INTRO TO DEEP LEARNING

What is Deep Learning?

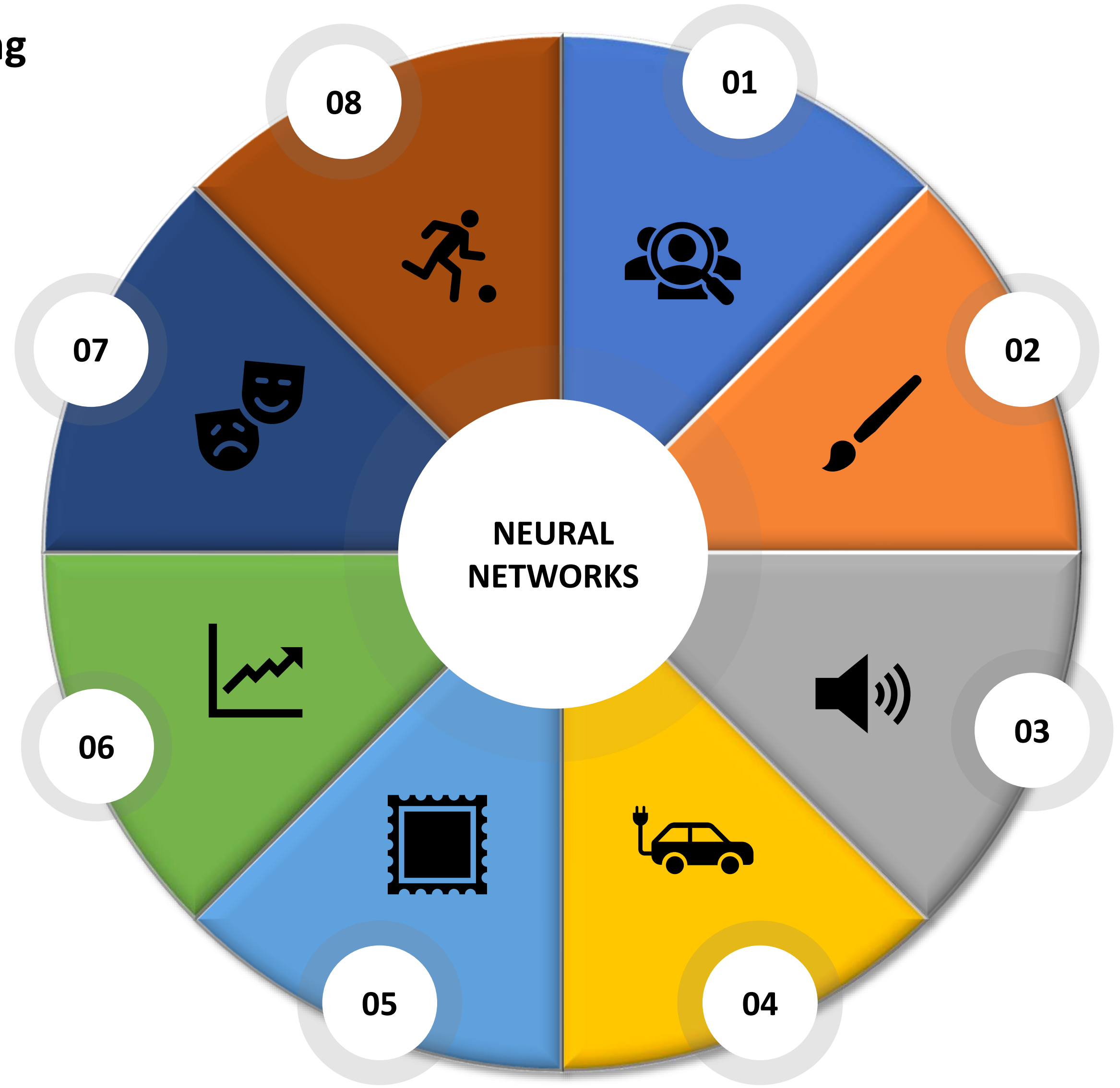
Deep Learning is a subset of artificial intelligence (AI) that mimics the workings of the human brain in processing data and creating patterns for use in decision making.
It is built around neural networks, which are algorithms modeled loosely after the human brain. These neural networks consist of layers of nodes, or "neurons," each layer designed to perform specific tasks and capable of learning from vast amounts of data.

How is Deep Learning different from Machine Learning?

Machine Learning	Deep Learning
• ML models are based on different techniques (Logistic Regression, SVM, Tree based models, etc)	• The core of DL models are Neural networks (Only the architectures may vary)
• Typically requires less data to train the model	• Requires large amount of data to be effective
• Can be less computationally intensive compared to deep learning	• Requires significant computational power (often GPUs) due to the complexity of the neural networks
• Models (especially simpler ones) tend to be more interpretable	• Models are generally considered "black boxes" because of their complexity

INTRO TO DEEP LEARNING

Applications of Deep Learning



- Computer Vision
- NLP
- Speech Analytics
- Self Driving Cars
- Robotics
- Regression & Classification
- Segmention & Product Recommendation
- Sports Analytics

RECAP

What do we need to know before we start learning Neural Networks?

- Python - Numpy, Pandas
- Maths - Derivatives (Good to know)
- Linear Regression (Theory)
- Logistic Regression (Theory)

RECAP

Linear Regression

The equation for Simple Linear Regression is :

$$y = a*X + b$$

where
Y = dependant variable
X = independent variable
a = weight
b = bias

Sample data

X1	X2	X3	y
20	30	10	58
30	10	25	67
21	15	31	78

Assuming the data is in the above format, the equation becomes :

$$y = a1*X1 + a2*X2 + a3*X3 + b$$

RECAP

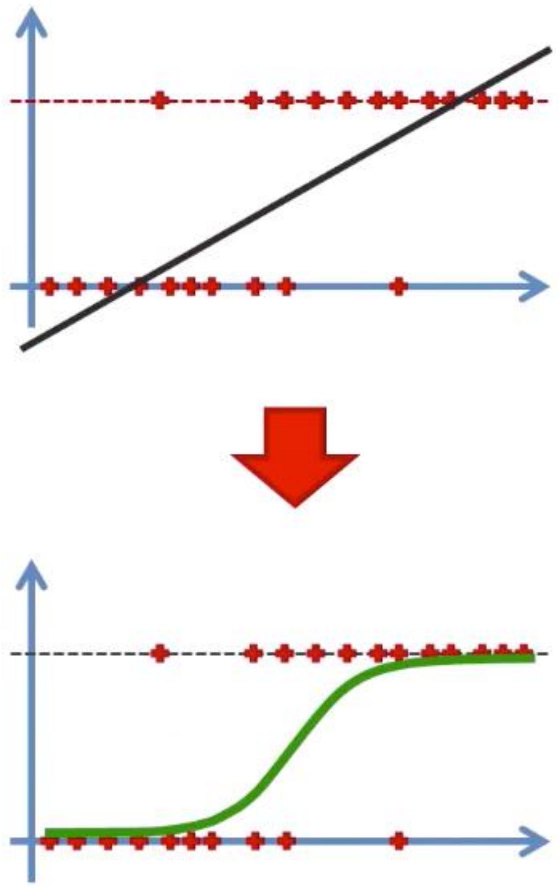
Logistic Regression

The equation for Linear Regression is :

f(z) : $y = a * X + b$

Applying Sigmoid Function to f(z) :

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



RECAP

Derivatives

Sigmoid Function: $\sigma(z) = \dfrac{1}{1+e^{-z}}$

$$\boxed{\text{Quotient rule:} \quad \frac{d}{dx} \cdot \frac{f(x)}{g(x)} = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}}$$

Taking derivative of the Sigmoid Function using Quotient Rule

$$\sigma'(z) = \frac{0-(-e^{-z})}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{\cancel{1+e^{-z}}}{\cancel{(1+e^{-z})}^2} - \frac{1}{(1+e^{-z})^2}$$

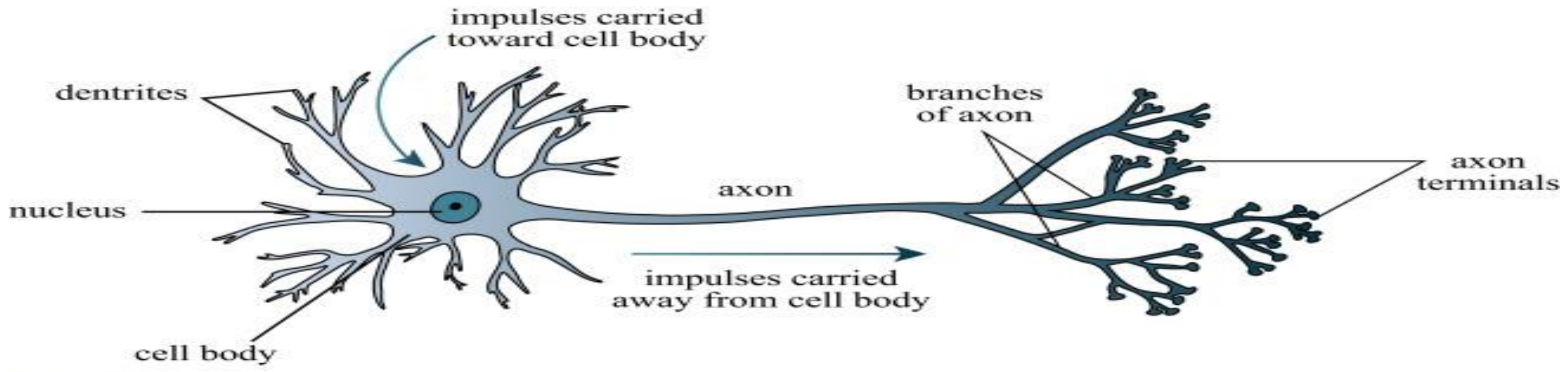
$$= \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2}$$

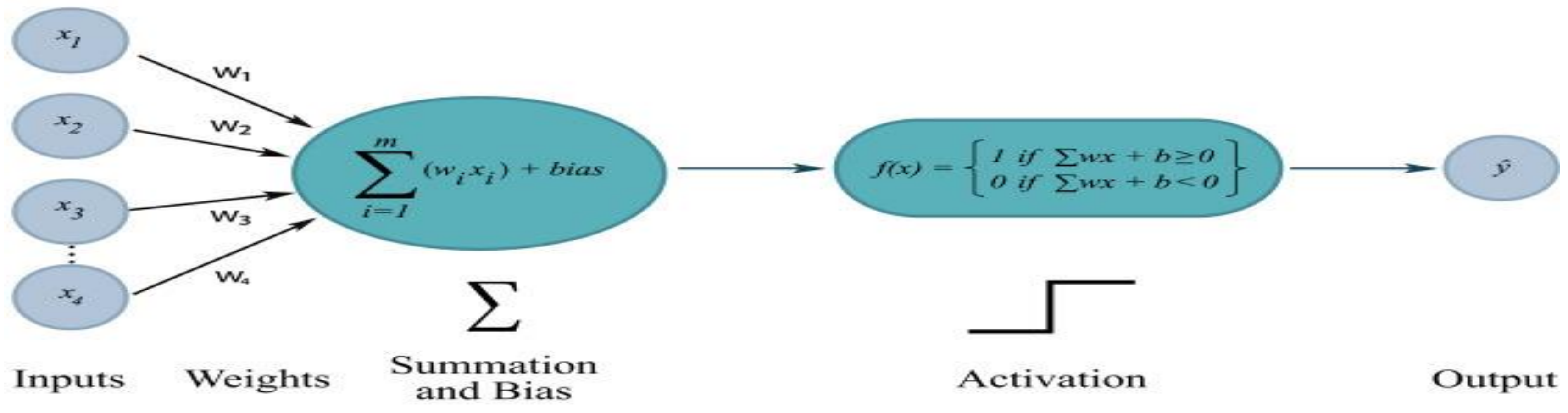
$$= \underbrace{\frac{1}{1+e^{-z}}}_{\sigma(z)}\left(1 - \underbrace{\frac{1}{1+e^{-z}}}_{\sigma(z)}\right) \longrightarrow \boxed{\sigma'(z) = \sigma(z)(1-\sigma(z))}$$

