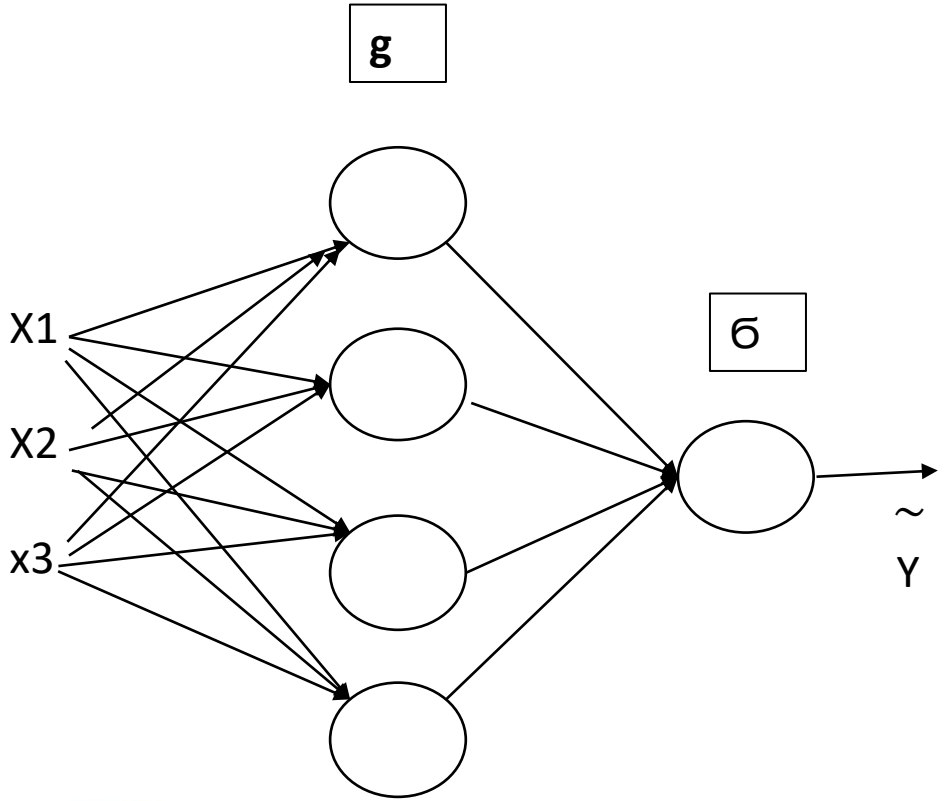
Neural Network Representation



Notation

$Z_i[l]$= input to l^{th} layer i^{th} neuron

$a_i[l]$ = output of l^{th} layer i^{th} neuron

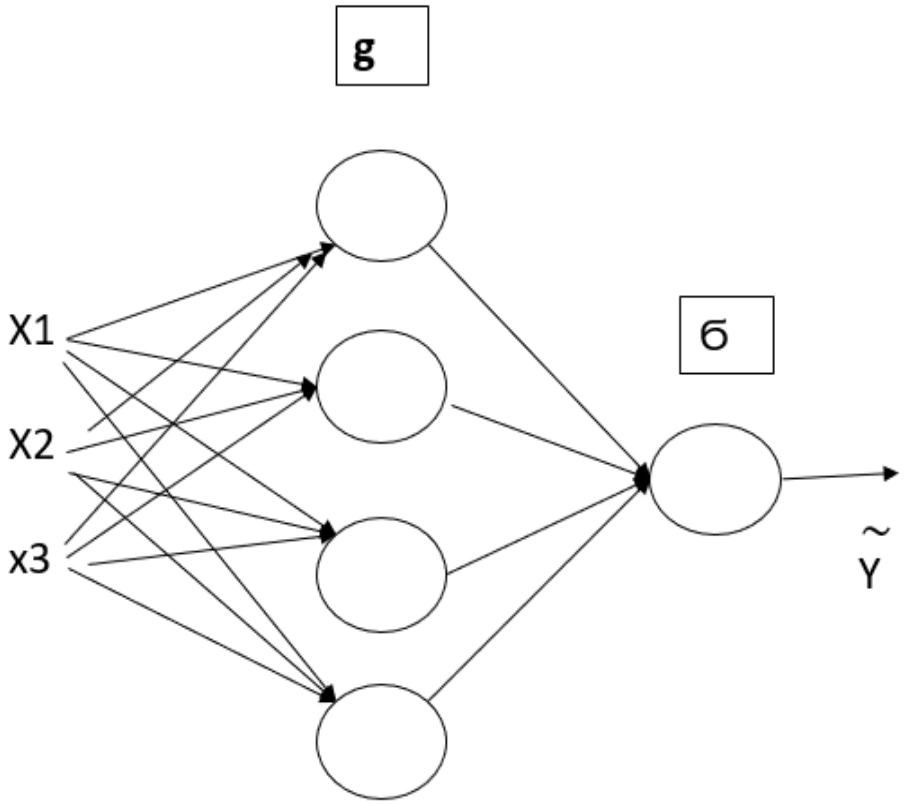
$$Z_1[1] = w_{11} * x1 + w_{12} * x2 + w_{13} * x3 + b_1$$

$$= (w_{11}\ w_{12}\ w_{13}) \begin{pmatrix} X1 \\ X2 \\ x3 \end{pmatrix} + b1$$

$$Z[1] = W[1] * X + b[1]$$

(4,1) (4,3) (3.1) (4,1)

$$a[1] = g(Z[1])$$

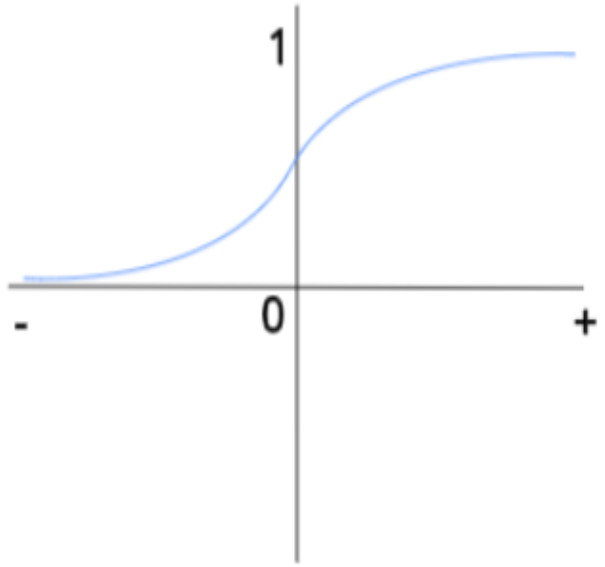


$$Z^{[2]}_{(1,1)} = W^{[2]}_{(1,4)} * a^{[1]}_{(4,1)} + b^{[2]}_{(1,1)}$$

$$\tilde{Y} = a^{[2]} = б\,(Z^{[2]})$$

Binary classification activation function

Sigmoid σ



Sigmoid Function Graph Visualization

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Considering Multiple training examples

