

# Unit – 5

## Learning With Neural Networks

# Topics

- Towards Cognitive Machine,
- Neuron Models: Biological Neuron, Artificial Neuron, Mathematical Model,
- Network Architectures: Feed forward Networks, Recurrent Networks,
- Perceptron's, Linear Neuron and the Widrow-Hoff Learning Rule,
- The Error-Correction Delta Rule,
- Multi-Layer Perceptron (MLP) Networks and the Error Backpropagation Algorithm,
- Multi-Class Discrimination with MLP Networks

# Towards cognitive machine

- Human intelligence possesses robust attributes with complex sensory, control, affective (emotional processes), and cognitive (thought processes) aspects of information processing and decision making.
- There are over a hundred billion biological neurons in our central nervous system (CNS), playing a key role in these functions.
- CNS obtains information from the external environment via numerous natural sensory mechanisms—vision, hearing, touch, taste, and smell.
- With the help of cognitive computing, it assimilates the information and offers the right interpretation.

# Towards Cognitive Machine (cont..)

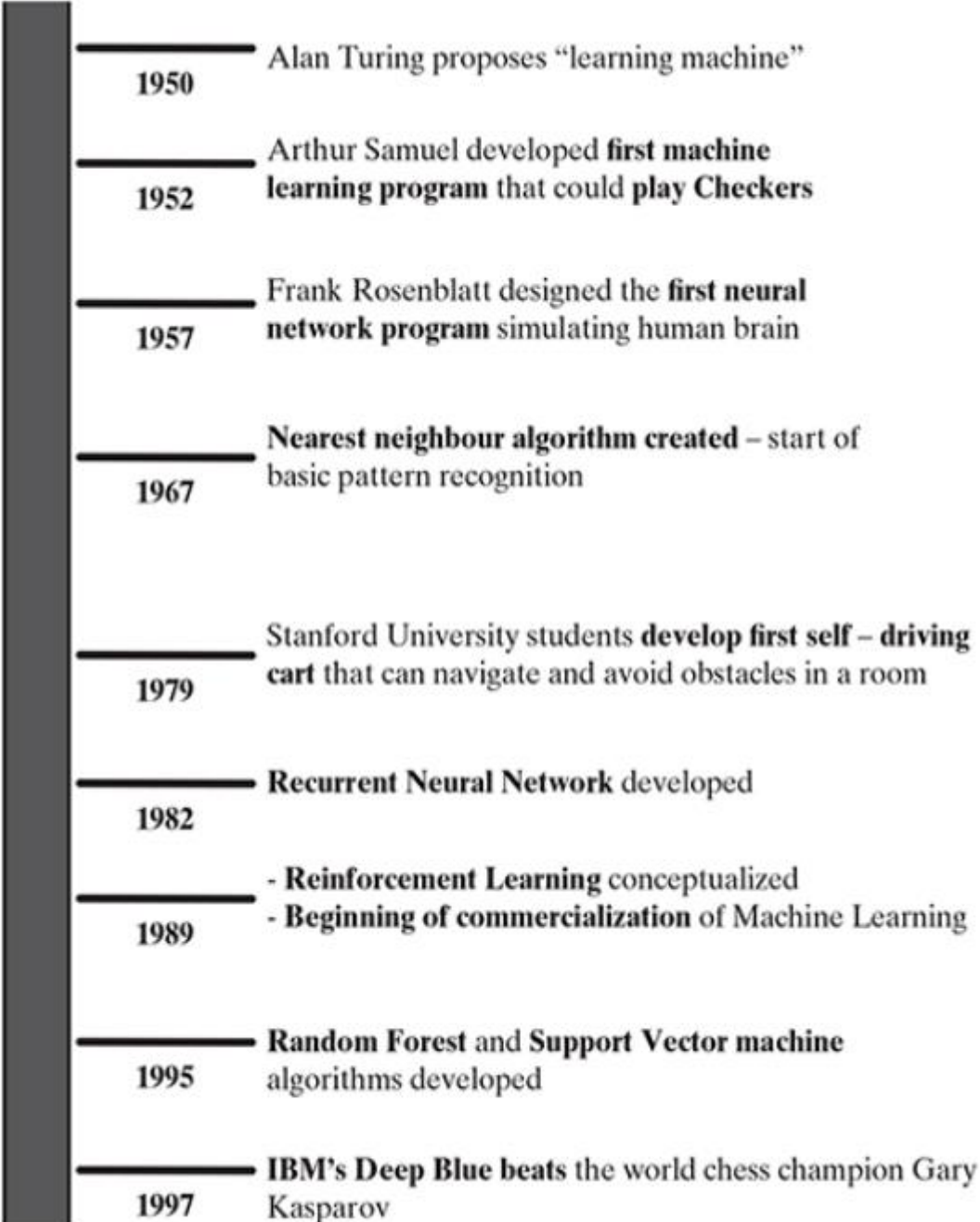
- The cognitive process then progresses towards some attributes, such as learning, recollection, and reasoning, which results in proper actions via muscular control.
- **The aim of system scientists is to create an intelligent cognitive system on the basis of this limited understanding of the brain—”a system that can help human beings to perform all kinds of tasks requiring decision making”.**
- Various new computing theories of the neural networks field have been developing, which it is hoped will be capable of providing a thinking machine.