NEURAL NETWORKS

Biological Reference

Neurons process the information from different sources in the body, processes the data, and pass on the “refined” information to the next neuron / node to comprehend a much complex task and generate the required action.


impulses carried toward cell body
dentrites
nucleus
cell body
axon
impulses carried away from cell body
branches of axon
axon terminals

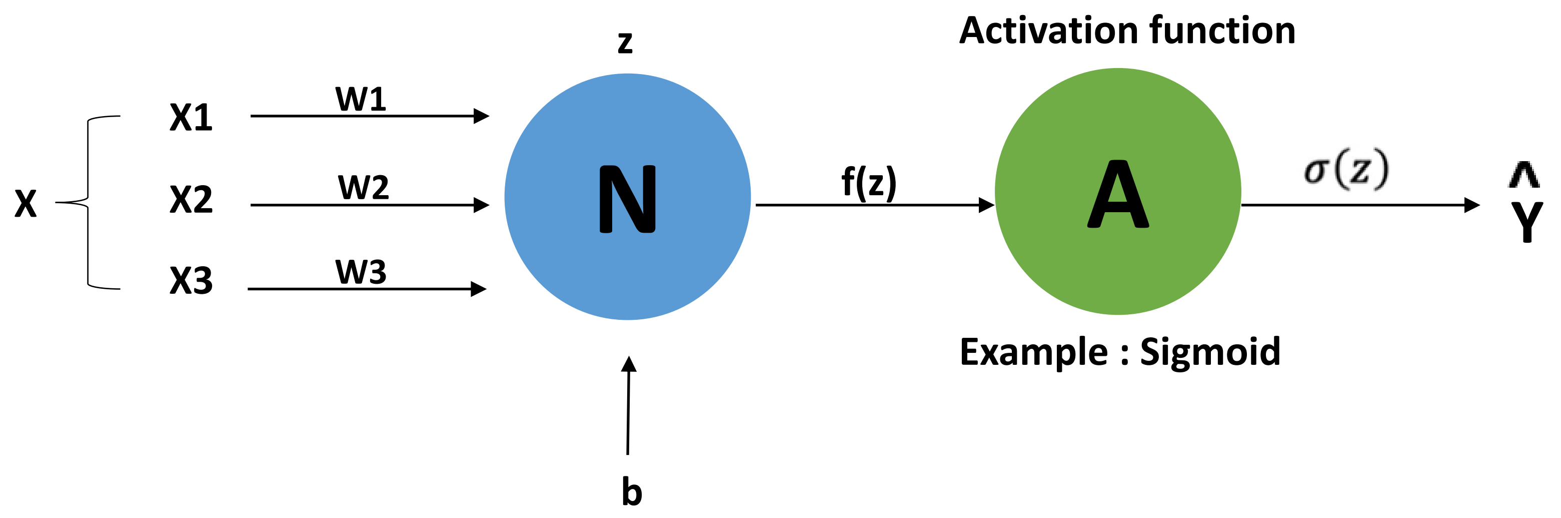

x_1
x_2
x_3
x_4
w_1
w_2
w_3
w_4
$\sum_{i=1}^{m}(w_i x_i) + bias$
$f(x) = \begin{cases} 1 \text{ if } \sum wx + b \geq 0 \\ 0 \text{ if } \sum wx + b < 0 \end{cases}$
$\hat{y}$
Σ
Inputs
Weights
Summation and Bias
Activation
Output

NEURAL NETWORKS

Fundamentals of Neural Networks

The basic architecture of a “unit” neural network consist of 3 stages :

- Input Layer
- Hidden Layers and Activation Function
- Output Layer

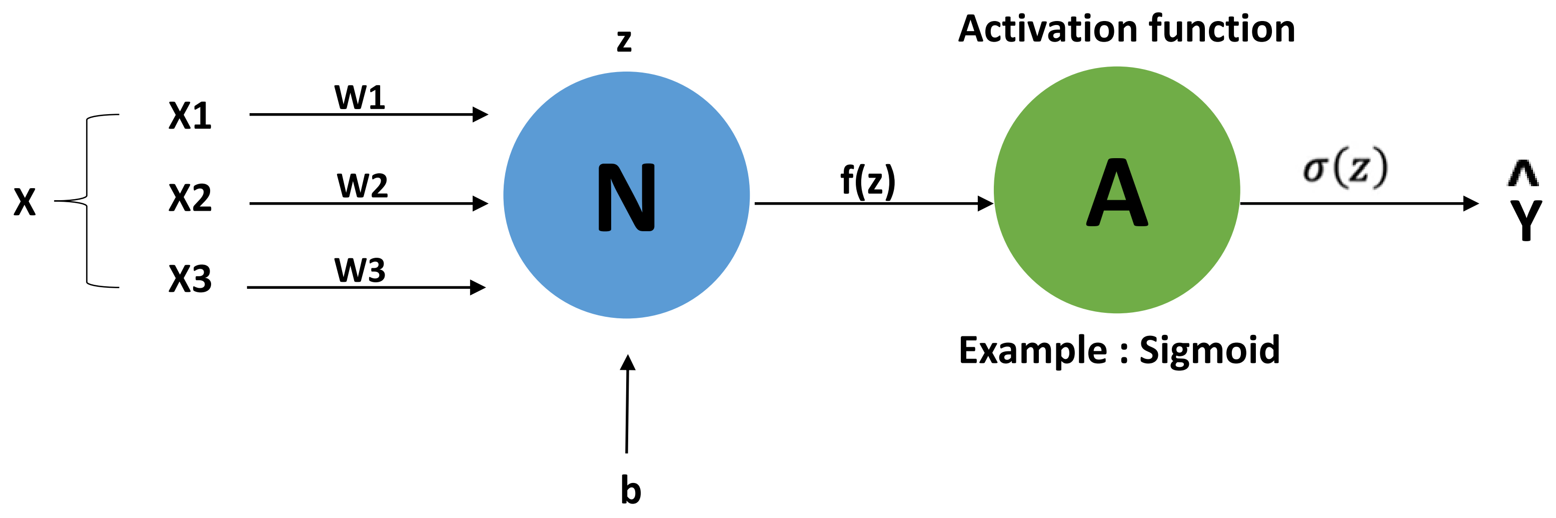


Note : This unit neuron architecture will give similar results to Logistic Regression as the working principle is same.
As there is a single node only, the number of hidden layers = 1 and number of nodes = 1.

NEURAL NETWORKS – WORKING OF A NEURON

Working of a Neuron

Let us assign some input values to the node, and calculate the output to understand how the computation works at neuron level.



X1 = 0.9	W1 = 2
X2 = 0.2	W2 = 3
X3 = 0.3	W3 = -1
	B = 0.5

z = w1*x1 + w2*x2 + w3*x3 + b

z = 0.9*2 + 0.2*3 + 0.3*(-1) + 0.5

z = 2.6

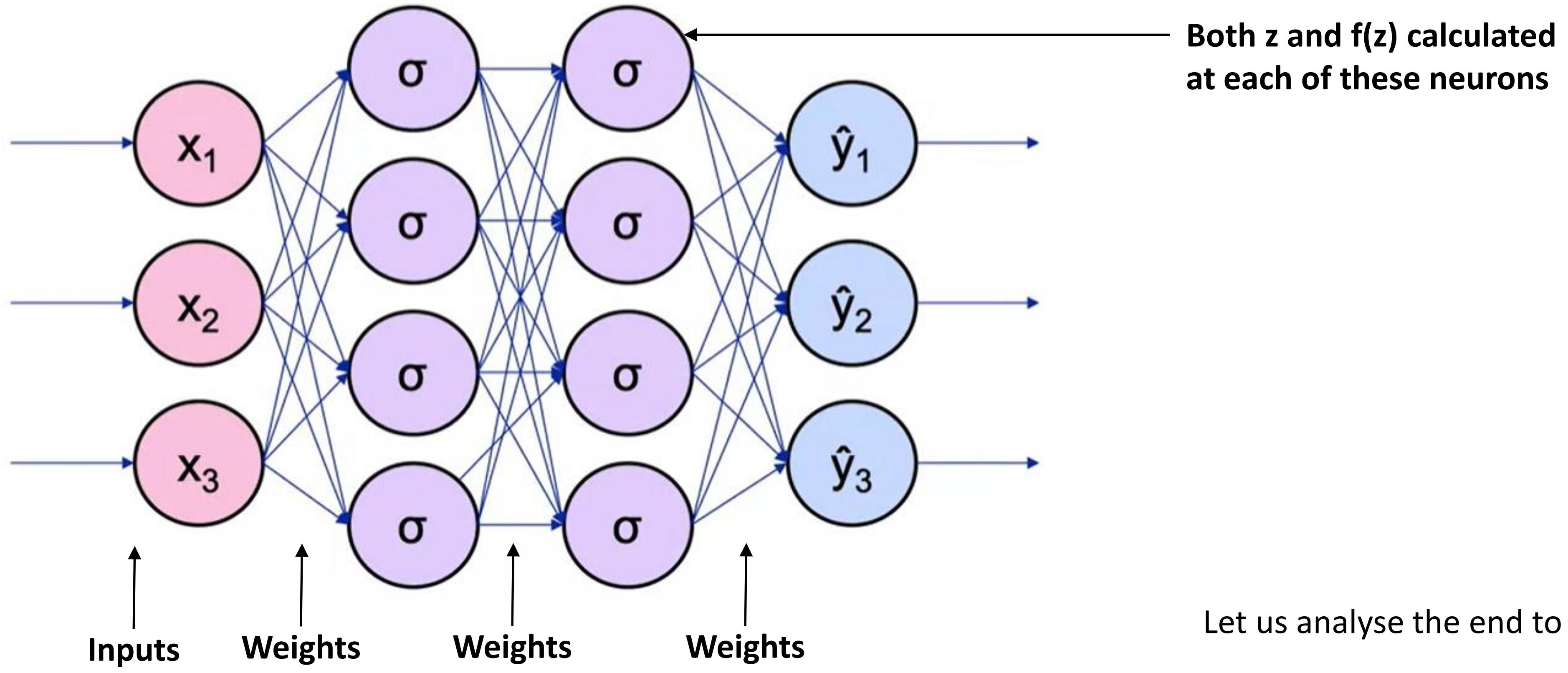
f(z) = f(2.6) = 1 / (1 + exp(-2.6))

f(z) = 0.93

NEURAL NETWORKS – FEED FORWARD NETWORK

Multi-layer Perceptron

Let us build an NN architecture of a multiclass classification model where
input features = 3,
hidden layers = 2 with nodes = 4 in each layer and
number of target classes = 3

