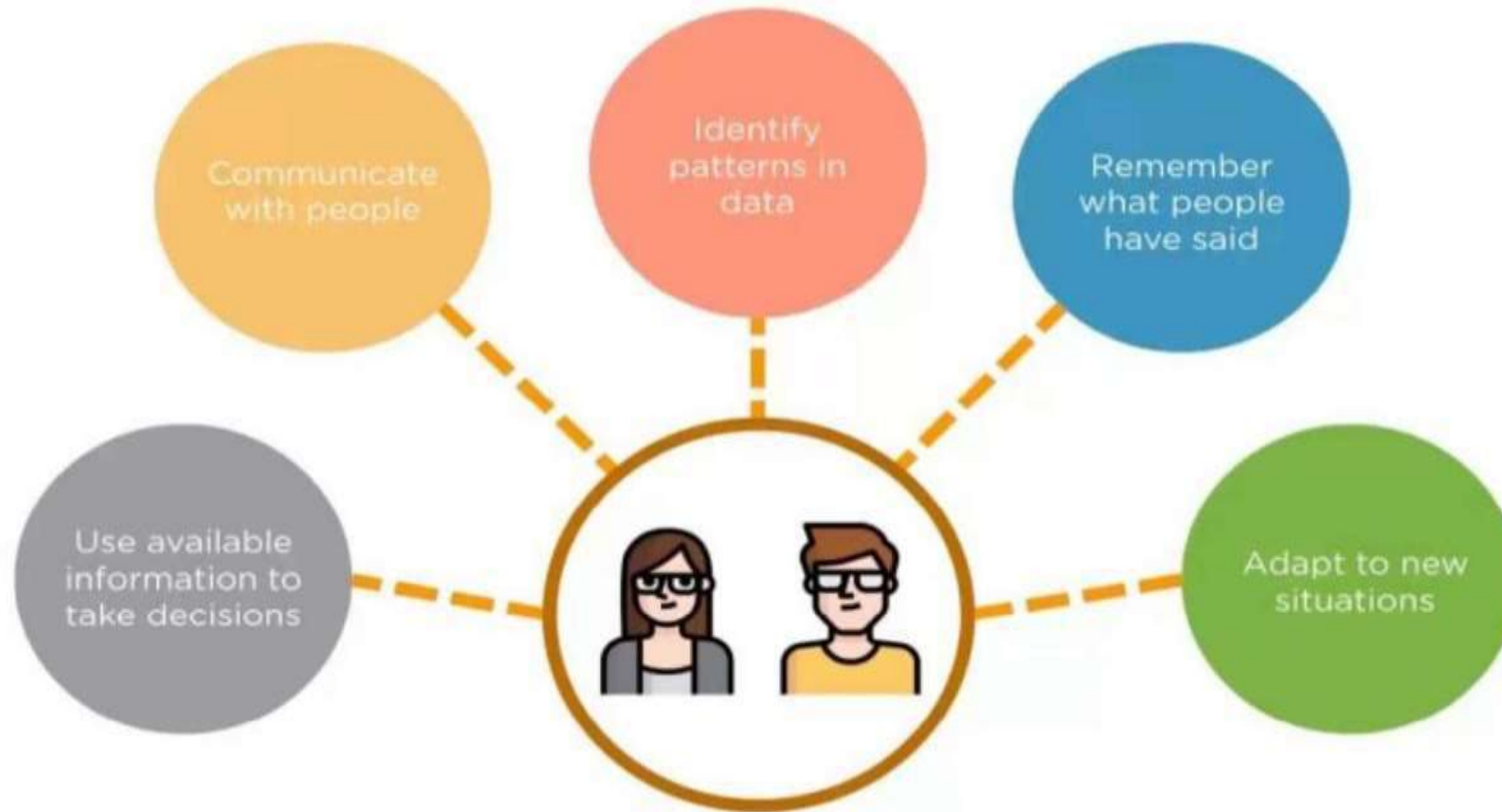


Course Content/Syllabus

Unit 1: Introduction to Deep Learning:

Neural networks, Training Neural Networks, Activation Functions, Multilayer Perceptrons, Implementation of Multilayer Perceptrons, Forward Propagation, Backward Propagation and Computational Graphs, Numerical Stability and Initialization, Generalization in Deep Learning, Dropout,
Case study: Simple Neural Networks' Implementation using keras.

TOPICS IN DEEP LEARNING HUMAN INTELLIGENCE



TOPICS IN DEEP LEARNING

Prerequisite

Interest in Deep Learning and AI



One programming language



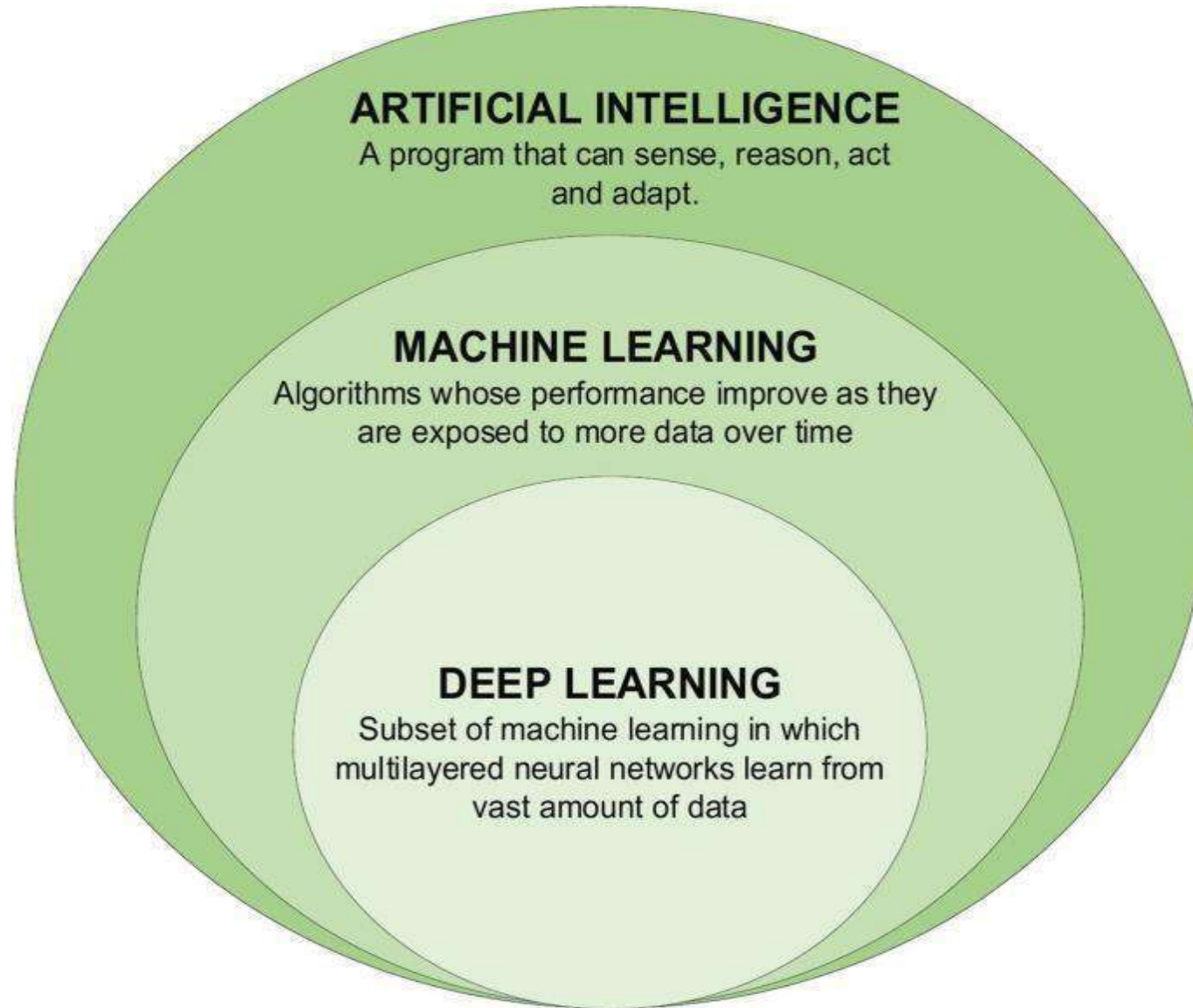
Commitment to Persevere



Little knowledge in Machine Intelligence

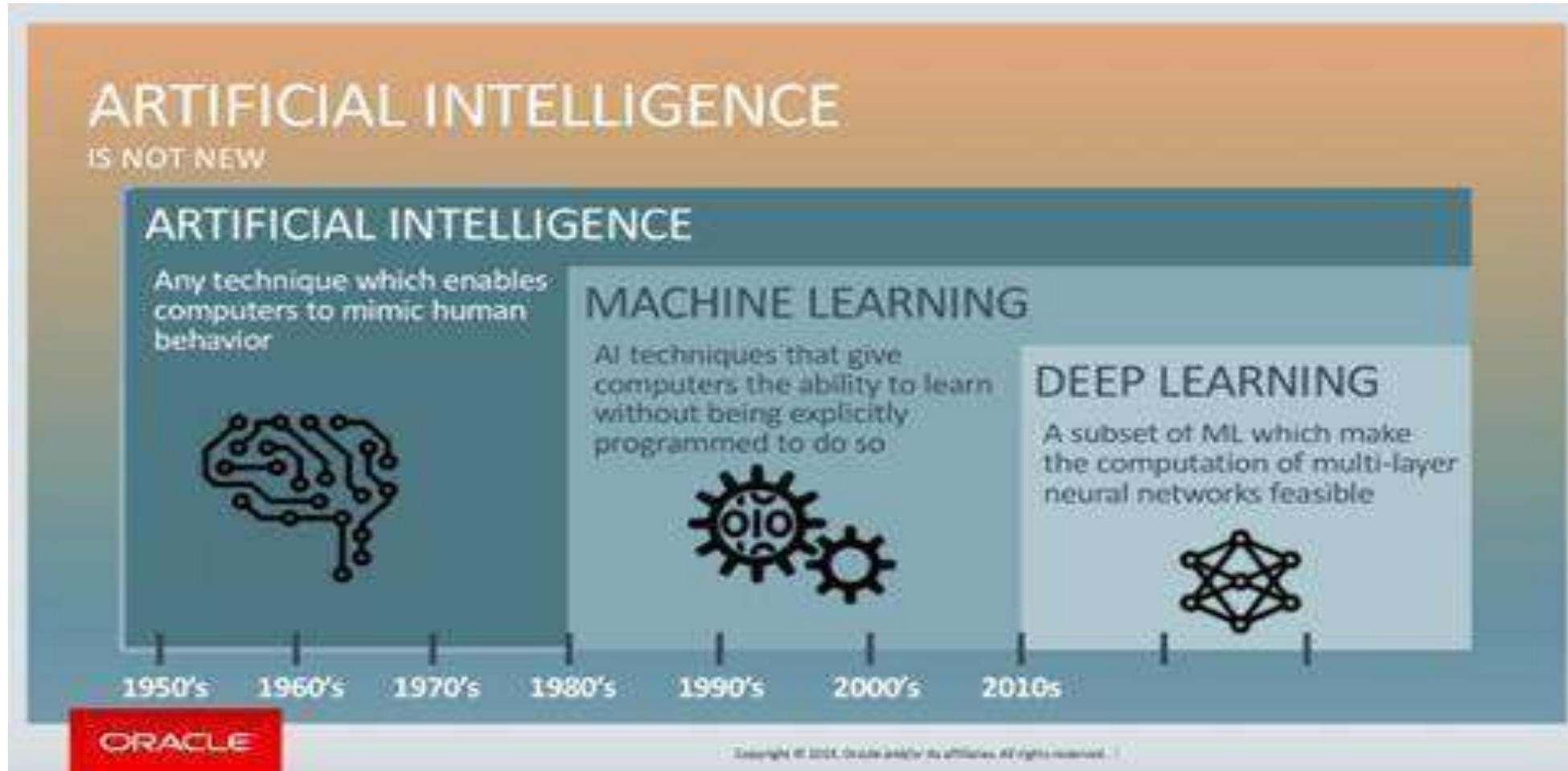
TOPICS IN DEEP LEARNING

AI V/S ML V/S DL



TOPICS IN DEEP LEARNING

AI V/S ML V/S DL



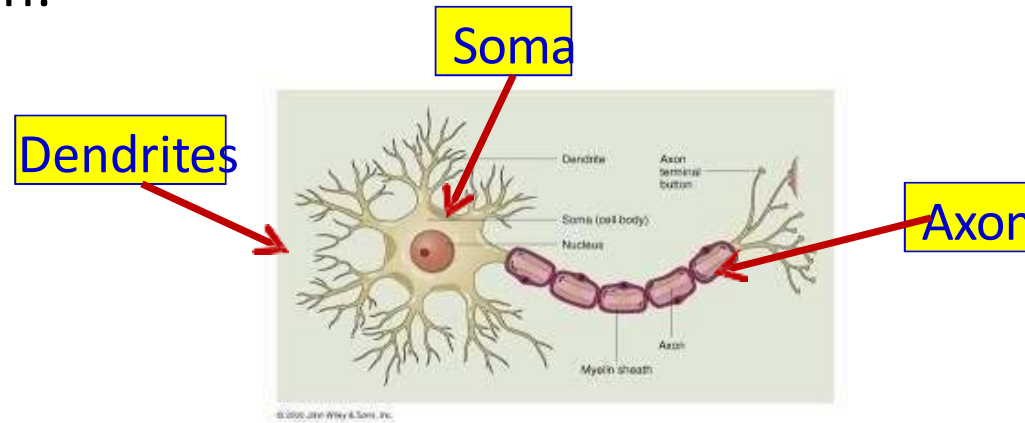
1.1 Neural Network

What is Artificial Neural Network?

- Artificial Neural Network (ANN) is an efficient computing system whose central theme is borrowed from the analogy of biological neural networks.

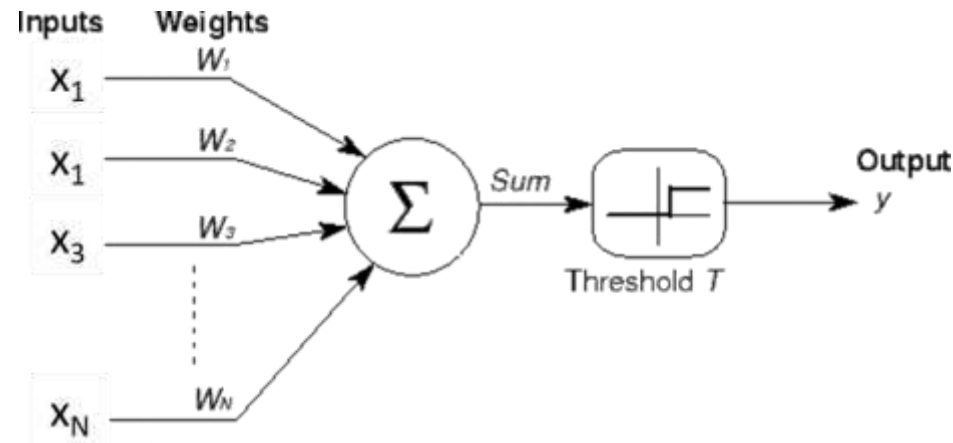
Modelling the brain

- What are the units?
- A neuron:



- Signals come in through the dendrites into the Soma
- A signal goes out via the axon to other neurons
 - Only one axon per neuron
- Factoid that may only interest me: Neurons do not undergo cell division

