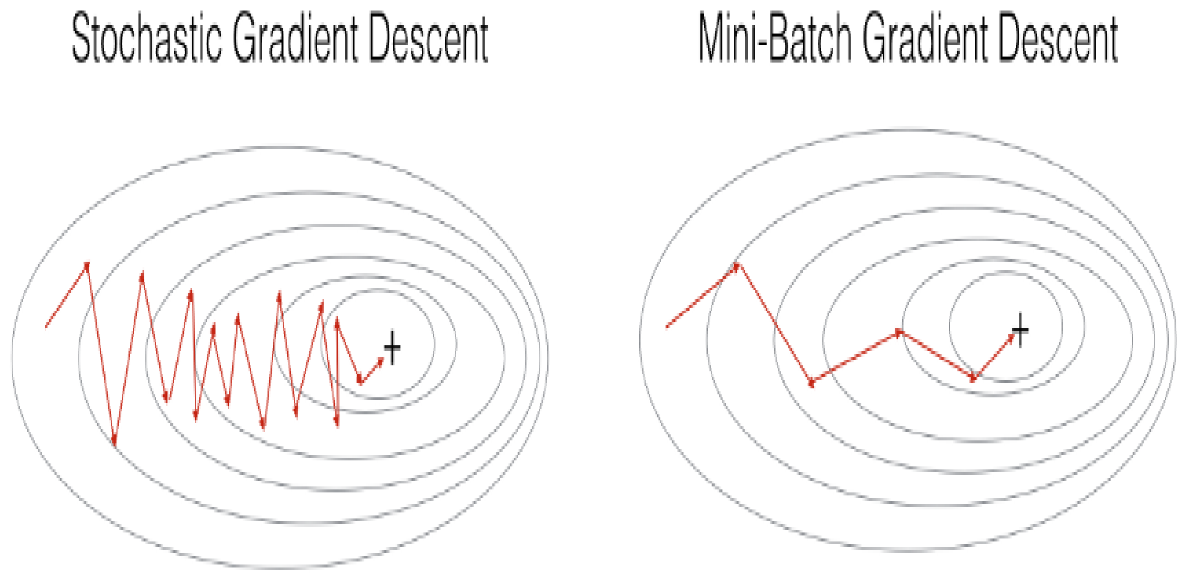
- This is another variant of the Gradient Descent optimizer with an additional capability of working with the data with a non-convex optimization problem.
- The problem with such data is that the cost function results to rest at the local minima which are not suitable for your learning algorithm.
- Instead of taking entire data at one time, in SGD we take single record at a time to feed neural network and to update weights.
- SGD is updated more frequently, the cost function will have severe oscillations as we can see in the figure.
- The oscillation of SGD may jump to a better local minimum.