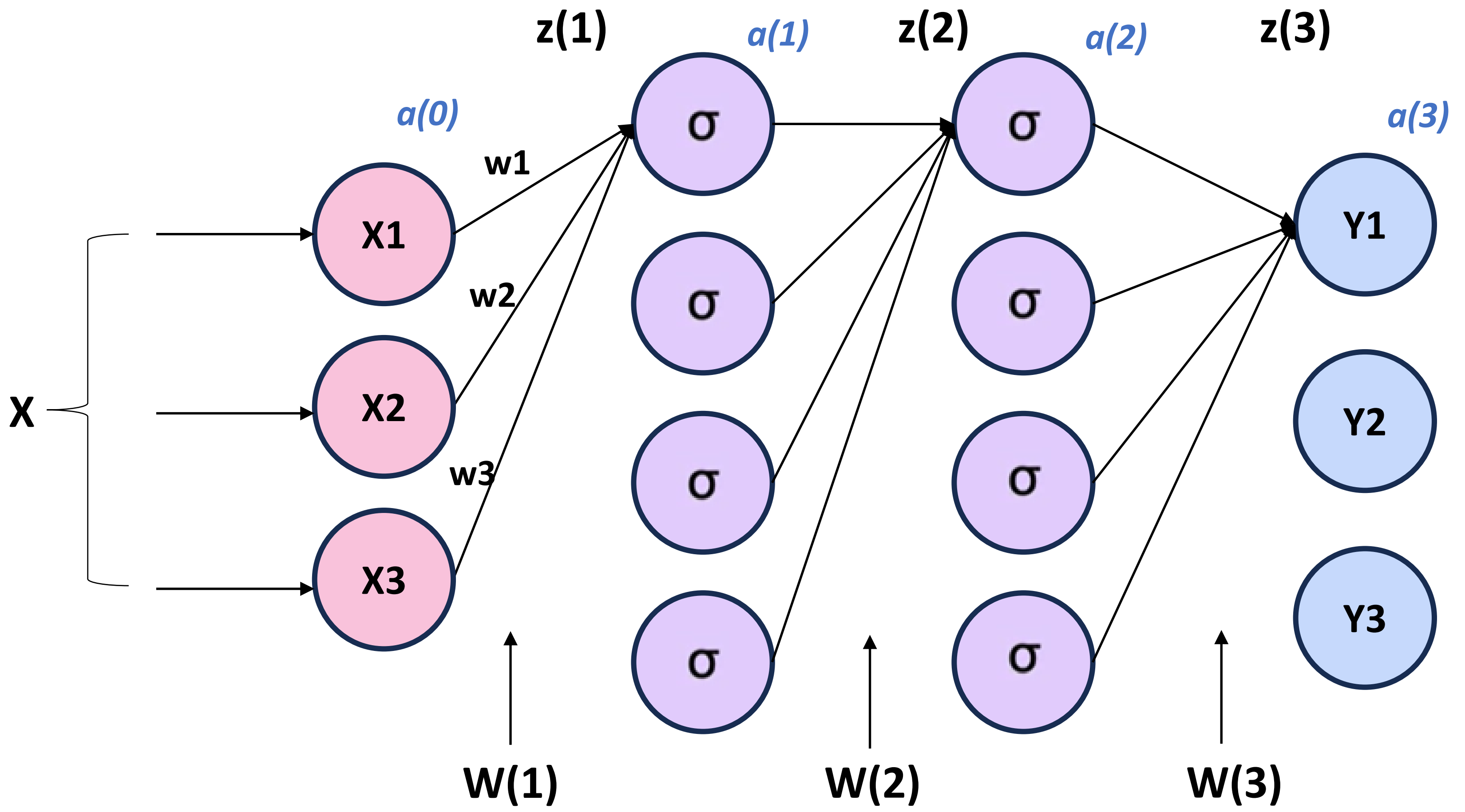


Let us analyse the end to end operation

NEURAL NETWORKS – FEED FORWARD NETWORK

Multi-layer Perceptron

input features = 3,
hidden layers = 2 with nodes = 4 in each layer and
number of target classes = 3



X : input dataset
X1, X2, X3 : Input variables

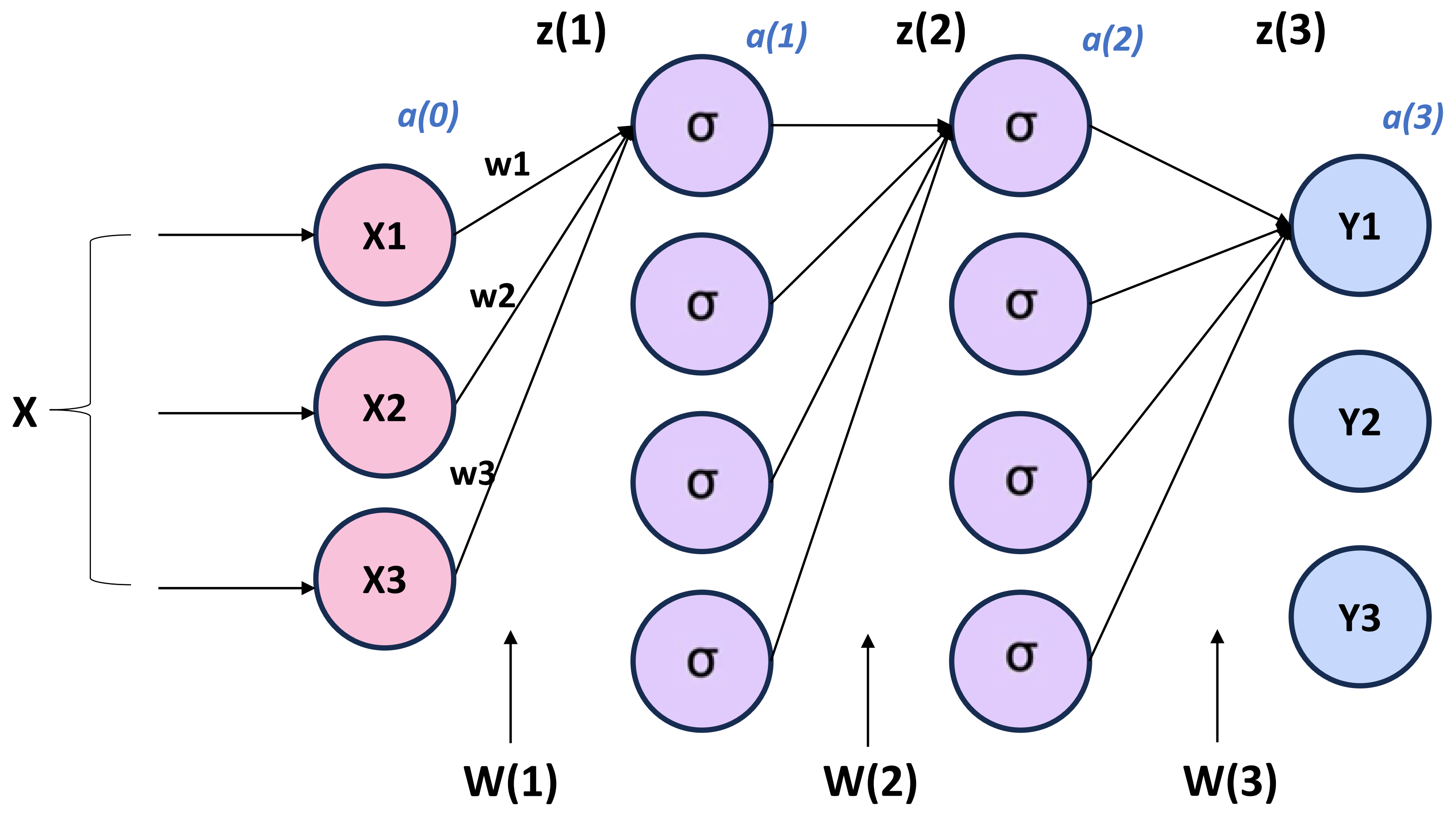
a : hidden / activation layer

W : weights matrix
W1, w2, w3 : individual weights

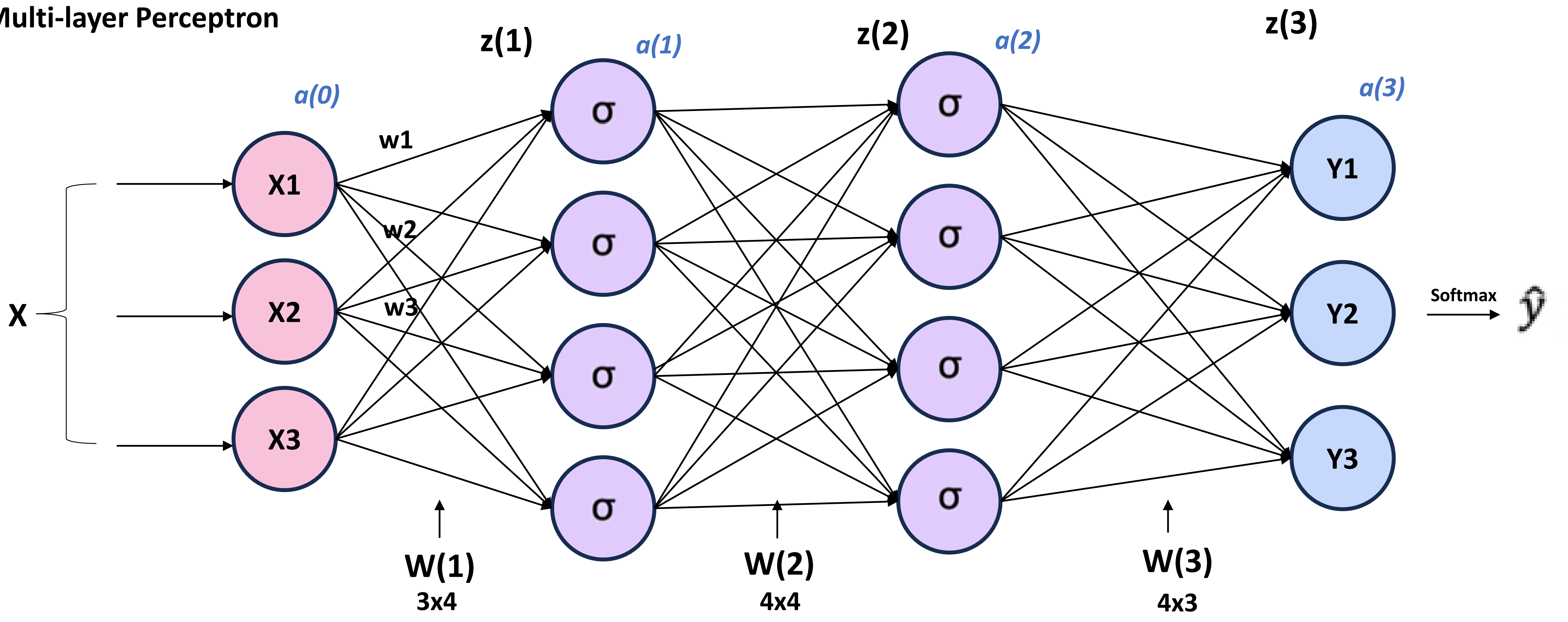
z : f(z) : y = w*X + b

NEURAL NETWORKS – FEED FORWARD NETWORK

Multi-layer Perceptron



NEURAL NETWORKS – FEED FORWARD NETWORK



$$z^{(1)} = xW^{(1)}$$
$$a^{(1)} = \sigma(z^{(1)})$$

$$z^{(2)} = a^{(1)}W^{(2)}$$
$$a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(3)}$$
$$\hat{y} = softmax(z^{(3)})$$

NEURAL NETWORKS – OPTIMIZATION

What does “Optimization” refer to in Neural Nets?

- **Initialization** - In a feed forward network, once the weights are assigned, the function f(z) is calculated and the output moves in a forward direction from one layer to another.
  The weights may or may not be the best combination for the node calculations, i.e. The weights “initialized” may not result in the best possible model at the final layer.
  We find the ‘*cost function*’ or ‘*loss function*’ or ‘*error*’ in the model to evaluate the model performance.

- **Weights second assignment** - Hence, after the first feed-forward network is built, the weights are then both increased & decreased to check how the model performance gets updated.

- **Re-Iterate** - Then we re-iterate the process in the same direction (either only increase or only decrease the weight) as long as the performance is improving.
  Each of the iterations are referred to as “steps”.