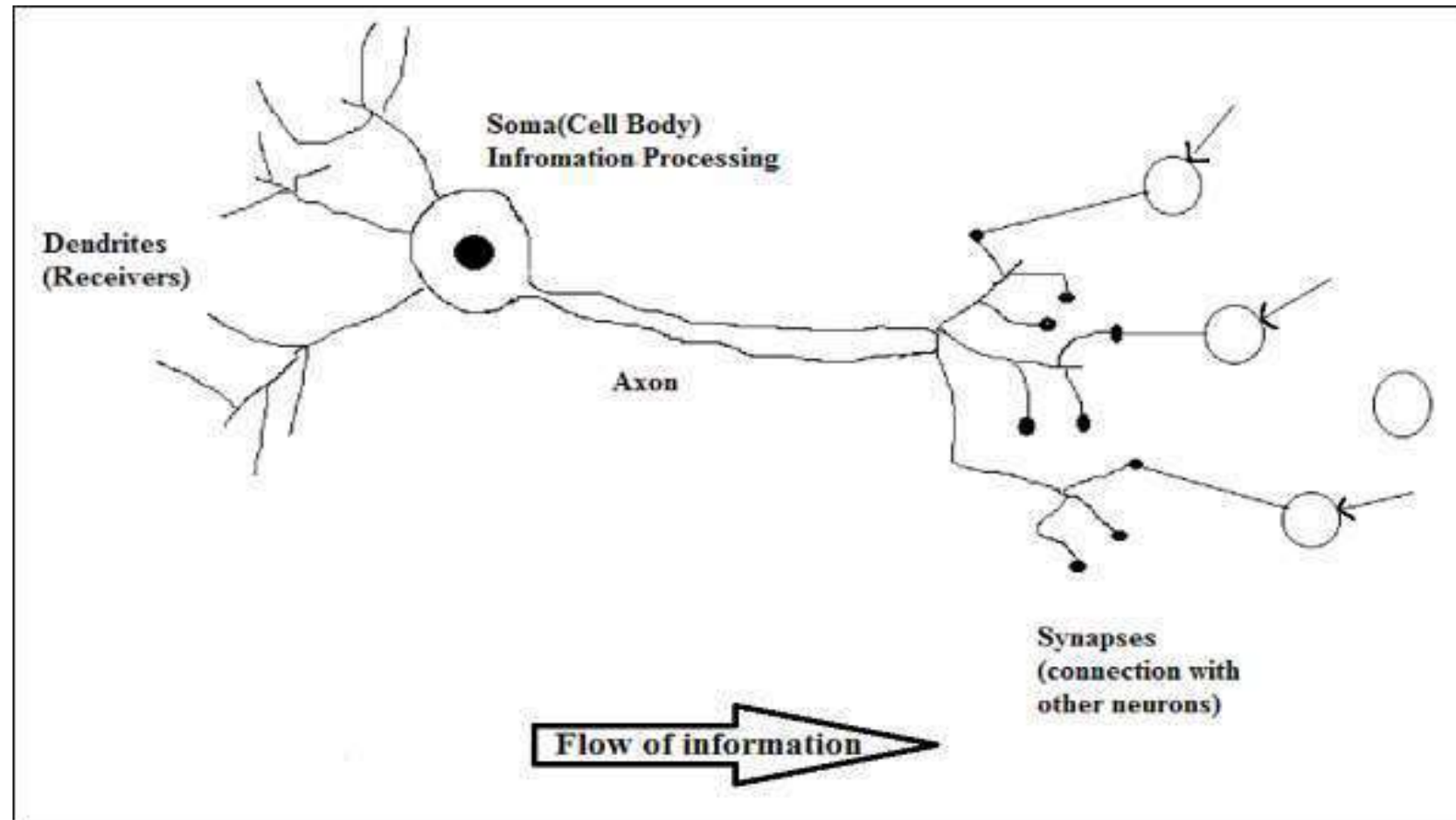
mathematical model of Perceptron



- Number of inputs combine linearly
 - Threshold logic: Fire if combined input exceeds or equal to threshold

$$Y = \begin{cases} 1 & \text{if } \sum_i w_i x_i - T \geq 0 \\ 0 & \text{else} \end{cases}$$

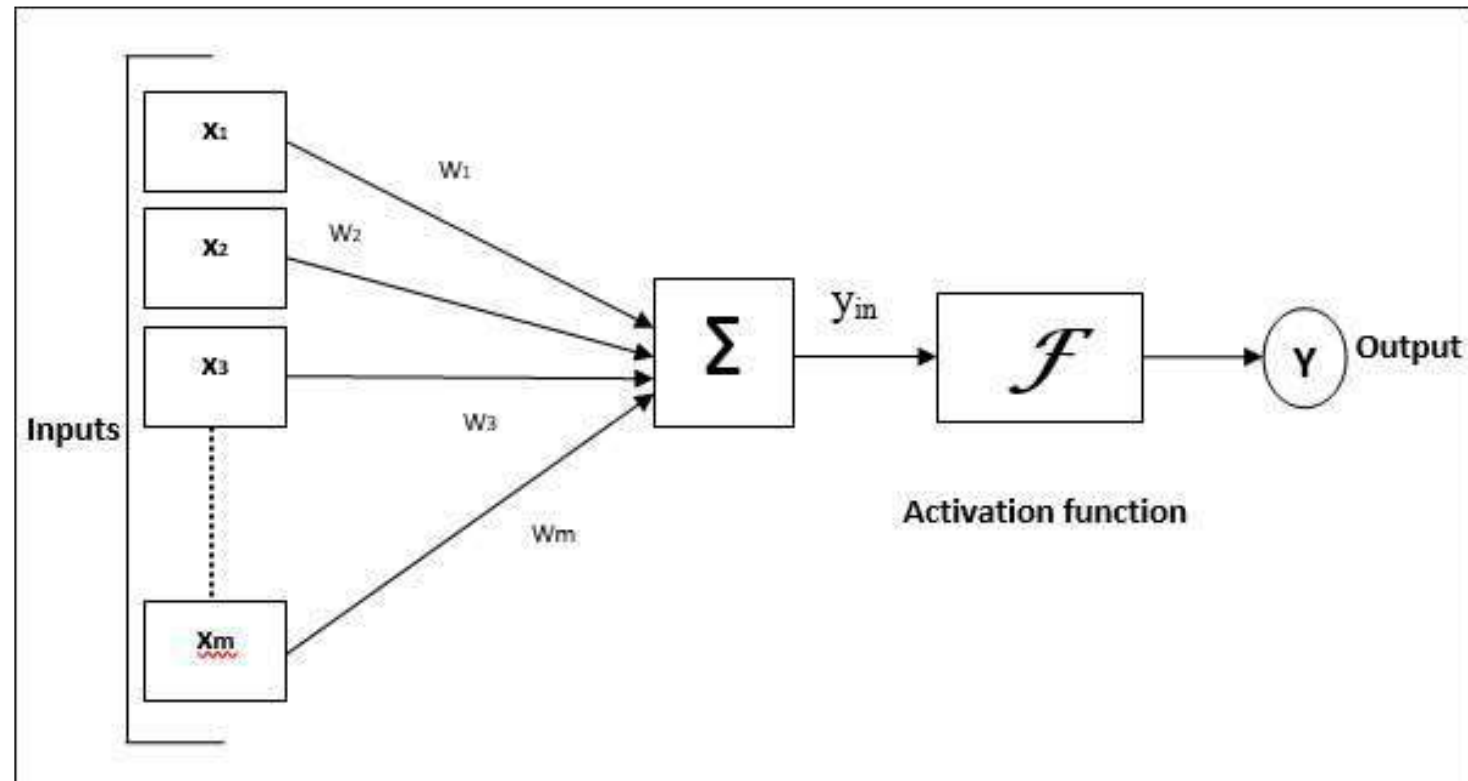
Biological Neuron



ANN versus BNN

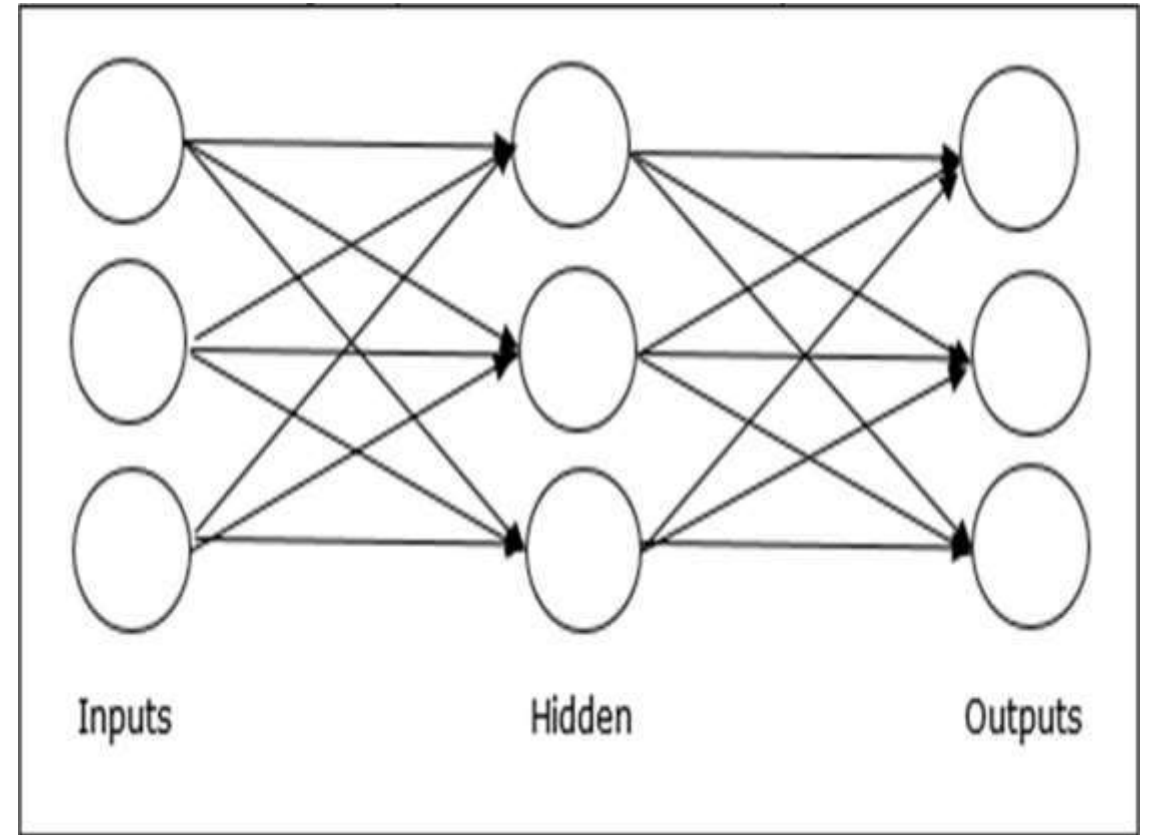
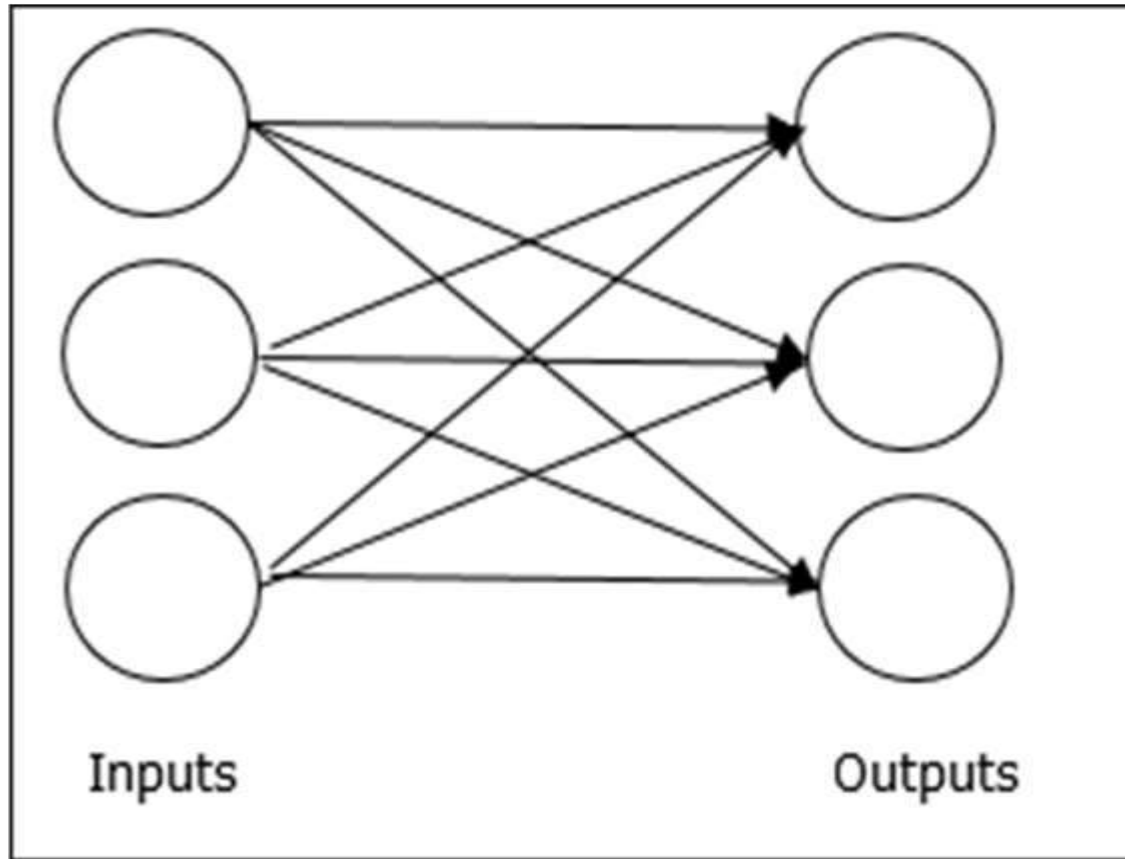
Biological Neural Network BNN	Artificial Neural Network ANN
Soma	Node
Dendrites	Input
Synapse	Weights or Interconnections
Axon	Output

Model of Artificial Neural Network

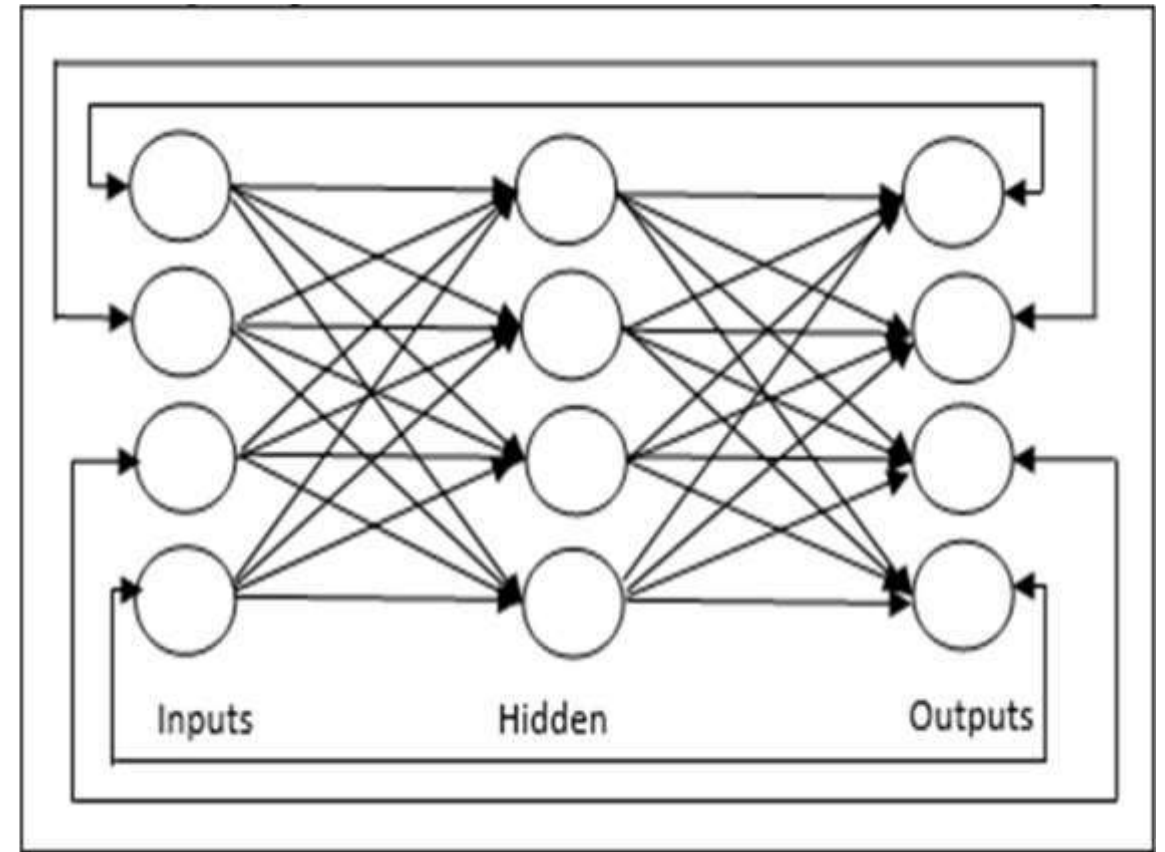
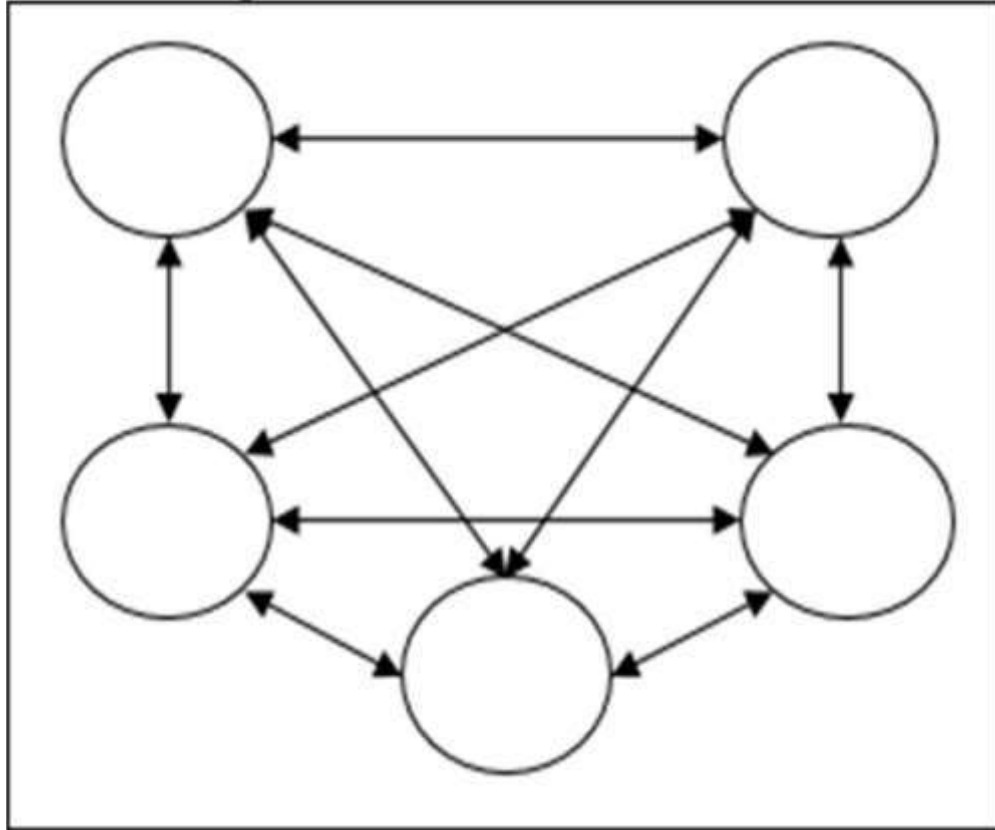