X(i)=ith training example

$\tilde{Y}(i)$= output of the ith training example

Z[l](i) = input to the lth layer ith training example

a[l](i) = output of the lth layer ith training example

For i= 1 to m

Z[1](i) =W[1] * X(i) +b[i]

a[1](i) = g(Z[1](i))

Z[2](i) = W[2] * a[1](i) +b[2]

a[2](i) = σ (Z[2](i))

$\tilde{y}$[i] = a[2](i)

Eliminating the for loop

X= X(1) X(2) X(3)…… X(m)

A[l]= a[l](1) a[l](2)…..a[l](m)

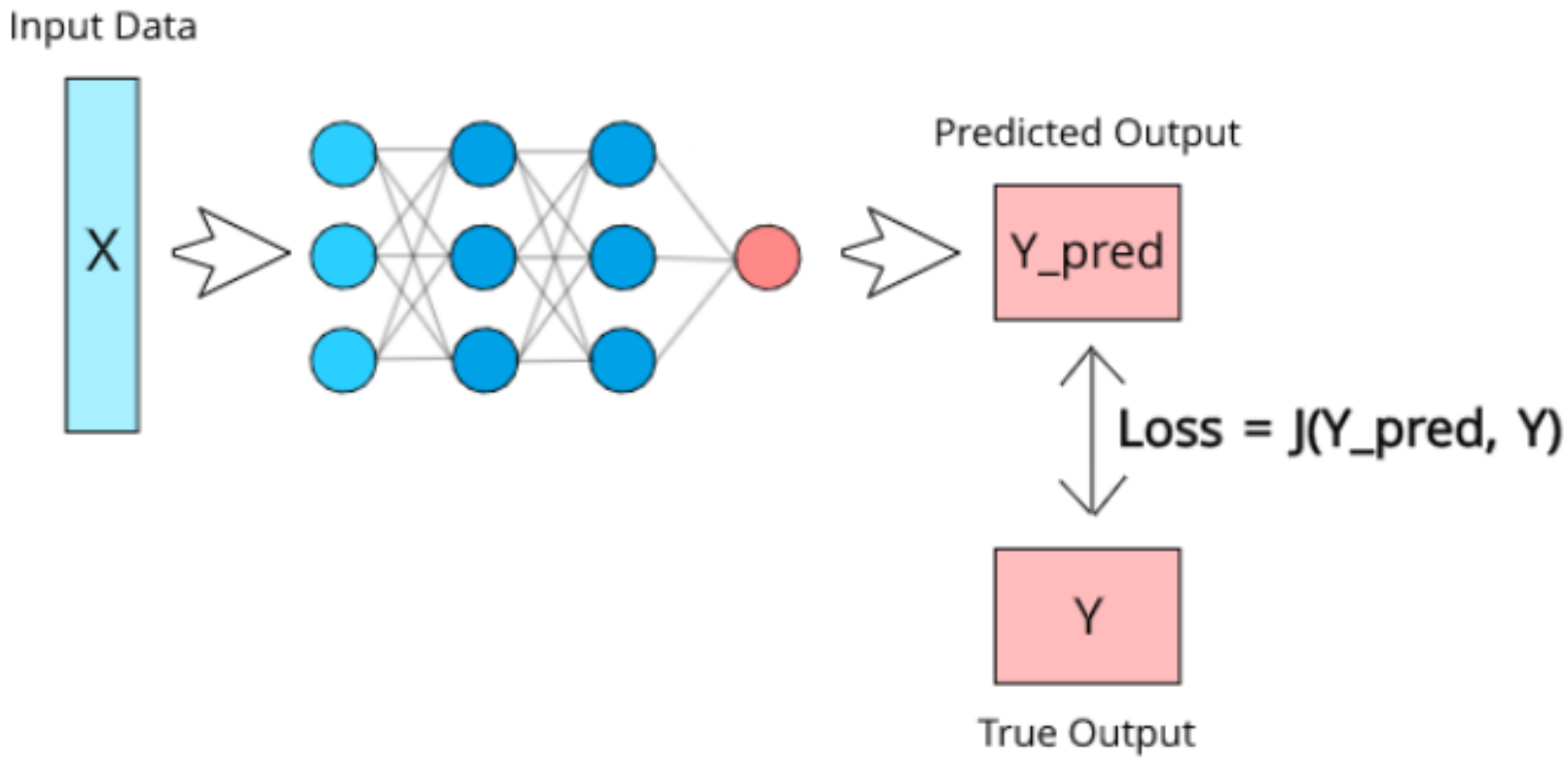
For i= 1 to m
    Z[1](i) =W[1] * X(i) +b[1]
    a[1](i) = g(Z[1](i))
    Z[2](i) = W[2] * a[1](i) +b[2]
    a[2](i) = σ (Z[2](i))

    y[i] = a [2](i)

Z[1] = W[1] * X + b[1]
A[1] = g (Z[1])
Z[2] = W[2] * A[1] + b[2]

Y=A[2] = σ (Z[2])

Neural Network Loss Functions



Neural Network Loss Visualization

Forward Propagation

$Z[1] = W[1]\, X + b[1]$

$A[1] = g[1](Z[1])$

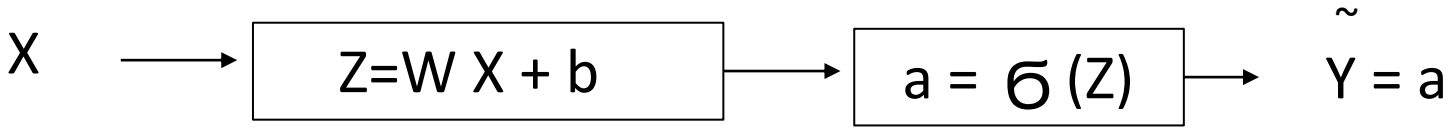
$Z[2] = W[2]\, A[1] + b[2]$

$A[2] = g[2]\,(Z[2])$

$= \sigma\,(Z[2]) = \tilde{y}$

Back Propagation

Consider a simple situation (Logistic Regression)