# Towards Cognitive Machine – From perceptron to deep networks

- Historically, research in neural networks was inspired by the desire to produce artificial systems capable of sophisticated ‘intelligent’ processing similar to the human brain.
- The perceptron is the earliest of the artificial neural networks paradigms. Frank Rosenblatt built this learning machine device in hardware in 1958.
- In 1959, Bernard Widrow and Marcian Hoff developed a learning rule, sometimes known as Widrow-Haff rule, for ADALINE (ADaptive LINear Elements). Their learning rule was simple and yet elegant.

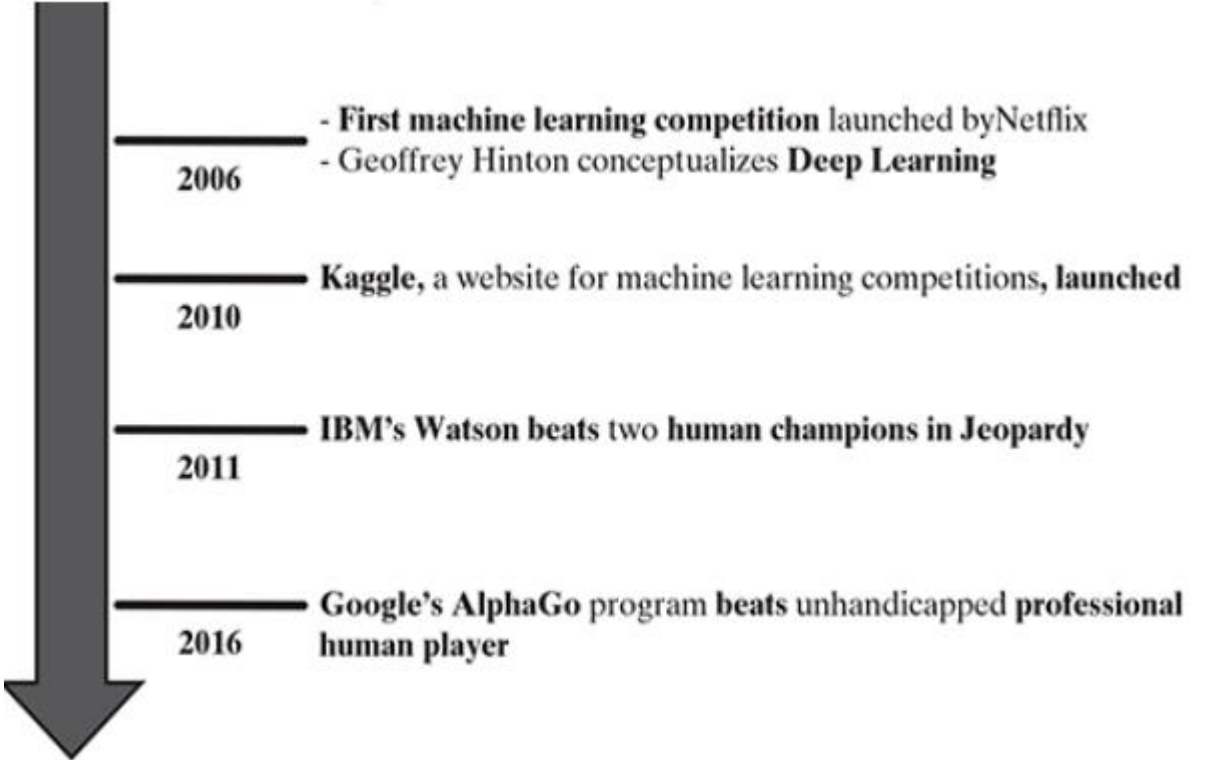
- In 1982, John Hopfield introduced a recurrent-type neural network that was based on the interaction of neurons through a feedback mechanism.
- The back-propagation learning rule arrived on the neural-network scene at approximately the same time from several independent sources.
- Essentially a refinement of the Widrow-Hoff learning rule, the back-propagation learning rule provided a systematic means for training multilayer feedforward networks.
- As the research in neural networks is evolving, more and more types of networks are being introduced. For reasonably complex problems, neural networks with back-propagation learning have serious limitations