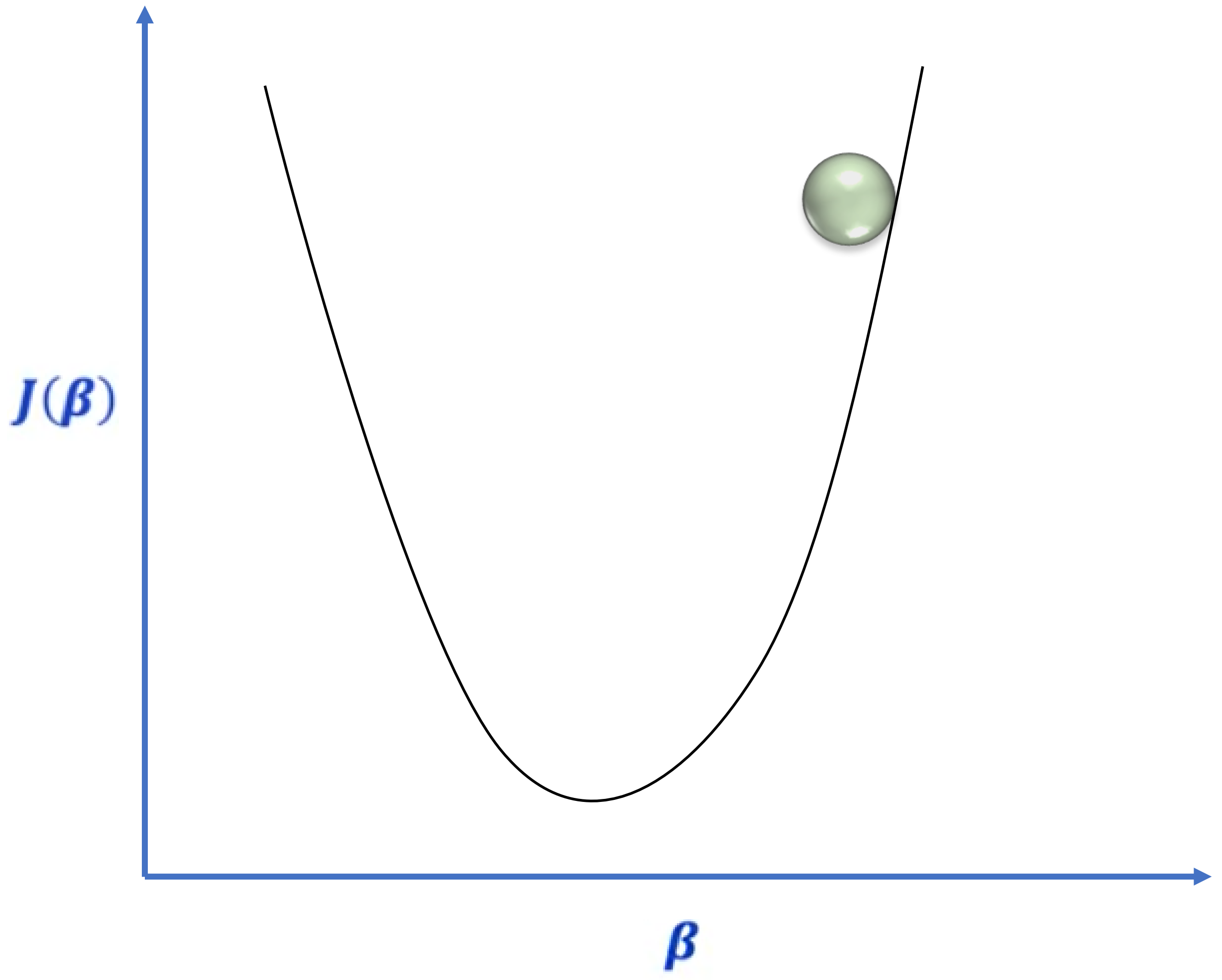
- **Peak Performance** - After a certain point, the model performance starts to decrease.
  This point of peak performance is called ‘**Gradient Descent’**.

Optimization refers to the method of finding the Gradient Descent in a neural network.

NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

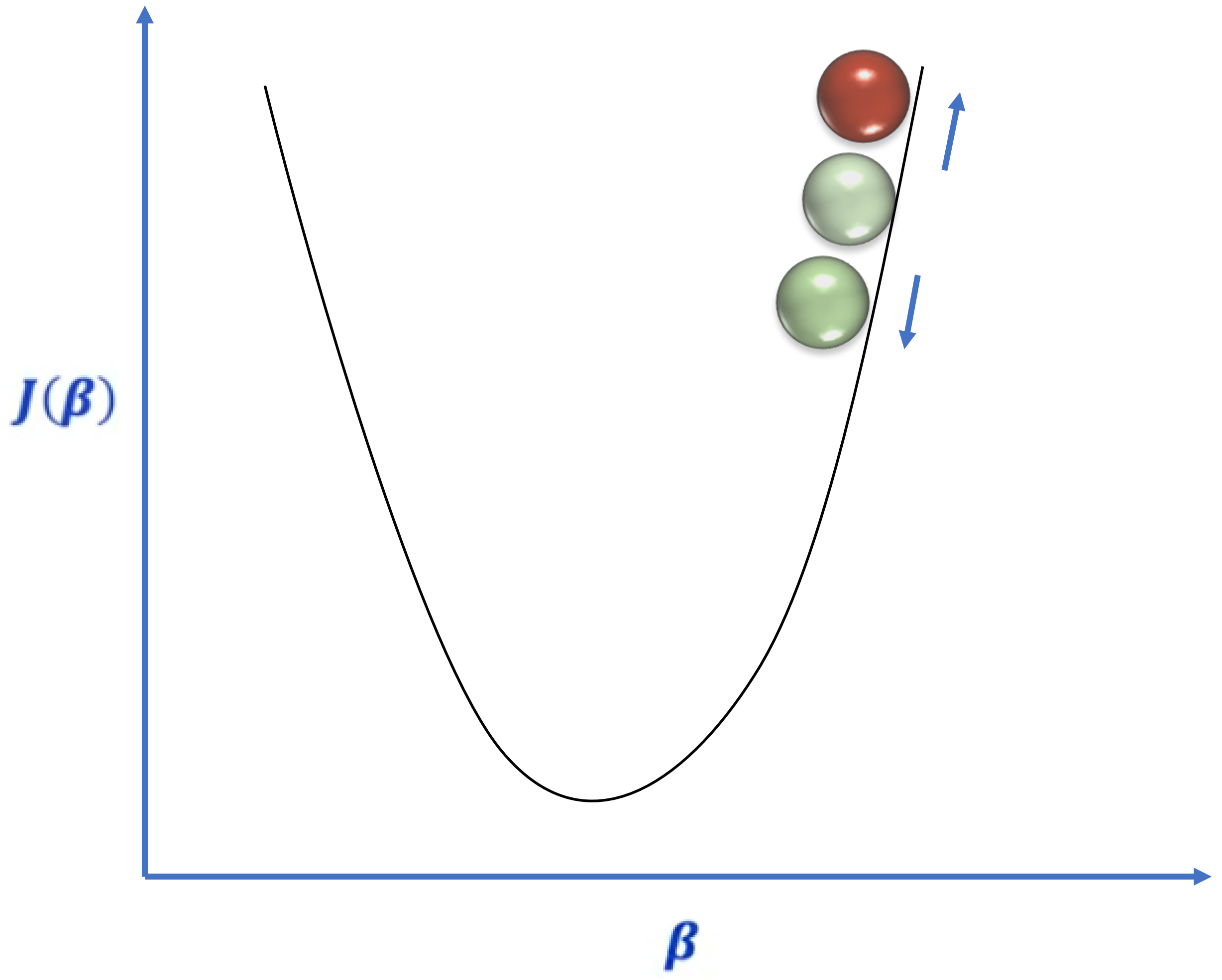


NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction



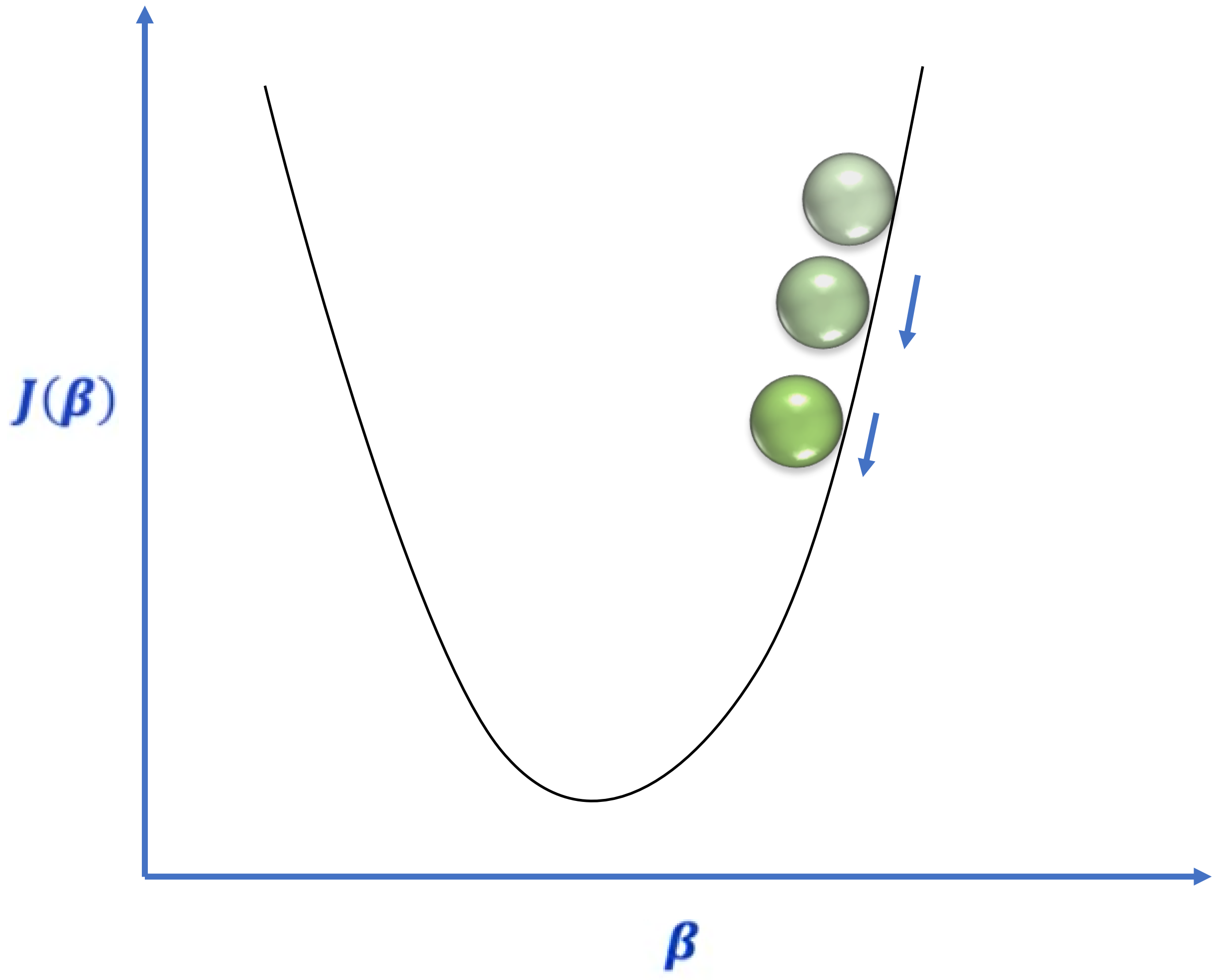
NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β)** is calculated at each 'step' to validate thar loss is getting reduced.