Network Topology

- Feedforward Network

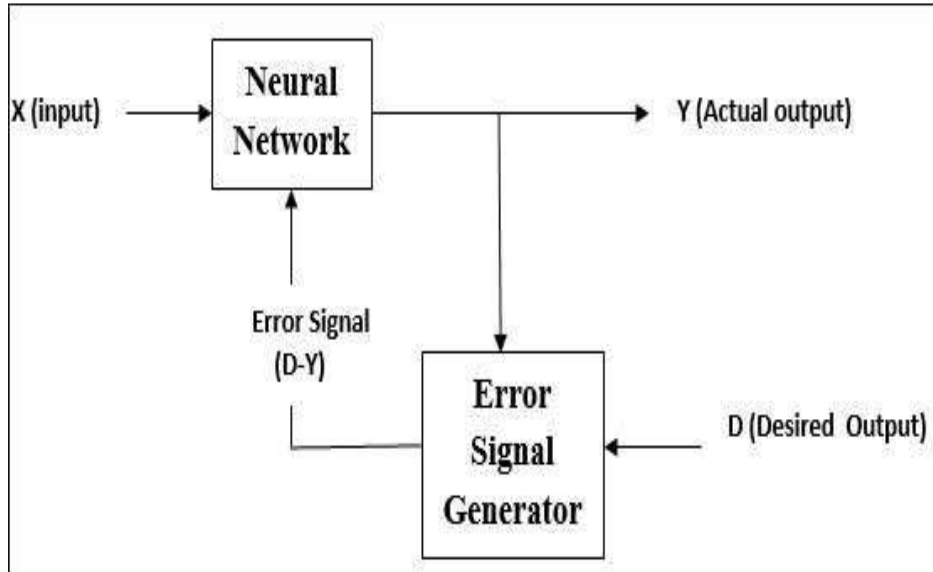


Feedback Network

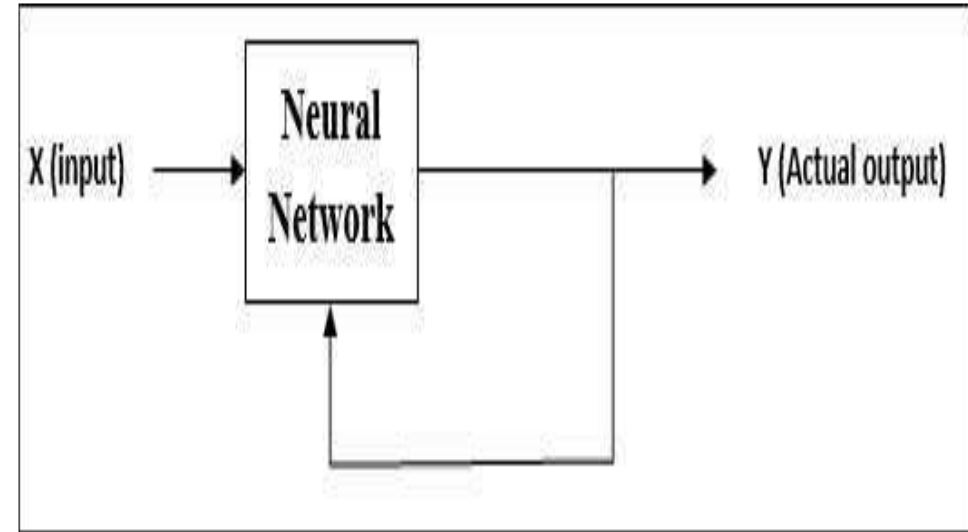


Adjustments of Weights or Learning

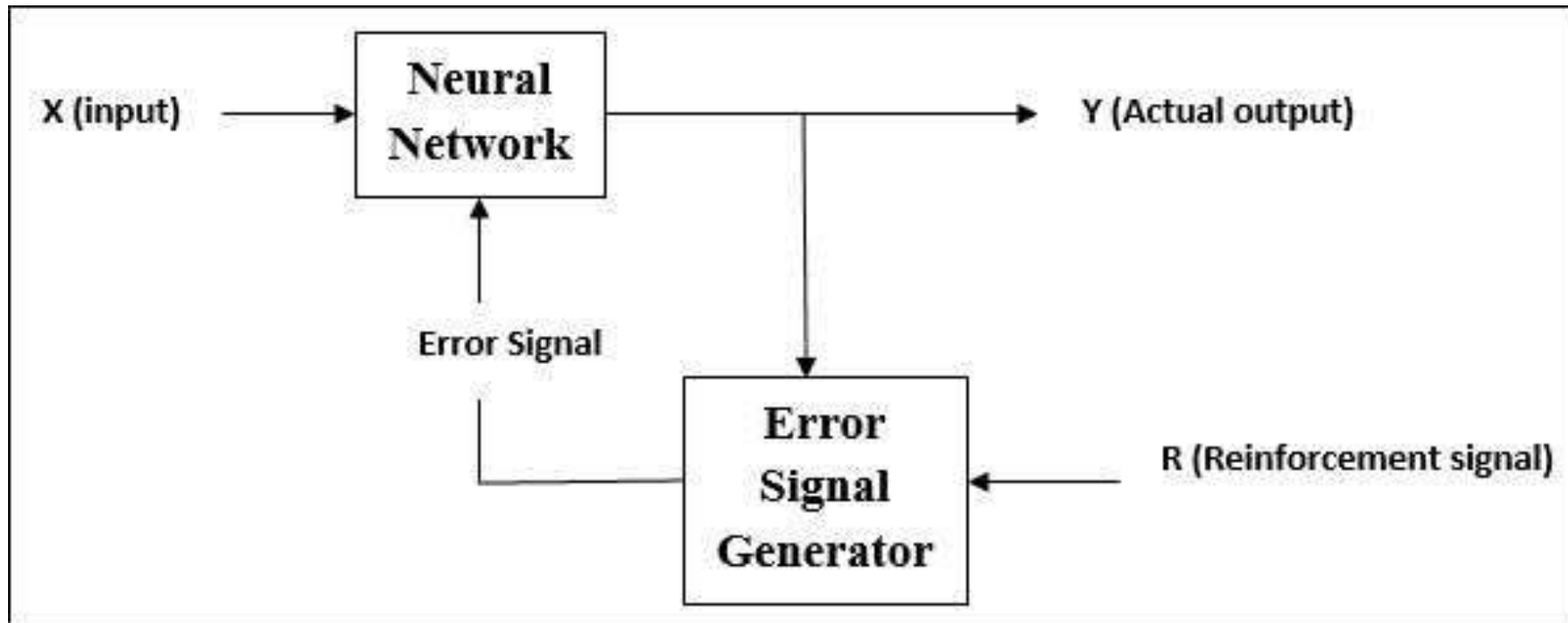
- Supervised Learning



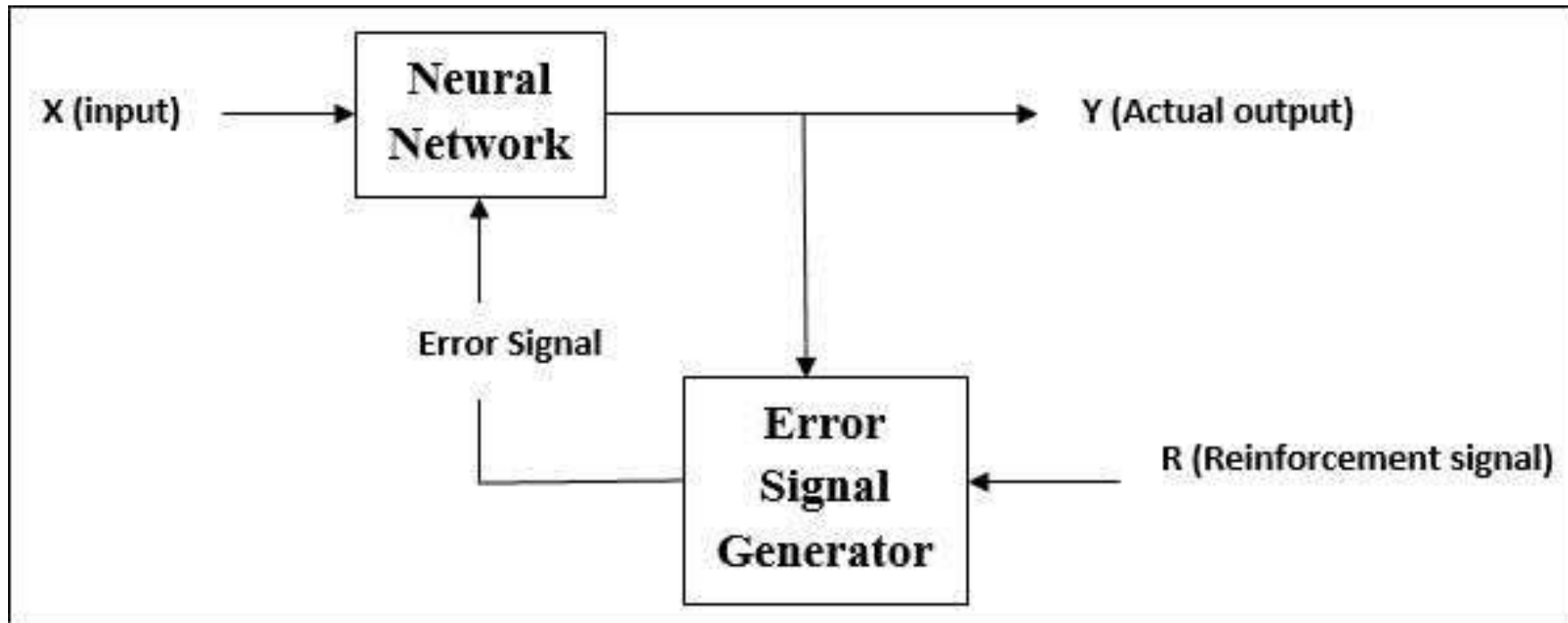
Unsupervised Learning



Reinforcement Learning

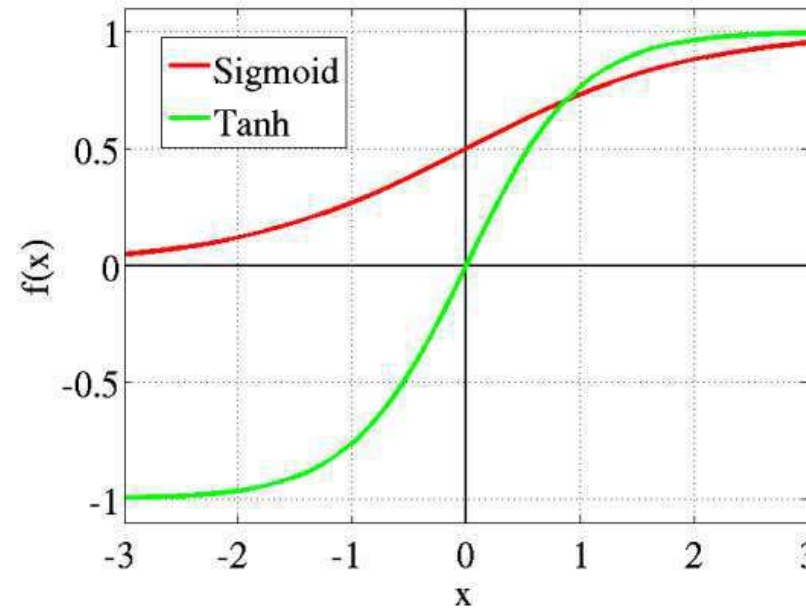
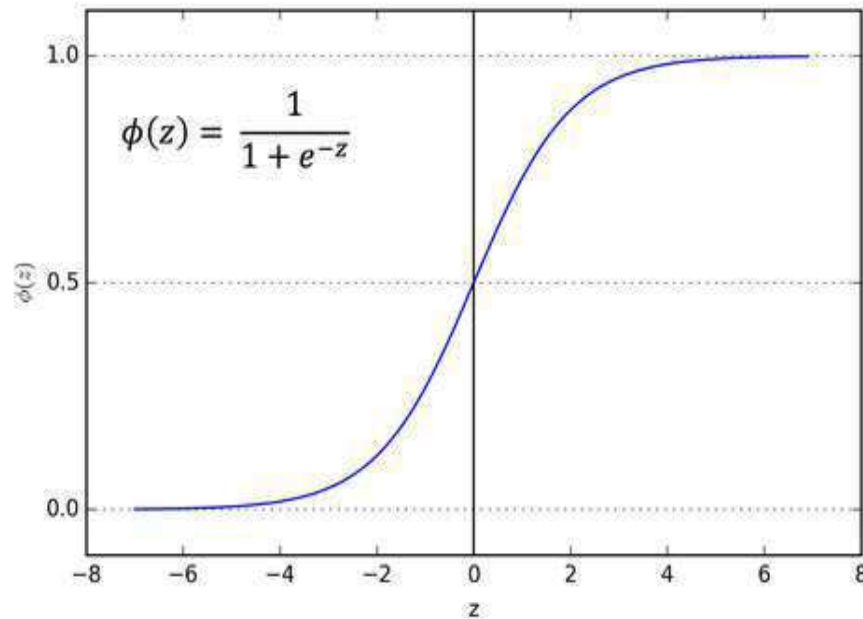


Reinforcement Learning



1.2 Activation Functions

- Sigmoid Activation Function Tanh or hyperbolic tangent Activation Function



$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Problems with Sigmoid and Tanh activation function

Problems with Tanh activation function

1. **Vanishing gradient:** looking at the function plot, you can see that when inputs become small or large, the function saturates at -1 or 1 , with a derivative extremely close to 0 . Thus, it has almost no gradient to propagate back through the network, so there is almost nothing left for lower layers.
2. **Computationally expensive:** the function has an exponential operation.



ReLU (Rectified Linear Unit) Activation Function

- The ReLU is the most used activation function in the world right now. Since, it is used in almost all the **convolutional neural networks or deep learning**.

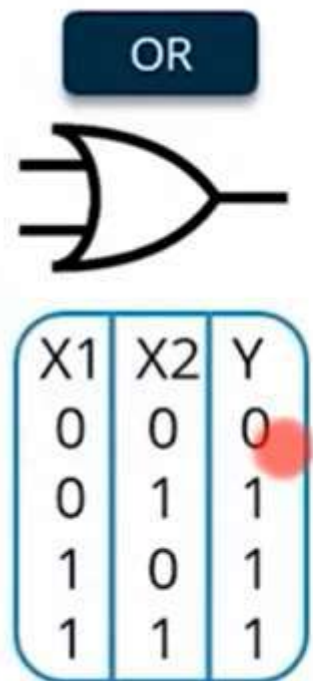
Range: [0 to infinity)

The function and its derivative both are monotonic.

But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

Applications

It can be used to implement Logic Gates.