- The learning speed of these feedforward neural networks is, in general, far slower than required and it has been a major bottleneck in their applications.
- Two reasons behind this limitation may be: (i) the slow gradient-based learning algorithms extensively used to train neural networks, and (ii) all the parameters of the network are tuned iteratively by using learning algorithms.
- “A new learning algorithm was proposed in 2006 called Extreme Learning Machine (ELM), for single hidden layer feedforward neural networks, in which the weights connecting input to hidden nodes are randomly chosen and never updated and weights connecting hidden nodes to output are analytically determined”.
- The subject of cognitive machines is in an exciting state of research and we believe that we are slowly progressing towards the development of truly intelligent systems. A step towards realizing strong AI has been taken through the recent research in ‘deep learning’.



A vertical timeline on the left side of the image, marked by a thick grey bar. It lists milestones from 1950 to 1997. Each year is preceded by a horizontal line segment. The text for each year is to the right of the line.

- 1950 Alan Turing proposes “learning machine”
- 1952 Arthur Samuel developed **first machine learning program** that could **play Checkers**
- 1957 Frank Rosenblatt designed the **first neural network program** simulating human brain
- 1967 **Nearest neighbour algorithm created** – start of basic pattern recognition
- 1979 Stanford University students **develop first self – driving cart** that can navigate and avoid obstacles in a room
- 1982 **Recurrent Neural Network** developed
- 1989
  - **Reinforcement Learning** conceptualized
  - **Beginning of commercialization** of Machine Learning
- 1995 **Random Forest** and **Support Vector machine** algorithms developed
- 1997 **IBM’s Deep Blue** beats the world chess champion Gary Kasparov



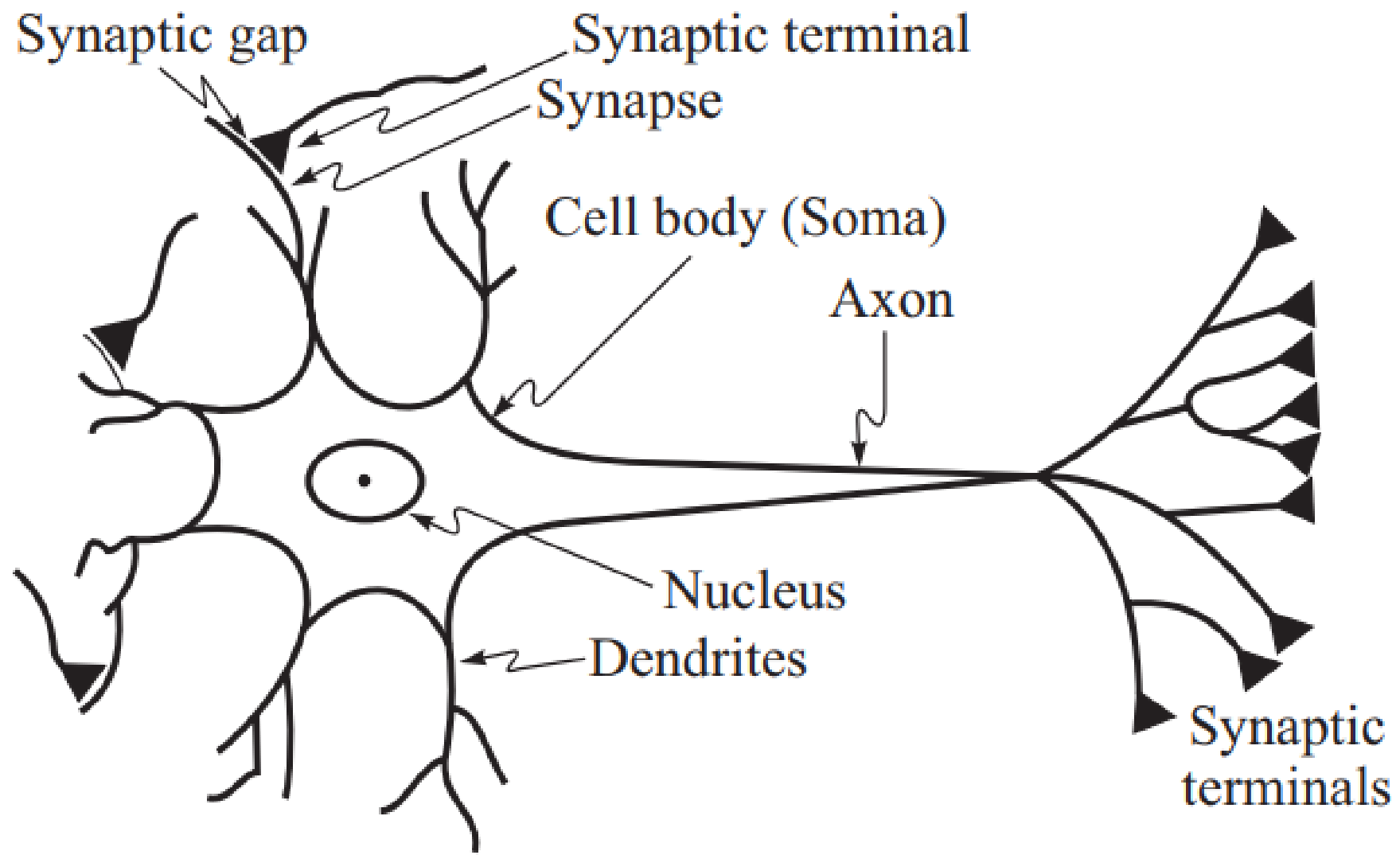
A vertical timeline on the right side of the image, marked by a thick grey bar with a downward-pointing arrow at the bottom. It lists milestones from 2006 to 2016. Each year is preceded by a horizontal line segment. The text for each year is to the right of the line.