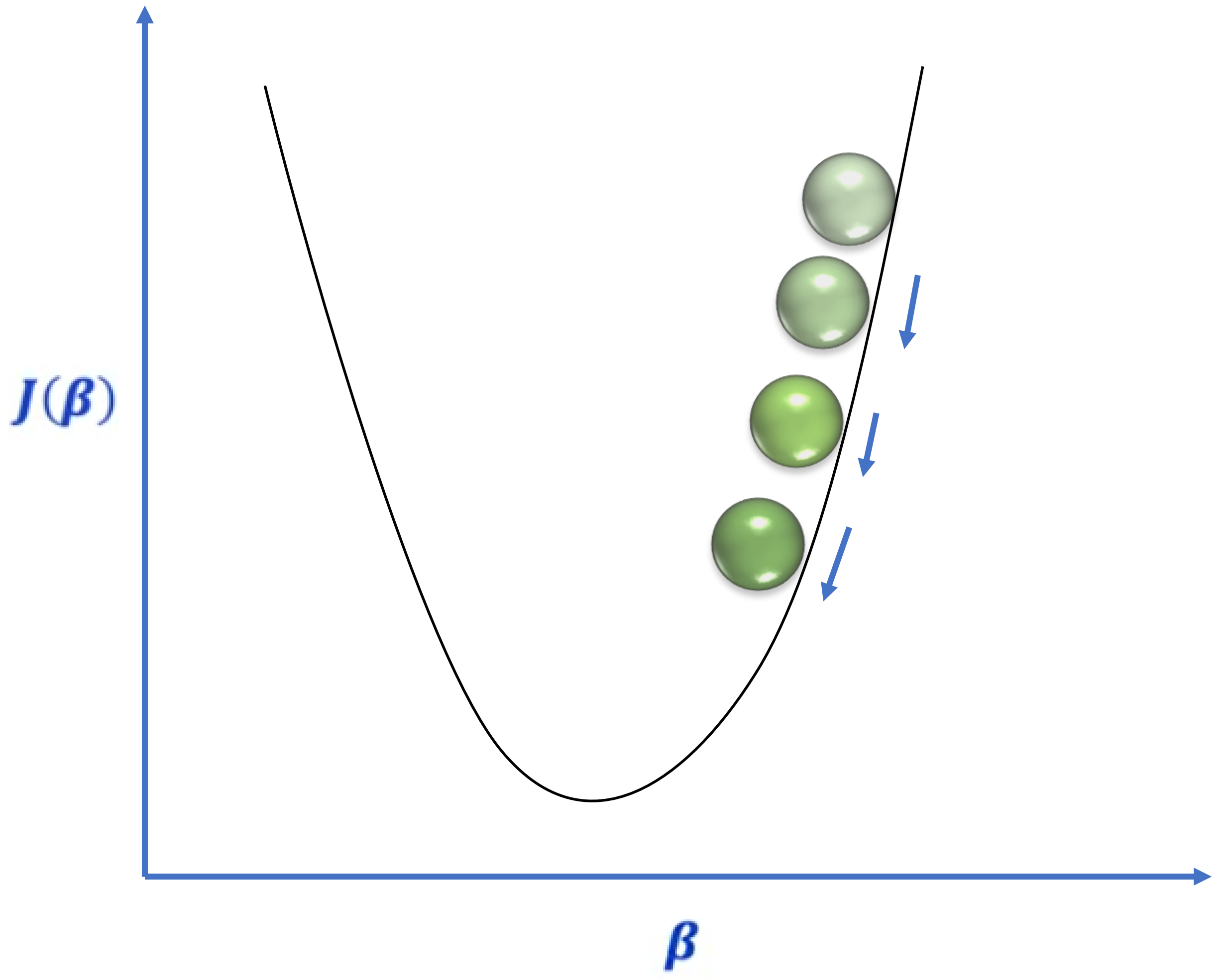
NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β)** is calculated at each 'step' to validate thar loss is getting reduced.



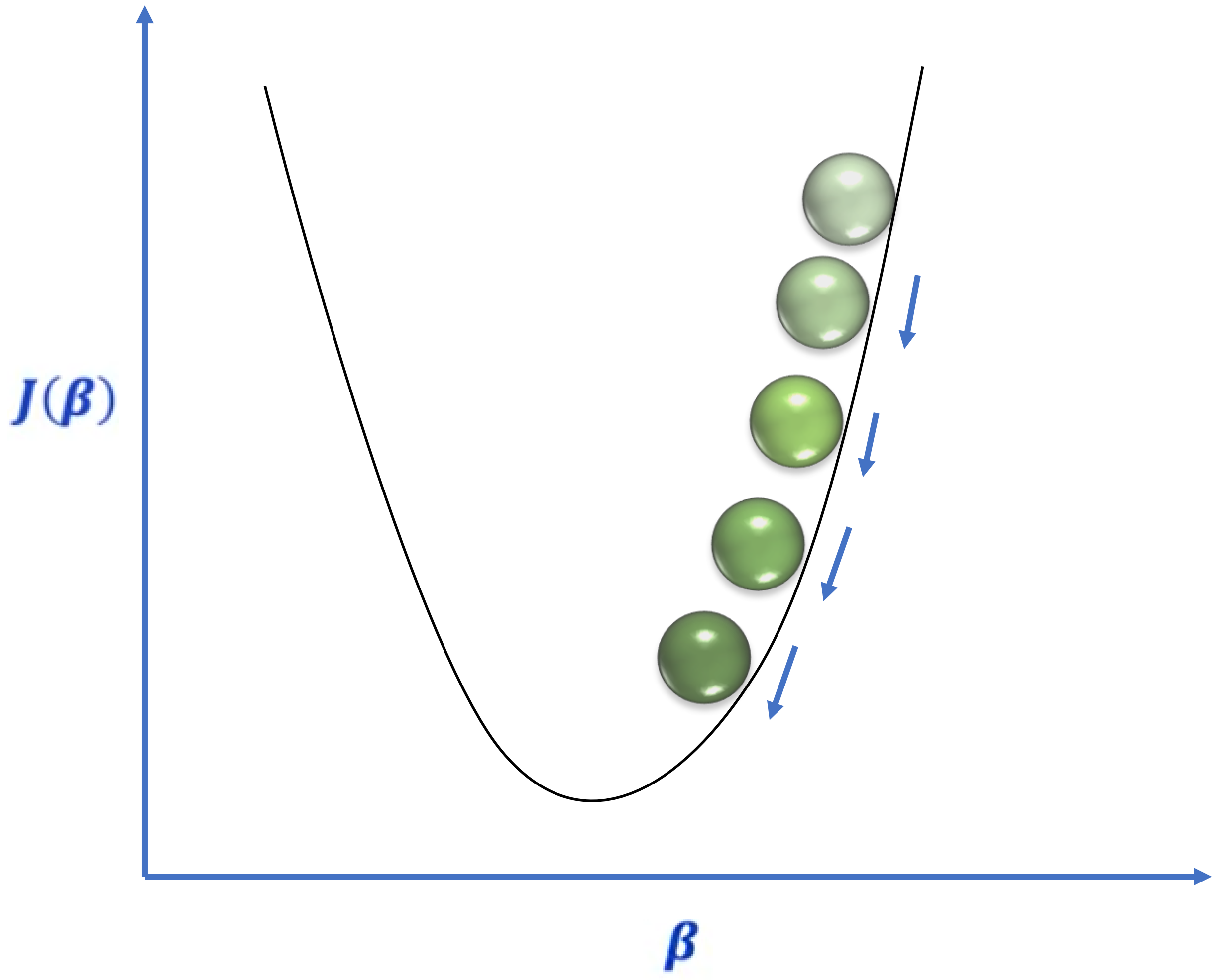
NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β)** is calculated at each 'step' to validate thar loss is getting reduced.



NEURAL NETWORKS – GRADIENT DESCENT

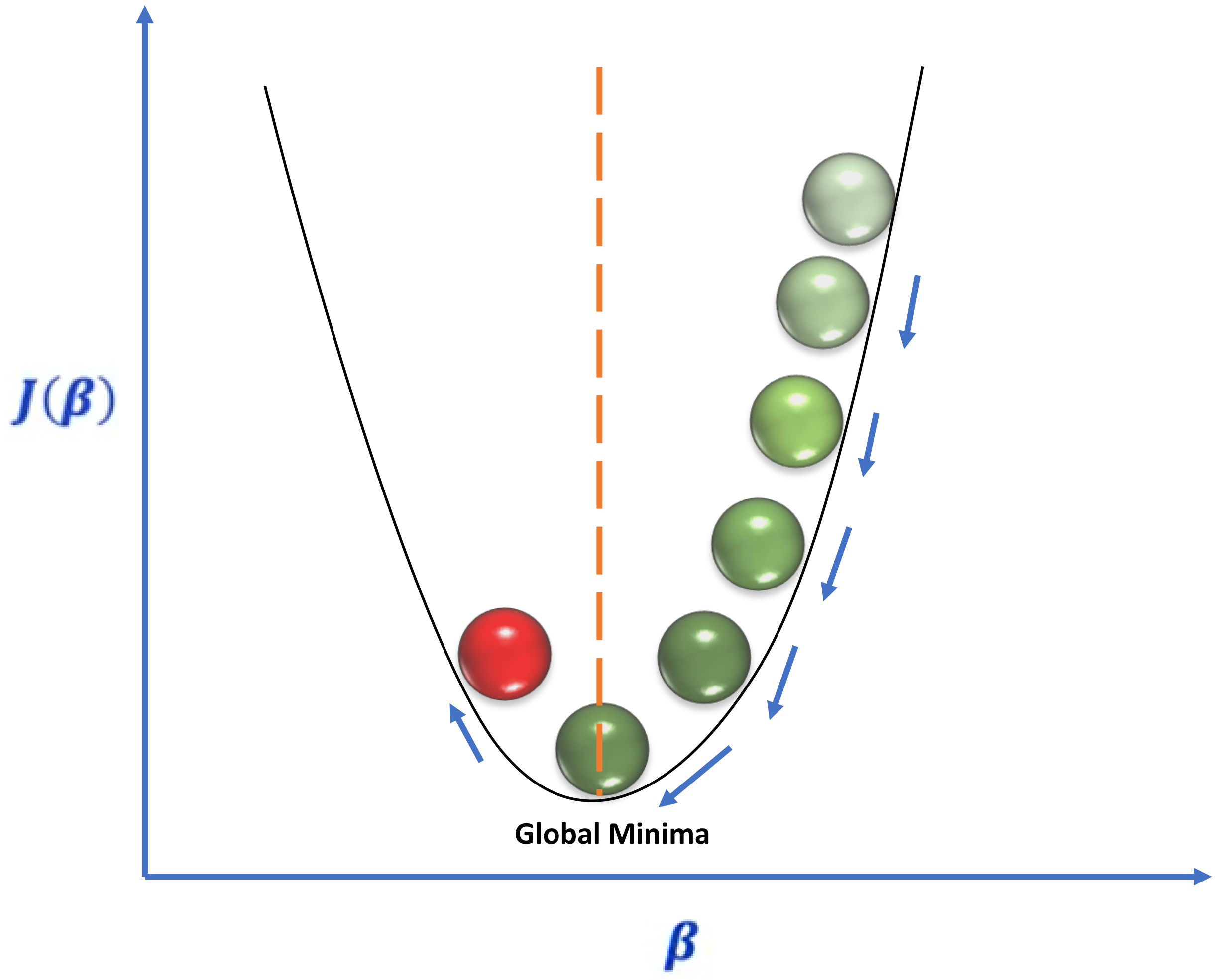
Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β)** is calculated at each 'step' to validate thar loss is getting reduced.

Step 4 : Peak Performance / Gradient Descent
Once the global minima is reached (*<u>**J(β)** is minimum</u>*), we can say that the Gradient Descent is achieved and the model is optimized.