Assume that the loss function use here is Binary Cross-entropy Loss Function

$$L(\tilde{Y}\text{-}Y) = -Y \log \tilde{Y} - (1\text{-}Y) \log (1\text{-}\tilde{Y})$$
$$= -Y \log a - (1\text{-}Y) \log (1\text{-}a)$$

Where Y is the ground truth value

and

Optimizer is Gradient Dissent

$$W_{new} = W_{old} - \alpha \frac{dL}{dW_{old}}$$

Where α is the learning rate

$$\frac{dL}{d\bar{a}} = \frac{-Y}{\bar{a}} + \frac{1-Y}{1-\bar{a}}$$

$$a = \sigma(Z) = \frac{1}{1+e^{-Z}}$$

Sigmoid activation function

$$\frac{da}{dZ} = a(1-a)$$

Prove

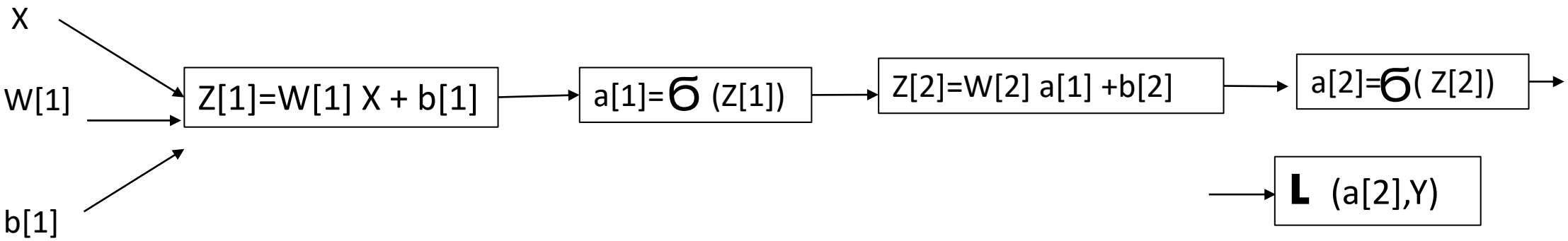
$$\frac{dL}{dZ} = \frac{da}{dZ} \cdot \frac{dL}{da} = a-Y$$

Chain rule

$$\frac{dL}{dW} = \frac{dZ}{dW} \cdot \frac{da}{dZ} \cdot \frac{dL}{da} = X\,(a-Y)$$

$$\frac{dL}{db} = (a-Y)$$

Two layer NN - Calculating gradients



Loss Functions

Model parameters :

$$W[1]_{(n[1],n[0])},\ b[1]_{(n[1],1)},\ W[2]_{(n[2],n[1])},\ b[2]_{(n[2],1)}$$

In the example n[0]=3, n[1]=4, n[2]=1

In general $(W[l]_{(n[l],\,n[l-1])},\ b[l]_{(n[l],\,1)})$ l= 1 to no. of layers

Loss Function $J(W[1], b[1], W[2], b[2]) = (1/m) \sum_{i=1}^{m} L(\tilde{y}(i) - y(i))$

Loss Function types

(1) Regression loss functions
- output of the model is a single value

(2) Classification loss functions -
- output of the model is a vector of values

Regression Loss Functions

Mean Squared Error

Mathematical formulation :

$$MSE = (1/m) \sum_{i=1}^{m} L(\tilde{y}(i) - y(i))^2$$

As the name suggests, *Mean square error* is measured as the average of squared difference between predictions and actual observations. It's only concerned with the average magnitude of error irrespective of their direction. However, due to squaring, predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions. Plus MSE has nice mathematical properties which makes it easier to calculate gradients.

Regression Loss Functions

Mean Absolute Error

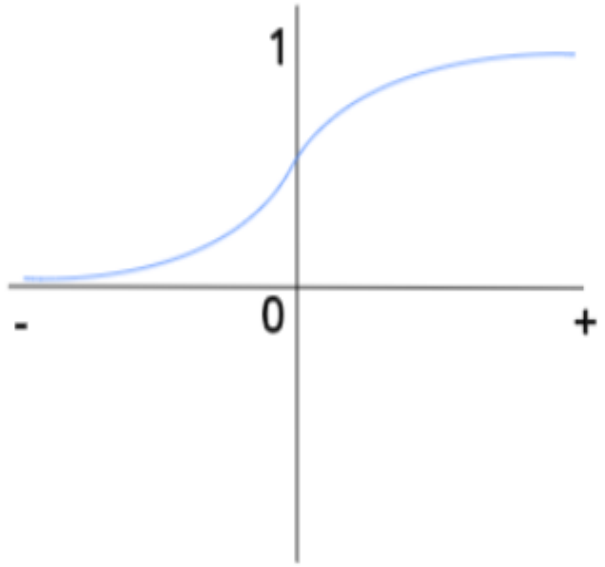
Mathematical formulation:

$$MAE = (1/m) \sum_{i=1}^{m} | L(\tilde{y}(i) - y(i)) |$$

Mean absolute error, on the other hand, is measured as the average of sum of absolute differences between predictions and actual observations. Like MSE, this as well measures the magnitude of error without considering their direction. Unlike MSE, MAE needs more complicated tools such as linear programming to compute the gradients. Plus MAE is more robust to outliers since it does not make use of square.