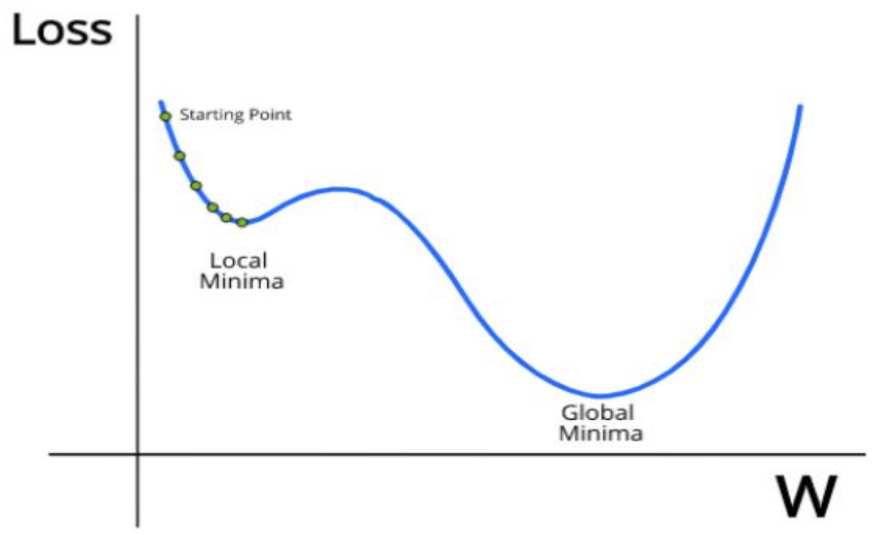


Mini-Batch Gradient Descent

- It is a combination of both batch gradient descent and stochastic gradient descent.
- Mini-batch gradient descent performs an update for a batch of observations.
- MBGD is used where the model parameters are updated in n small batch sizes, n samples to calculate each time. This results in less memory usage and low variance in the model.
- It is the algorithm of choice for neural networks, and the batch sizes are usually from 50 to 256.



Momentum



- Here, we are starting from the labelled green dot.
- Every subsequent green dot represents the loss and new weight value after a single update has occurred.
- The gradient descent will only happen till the local minima since the partial derivative (gradient) near the local minima is near zero.
- Hence it will stay near there after reaching the local minima and will not try to reach the global minima.
- This is a rather simple graph and in reality the graph will be much more complicated than this with many local minima's present.
- Hence if we use just gradient descent we are not guaranteed to reach a good loss.
- We can combat this problem by using the concept of momentum.

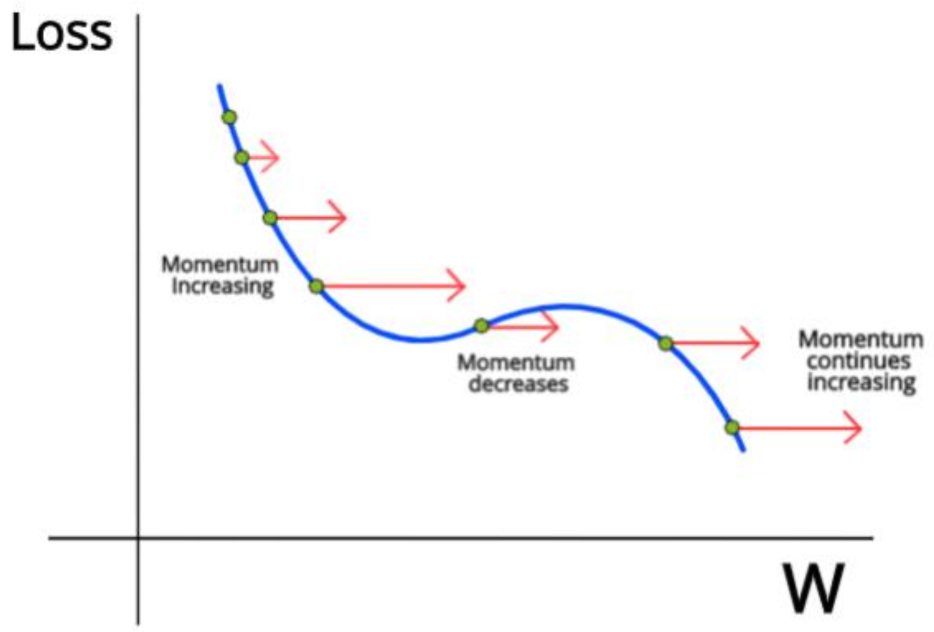
- In momentum, what we are essentially going to do is to try to capture some information regarding the previous updates a weight has gone through before performing the current update.
- Essentially, if a weight is constantly moving in a particular direction (increasing or decreasing), it will slowly accumulate some “momentum” in that direction.
- Hence when it faces some resistance and actually has to go the opposite way, it will still continue going in the original direction for a while because of the accumulated momentum.