NEURAL NETWORKS – GRADIENT DESCENT

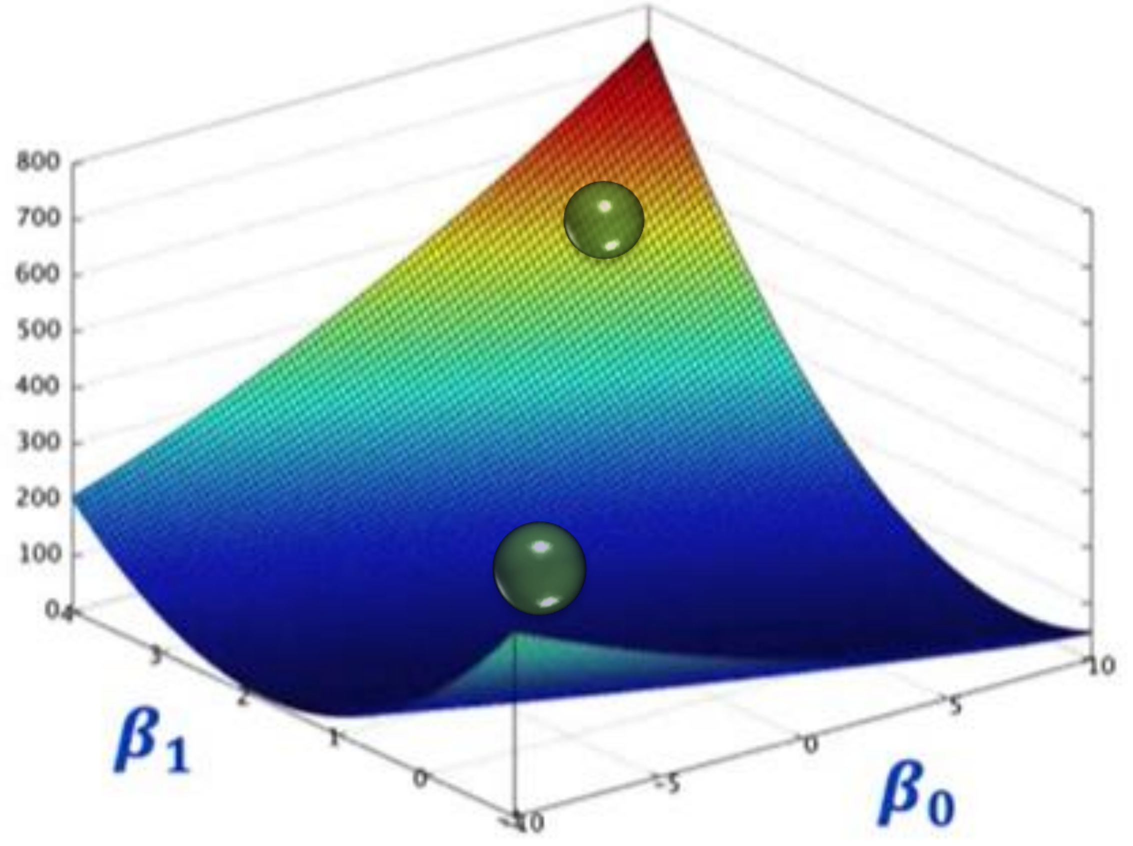
Working of Gradient Descent with (β0, β1):

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β0, β1)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β0, β1)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β0, β1)** is calculated at each ‘step’ to validate thar loss is getting reduced.

Step 4 : Peak Performance / Gradient Descent
Once the global minima is reached (<u>**J(β0, β1)** *is minimum*</u>), we can say that the Gradient Descent is achieved and the model is optimized.



NEURAL NETWORKS – GRADIENT DESCENT

Types of Gradient Descent

There are 3 types of Gradient Descent –

- Full Batch Gradient Descent
- Stochastic Gradient Descent
- Mini Batch Gradient Descent

The purpose of having different types of GD is only to have a system with balanced space and time complexity.

Full batch Gradient Descent

- Uses all the data (without sampling) to run each step.
  This makes the space complexity very high.

- Since it uses all the data at once, the number of steps / iterations required are significantly lower.
  This makes the time complexity low.

- Path to gradient descent is more direct.

NEURAL NETWORKS – GRADIENT DESCENT

Stochastic Gradient Descent

- Uses only one data point to run each step.
  This makes the space complexity very low.

- Since it uses only one data point at once, the number of steps / iterations required are significantly high.
  This makes the time complexity high / the system is very slow.

- Path to gradient descent is very complex. Prone to noise in the data.

Mini batch Gradient Descent

- Uses sampling technique to run each step in small batches of the data.
  This makes the space complexity comparatively lower than Full batch GD.

- Since it uses the data in small batches, the number of steps / iterations required are still higher than Full Batch GD but much faster than Stochastic GD.
  This makes the time complexity moderate.

- Due to having a moderate space as well as time complexity, this is often considered as the best of both worlds.

NEURAL NETWORKS – BACK PROPOGATION

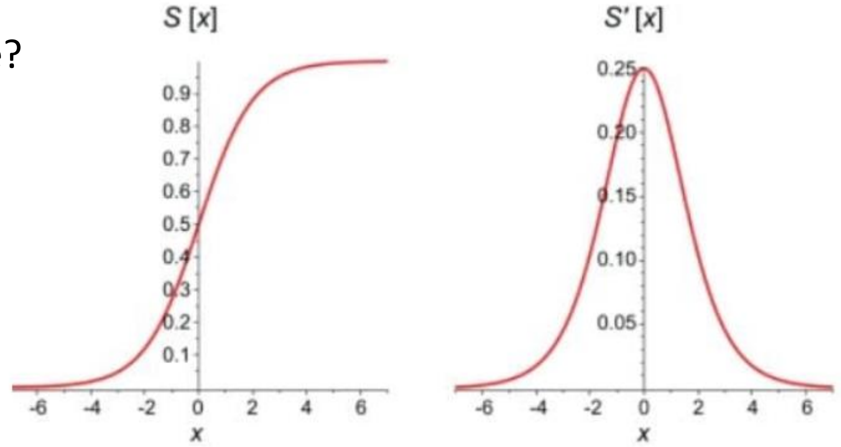
We have studied how to train Neural Networks.

- Put in the input variables.
- Initialize Weights and get output
- Calculate Loss Function. Adjust weights (increase or decrease) and repeat to achieve the Gradient Descent.

But how do we actually change the weights? How do we know how much to change?

$$W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$$

where
η = learning rate (usually very small like 0.1)
C = cost function



With the chain rule of partial derivatives, we can represent gradient of the loss function as a product of gradients of all the activation functions of the nodes with respect to their weight. Therefore, the updated weights of nodes in the network depend on the gradients of the activation functions of each node.

For the nodes with sigmoid activation functions, we know that the partial derivative of the sigmoid function reaches a maximum value of 0.25.

When there are more layers in the network, the value of the product of derivative decreases until at some point the partial derivative of the loss function approaches a value close to zero, and the partial derivative vanishes. This is called as **Vanishing gradient** problem.

NEURAL NETWORKS – ACTIVATION FUNCTION

What are Activation functions?

Activation functions are the mathematical function that is applied to the linear output of the neurons which will be able to define the use of our neural network application.
For example, Sigmoid activation function can be used to classify the output into 0 and 1.

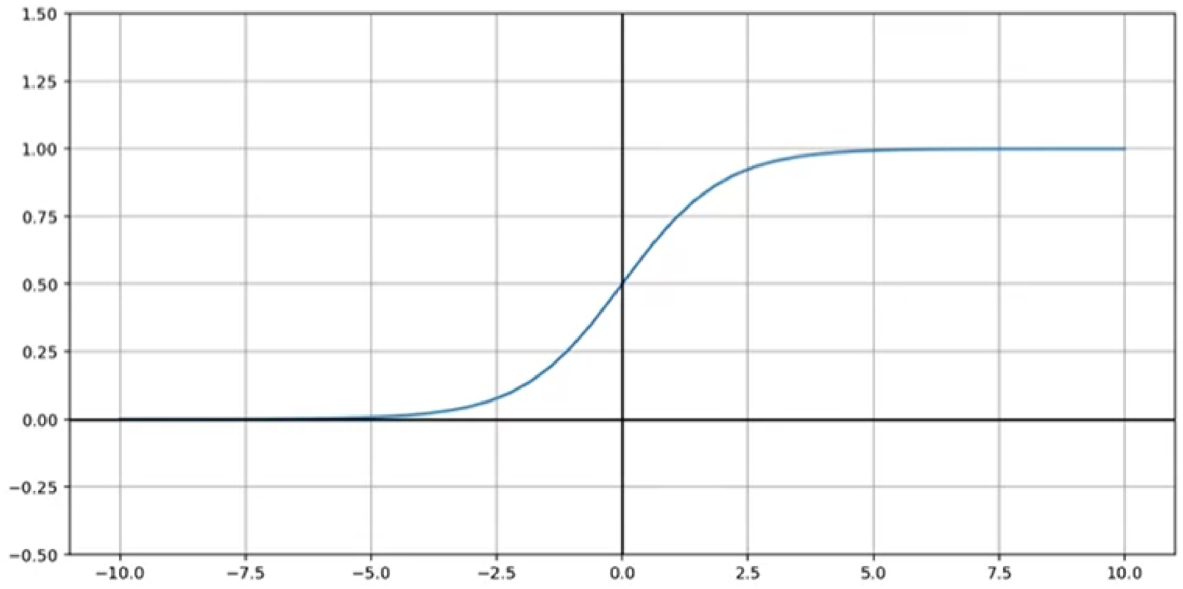
Different activation functions are used based on the type of problem statement.

Sigmoid activation function

This is called the "sigmoid" function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Useful when outcomes should be in (0,1)

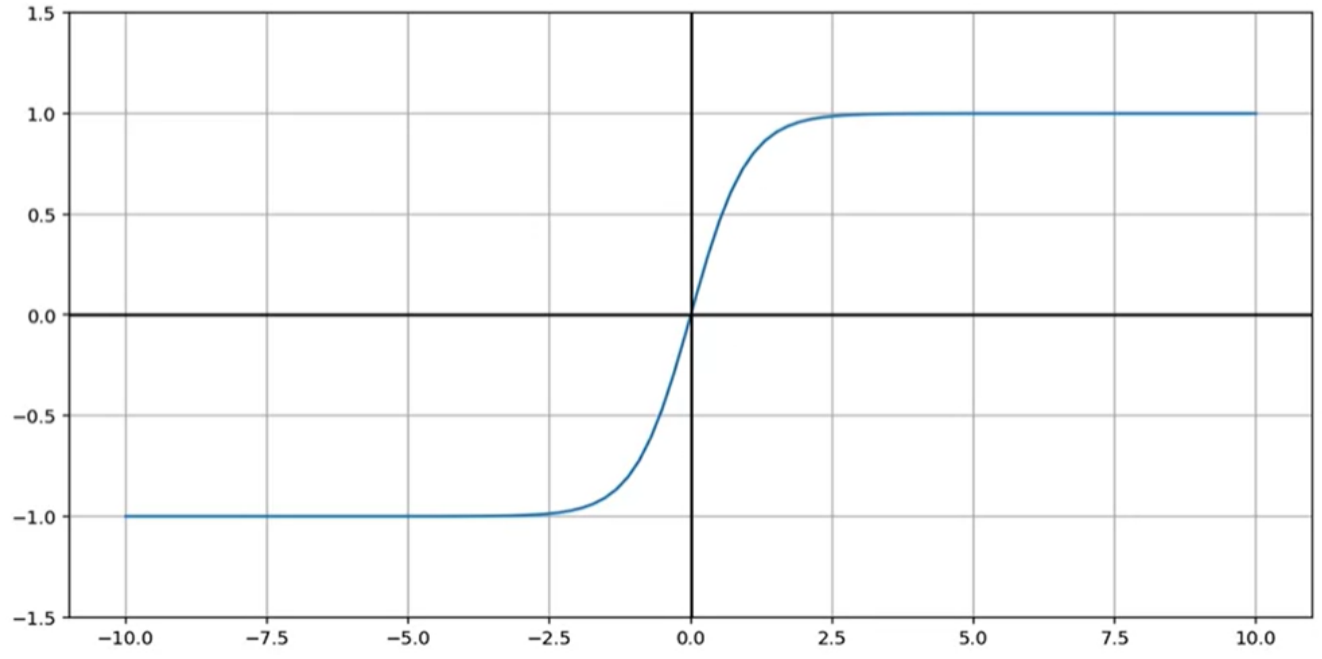
Not immune to Vanishing Gradient Problem