- 2006
  - **First machine learning competition** launched by Netflix
  - Geoffrey Hinton conceptualizes **Deep Learning**
- 2010 **Kaggle**, a website for machine learning competitions, **launched**
- 2011 **IBM’s Watson** beats two **human champions in Jeopardy**
- 2016 **Google’s AlphaGo** program **beats unhandicapped professional human player**



# Neuron Models – 1. Biological Neuron

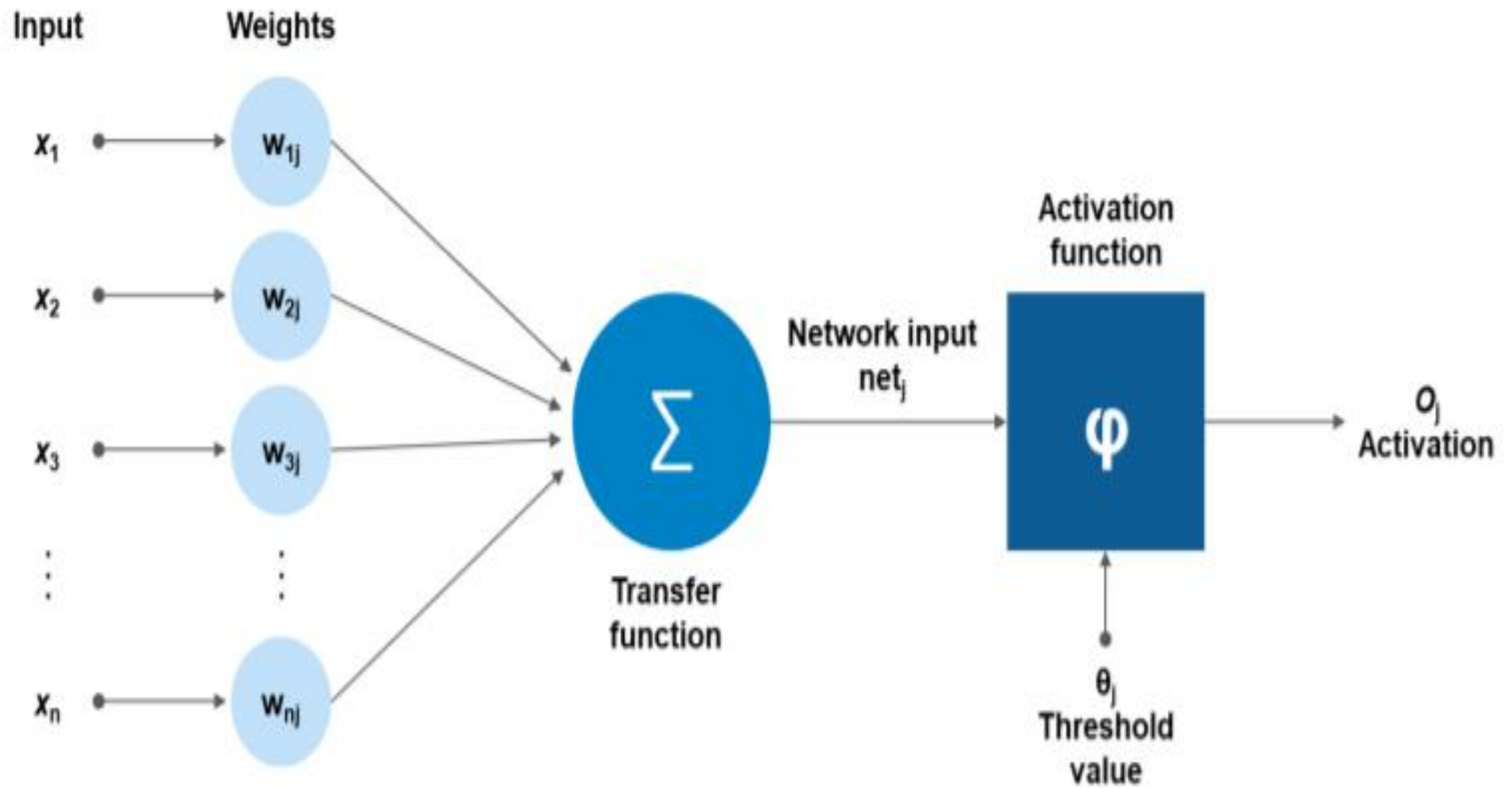
- To the extent a human brain is understood today, it seems to operate as follows: bundles of neurons, or nerve fibers, form nerve structures.
- There are many different types of neurons in the nerve structure, each having a particular shape, size and length depending upon its function and utility in the nervous system.
- While each type of neuron has its own unique features needed for specific purposes, all neurons have two important structural components in common.
- At one end of the neuron are called dendrites, which come together to form larger branches and trunks where they attach to soma, the body of the nerve cell.



- At the other end of the neuron is a single filament leading out of the soma, called an axon, which has extensive branching on its far end.
- These two structures have special electrophysiological properties which are basic to the function of neurons as information processors, as we shall see next.
- Neurons are connected to each other via their axons and dendrites. Signals are sent through the axon of one neuron to the dendrites of other neurons
- Hence, dendrites may be represented as the inputs to the neuron, and the axon as its output. Note that each neuron has many inputs through its multiple dendrites, whereas it has only one output through its single axon.