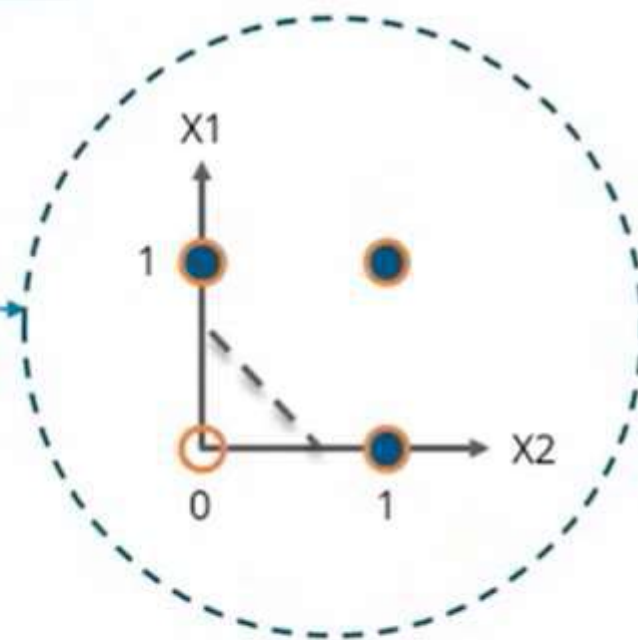


X1	X2
0	0
0	1
1	0
1	1

$W = 1$

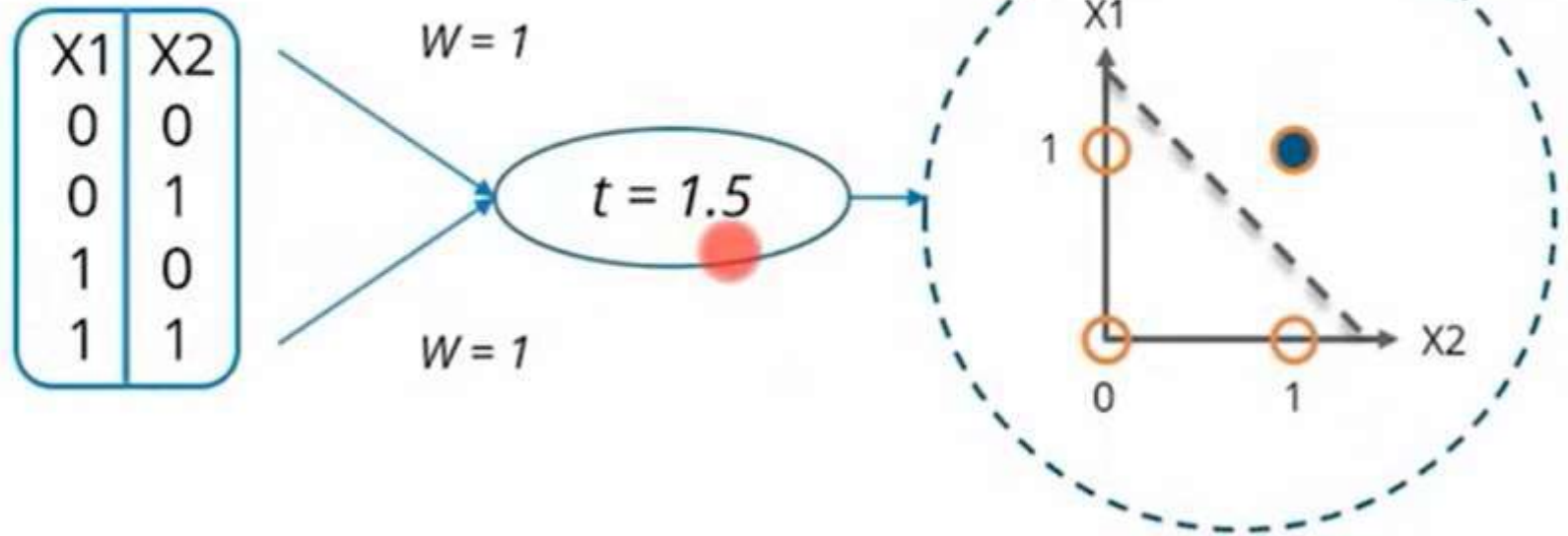
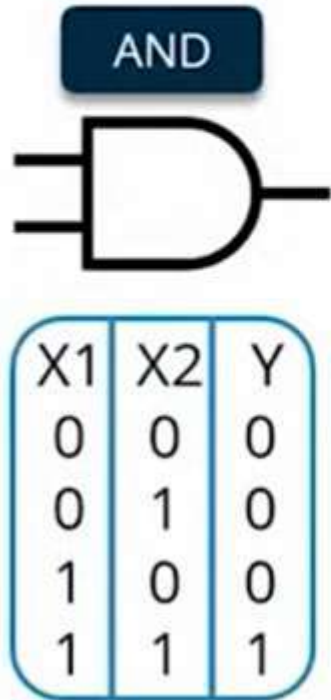
$t = 0.5$

$W = 1$



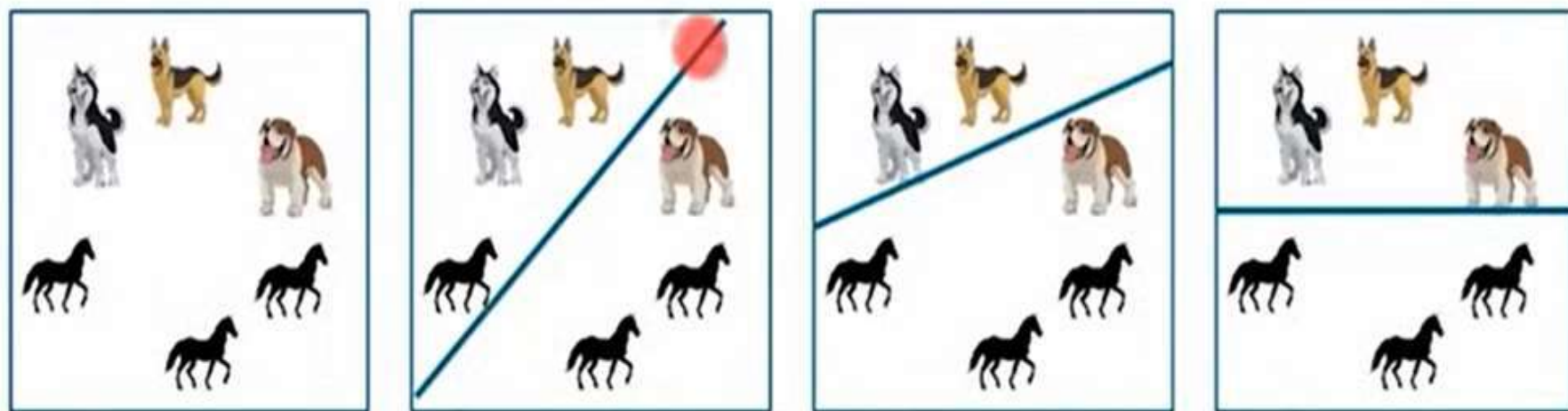
Applications

It can be used to implement Logic Gates.



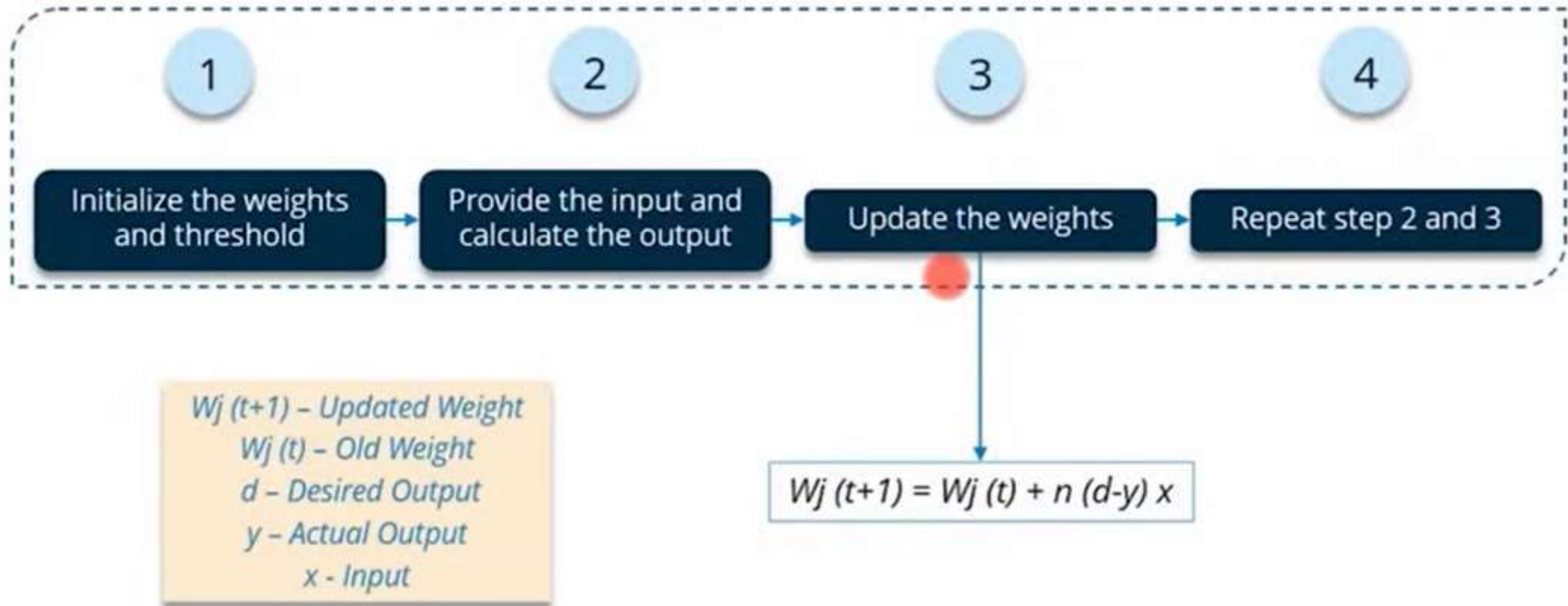
Applications

It is used to classify any linearly separable set of inputs.



Adjustments of Weights or Learning

Perceptron Learning Algorithm



1.3 Training Algorithm

- Training Algorithm for Single Output Unit
- Step 1 – Initialize the following to start the training –
 - Weights
 - Bias
 - Learning rate α
- Step 2 – Continue step 3-8 when the stopping condition is not true.
- Step 3 – Continue step 4-6 for every training vector x .
- Step 4 – Activate each input unit as follows –
$$x_i = s_i (i=1 \text{ to } n)$$
- Step 5 – Now obtain the net input with the following relation –

$$y_{in} = b + \sum_i^n x_i w_i$$

Step 6 – Apply the following activation function to obtain the final output.

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Step 7 – Adjust the weight and bias as follows –

Case 1 – if $y \neq t$ then,

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$

$$b(\text{new}) = b(\text{old}) + \alpha t$$

Case 2 – if $y = t$ then,

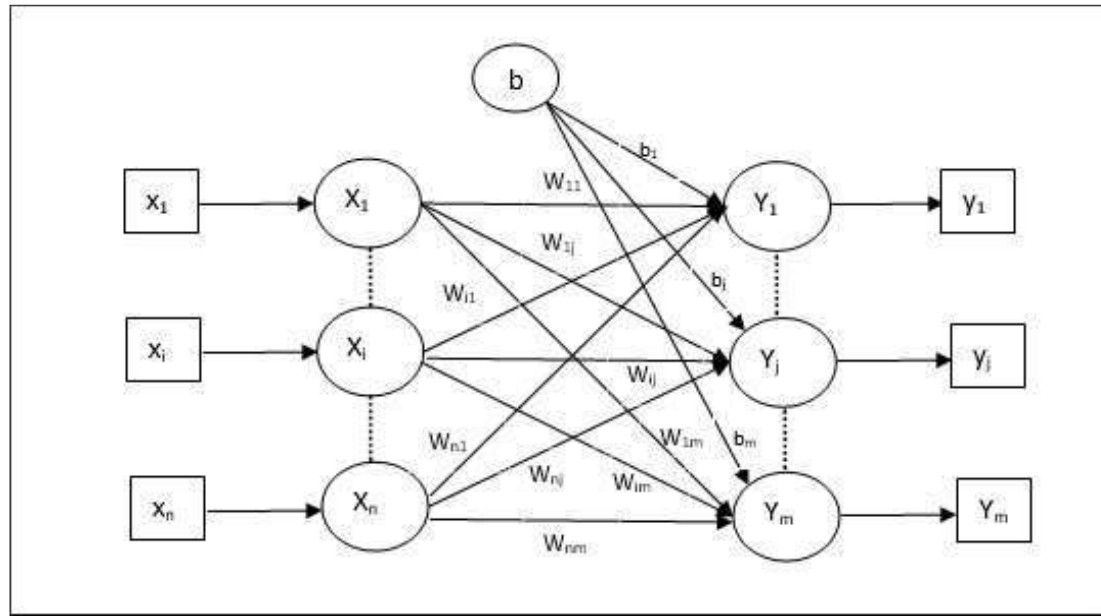
$$w_i(\text{new}) = w_i(\text{old})$$

$$b(\text{new}) = b(\text{old})$$

Step 8 – Test for the stopping condition, which would happen when there is no change in weight. (α -learning rate, t -target output, X_i -Input associated with w_i)

Training Algorithm for Multiple Output Units

- Step 1 – Initialize the following to start the training –
 - Weights
 - Bias
 - Learning rate α
- Step 2 – Continue step 3-8 when the stopping condition is not true.
- Step 3 – Continue step 4-6 for every training vector x .



- Step 4 – Activate each input unit as follows –
- $x_i = s_i$ (i=1 to n)
- Step 5 – Obtain the net input with the following relation –

$$y_{in} = b + \sum_i^n x_i w_{i y_{in}}$$

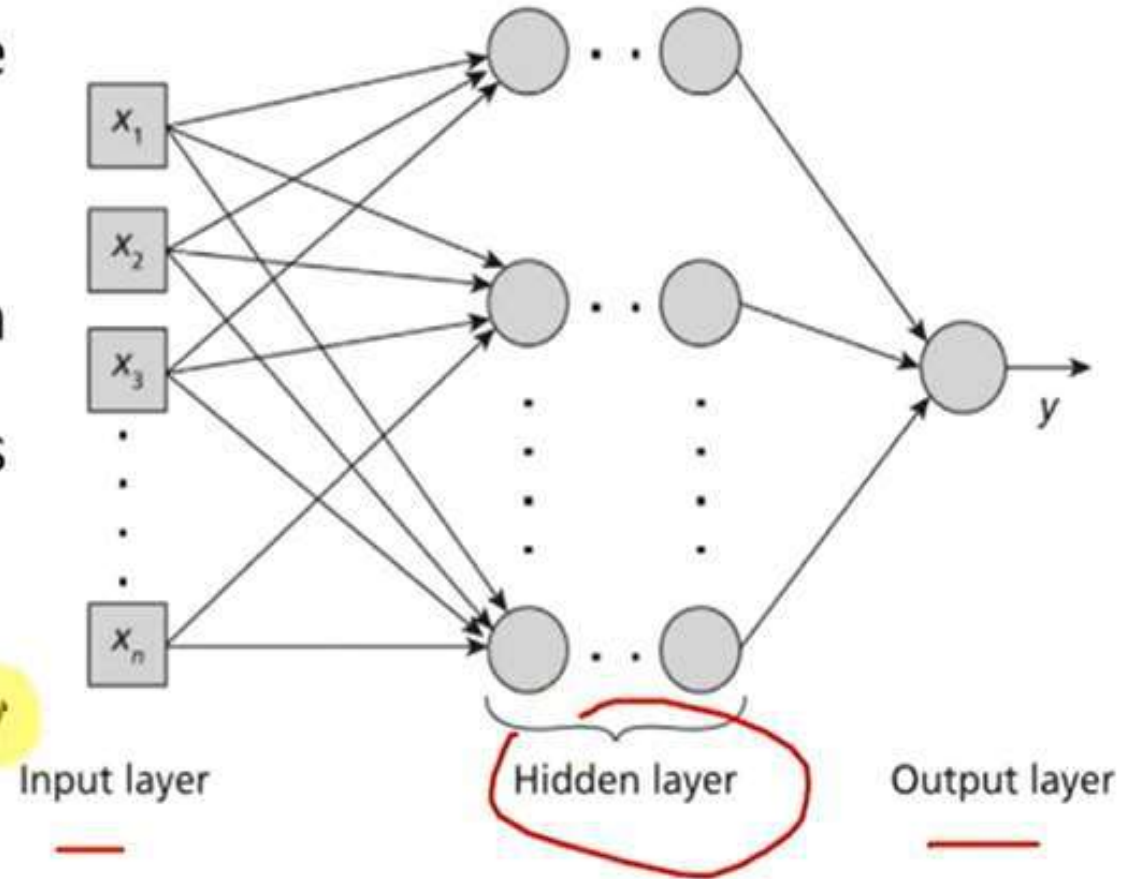
- Step 6 – Apply the following activation function to obtain the final output for each output unit $j = 1$ to m –

$$(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

- Step 7 – Adjust the weight and bias for $x = 1$ to n and $j = 1$ to m as follows –
- Case 1 – if $y_j \neq t_j$ then,
 - $w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha t_j x_i$
 - $b_j(\text{new}) = b_j(\text{old}) + \alpha t_j$
- Case 2 – if $y_j = t_j$ then,
 - $w_{ij}(\text{new}) = w_{ij}(\text{old})$
 - $b_j(\text{new}) = b_j(\text{old})$
- Step 8 – Test for the stopping condition, which will happen when there is no change in weight.