NEURAL NETWORKS – ACTIVATION FUNCTION

Hyperbolic Tangent activation function

$$tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2x} - 1}{e^{2x} + 1}$$



Useful when outcomes should be in (-1,1)

Not immune to Vanishing Gradient Problem

$$tanh(0) = 0$$
$$tanh(\infty) = 1$$
$$tanh(-\infty) = -1$$

NEURAL NETWORKS – ACTIVATION FUNCTION

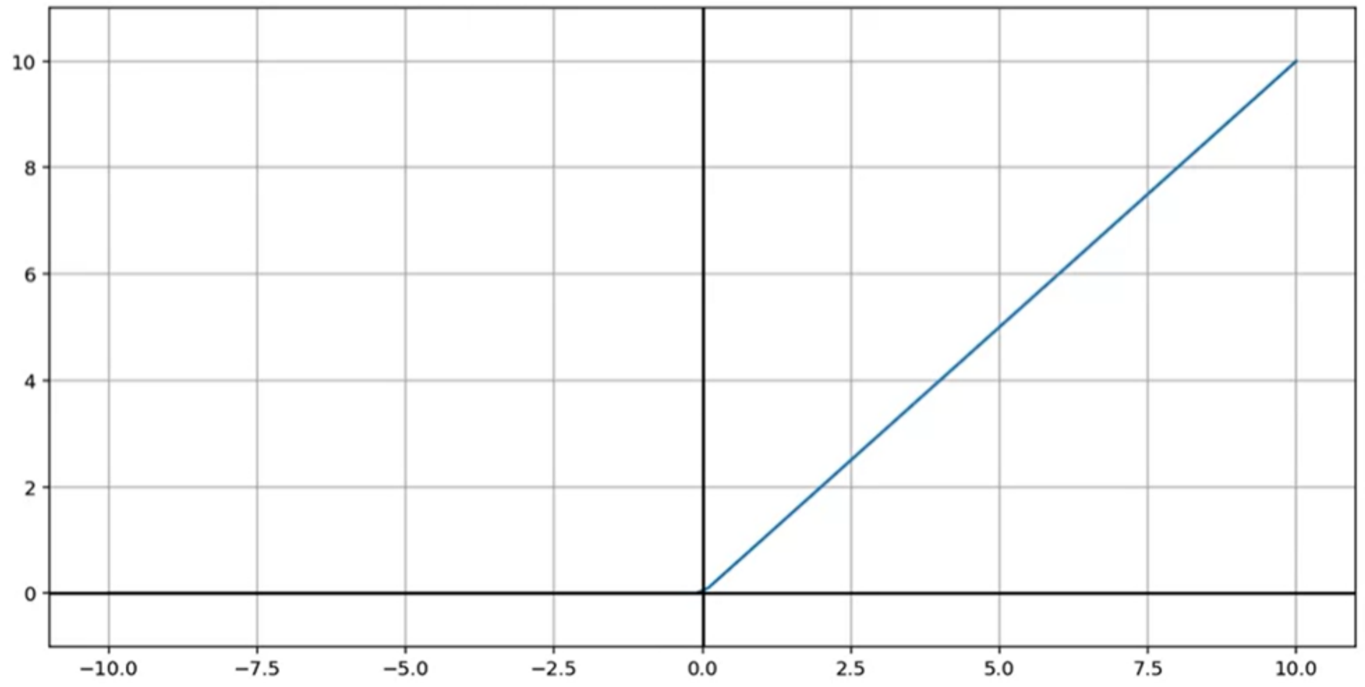
ReLU activation function

$$ReLU(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$= \max(0, z)$$

Useful to capture large effects but does not allow negative outcomes.

Immune to Vanishing Gradient Problem



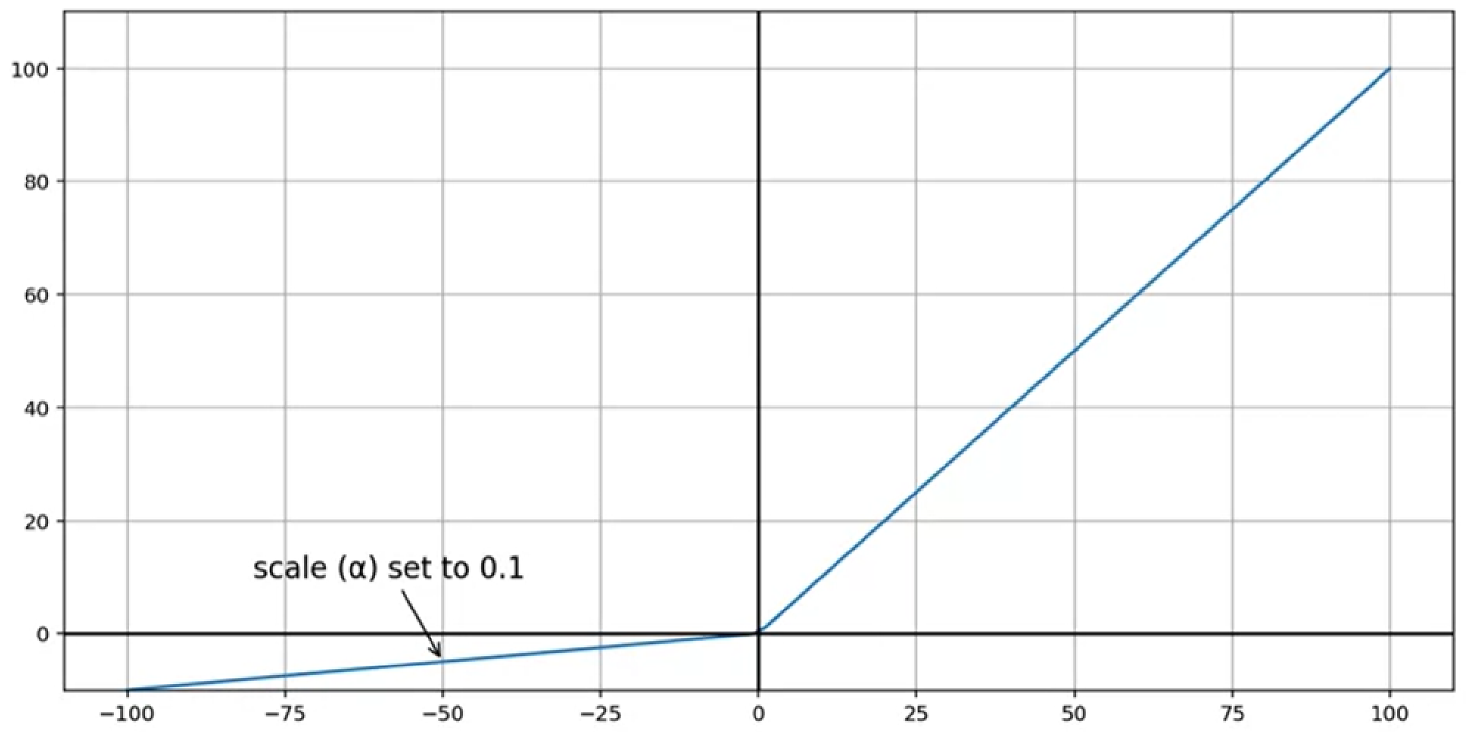
$$ReLU(0) = 0$$
$$ReLU(z) = z \;; \quad \text{for } (z \gg 0)$$
$$ReLU(-z) = 0$$

NEURAL NETWORKS – ACTIVATION FUNCTION

Leaky ReLU activation function

$$LReLU(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$= \max(\alpha z, z)\,; \quad \text{for } (\alpha < 1)$$



Useful to capture large effects but allows negative outcomes as well (unlike ReLU).

Immune to Vanishing Gradient Problem

$$LReLU(0) = 0$$
$$LReLU(z) = z\,; \quad \text{for } (z \gg 0)$$
$$LReLU(-z) = -\alpha z$$

NEURAL NETWORKS – Regularization

“Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.” – Goodfellow et al. (2016)

Different Regularization Methods

- Regularization penalty in Cost function
- Dropout
- Early Stopping

NEURAL NETWORKS – Regularization

Regularization Penalty in Cost Function

This is similar to regularization in Ridge Regression for numeric variables

$$J = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda\sum_{j=1}^{m}W_i^2$$

For categorical variables, we use **categorical cross entropy**.

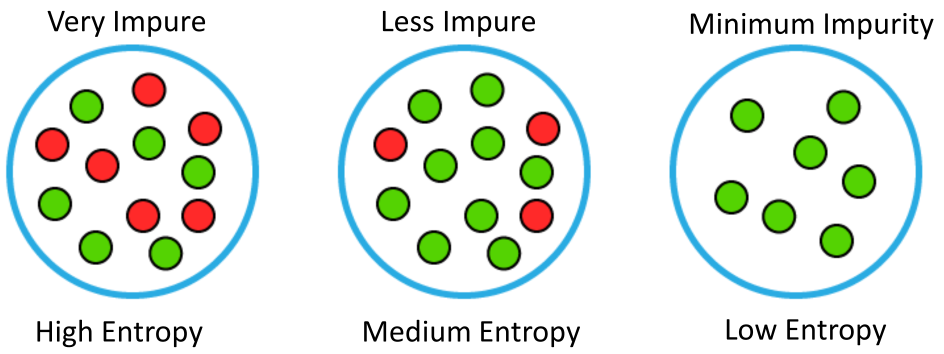
Let's Learn How To Build A DT using Entropy & Information Gain

Entropy :

Entropy is a measure of disorder or impurity in a node. i.e. measures homogeneity of a node

A **high value of Entropy** means that the **randomness** in the system is **high** and thus making accurate predictions is **tough**.

A **low value of Entropy** means that the **randomness** in the system is **low** and thus making accurate predictions is **easier**.



$$E = -\sum_{i=1}^{N} p_i log_2 p_i$$

N -> number of classes

Pi -> probability of the ith class

Suppose you have marbles of three colors; red, purple, and yellow

If we have **one red, three purple**, and **four yellow** observations in our set, our equation becomes:

$$E = -(p_r log_2 p_r + p_p log_2 p_p + p_y log_2 p_y)$$

where pr, pp and py are the probabilities of choosing a red, purple and yellow. So,

pr=1/8

pp=3/8

py=4/8

Our equation now becomes:

E=−((1/8)*log(1/8)+(3/8)*log(3/8)+(4/8)*log(4/8))

Our entropy would be: **0.97**

Let's Learn How To Build A DT using Entropy & Information Gain

What happens to Entropy when all observations belong to the same class say red?

$$E = -(p_r log_2 p_r + p_p log_2 p_p + p_y log_2 p_y)$$

$$E = -((8/8)log(8/8)+(0)log(0)+(0)log(0))$$

$$E = -(1 log_2 1) \quad = 0$$

Such dataset has no impurity.
This implies that such a dataset would not be useful for learning.