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Momentum Update Formula

$$W_{new} = \nu_{new} + W_{old}$$

Weight Update Formula (Momentum)



Here, V is the momentum factor.
As you can see, in each update it essentially adds the current derivative with a part of the previous momentum factor.
And then we just add this to the weight to get the updated weight.
η here is the coefficient of momentum and it decides how much momentum gets carried forward each time.
In most cases, the momentum factor usually is enough to make the weights overcome the local minima.

Nesterov Accelerated Gradients (NAG)

- In NAG, instead of calculating the gradients at the current position we try to calculate it from an approximate future position.
- Just before reaching a minima, the momentum will start reducing before it reaches it because we are using gradients from a future point.
- This results in improved stability and lesser oscillations while converging, furthermore, in practice it performs better than pure momentum.

How NAG helps in optimizing a weight in a neural network.

- If we look at the whole momentum equation expanded:

$$W_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})} + W_{old}$$

- Considering the fact that alpha*current gradient is going to be a very small value, we can approximate the next W value by just adding η *v_old with the current W value.
- This will give an approximate future W value.

$$W_{future} = \eta * \nu_{old} + W_{old}$$

Calculating approximate future weight value

- Now what we want to do is instead of calculating gradients with respect to the current W value, we will calculate them with respect to the future W value.
- This allows the momentum factor to start adapting to sharp gradient changes before they even occur, leading to increased stability while training.

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{future})}$$

Nesterov Momentum Formula

- This is the new momentum equation with NAG. As you can see, we are taking gradients from an approximate future value of W instead of the current value of W.

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Momentum Update Formula

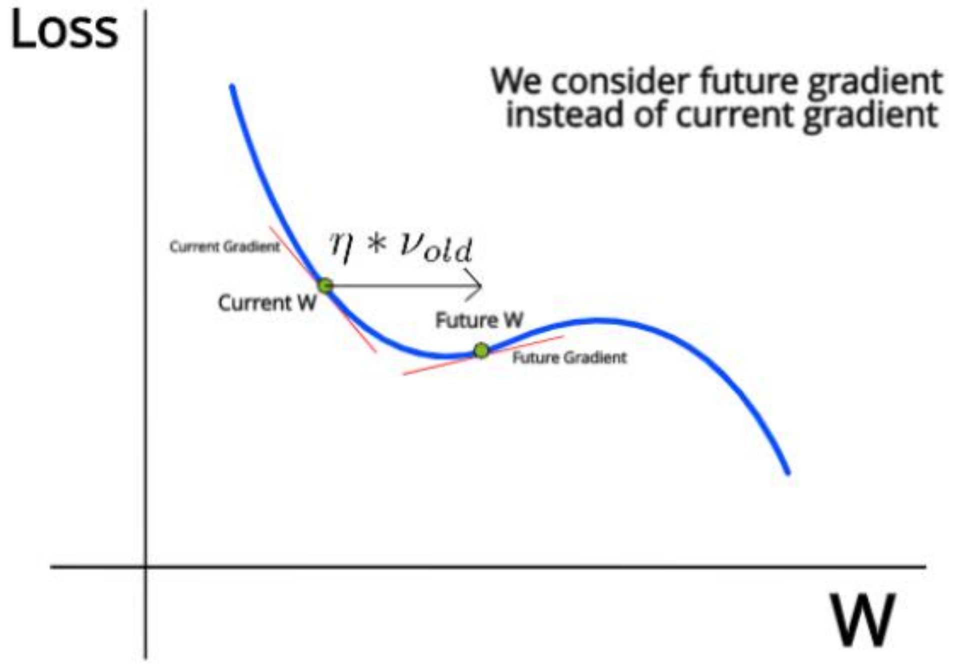
$$W_{new} = \nu_{new} + W_{old}$$

Weight Update Formula (Momentum)

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{future})}$$

Nesterov Momentum Formula

- Here is an example of using NAG. At the current W value, we add η *v_old to approximate the future W value.
- Then we calculate the gradient at that point and use that instead of current gradient value while calculating the v_new value.
- As you can see in this example, even though the momentum is technically supposed to increase in this scenario, it will start decreasing at this point itself because the future W value has a gradient pointing in the opposite direction.