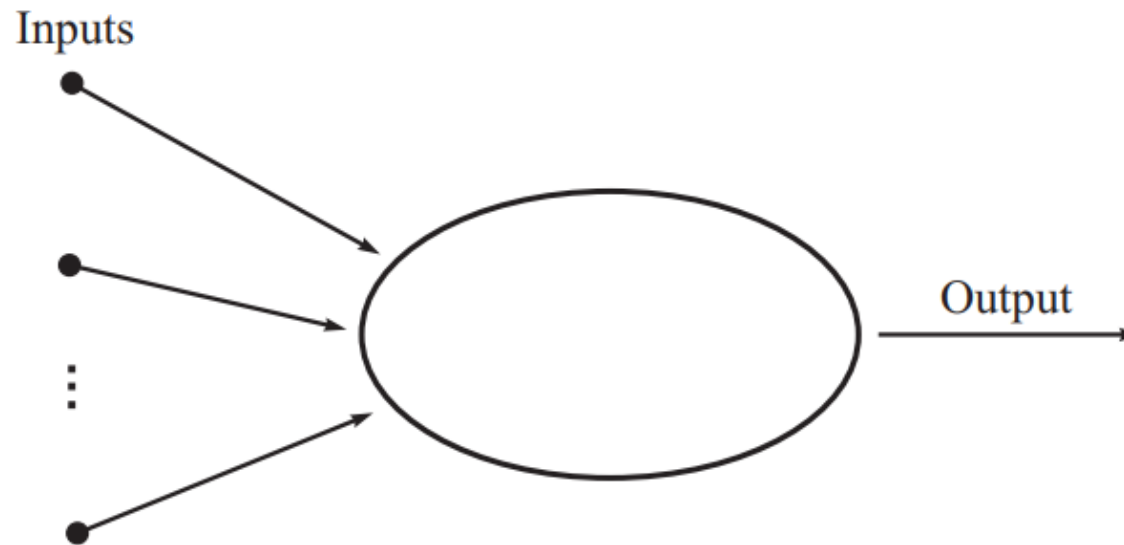
# Neuron Models – 2. Artificial Neuron

- An artificial neuron is a digital construct that seeks to simulate the behavior of a biological neuron in the brain.
- an artificial neuron is composed of a set of weighted inputs, along with a transformation function and an activation function.
- The activation function at the end would correspond to the axon of a biological neuron.
- The weighted inputs would correspond to the inputs of a biological neuron that take electrical impulses moving through the brain and work on them to transmit them to subsequent layers of neurons.
- Artificial neurons, as parts of artificial neural networks, are driving deep learning and machine learning capabilities. They are helping computers to “think more like humans” and produce more sophisticated cognitive results.



## 1. Input and outputs:

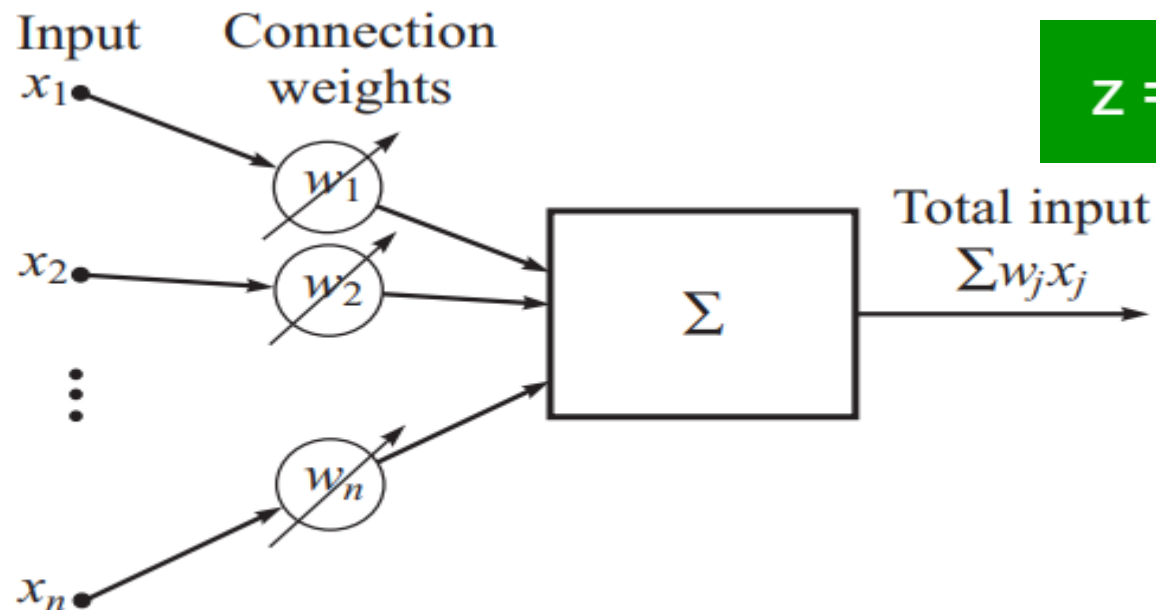
Just as there are many inputs (stimulation levels) to a biological neuron, there should be many input signals to our artificial neuron (AN). All of them should come to our AN simultaneously. In response, a biological neuron either ‘fires’ or ‘doesn’t fire’ depending upon some threshold level. Our AN will be allowed a single output signal, just as is present in a biological neuron: many inputs, one output.



**Figure 5.2** Many inputs, one output model of a neuron

## 2. Weighting factors:

Each input will be given a relative weighting, which will affect the impact of that input. Weights are adaptive coefficients within the network, that determine the intensity of the input signal. In fact, this adaptability of connection strength is precisely what provides neural networks their ability to learn and store information, and, consequently, is an essential element of all neuron models.