Binary classification activation function

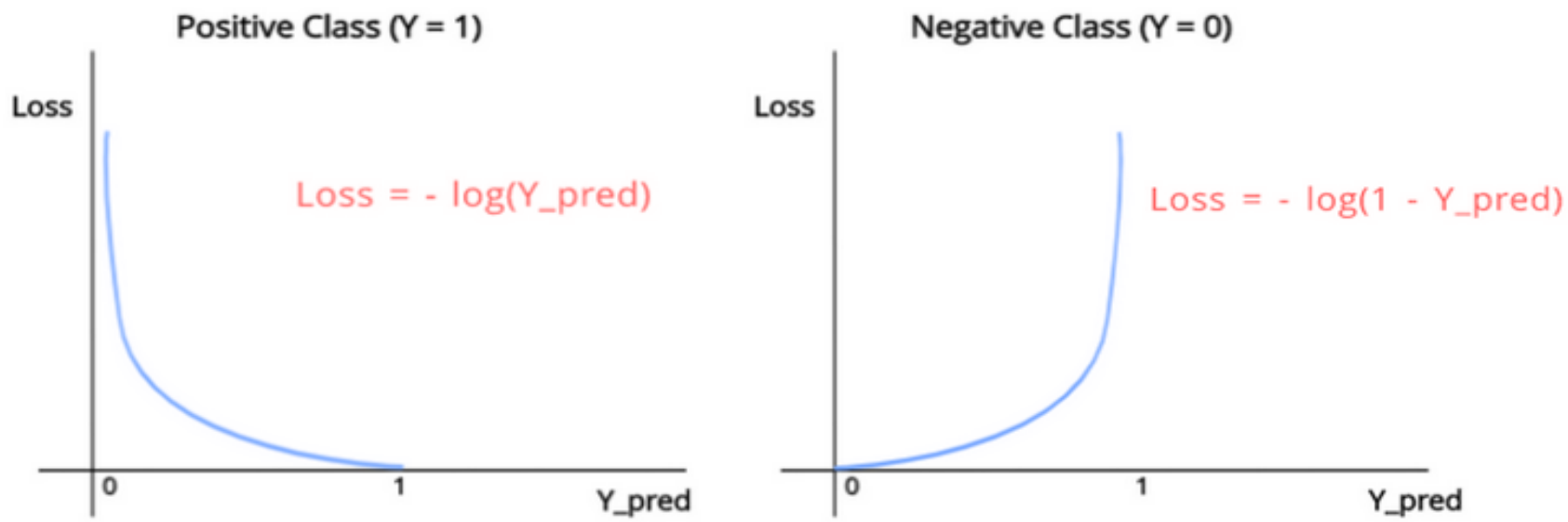
Sigmoid σ



Sigmoid Function Graph Visualization

Binary Cross Entropy Loss

The loss function we use for binary classification is called **binary cross entropy (BCE)**. This function effectively penalizes the neural network for binary classification task. Let's look at how this function looks.



Binary Cross Entropy Loss Graphs

Binary Cross Entropy Loss

This is the most common loss function used for classification problems. Cross-entropy loss increases as the predicted probability diverges from the actual label.

Mathematical Formulation:

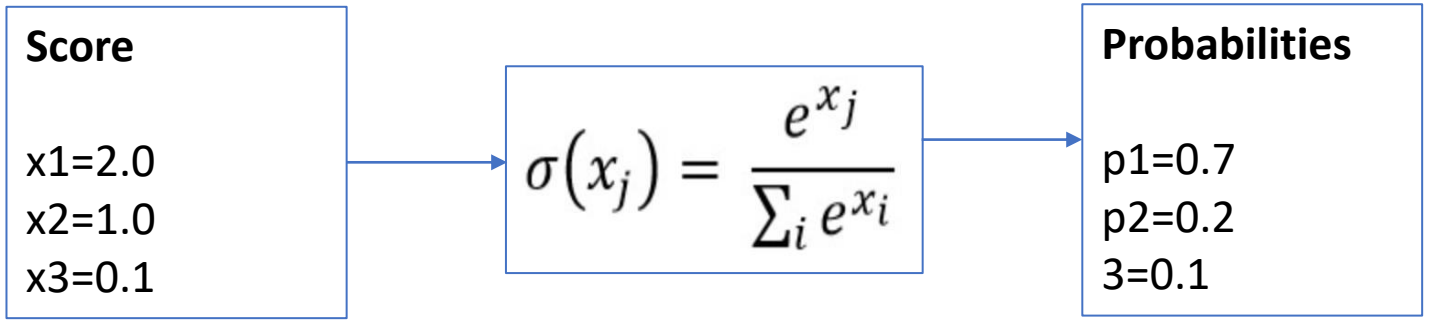
For binary classification

$$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)) \quad = \mathrm{L}(\tilde{y}(i) - y(i))$$

Notice that when actual label is 1 (y(i) = 1), second half of function disappears whereas in case actual label is 0 (y(i) = 0) first half is dropped off. In short, we are just multiplying the log of the actual predicted probability for the ground truth class. An important aspect of this is that cross entropy loss penalizes heavily the predictions that are *confident but wrong*.

Multiclass Classification Activation function
Softmax Function

The output layer is in charge of producing the probability of each class given the input image. To obtain these probabilities, initialize the final Dense layer to contain the same number of neurons as there are classes. The output of this dense layer then passes through the **Softmax activation function**, which maps all the final dense layer outputs to a probability distribution of a list of potential outcome with a vector whose elements sum up to one:



Where x denotes each element in the final layer's outputs.

Categorical Cross Entropy Loss

A common loss function to use when predicting multiple output classes is the **Categorical Cross-Entropy Loss function,** defined as follows:

$$L(y(i) - \tilde{y}(i)) = - \sum_{J=1}^{k} y(i)(j) \log (\tilde{y}(i)(j))$$

Where k is the number of categories

This loss function is used when labels are one hot encoded vectors.

Since only one position is one and all others zeros in the label vector we can write

$$L(\Theta) = - \sum_{i=1}^{m} y_i \log (\hat{y}_i)$$

Where m is the number of training examples

Sparse Categorical Cross Entropy Loss

Sparse Categorical Cross Entropy loss function is used when the target labels are single values such as integers.

When number of classes are very large, this can speed up the execution and save lot of memory by avoiding lots of logs and sum over zero values therefore more efficient than **Categorical Cross Entropy** loss function.