Nesterov VIsual Example

Adaptive Optimization

- In these methods, the parameters like learning rate (alpha) and momentum of coefficient (η) will not stay constant throughout the training process.
- Instead, these values will constantly adapt for each and every weight in the network and hence will also change along with the weights.
- These kind of optimization algorithms fall under the category of adaptive optimization.

Adagrad

- Adagrad is short for adaptive gradients. In this we try to change the learning rate (alpha) for each update.
- The learning rate changes during each update in such a way that it will decrease if a weight is being updated too much in a short amount of time and it will increase if a weight is not being updated much.

- First, each weight has its own cache value, which collects the squares of the gradients till the current point.

$$cache_{new} = cache_{old} + \left(\frac{\partial(Loss)}{\partial(W_{old})}\right)^2$$

Cache updation for Adagrad

- The cache will continue to increase in value as the training progresses. Now the new update formula is as follows:

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adagrad Update Formula

This is of the same form as the original gradient descent formula except for the fact that the learning rate (alpha) constantly changes throughout the training. The ϵ in the denominator is a very small value to ensure division by zero does not occur.

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adagrad Update Formula

- Essentially what's happening here is that if a weight has been having very huge updates, it's cache value is also going to increase.
- As a result, the learning rate will become lesser and that weight's update magnitudes will decrease over time.
- On the other hand, if a weight has not been having any significant update, it's cache value is going to be very less, and hence its learning rate will increase, forcing it to take bigger updates.
- This is the basic principle of the Adagrad optimizer.

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new} + \epsilon}} * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adagrad Update Formula

- However the disadvantage of this algorithm is that regardless of a weight’s past gradients, the cache will always increase by some amount because square cannot be negative.
- Hence the learning rate of every weight will eventually decrease to a very low value till training does not happen significantly anymore.
- The next adaptive optimizer, RMSProp effectively solves this problem.

RMSProp

- In RMSProp the only difference lies in the cache updating strategy. In the new formula, we introduce a new parameter, the decay rate (gamma).

$$cache_{new} = cache_{old} + (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Cache updation for Adagrad

$$cache_{new} = \gamma * cache_{old} + (1 - \gamma) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

RMSProp Cache Update Formula

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adagrad Update Formula

- Here the gamma value is usually around 0.9 or 0.99. Hence for each update, the square of gradients get added at a very slow rate compared to Adagrad.
- This ensures that the learning rate is changing constantly based on the way the weight is being updated, just like Adagrad, but at the same time the learning rate does not decay too quickly, hence allowing training to continue for much longer.

Adam- Adaptive Moment Estimation

- Adam is a little like combining RMSProp with Momentum.
- First we calculate our m value, which will represent the momentum at the current point.

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Momentum Update Formula

$$m_{new} = \beta_1 * m_{old} - (1 - \beta_1) * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adam Momentum Update Formula

The only difference between this equation and the momentum equation is that instead of the learning rate we keep $(1 - \beta_1)$ to be multiplied with the current gradient.

- Next we will calculate the accumulated cache, which is exactly the same as it is in RMSProp:

$$cache_{new} = \beta_2 * cache_{old} + (1 - \beta_2) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

- Now we can get the final update formula:

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

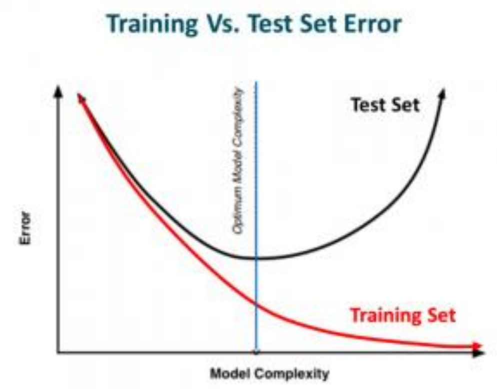
Adagrad Update Formula

$$W_{new} = W_{old} - \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * m_{new}$$