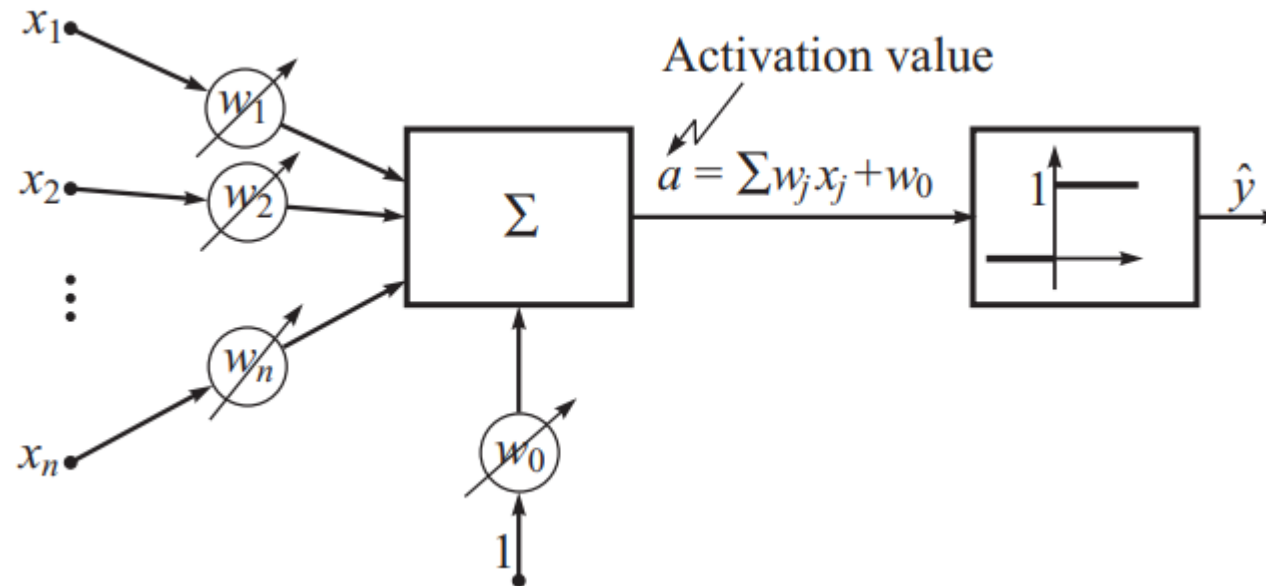


$$z = (w_1.x_1 + w_2.x_2 + \dots + w_n.x_n) + b$$

**Figure 5.3** A neuron with weighted inputs

**3. Activation function:** Artificial neurons use an activation function, often called a transfer function, to compute their activation as a function of total input stimulus.

- Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it.
- In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output. This process is known as *back-propagation*.
- Activation functions make the back-propagation possible since the *gradients* are supplied along with the error to update the weights and biases.