Neural Network Optimizers

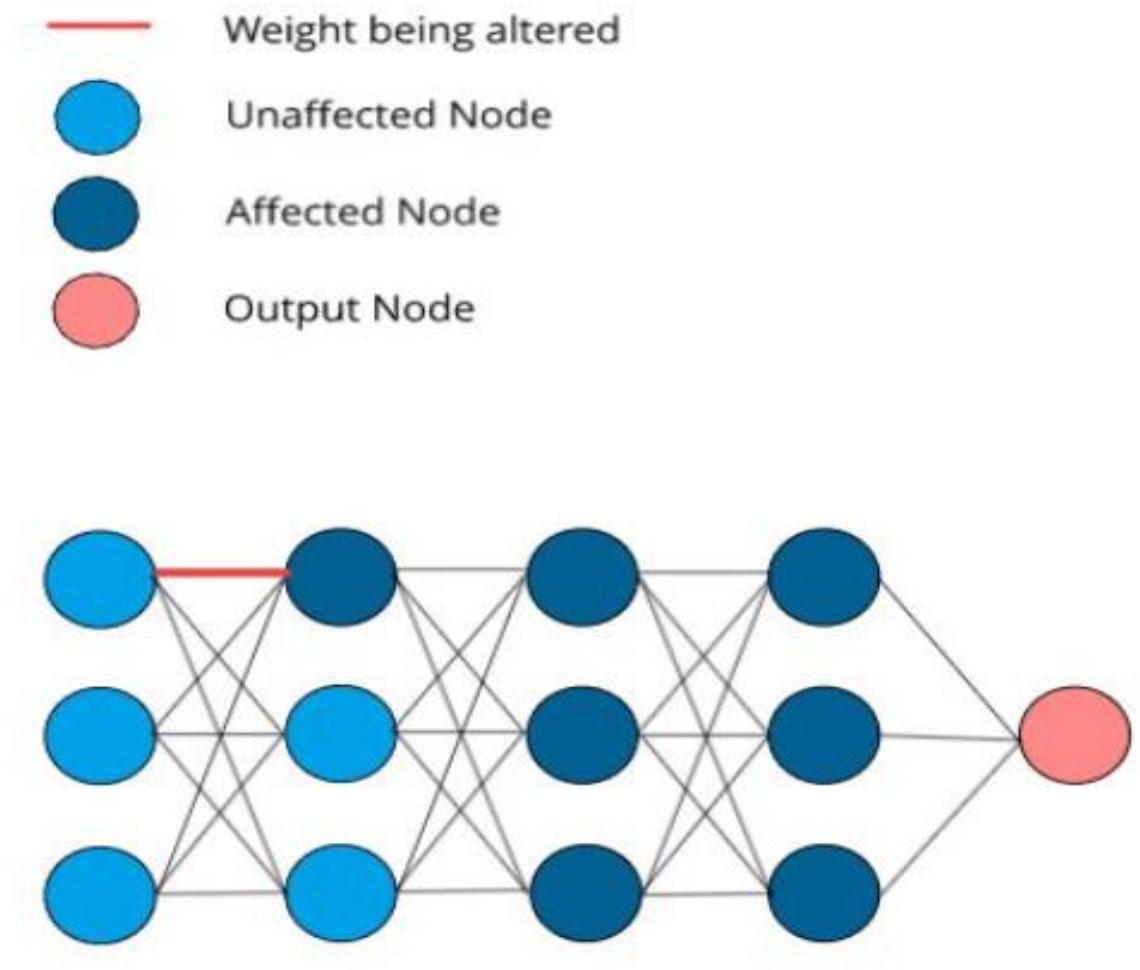
The purpose of the training stage of a NN is to find the set of model parameters which minimizes the Loss Function. The operation which perform this operation is called 'optimizer'.

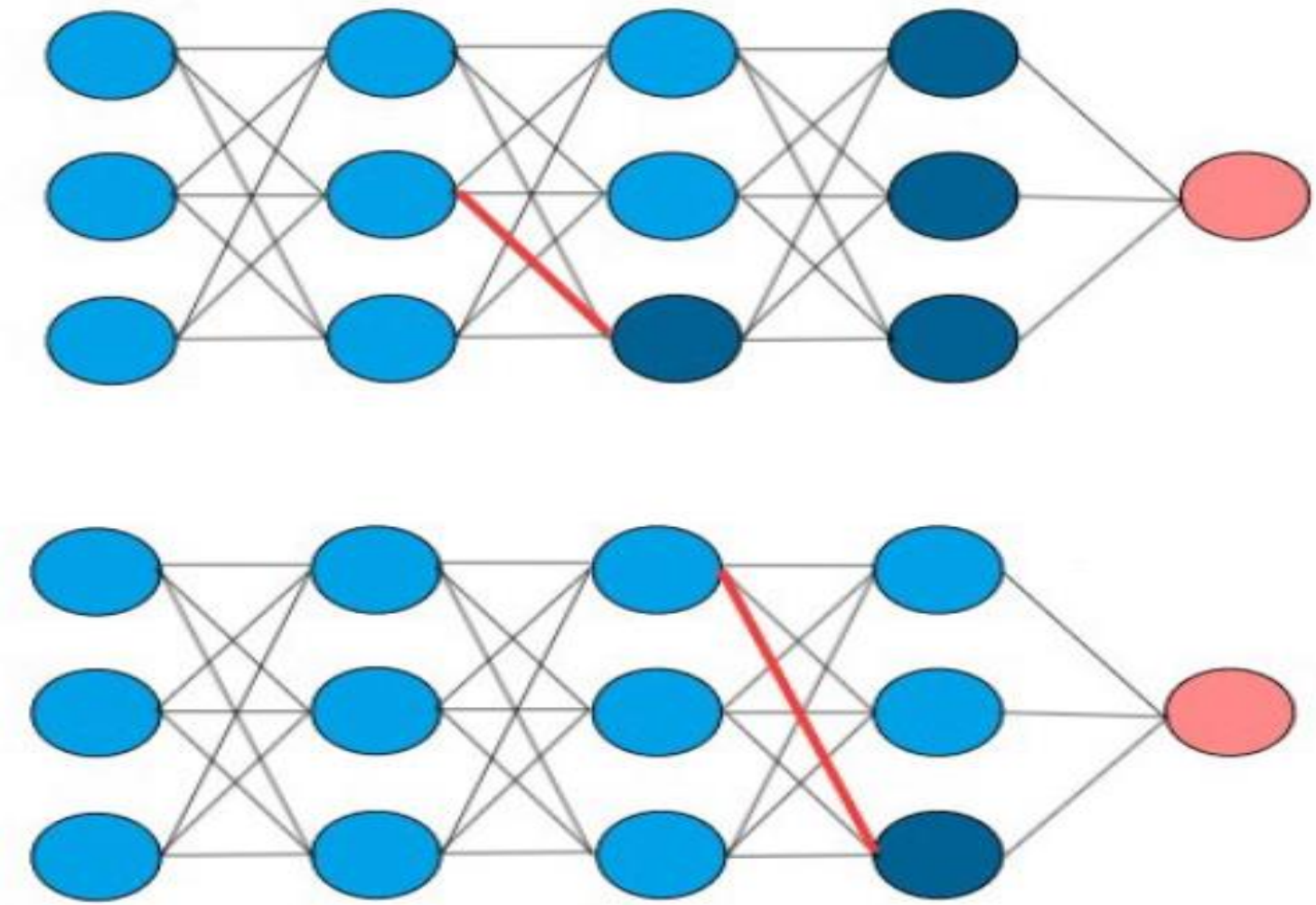
$$\textbf{Loss Function } J(W[1], b[1], W[2], b[2]) = (1/m) \sum_{i=1}^{m} L(\tilde{y}(i) - y(i))$$