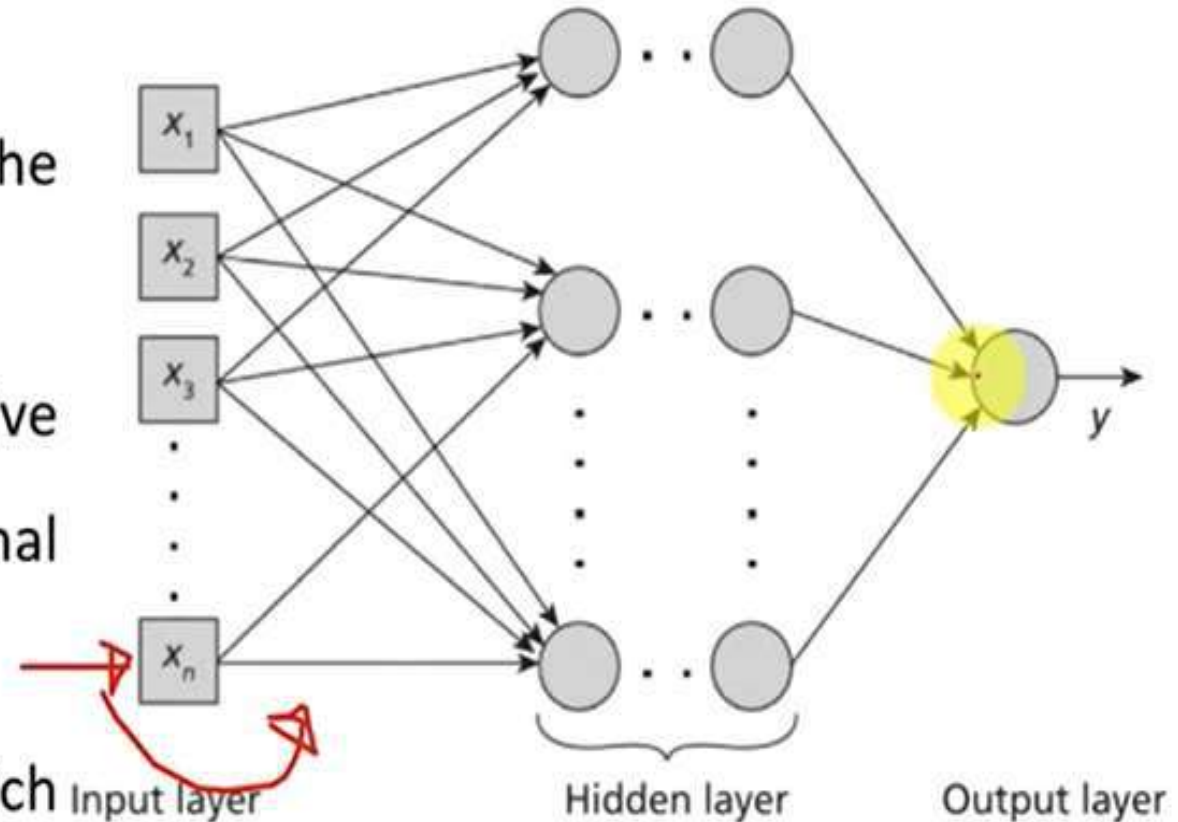
1.4 Multi-Layer Perceptron Learning Algorithm

- A multi-layer perceptron is a type of Feed Forward Neural Network with multiple neurons arranged in layers.
- The network has at least three layers with an input layer, one or more hidden layers and an output layer.
- All the neurons in a layer are fully connected to the neurons in the next layer.



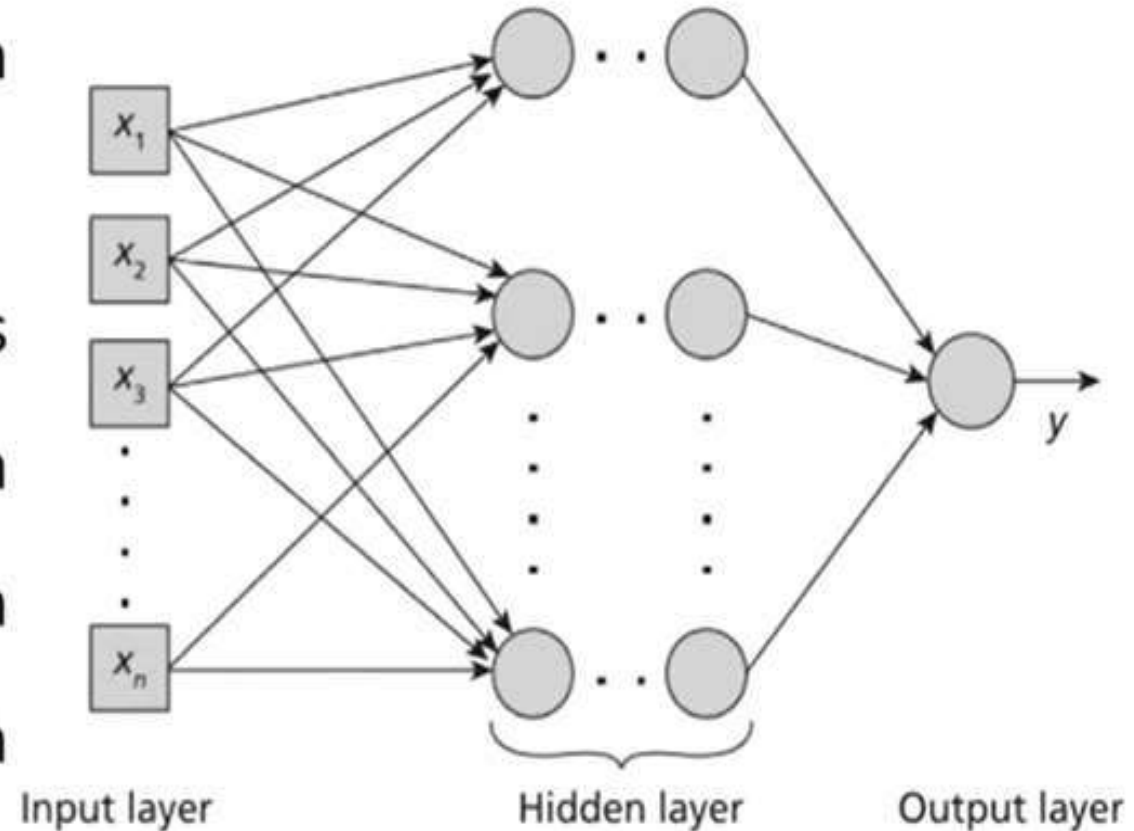
Multi-Layer Perceptron Learning Algorithm

- The input layer is the visible layer.
- It just passes the input to the next layer.
- The layers following the input layer are the hidden layers.
- The hidden layers neither directly receive inputs nor send outputs to the external environment.
- The final layer is the output layer which outputs a single value or a vector of values.



Multi-Layer Perceptron Learning Algorithm

- The activation functions used in the layers can be linear or non-linear depending on the type of the problem modelled.
- Typically, a sigmoid activation function is used if the problem is a binary classification problem and a softmax activation function is used in a multi-class classification problem.



Multi-Layer Perceptron Learning Algorithm

Input: Input vector (x_1, x_2, \dots, x_n)

Output: Y_n

Learning rate: α

Assign random weights and biases for every connection in the network in the range $[-0.5, +0.5]$.

Step 1: Forward Propagation

1. Calculate Input and Output in the Input Layer:

(Input layer is a direct transfer function, where the output of the node equals the input).

where,

Input at Node j ' I_j ' in the Input Layer is

$$I_j = x_j$$

x_j is the input received at Node j

Output at Node j ' O_j ' in the Input Layer is

$$O_j = I_j$$

Multi-Layer Perceptron Learning Algorithm

Net Input at Node j in the Output Layer is

$$\underline{I_j} = \sum_{i=1}^n O_i w_{ij} + x_0 \times \theta_j$$

where,

O_i is the output from Node i

w_{ij} is the weight in the link from Node i to Node j

x_0 is the input to bias node '0' which is always assumed as 1

θ_j is the weight in the link from the bias node '0' to Node j

Output at Node j

$$O_j = \frac{1}{1 + e^{-I_j}}$$

where,

I_j is the input received at Node j

Multi-Layer Perceptron Learning Algorithm

3. Estimate error at the node in the *Output Layer*:

$$\text{Error} = O_{\text{Desired}} - O_{\text{Estimated}}$$

where,

O_{Desired} is the desired output value of the Node in the Output Layer

$O_{\text{Estimated}}$ is the estimated output value of the Node in the Output Layer

Multi-Layer Perceptron Learning Algorithm

Step 2: Backward Propagation

1. Calculate Error at each node:

For each Unit k in the Output Layer

$$\text{Error}_k = O_k (1 - O_k) (O_{\text{Desired}} - O_k)$$

where,

O_k is the output value at Node k in the Output Layer.

O_{Desired} is the desired output value of the Node in the Output Layer.

For each unit j in the Hidden Layer

$$\text{Error}_j = O_j (1 - O_j) \sum_k \text{Error}_k w_{jk}$$

where,

O_j is the output value at Node j in the Hidden Layer.

Error_k is the error at Node k in the Output Layer.

w_{jk} is the weight in the link from Node j to Node k .

Multi-Layer Perceptron Learning Algorithm

2. Update all weights and biases:

Update weights

$$\begin{aligned}\Delta w_{ij} &= \alpha \times \text{Error}_j \times O_i \\ w_{ij} &= w_{ij} + \Delta w_{ij}\end{aligned}$$

where,

O_i is the output value at Node i .

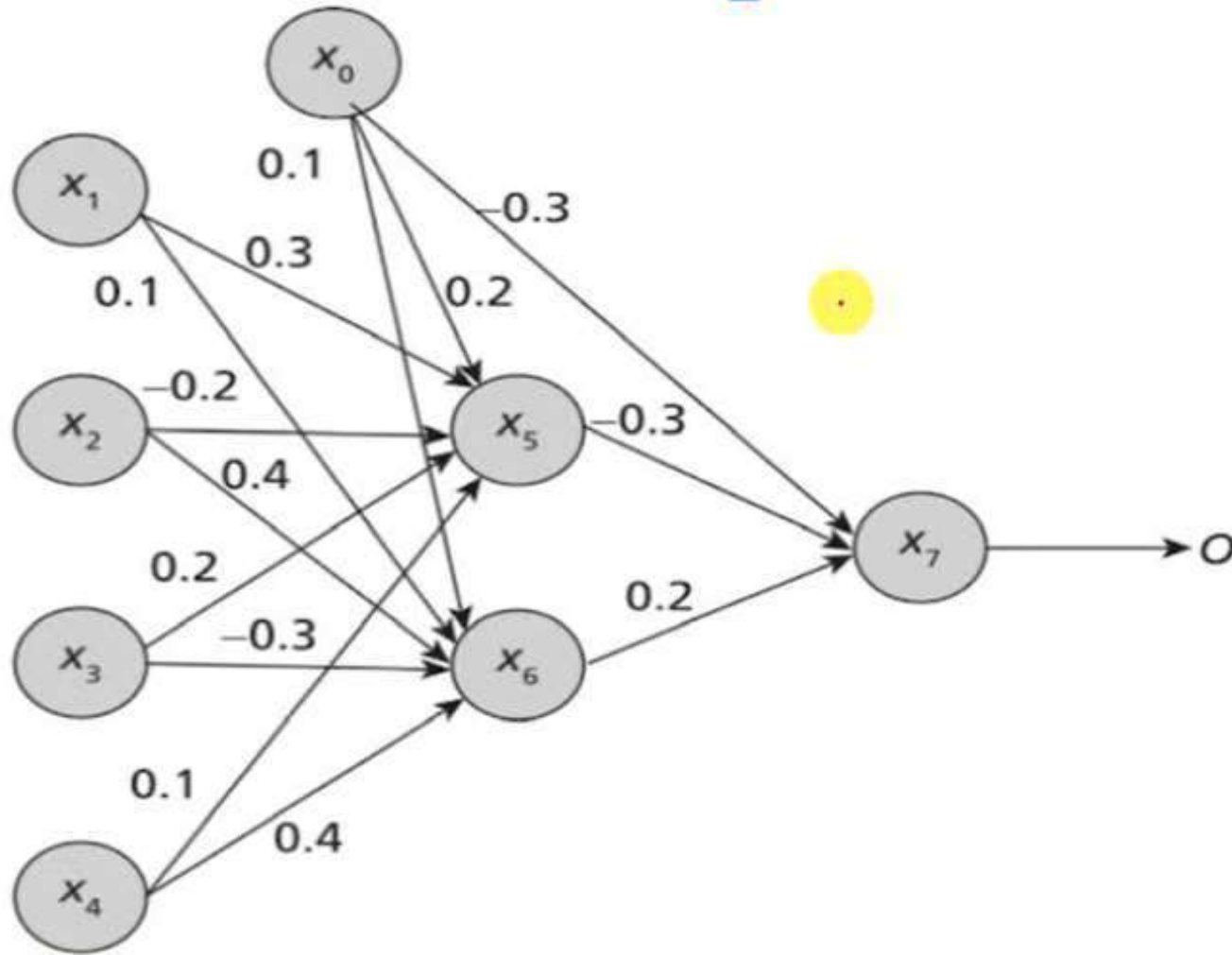
Error_j is the error at Node j .

α is the learning rate.

w_{ij} is the weight in the link from Node i to Node j .

Δw_{ij} is the difference in weight that has to be added to w_{ij} .

Multi-Layer Perceptron Learning



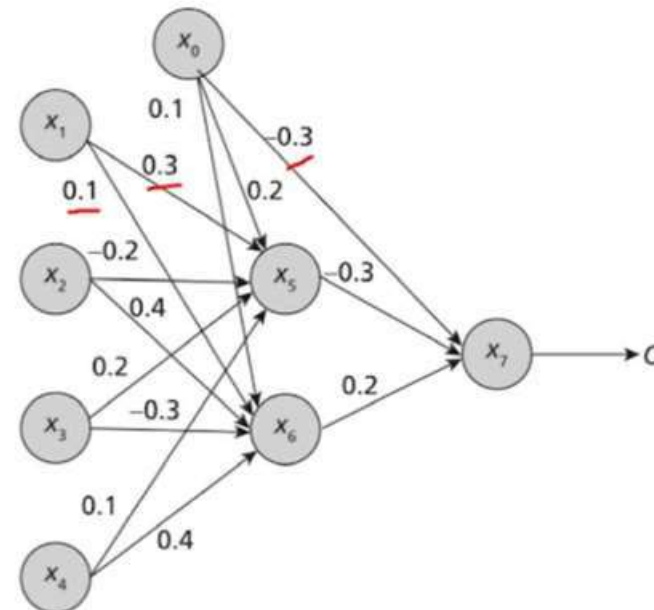
**Solved
Example**

Multi-Layer Perceptron Learning – Solved Example

x_1	x_2	x_3	x_4	w_{15}	w_{16}	w_{25}	w_{26}	w_{35}	w_{36}	w_{45}	w_{46}	w_{57}	w_{67}	θ_5	θ_6	θ_7
1	1	0	1	0.3	0.1	-0.2	0.4	0.2	-0.3	0.1	0.4	-0.3	0.2	0.2	0.1	-0.3

- Calculate Input and Output in the Input Layer
Net Input and Output Calculation


Input layer	I_j	O_j
x_1	1	1
x_2	1	1
x_3	0	0
x_4	1	1



Multi-Layer Perceptron Learning – Solved Example

x_1	x_2	x_3	x_4	w_{15}	w_{16}	w_{25}	w_{26}	w_{35}	w_{36}	w_{45}	w_{46}	w_{57}	w_{67}	θ_5	θ_6	θ_7
1	1	0	1	0.3	0.1	-0.2	0.4	0.2	-0.3	0.1	0.4	-0.3	0.2	0.2	0.1	-0.3

2. Calculate Net Input and Output in the Hidden Layer and Output Layer as

Unit _j	Net Input I_j	Output O_j
x_5 	$I_5 = x_1 \times w_{15} + x_2 \times w_{25} + x_3 \times w_{35} + x_4 \times w_{45} + x_0 \times \theta_5$ $I_5 = 1 \times 0.3 + 1 \times -0.2 + 0 \times 0.2 + 1 \times 0.1 + 1 \times 0.2 = 0.4$	$O_5 = \frac{1}{1 + e^{-I_5}} = \frac{1}{1 + e^{-0.4}} = 0.599$
x_6	$I_6 = x_1 \times w_{15} + x_2 \times w_{26} + x_3 \times w_{36} + x_4 \times w_{46} + x_0 \times \theta_6$ $I_6 = 1 \times 0.3 + 1 \times 0.4 + 0 \times -0.3 + 1 \times 0.4 + 1 \times 0.1 = 1.2$	$O_6 = \frac{1}{1 + e^{-I_6}} = \frac{1}{1 + e^{-1.2}} = 0.769$
x_7	$I_7 = O_5 \times w_{57} + O_6 \times w_{67} + x_0 \times \theta_7$ $I_7 = 0.599 \times -0.3 + 0.769 \times 0.2 + 1 \times -0.3 = -0.326$	$O_7 = \frac{1}{1 + e^{-I_7}} = \frac{1}{1 + e^{0.326}} = 0.419$

Multi-Layer Perceptron Learning – Solved Example

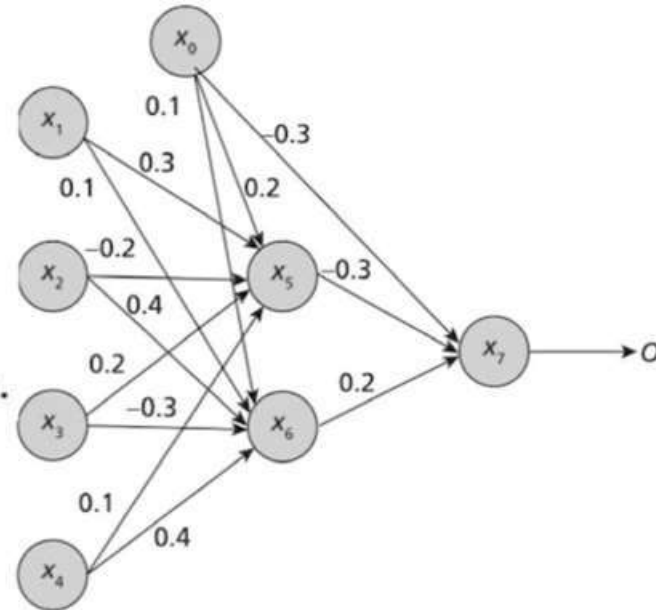
x_1	x_2	x_3	x_4	w_{15}	w_{16}	w_{25}	w_{26}	w_{35}	w_{36}	w_{45}	w_{46}	w_{57}	w_{67}	θ_5	θ_6	θ_7
1	1	0	1	0.3	0.1	-0.2	0.4	0.2	-0.3	0.1	0.4	-0.3	0.2	0.2	0.1	-0.3

3. Calculate Error = $O_{desired} - O_{Estimated}$

So, error for this network is:

$$\text{Error} = O_{desired} - O_7 = 1 - 0.419 = 0.581$$

So, we need to back propagate to reduce the error.