Adam weight updation formula

- As we can observe here, we are performing accumulating the gradients by calculating momentum and also we are constantly changing the learning rate by using the cache.
- Due to these two features, Adam usually performs better than any other optimizer out there and is usually preferred while training neural networks.
- In the paper for Adam, the recommended parameters are 0.9 for beta1(momentum) , 0.99 for beta2 (cache) and 1e-08 for epsilon.

Regularization for Deep Learning



- Overfitting is a major issue that occurs during training.
- A model is considered as overfitting the training data when the training error keeps decreasing but the test error (or the generalization error) starts increasing.
- At this point we tend to believe that the model is learning the training data distribution and not generalizing to unseen data.
- Regularization is a modification we make to the learning algorithm or the model architecture that reduces its generalization error, possibly at the expense of increased training error.

Different Regularization Techniques in Deep Learning

- Data Augmentation
- Weight Regularization
- Dropouts
- Early Stopping

1. Decrease the network complexity

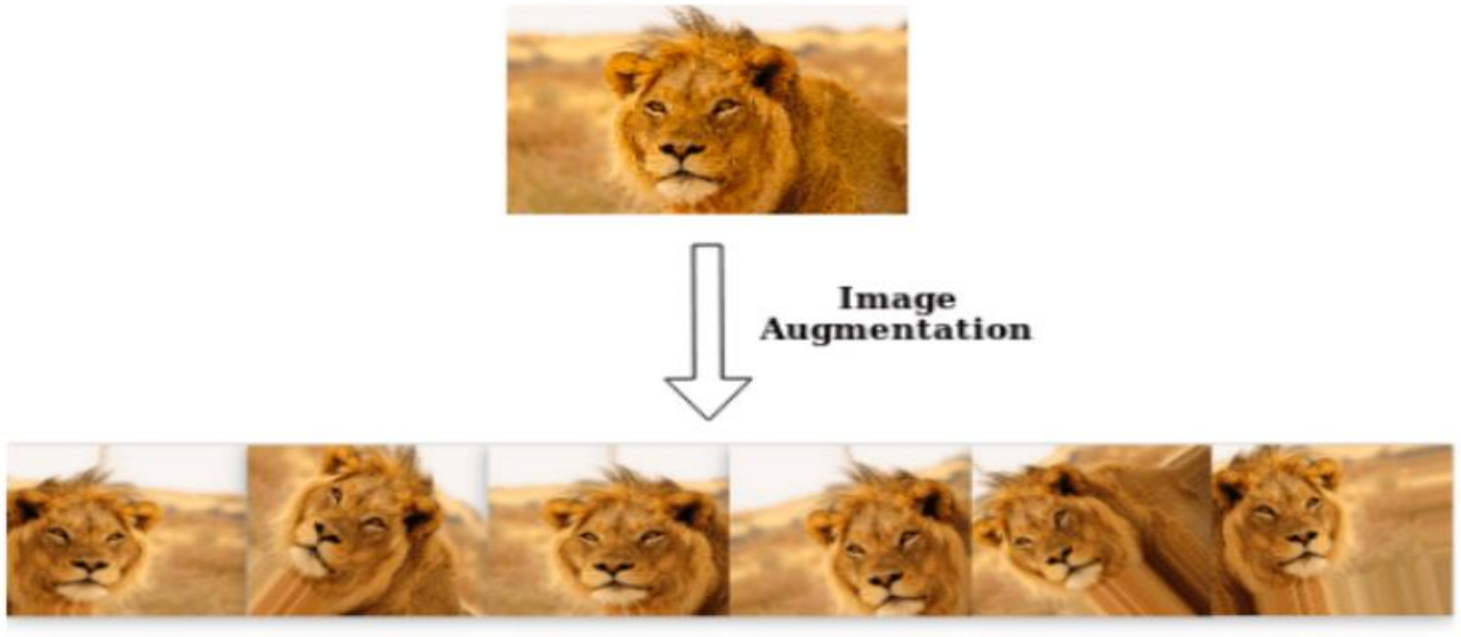
- Deep neural networks like CNN are prone to overfitting because of the millions or billions of parameters it encloses.
- A model with these many parameters can be overfit on the training data because it has sufficient capacity to do so.
- By removing certain layers or decreasing the number of neurons (filters in CNN) the network becomes less prone to overfitting as the neurons contributing to overfitting are removed or deactivated.
- The network also has a reduced number of parameters because of which it cannot memorize all the data points & will be forced to generalize.