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.

Hence we can say that if we change any particular weight in the neural network, the output of the network will also change.

Effect of changing model parameters

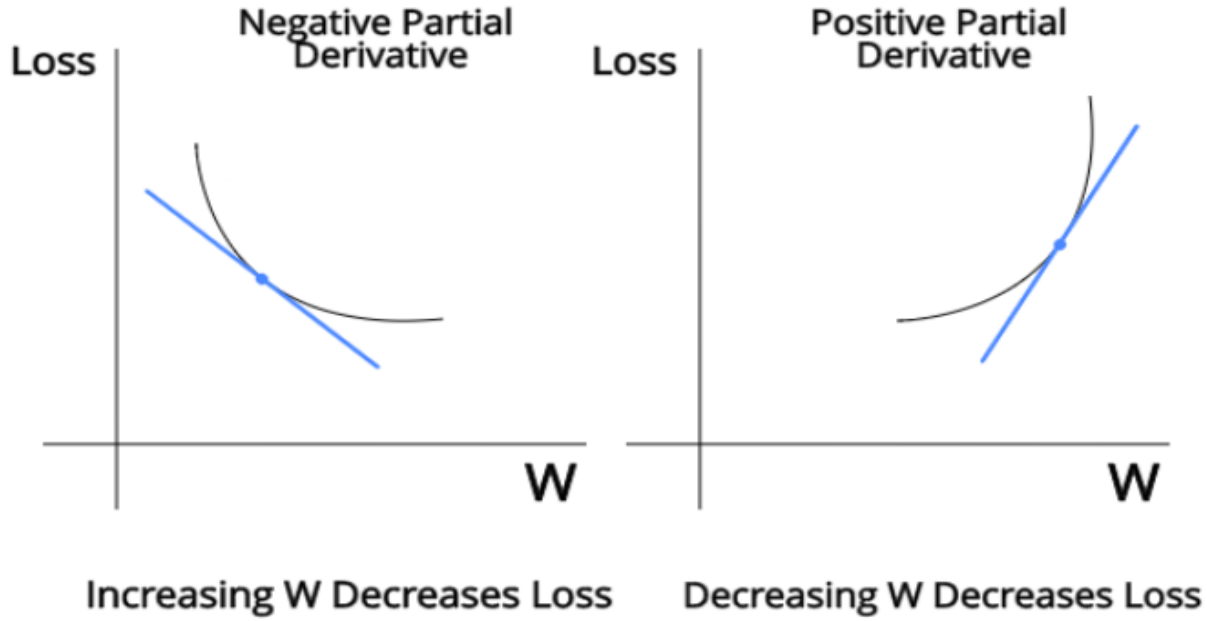




Visualizing which parts of a network altering particular weights will affect

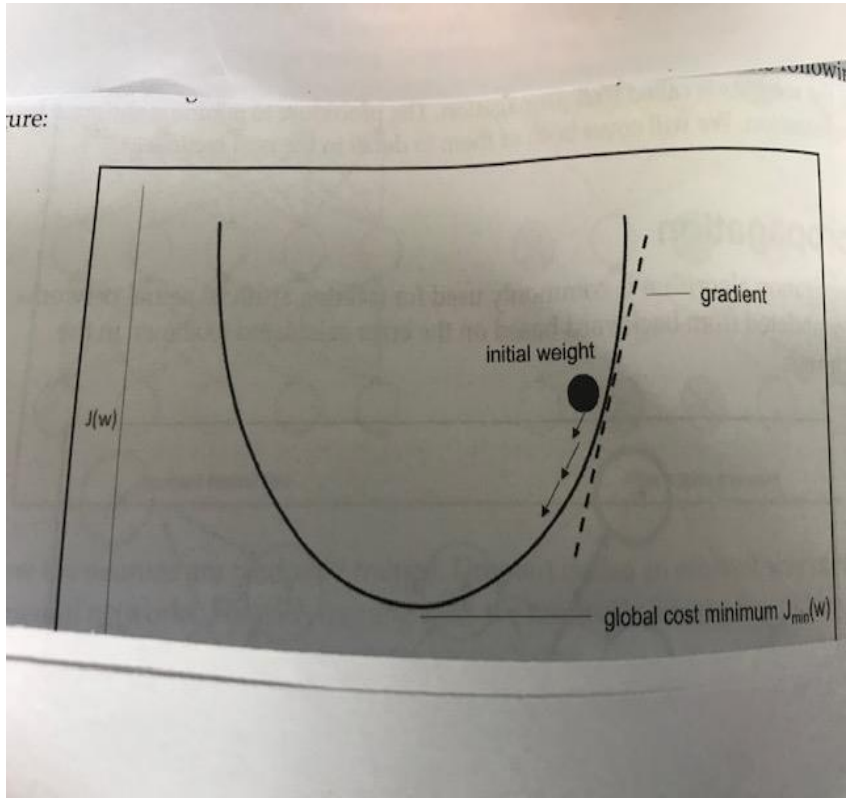
Gradient Descent

We need to find the partial derivatives of each and every **weight** in the neural network **with respect to** the 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. Our ultimate goal is to decrease the loss after all. This algorithm is called **Gradient Descent** or **Stochastic Gradient Descent (SGD)**



Gradient Descent

The Gradient Descent algorithm performs multidimensional optimization. The objective is to reach the global minimum of the loss function. It is used to improve or optimize the model prediction.