Multi-Layer Perceptron Learning – Solved Example

x_1	x_2	x_3	x_4	w_{15}	w_{16}	w_{25}	w_{26}	w_{35}	w_{36}	w_{45}	w_{46}	w_{57}	w_{67}	θ_5	θ_6	θ_7
1	1	0	1	0.3	0.1	-0.2	0.4	0.2	-0.3	0.1	0.4	-0.3	0.2	0.2	0.1	-0.3

Step 2: Backward Propagation

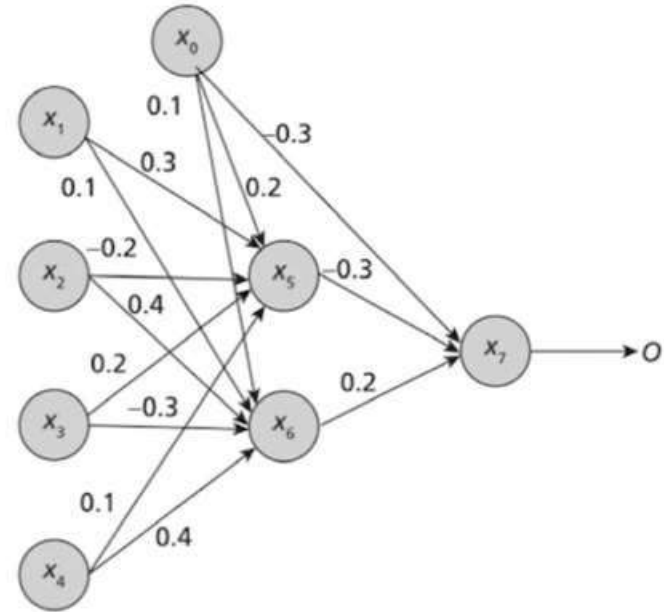
1. Calculate Error at each node

For each unit k in the output layer, calculate:

$$\checkmark \text{Error}_k = O_k (1 - O_k) (O_{\text{desired}} - O_k)$$

For each unit j in the hidden layer, calculate:

$$\checkmark \text{Error}_j = O_j (1 - O_j) \sum_k \text{Error}_k w_{jk}$$



Multi-Layer Perceptron Learning – Solved Example

x_1	x_2	x_3	x_4	w_{15}	w_{16}	w_{25}	w_{26}	w_{35}	w_{36}	w_{45}	w_{46}	w_{57}	w_{67}	θ_5	θ_6	θ_7
1	1	0	1	0.3	0.1	-0.2	0.4	0.2	-0.3	0.1	0.4	-0.3	0.2	0.2	0.1	-0.3

Error Calculation for Each Unit in the Output Layer and Hidden Layer

For Output Layer Unit _k	Error _k
x_7	$\text{Error}_7 = O_7(1 - O_7)(Y_n - O_7)$ $= 0.419 \times (1 - 0.419) \times (1 - 0.419) = 0.141$
For Hidden Layer Unit _j	Error _j
x_6	$\text{Error}_6 = O_6(1 - O_6) \sum_k \text{Error}_k w_{jk} = O_6(1 - O_6) \text{Error}_7 w_{67}$ $= 0.769(1 - 0.769) \times 0.2 \times 0.141 = 0.005$
x_5	$\text{Error}_5 = O_5(1 - O_5) \sum_k \text{Error}_k w_{jk} = O_5(1 - O_5) \text{Error}_7 w_{57}$ $= 0.599(1 - 0.599) \times 0.141 \times -0.3 = -0.0101$

Multi-Layer Perceptron Learning – Solved Example

x_1	x_2	x_3	x_4	w_{15}	w_{16}	w_{25}	w_{26}	w_{35}	w_{36}	w_{45}	w_{46}	w_{57}	w_{67}	θ_5	θ_6	θ_7
1	1	0	1	0.3	0.1	-0.2	0.4	0.2	-0.3	0.1	0.4	-0.3	0.2	0.2	0.1	-0.3

2. Update weight using the below formula:

Learning rate $\alpha = 0.8$. ✓

$$w_{ij} = w_{ij} + \alpha \times \text{Error}_j \times O_i$$

The updated weights and bias

w_{ij}	$w_{ij} = w_{ij} + \alpha \times \text{Error}_j \times O_i$	New Weight
w_{15}	$w_{15} = w_{15} + 0.8 \times \text{Error}_5 \times O_1$ $= 0.3 + 0.8 \times -0.0101 \times 1$	0.292
w_{16}	$w_{16} = w_{16} + 0.8 \times \text{Error}_6 \times O_1$ $= 0.1 + 0.8 \times 0.005 \times 1$	0.104
w_{25}	$w_{25} = w_{25} + 0.8 \times \text{Error}_5 \times O_2$ $= -0.2 + 0.8 \times -0.0101 \times 1$	-0.208
w_{26}	$w_{26} = w_{26} + 0.8 \times \text{Error}_6 \times O_2$ $= 0.4 + 0.8 \times 0.005 \times 1$	0.404

Multi-Layer Perceptron Learning – Solved Example

2. Update weight using the below formula:

Learning rate $\alpha = 0.8$.

$$w_{ij} = w_{ij} + \alpha \times \text{Error}_j \times O_i$$

The updated weights and bias

w_{ij}	$w_{ij} = w_{ij} + \alpha \times \text{Error}_j \times O_i$	New Weight
w_{35}	$w_{35} = w_{35} + 0.8 \times \text{Error}_5 \times O_3$ $= 0.2 + 0.8 \times -0.0101 \times 0$	0.2
w_{36}	$w_{36} = w_{36} + 0.8 \times \text{Error}_6 \times O_3$ $= -0.3 + 0.8 \times 0.005 \times 0$	-0.3
w_{45}	$w_{45} = w_{45} + 0.8 \times \text{Error}_5 \times O_4$ $= 0.1 + 0.8 \times -0.0101 \times 1$	0.092
w_{46}	$w_{46} = w_{46} + 0.8 \times \text{Error}_6 \times O_4$ $= 0.4 + 0.8 \times 0.005 \times 1$	0.404
w_{57}	$w_{57} = w_{57} + 0.8 \times \text{Error}_7 \times O_5$ $= -0.3 + 0.8 \times 0.141 \times 0.599$	-0.232
w_{67}	$w_{67} = w_{67} + 0.8 \times \text{Error}_7 \times O_6$ $= 0.2 + 0.8 \times 0.141 \times 0.769$	0.287

Multi-Layer Perceptron Learning – Solved Example

x_1	x_2	x_3	x_4	w_{15}	w_{16}	w_{25}	w_{26}	w_{35}	w_{36}	w_{45}	w_{46}	w_{57}	w_{67}	θ_5	θ_6	θ_7
1	1	0	1	0.3	0.1	-0.2	0.4	0.2	-0.3	0.1	0.4	-0.3	0.2	0.2	0.1	-0.3

Update bias using the below formula:

$$\theta_j = \theta_j + \alpha \times \text{Error}_j$$

θ_j	$\theta_j = \theta_j + \alpha \times \text{Error}_j$	New Bias
θ_5	$\theta_5 = \theta_5 + \alpha \times \text{Error}_5$ $= 0.2 + 0.8 \times -0.0101$	0.192
θ_6	$\theta_6 = \theta_6 + \alpha \times \text{Error}_6$ $= 0.1 + 0.8 \times 0.005$	0.104
θ_7	$\theta_7 = \theta_7 + \alpha \times \text{Error}_7$ $= -0.3 + 0.8 \times 0.141$	-0.187

Multi-Layer Perceptron Learning – Solved Example

2. Calculate Net Input and Output in the Hidden Layer and Output Layer

Net Input and Output Calculation in the Hidden Layer and Output Layer

Unit j	Net Input I_j	Output O_j
x_5	$I_5 = x_1 \times w_{15} + x_2 \times w_{25} + x_3 \times w_{35} + x_4 \times w_{45} + x_0 \times \theta_5$ $I_5 = 1 \times 0.292 + 1 \times -0.208 + 0 \times 0.2 + 1 \times 0.092 + 1 \times 0.192 = 0.368$	$O_5 = \frac{1}{1 + e^{-I_5}} = \frac{1}{1 + e^{-0.368}} = 0.591$
x_6	$I_6 = x_1 \times w_{16} + x_2 \times w_{26} + x_3 \times w_{36} + x_4 \times w_{46} + x_0 \times \theta_6$ $I_6 = 1 \times 0.292 + 1 \times 0.404 + 0 \times -0.3 + 1 \times 0.404 + 1 \times 0.104 = 1.204$	$O_6 = \frac{1}{1 + e^{-I_6}} = \frac{1}{1 + e^{-1.204}} = 0.7692$
x_7	$I_7 = O_5 \times w_{57} + O_6 \times w_{67} + x_0 \times \theta_7$ $I_7 = 0.591 \times -0.232 + 0.7692 \times 0.287 + 1 \times -0.187 = -0.326$	$O_7 = \frac{1}{1 + e^{-I_7}} = \frac{1}{1 + e^{0.1034}} = 0.474$

Multi-Layer Perceptron Learning – Solved Example

- The output we receive in the network at node 7 is 0.474

$$\text{Error} = 1 - 0.474 = 0.526$$

- Now, when we compare the error, we get in the previous iteration and in the current iteration,

$$\text{Error reduced is } 0.581 - 0.526 = 0.055$$

- It is visible that the network has learnt and reduced the error by 0.055.
- Thus, the training is continued for a predefined number of epochs or until the training error is reduced below a threshold value.

1.5 Computational graphs

Computational graphs are a type of graph that can be used to represent mathematical expressions. This is similar to descriptive language in the case of deep learning models, providing a functional description of the required computation.

In general, the computational graph is a directed graph that is used for expressing and evaluating mathematical expressions.

These can be used for two different types of calculations:

- Forward computation

- Backward computation

Computational graphs

key terminologies in computational graphs:

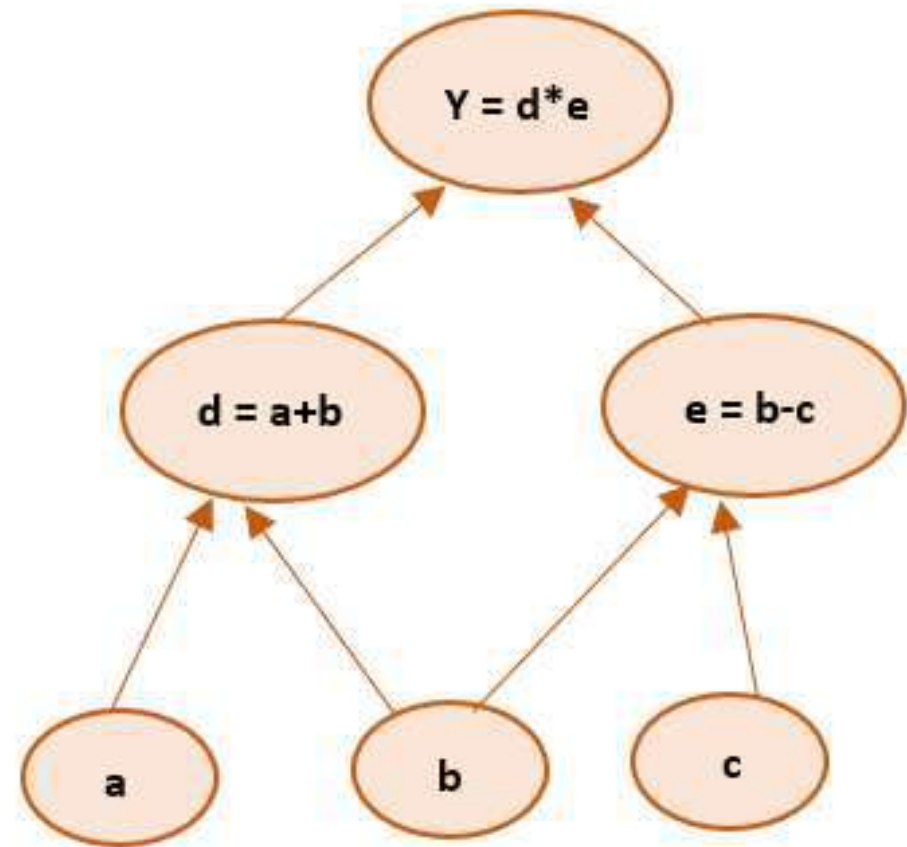
- A variable is represented by a node in a graph. It could be a scalar, vector, matrix, tensor, or even another type of variable.
- A function argument and data dependency are both represented by an edge. These are similar to node pointers.
- A simple function of one or more variables is called an operation. There is a set of operations that are permitted. Functions that are more complex than these operations in this set can be represented by combining multiple operations.

Computational graphs

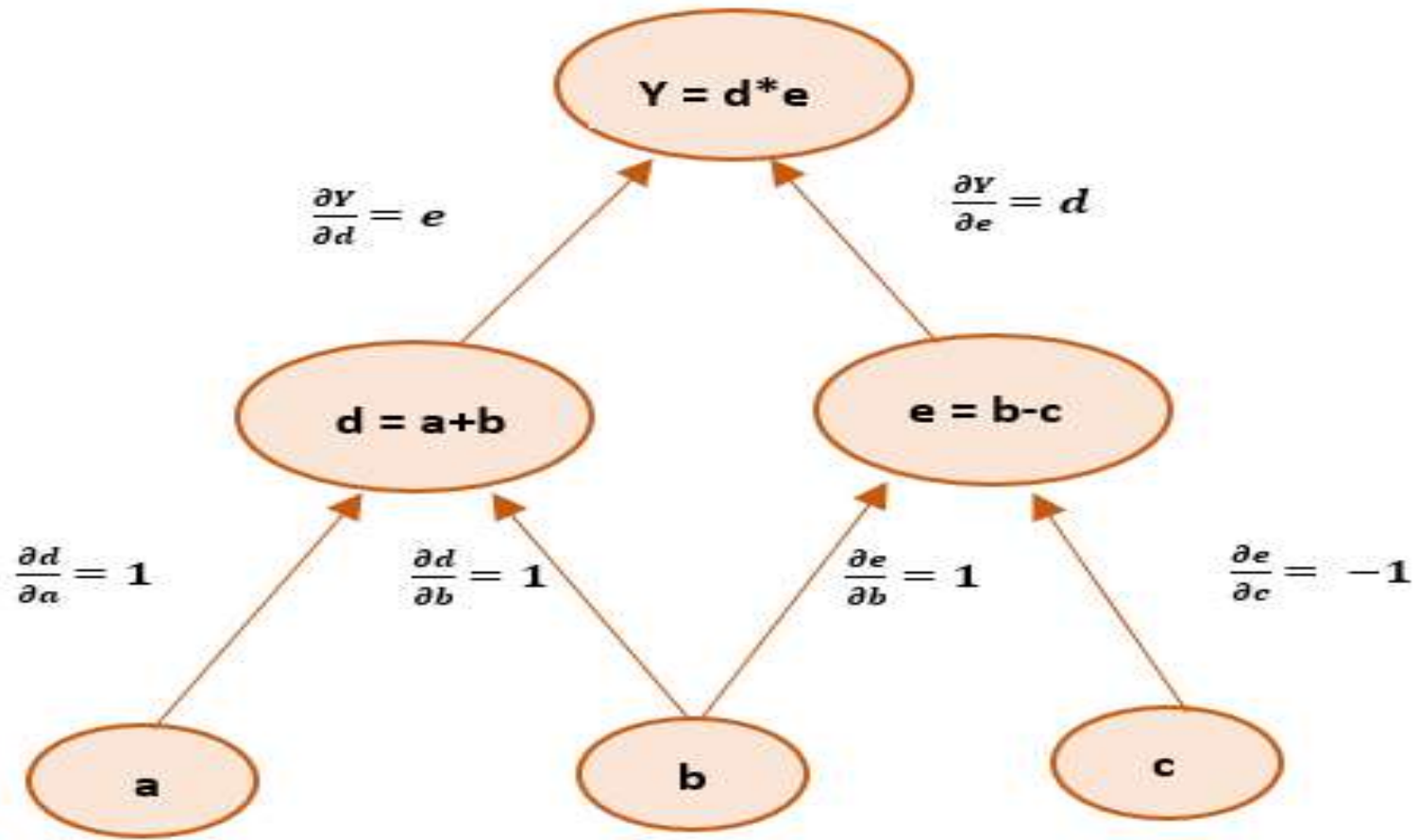
For example, consider this
: $Y = (a+b) * (b-c)$