- There is no general rule as to how many layers are to be removed or how many neurons must be in a layer before the network can overfit.
- The popular approach for reducing the network complexity is

1. Grid search can be applied to find out the number of neurons and/or layers to reduce or remove overfitting.
2. The overfit model can be pruned (trimmed) by removing nodes or connections until it reaches suitable performance on test data.

2. Data Augmentation

- One of the best strategies to avoid overfitting is to increase the size of the training dataset.
- But in real-world scenarios gathering of large amounts of data is a tedious & time-consuming task, hence the collection of new data is not a viable option.



3. Weight Regularization

- Weight regularization is a technique which aims to stabilize an overfitted network by penalizing the large value of weights in the network.
- Weight regularization penalizes the network’s large weights & forcing the optimization algorithm to reduce the larger weight values to smaller weights, and this leads to stability of the network & presents good performance.
- In weight regularization, the network configuration remains unchanged only modifying the value of weights.

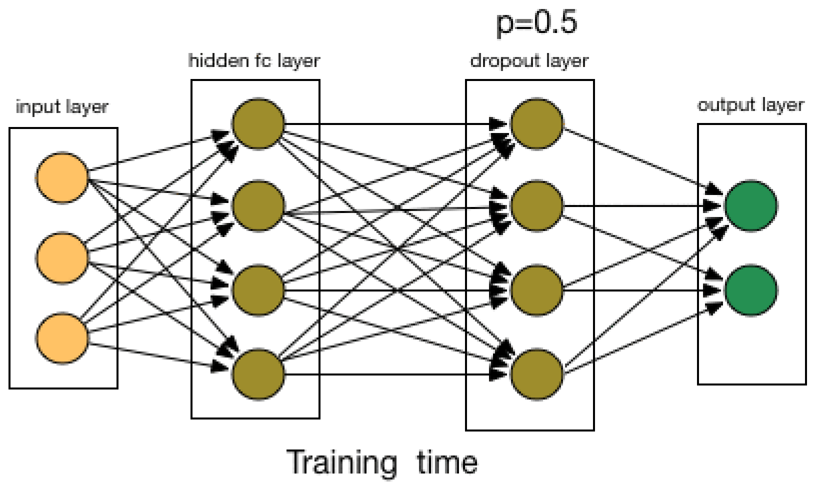
$$E = \underbrace{\frac{1}{2} * \sum (t_k - o_k)^2}_{\text{plain error}} + \underbrace{\frac{\lambda}{2} * \sum {w_i}^2}_{\text{weight penalty}}$$

- **L1 regularization** adds the sum of absolute values of the weights in the network as the weight penalty.
- **L2 regularization** adds the squared values of weights as the weight penalty.

- The **lambda term** is a hyperparameter which defines how much of the network's weights must be reflected on the loss function or simply the term which controls the influence of weight penalty on the loss function.
- If the **data is too complex**, **L2 regularization** is a better choice as it can model the inherent pattern in the data. If the **data is simple**, **L1 regularization** can be used.
- For most computer vision L2 regularization is applied.