Gradient Descent

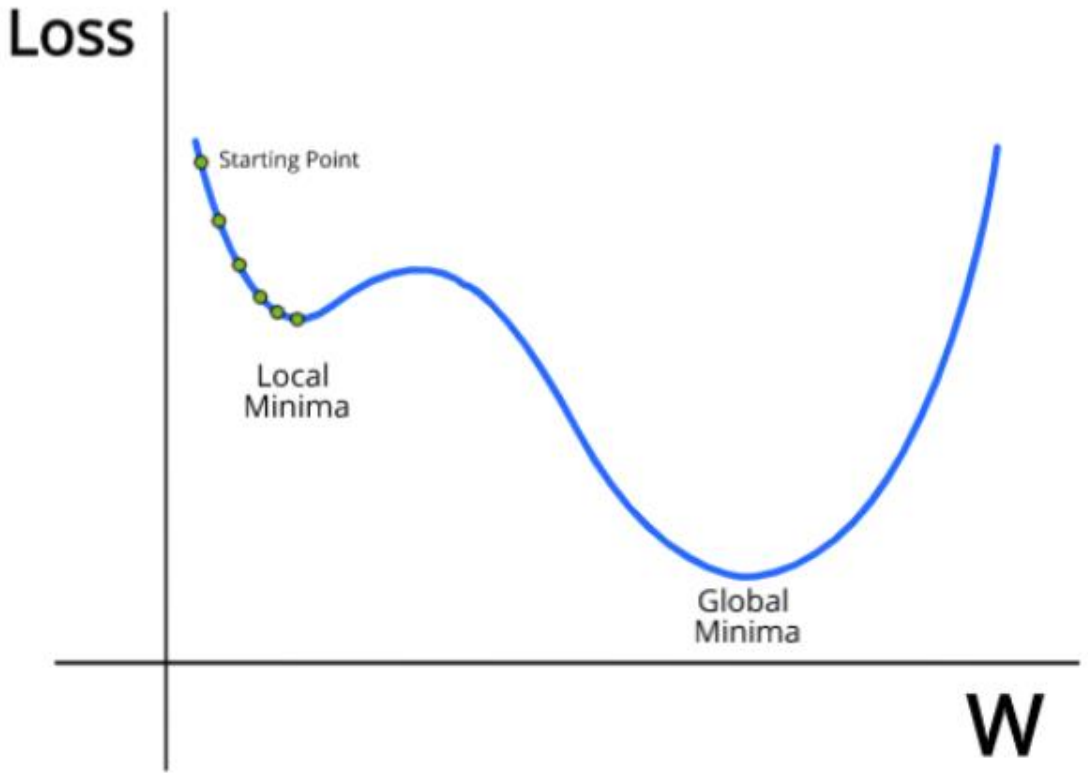
And this is the most basic method of optimizing neural networks. This happens as an iterative process and hence we will update the value of each weight **multiple** times before the Loss converges at a suitable value. Let's look at the mathematical way of representing a single update:

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

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.

SGD Local Minima Problem

There is, however, one big problem with just gradient descent. We are not pushing the loss towards the **global minima**, we are merely pushing it towards the closest **local minima.**



Here, 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 therefore the training process may not proceed beyond this point.

This is a rather simple graph and in reality the graph will be much more complicated than this with **many** local minimas 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**.

Momentum

In momentum, what we are going to do is essentially 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.

This approach can be used to the gradient descent algorithm to bypass local minima and try to reach the global minima.

Momentum

Mathematical formulation:

Momentum update

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

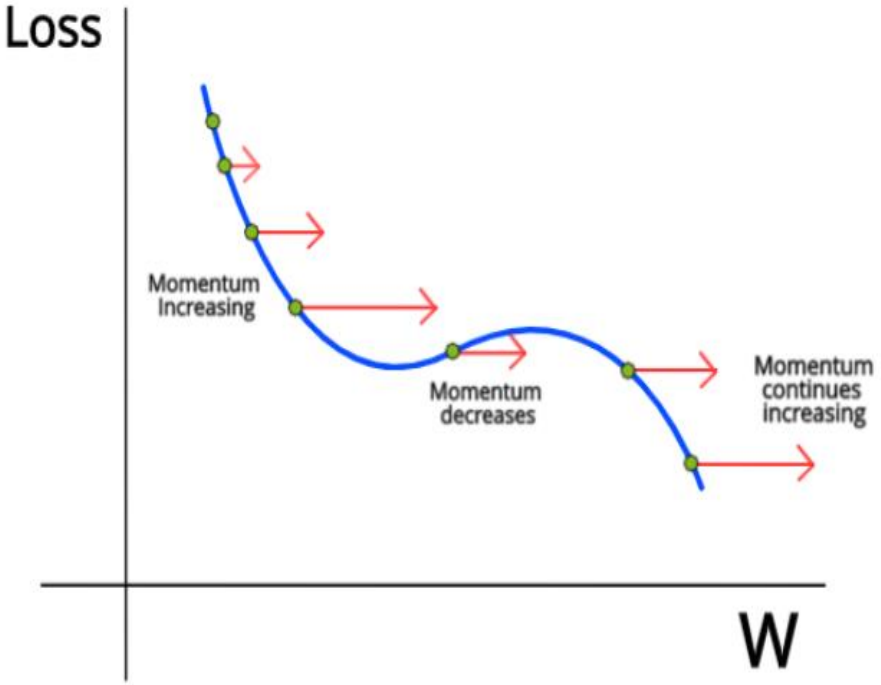
Wight update with momentum

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

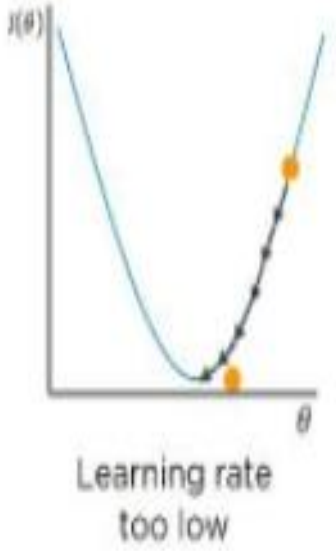
Here, V is the **momentum factor.** As you can see, in each update it essentially adds the current derivative with a part of the **previous** momentum factor. And then we just add this to the weight to get the updated weight. ɳ here is the **coefficient of momentum** and it decides how much momentum gets **carried forward** each time.

```
tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.01
)
```

As the weight gets updated, it will essentially store a *portion* of all the **previous gradients** in the momentum factor. Hence once it faces a change in the opposite direction, like a local minima, it will continue moving in the same direction until the magnitude of the momentum factor **gradually decreases** and starts pointing in the **opposite direction**. In most cases, the momentum factor usually is enough to make the weights overcome the local minima.