For better understanding, we introduce two variables d and e such that every operation has an output variable. We now have:

$d = a + b$
 $e = b - c$
 $Y = d * e$

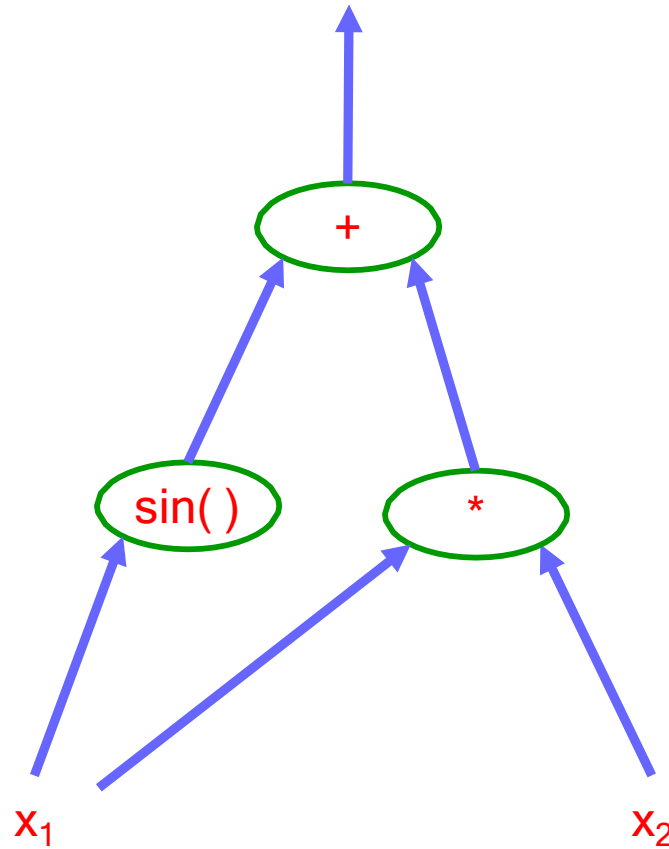


Computational graphs



Computational Graph: Example

$$f(x_1, x_2) = x_1x_2 + \sin(x_1)$$



1.6 Numerical Stability in Deep Learning

Numerical stability refers to how well an algorithm can handle small numerical errors (such as rounding errors) without leading to incorrect or extreme results. In deep learning, instability often arises due to the following:

1. Exploding and Vanishing Gradients
2. Ill-Conditioning
3. Floating Point Precision Issues

Numerical Stability in Deep Learning

1. Exploding and Vanishing Gradients

- **Exploding Gradients:** When gradients become excessively large during backpropagation, leading to divergence in weight updates.
- **Vanishing Gradients:** When gradients shrink towards zero, preventing deep layers from learning.

Solutions:

- **Proper weight initialization (e.g., Xavier, He initialization)**
- **Gradient clipping** (limits the maximum gradient value)
- **Batch normalization** (normalizes activations to stabilize training)
- **Residual connections** (helps gradients flow better in deep networks)

Numerical Stability in Deep Learning

2. Ill-Conditioning

- This occurs when small changes in input lead to disproportionately large changes in output, making optimization difficult.

Solutions:

- Use well-conditioned loss functions
- Ensure appropriate scaling of inputs and weights
- Use adaptive optimizers like Adam or RMSprop

Numerical Stability in Deep Learning

3. Floating Point Precision Issues

- Deep learning operations involve floating-point arithmetic, where numerical precision (e.g., single vs. double precision) affects results.
- **Solutions:**
- Mixed-precision training (combining lower and higher precision for speed and stability)
- Regularization techniques like weight decay to prevent extreme values

Weight Initialization in Deep Learning

Both numerical stability and initialization play a significant role in deep learning efficiency. Proper weight initialization prevents gradient problems, while stability techniques like batch normalization and gradient clipping ensure smooth training.

1.7 Generalization in Deep Learning

- **Hyperparameter Tuning**
- **Normalization**
- **Regularization: Dropout, Data Augmentation, Early Stopping**
- **Optimizers**

Parameters	Hyperparameters
parameters are the internal variables within a model that are automatically learned from the training data during the learning process, like weights and biases .	hyperparameters are variables set before training begins and control the learning process itself, such as learning rate, batch size, and the number of hidden layers , which need to be manually tuned to optimize model performance.
Parameters are updated during training based on the data	Hyperparameters are set before training starts and remain fixed throughout.
Parameters are automatically adjusted by the learning algorithm	Hyperparameters are manually tuned by the user

Batch size	The number of data samples processed before updating the model parameters.
Learning rate	How quickly the model updates its parameters during training.
Optimizer type	The algorithm used to update the model parameters.
Number of hidden layers	How many layers a neural network has.
Activation Functions, Dropout Rate, etc.	

Normalization stabilizes training by keeping activations and gradients in a manageable range.

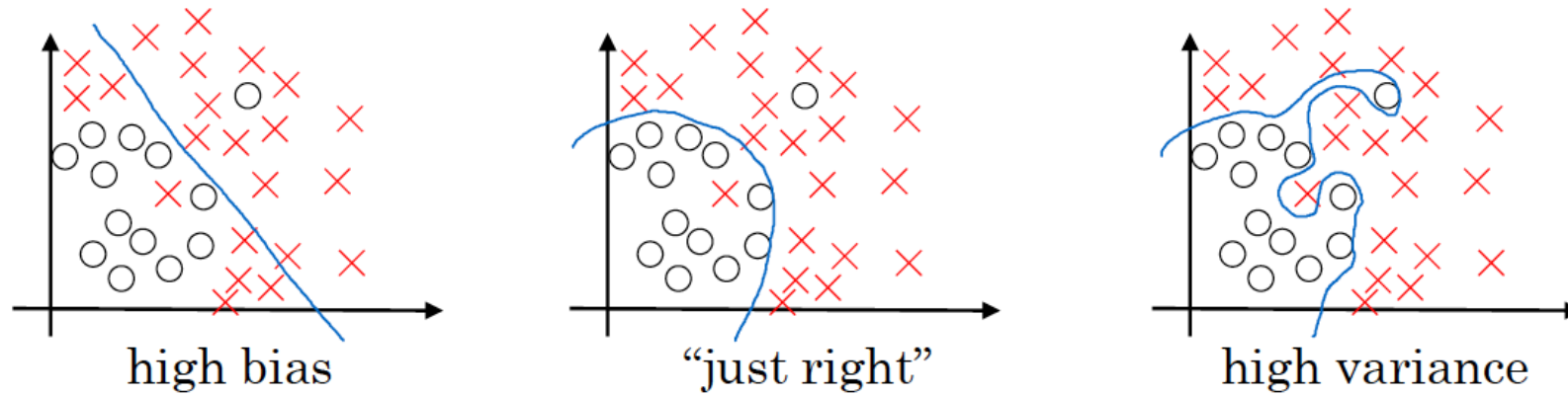
BatchNorm is widely used in CNNs, but **LayerNorm** is better for RNNs and Transformers.

Different normalizations help in different scenarios, such as **GroupNorm** for small batches and **InstanceNorm** for style transfer.

By using the right normalization technique, deep learning models train faster, generalize better, and become more stable.

Normalization Type	Best for	Pros	Cons
Batch Norm	CNNs, DNNs	Fast training, reduces internal covariate shift	Depends on batch size
Layer Norm	RNNs, Transformers	Works with variable batch sizes	Slightly slower
Instance Norm	Style transfer, GANs	Works well for per-instance normalization	May not generalize well
Group Norm	Small batch CNNs	Effective with small batch sizes	Needs tuning for groups
Input Norm	Any model	Helps convergence	Needs preprocessing

Bias and Variance: Introduction



- Bias / Variance techniques are Easy to learn, but difficult to master.
- So here the explanation of Bias / Variance:
 - If your model is underfitting (logistic regression of non linear data) it has a "high bias"
 - If your model is overfitting then it has a "high variance"
 - Your model will be alright if you balance the Bias / Variance

TOPICS IN DEEP LEARNING

Bias and Variance: Example

Training error	1%	12%	12%	0.5%
Dev error/ Test error	10%	13%	24%	1%
	High Variance	High bias	High bias and High Variance	Low Bias Low Variance
	Over fitting	Underfitting	Underfitting	Best/ Good fit

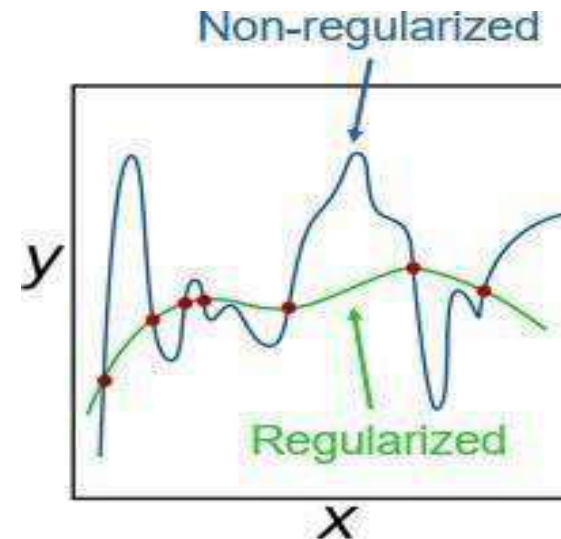
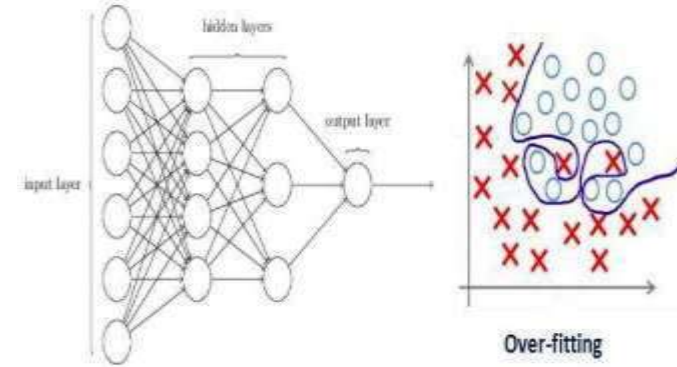
Regularization

How do you prevent overfitting?
Regularization

TOPICS IN DEEP LEARNING

Regularization

- Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.
- Regularization is a process of introducing additional information in order to prevent overfitting.



Type of Regularization

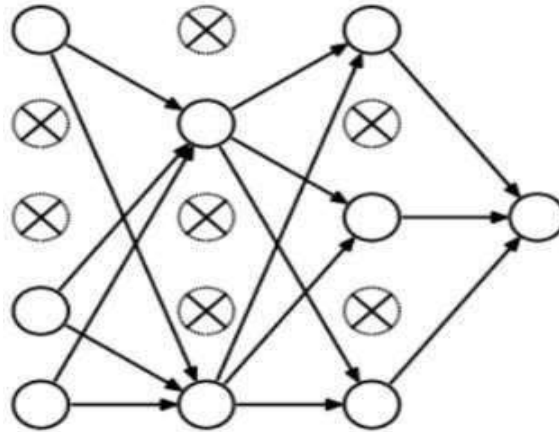
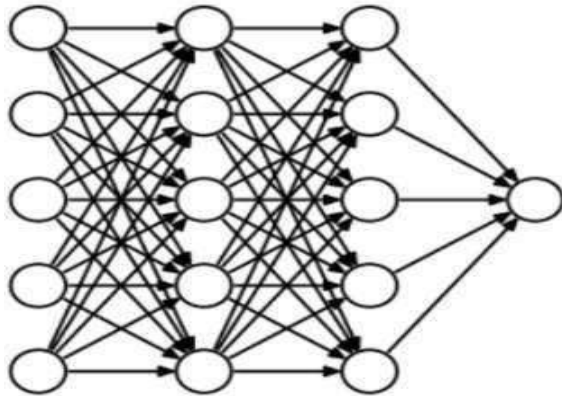
Dropout

TOPICS IN DEEP LEARNING

Dropout

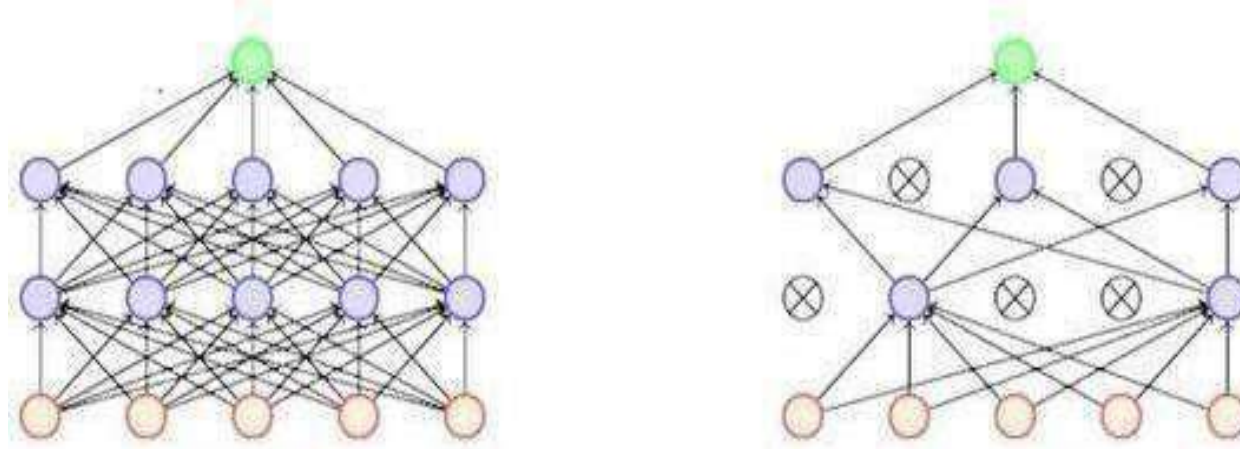
This is the one of the [most interesting types of regularization techniques](#). It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.

To understand dropout, let's say our neural network structure is akin to the one shown below



So what does dropout do? At every iteration, it **randomly selects some nodes and removes them along with all of their incoming and outgoing connections** as shown in right figure

What is Dropout?



- Dropout means leaving some units
- Temporarily remove a node and both its incoming and outgoing connections.
- This thins the neural network

Note: Given a total of “n” nodes what are the total number of thinned networks that can be formed?

$$2^n$$

TOPICS IN DEEP LEARNING

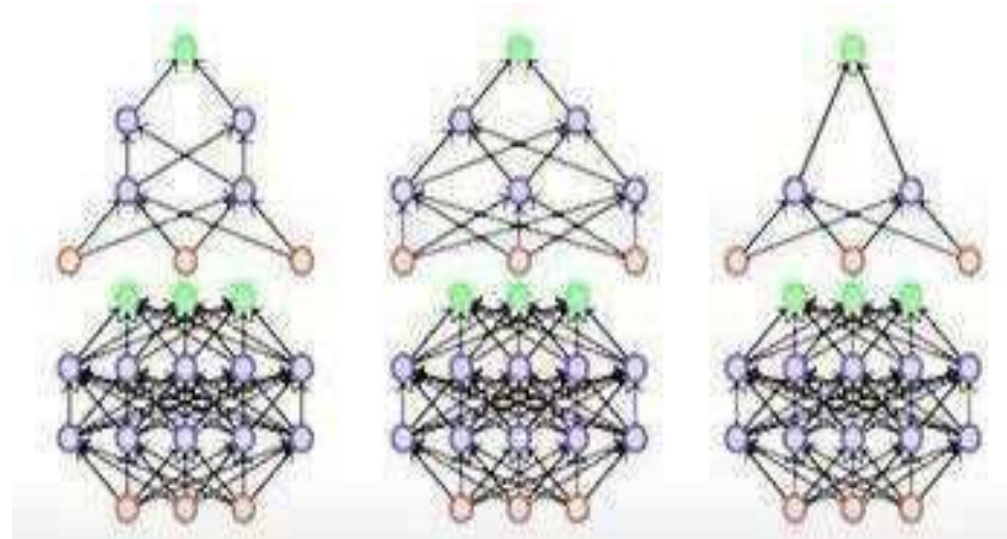
Why dropout regularization?

- Train several neural networks having different architectures
- Train multiple instances of the same network using different training samples.

“ Expensive options”

Both at training as well as at testing levels

Hence solution is dropout



Type of Regularization **Data Augmentation**

1. More training data is one more solution for over fitting
2. As getting additional data may be expensive and may not be possible
3. Flipping of all images can be one of the ways to increase the data set size.
4. Randomly zooming in and zooming out can be another way.
5. Distorting some of the images based on your application may be an another way to increase the data set size.