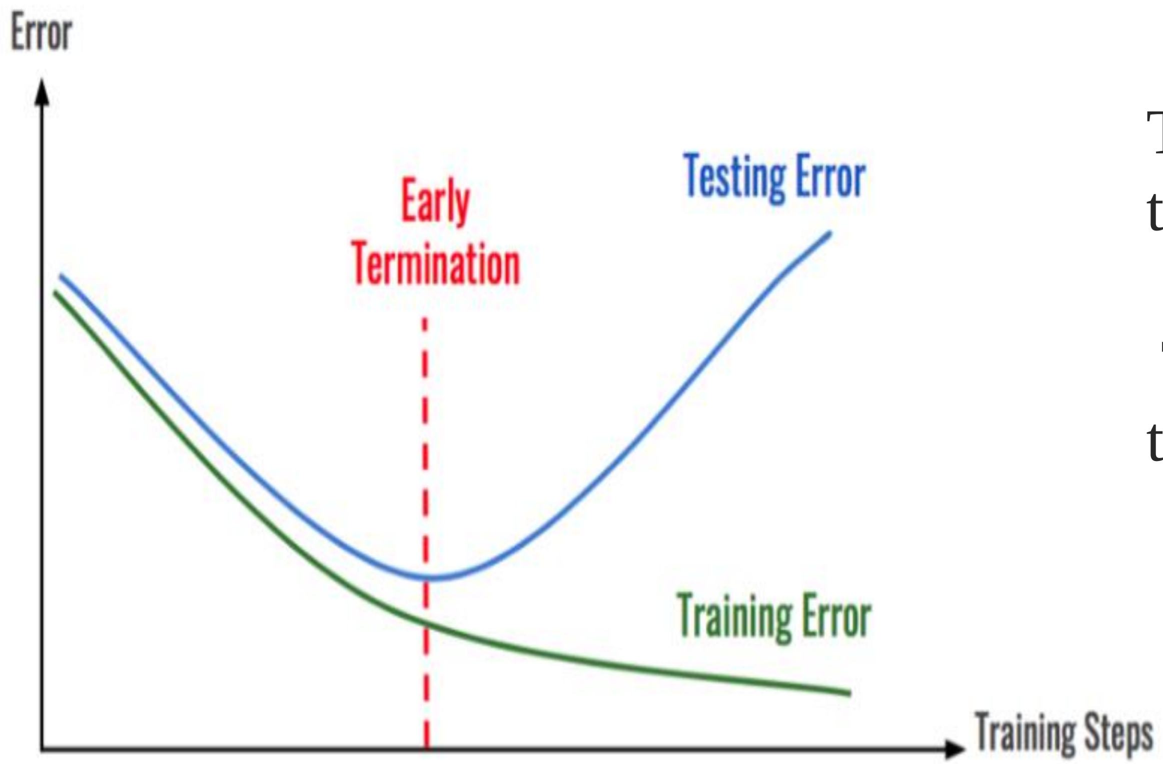
4. Dropouts

- Dropout is a regularization strategy that prevents deep neural networks from overfitting.
- While L1 & L2 regularization reduces overfitting by modifying the loss function, dropouts, on the other hand, deactivate a certain number of neurons at a layer from firing during training.



5. Early Stopping

- While training a neural network using an optimization algorithm like **Gradient Descent,** the model parameters (weights) are updated to reduce the training error.
- At the end of each forward propagation, the network parameters are updated to reduce error in the next iteration.
- Too much training can result in network overfitting on the training data.
- Early stopping provides guidance as to how many iterations can be run before the network begins to overfit.



The above graph indicates the point after which the network begins to overfit.

The network parameters at the point of early termination are the best fit for the model.

- To decrease the test error beyond the point of early termination can be done by

1. Decreasing the learning rate. Applying a learning rate scheduler algorithm would be recommended.
2. Applying a different optimization algorithm.
3. Applying L1 or L2 regularization.