What happens to Entropy if we have two classes, half made up of yellow and the other half being purple?

$$E = -(p_r log_2 p_r + p_p log_2 p_p + p_y log_2 p_y)$$

$$E = -((4/8)log(4/8)+(4/8)log(4/8)+(0)log(0))$$

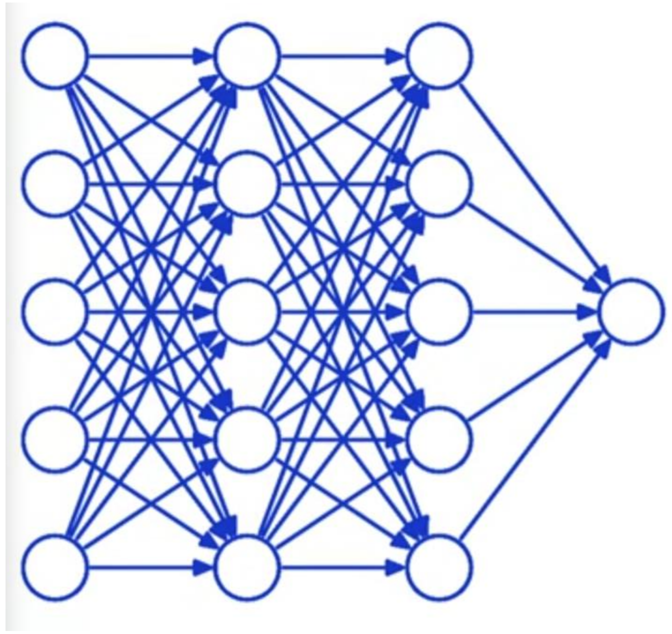
$$E = -((0.5 log_2 0.5) + (0.5 log_2 0.5)) \quad = 1$$

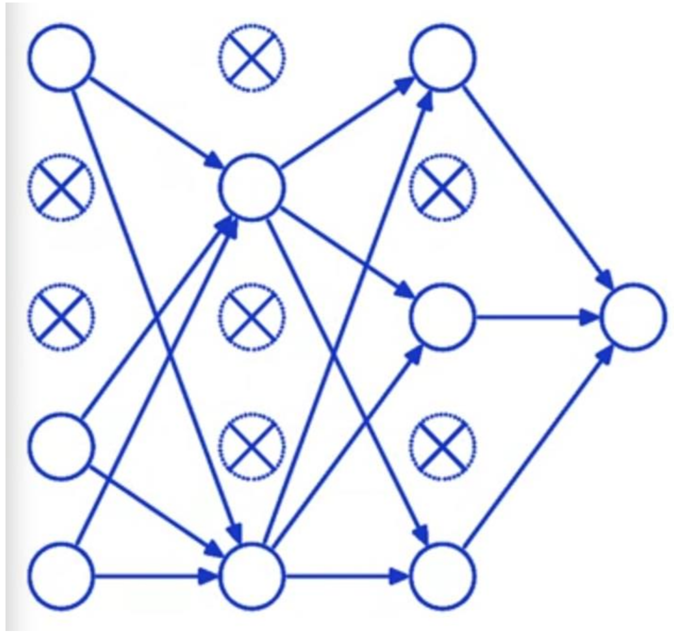
Such dataset has high impurity.
This implies that such a dataset is useful for learning.

NEURAL NETWORKS – Regularization

Dropout

Dropout mechanism is used to rescale the weights of neurons and dropping some nodes based on the activity status.
In simple words, we drop the nodes which do not create any meaningful output (or the output wont change even if they are removed).


Standard Neural Network

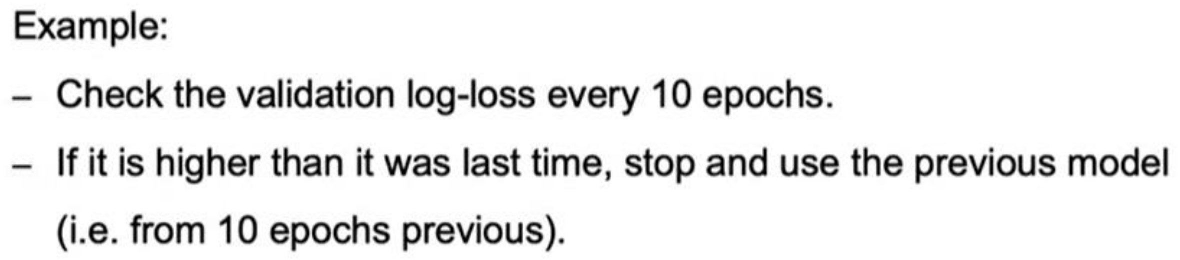

After applying Dropout

NEURAL NETWORKS – Regularization

Early Stopping

This refers to choosing some rules, after which the model stops training.
For example, we can set the early stopping criteria as Accuracy = 0.9 and this will be used at each iteration of the training network.

Once the criteria has been met, the model will stop training.



Example:

- Check the validation log-loss every 10 epochs.
- If it is higher than it was last time, stop and use the previous model (i.e. from 10 epochs previous).

NEURAL NETWORKS – Optimizers

Optimizers

We have considered the different optimization methods such as Full Batch GD, Stochastic GD and Mini Batch GD.
However, all of them have used the same formula to update the new weights as follows :

$$W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$$

$$\mathrm{W} \coloneqq W - \alpha \cdot \nabla J$$

There are some more methods to find the new weights. These methods are called as **Optimizers**

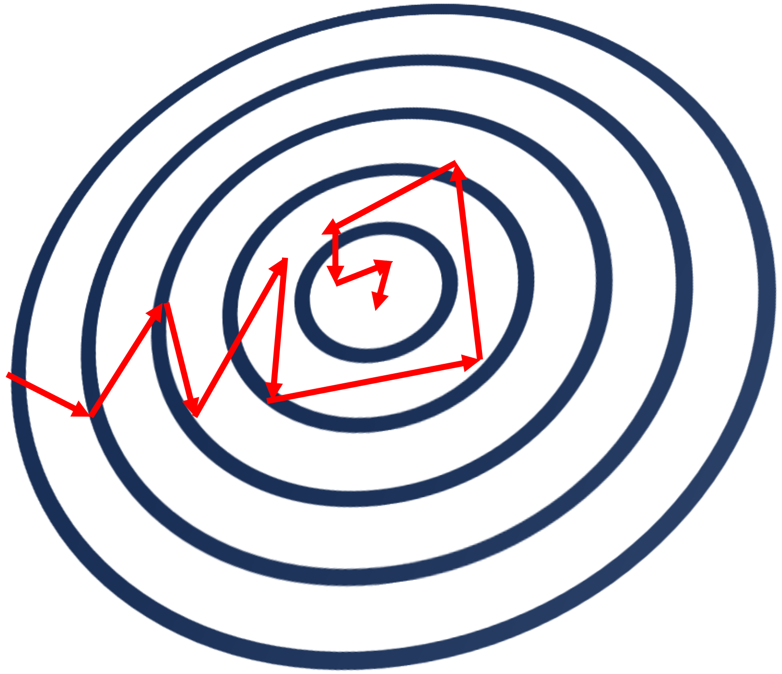
NEURAL NETWORKS – Optimizers

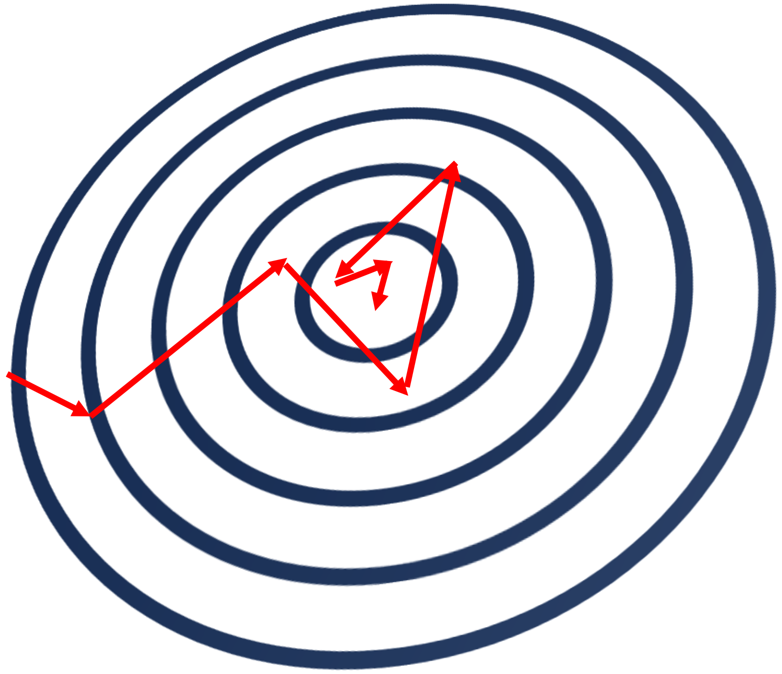
Momentum

The idea here is to keep a "running average" of the step directions, smoothing out the variation at each points.

$$v_{\mathrm{t}} \coloneqq \eta \cdot v_{t-1} - \alpha \cdot \nabla J$$
$$\mathrm{W} \coloneqq W - v_t$$

Here η is referred to as momentum. It is generally < 1.
This method is quite prone to the "**overshooting**" problem.


Gradient Descent


Gradient Descent with Momentum

NEURAL NETWORKS – Optimizers

AdaGrad

Idea: scale the update for each weight separately.

1. Update frequently-updated weights less.
2. Keep running sum of previous updates.
3. Divide new updates by factor of previous sum.

With starting point $G_i(0) = 0$:

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(\mathrm{t})\right)^2$$

G will continue to increase

$$W := W - \frac{\eta}{\sqrt{G_t} + \epsilon} \cdot \nabla J$$

This leads to smaller updates each iteration

RMSProp

Quite similar to AdaGrad.

- Rather than using the sum of previous gradients, decay older gradients more than more recent ones.
- More adaptive to recent updates.

NEURAL NETWORKS – Optimizers

Adam (Adaptive Moment Estimation)

The idea is to utilize both momentum and RMSProp concepts

RMSProp and Adam seem to be quite popular. From 2012 to 2017, approximately 23% of deep learning papers submitted to arXiv (a popular platform for research in Deep Learning) mentioned using the Adam approach.

NEURAL NETWORKS – Optimizers

	Momentum	AdaGrad	RMSProp	ADAM
Concept	Incorporates the 'momentum' of its updates to keep on moving in the direction of steepest descent. It helps to accelerate the gradient descent algorithm by considering the past gradients to smooth out the update. It does this by adding a fraction of the direction of the previous step to the current step, effectively increasing the speed of convergence.	Adapts the learning rate for each parameter, giving low learning rates to parameters with frequently occurring features and higher learning rates to parameters with infrequent features. It achieves this by accumulating the square of the gradients in the denominator; as the accumulated value grows, the learning rate shrinks.	Tries to resolve AdaGrad's radically diminishing learning rates by using a moving average of squared gradients. It adjusts the learning rate for each weight based on the mean of recent magnitudes of the gradients for that weight.	Combines ideas from both Momentum and RMSProp. Besides storing an exponentially decaying average of past squared gradients like RMSProp, Adam also keeps an exponentially decaying average of past gradients, similar to momentum. It calculates adaptive learning rates for each parameter.
Advantage	Helps to prevent oscillations and speeds up convergence, particularly in areas of the objective function that have a shallow gradient.	Very effective for sparse data (lots of zeros in data), as it adapts the learning rate to the frequency of parameters.	Solves the diminishing learning rate problem of AdaGrad, making it suitable for both non-stationary and stationary problems.	Requires less memory and is computationally efficient. It is well suited for problems that are large in terms of data/parameters or problems with noisy/spare gradients. Adam generally works well in practice and outperforms other adaptive learning-method algorithms.
Disadvantage	Might overshoot the minimum because of the momentum it accumulates.	The continuously accumulating squared gradients in the denominator can cause the learning rate to shrink and become infinitesimally small, which essentially stops the network from learning further.	Still requires manual tuning of the learning rate.	The adaptive learning rate can sometimes lead to overshooting in the initial stages of the training because of the high moments. Also, it might not converge to the optimal solution under certain conditions as theoretically expected, requiring more tuning of hyperparameters like the initial learning rate.