Affine: Translate



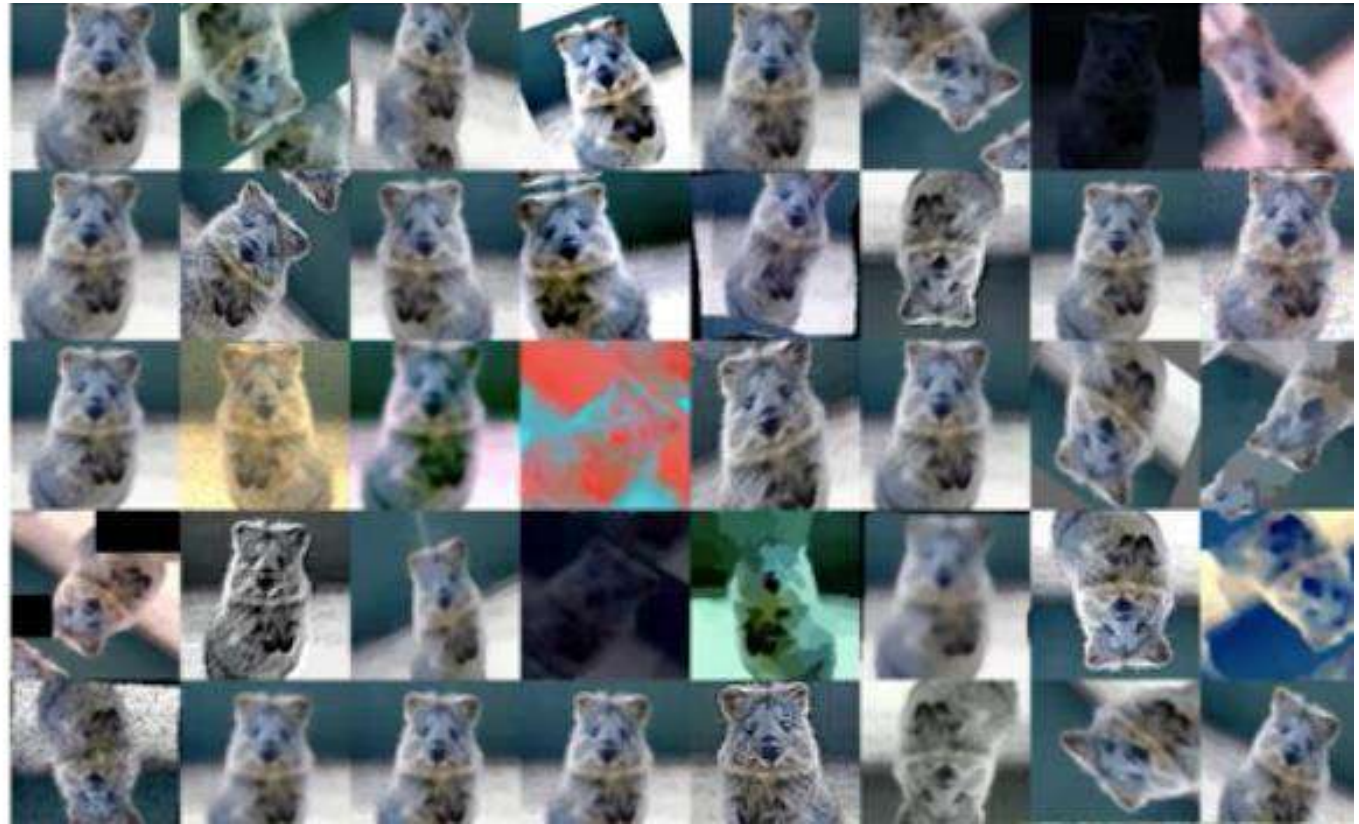
Affine: Rotate



Affine: Shear



Noise
Robustness



TOPICS IN DEEP LEARNING

Data Augmentation

Popular Augmentation Techniques:

1. Flip
2. Rotation
3. Scale
4. Crop
5. Translation
6. Gaussian Noise
7. Conditional GANs (Advanced Augmentation Technique)

Conditional GANs can transform an image from one domain to an image to another domain.

An example of conditional GANs used to transform photographs of summer sceneries to winter sceneries.



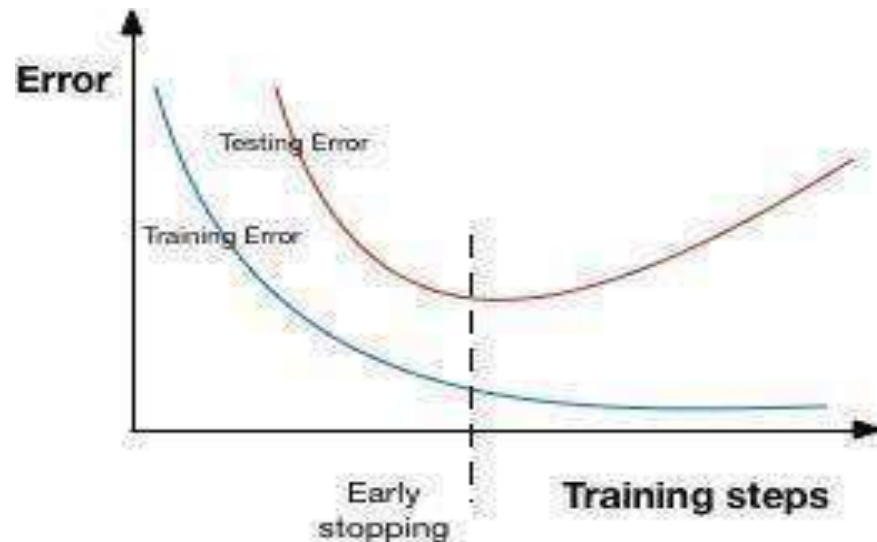
Type of Regularization

Early Stopping

TOPICS IN DEEP LEARNING

Early Stopping

Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that the performance on the validation set (or test set) is getting worse, we immediately stop the training on the model. This is known as early stopping.



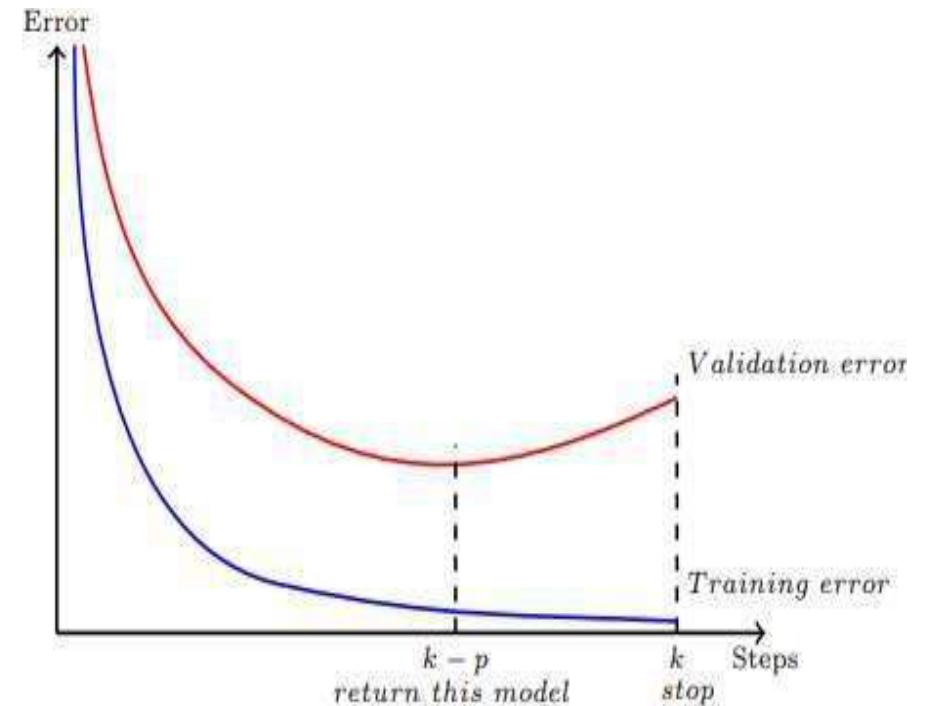
The above image, we will stop training at the dotted line since after that our model will start overfitting on the training data.

How long to train a neural network?

- Too little training will mean that the model will underfit the train and the test sets.
- Too much training will mean that the model will overfit the training dataset and have poor performance on the test set.
- A compromise is to train on the training dataset but to stop training at the point when performance on a validation dataset starts to degrade.
- This simple, effective, and widely used approach to training neural networks is called early stopping.

Early Stopping

- Track the validation error
- Have a patience parameter p
- If you are at step k and there was no improvement in validation error in the previous p steps then stop training and return the model stored at step $k - p$
- Basically, stop the training early before it drives the training error to 0 and blows up the validation error



Its advantage is that you don't need to search a hyperparameter like in other regularization approaches (like lambda in L2 regularization).

TOPICS IN DEEP LEARNING

Optimizers

TOPICS IN DEEP LEARNING

Optimizers

- Optimizers are algorithm that is used to tweak the parameters such that we reach the minimum of the loss function.
- How should one change the weights and biases is defined by the optimizers we use.
- Optimizers have been undergoing constant evolution, adding techniques that makes them perform faster and more efficiently

TOPICS IN DEEP LEARNING

Optimizers

Different types of Optimizers used frequently:

1. Gradient Descent
2. Stochastic Gradient Descent
3. Mini-Batch Gradient Descent
4. SGD with Momentum
5. Adagrad
6. RMS-Prop
7. Adam (**Adaptive Moment Estimation** -RMS prop and momentum together)

TOPICS IN DEEP LEARNING

Optimizers - Gradient Descent

Gradient Descent is something we have learnt before.

- We find the gradient of the loss function with respect to the weights and biases
- And move in small steps towards the opposite direction of the gradient (Gradient points to ascent, we want direction of descent to reach the minimum)
- Gradient Descent **iteratively reduces a loss function by moving in the direction opposite to that of steepest ascent.**

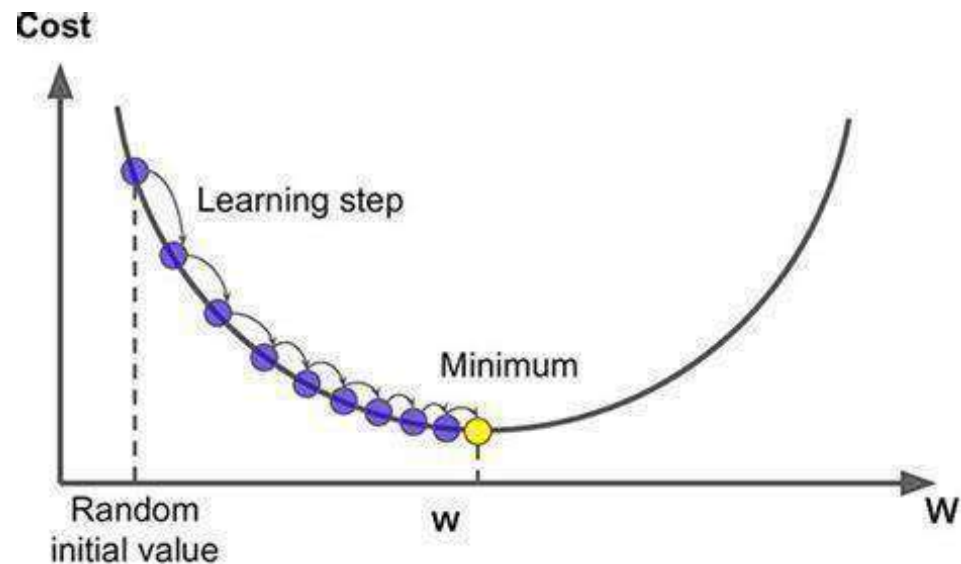
Gradient Descent Algorithm:

$$\Theta_{n+1} = \Theta_n - \alpha \frac{\partial}{\partial \Theta_n} J(\Theta_n)$$

$\Theta \rightarrow$ Parameter Vector

$J \rightarrow$ Cost Function

$\alpha \rightarrow$ Slope Parameter



TOPICS IN DEEP LEARNING

Optimizers - Gradient Descent

Advantages

- Easy to implement
- Good results most of the times

Disadvantages

- Can get stuck in local minimas
- Because this method calculates the gradient for the entire data set in one update, the calculation is very slow
- It requires large memory and it is computationally expensive

TOPICS IN DEEP LEARNING

Optimizers - Stochastic Gradient Descent

Vanilla Gradient descent, find the gradient for the entire dataset, and then updates the parameters, this method is slow and hence the convergence is slow. Further this may not be suitable for huge datasets.

Stochastic GD is a variation of Vanilla Gradient descent where, the gradient is updated every data item, instead of after going through the entire dataset.

- As the model parameters are frequently updated **parameters have high variance and fluctuations** in loss functions at different intensities
- SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation time increases. But even after increasing the number of iterations, the **computation cost is still less than that of the vanilla gradient descent optimizer.**
- If the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm

TOPICS IN DEEP LEARNING

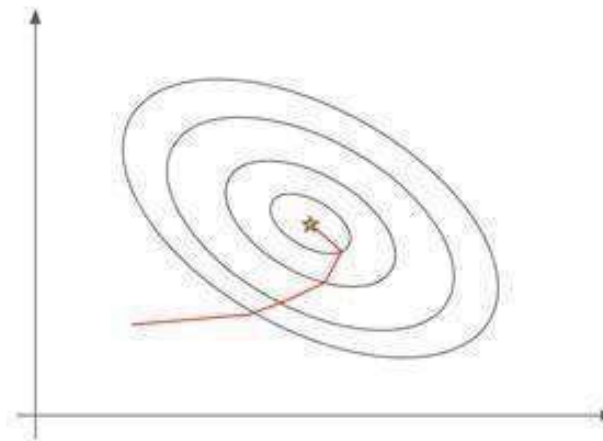
Optimizers - Stochastic Gradient Descent

Advantages

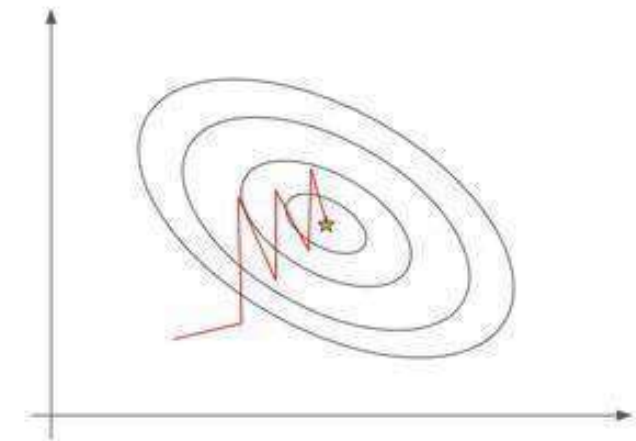
- Frequent Updation of parameters
- Less computationally expensive than Vanilla Gradient Descent
- Allows the use of large data sets as it has to update only one example at a time
- Uses less memory

Disadvantages

- The frequent can also result in noisy gradients which may cause the error to increase instead of decreasing it.
- High Variance.
- Frequent computationally expensive
- Still prone to local minima



Gradient Descent



Stochastic Gradient Descent

TOPICS IN DEEP LEARNING

Optimizers - Mini Batch Gradient Descent

Mini Batch Gradient Descent **combines the ideas of Vanilla GD and Stochastic GD.**

We shuffle the data, and choose a mini batch of K items, we then calculate gradients for this mini batch, and update the gradients.

This avoids the problem of computing gradients for the entire dataset and also the problem of frequent updation which causes noisy gradients. By finding a middle ground it is faster than both the variants.

- As the algorithm uses batching, all the training data need not be loaded in the memory, thus making the process more efficient to implement
- Moreover, the cost function in mini-batch gradient descent is noisier than the batch gradient descent algorithm but smoother than that of the stochastic gradient descent algorithm.
- Because of this, mini-batch gradient descent is ideal and provides a good balance between speed and accuracy.

TOPICS IN DEEP LEARNING

Optimizers - Mini Batch Gradient Descent

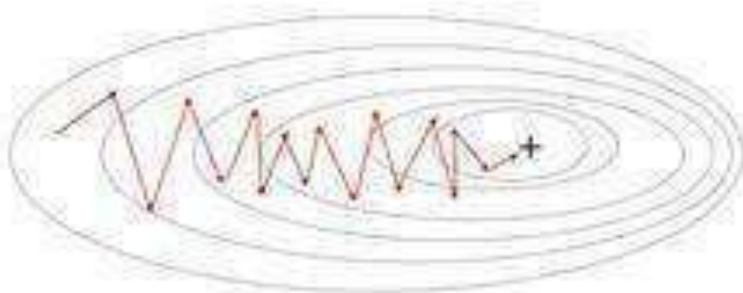
Advantages:

- Faster and more efficient than previous variants of GD
- More efficient gradient calculation
- Requires less amount of memory (Loads only one batch per updation)

Disadvantages:

- Still prone to local minima
- It introduces another hyperparameter K (the batch size)

Stochastic Gradient Descent



Mini-Batch Gradient Descent