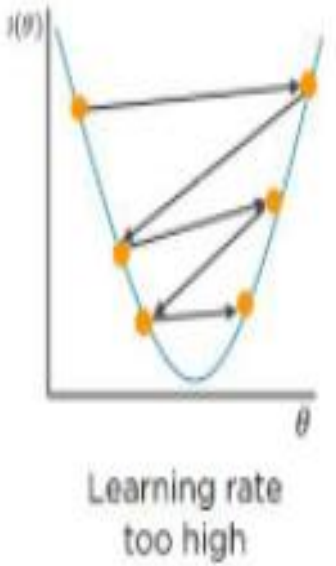


Learning Rate



Learning rate too low

Learning rate too low	Learning rate too high
• Training of the model will progress very slowly as we are making very tiny updates to the weights	• Causes undesirable divergent behavior to the loss function due to drastic updates in weights
• Will take many updates before reaching the minimum point	• At times it may fail to converge or even diverge



Learning rate too high

Adaptive Optimization

In these methods, the parameter learning rate 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
- RMSProp
- Adam

Adagrad

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

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

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

Cache update for Adagrad

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

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

Weight update

However, cache will become large even for a small gradient because of the square term. Therefore the learning rate will be very low and therefore the training process will become slow. The next adaptive optimizer, **RMSProp** effectively solves this problem.

RMSProp (Root Mean Square Propagation)

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

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

RMSProp cache 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

Adam is widely regarded as one of the best go to optimizers for deep learning in general.

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

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

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:

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

Adam cache Update Formula

Adam

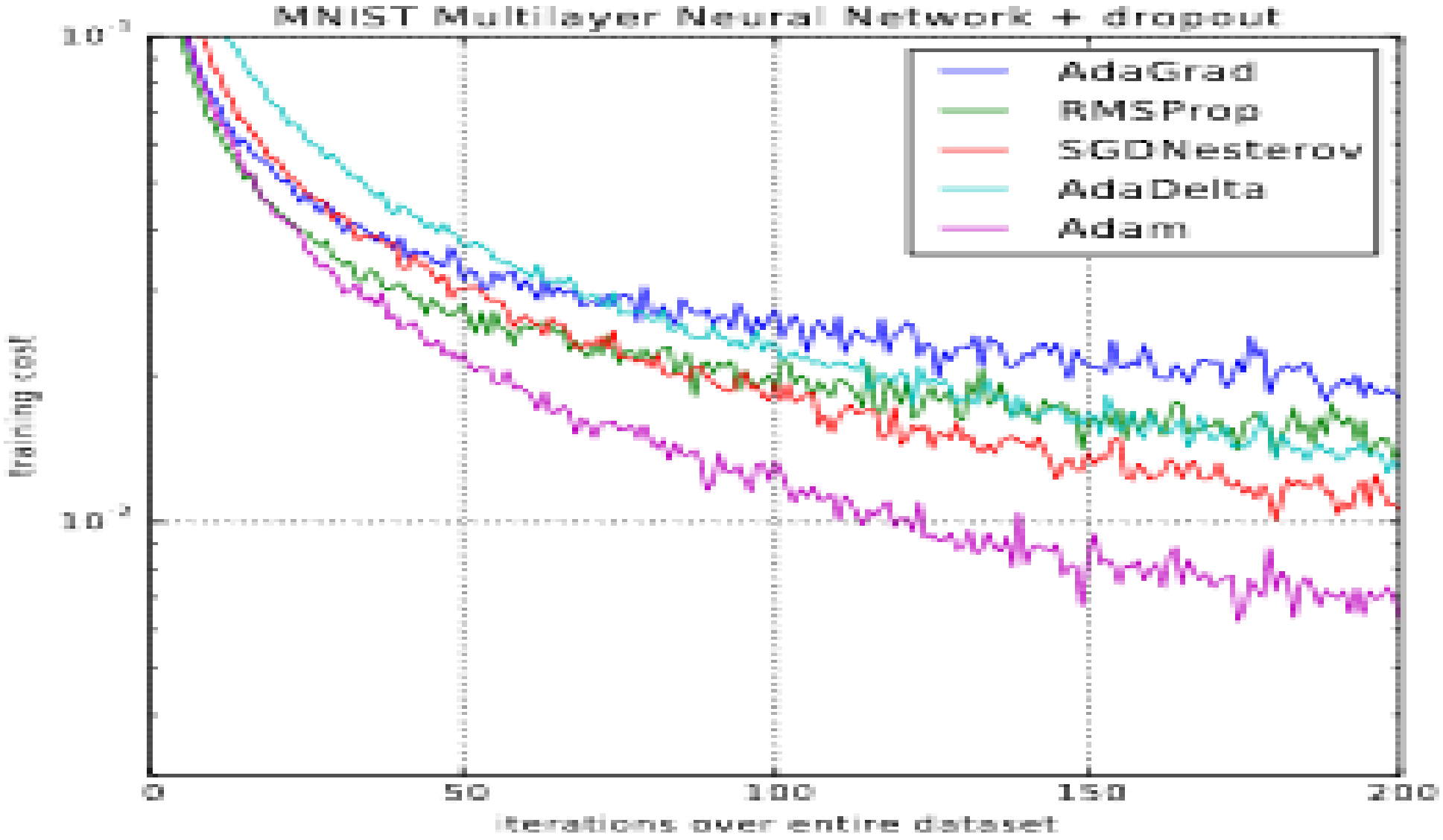
The weight update formula:

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

Adam weight update formula

The Adam is performing accumulation of the gradients by calculating momentum and also w 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 (https://arxiv.org/pdf/1412.6980.pdf) the recommended parameters are 0.9 for beta_1, 0.99 for beta_2 and 1e-08 for epsilon.



Source : Adam paper