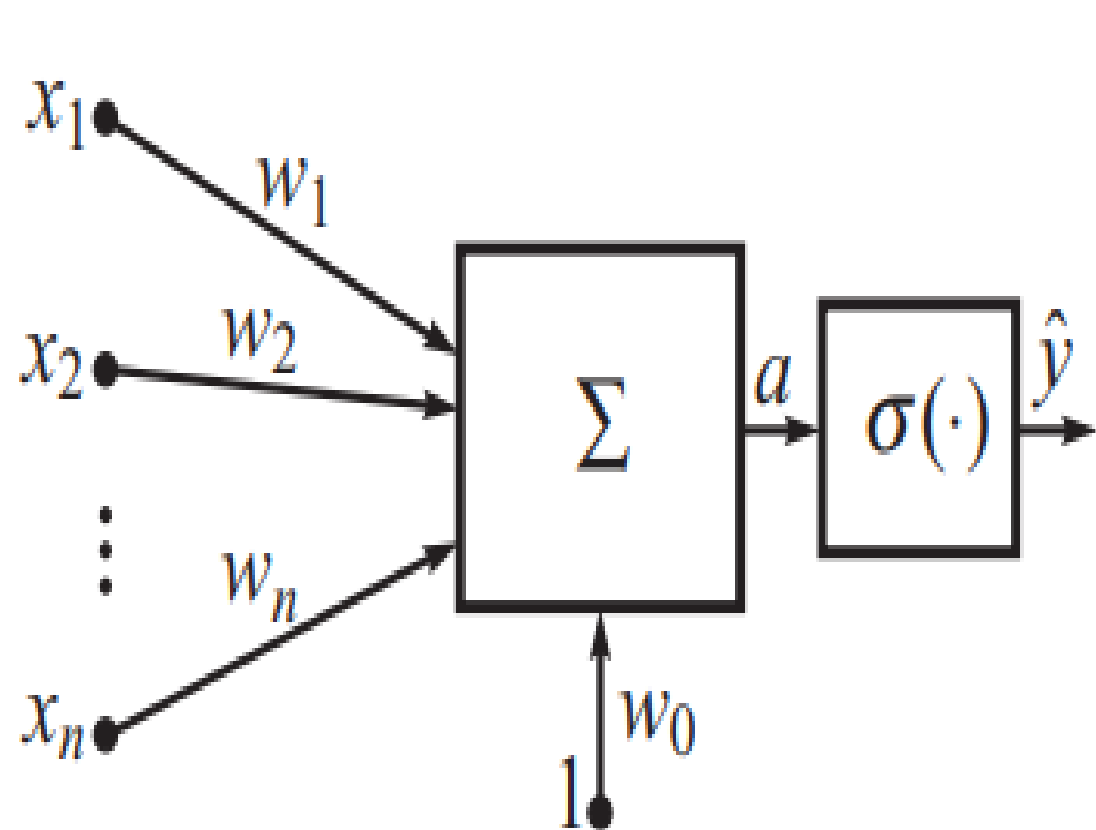
# Neuron Models – 3. Mathematical model

- The artificial neuron is really nothing more than a simple mathematical equation for calculating an output value from a set of input values.
- A neuron model (a processing element/a unit/a node/a cell of our neural network), will be represented as follows:

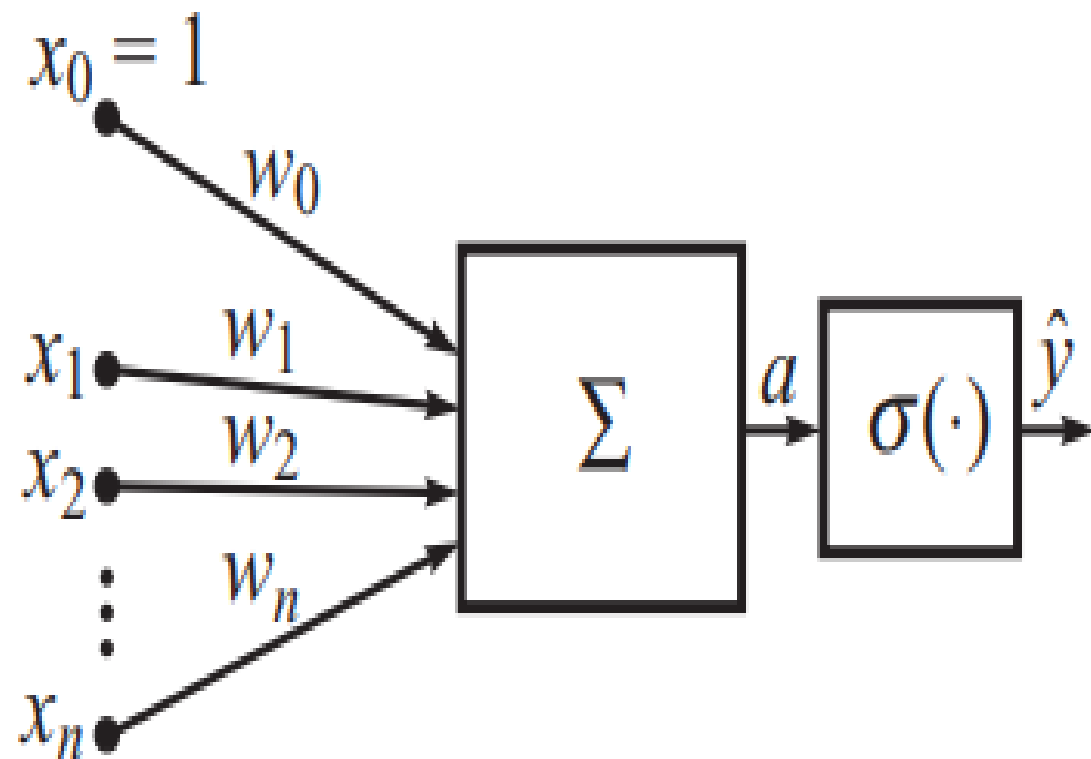
The input vector  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_n]^T$ ;

- the connection weight vector :  $\mathbf{w}^T = [w_1 \ w_2 \ \dots \ w_n]$ ;
- the unity-input weight  $w_0$  (bias term), and the output  $\hat{y}$  of the neuron are related by the following equation:

$$\hat{y} = \sigma(\mathbf{w}^T \mathbf{x} + w_0) = \sigma\left(\sum_{j=1}^n w_j x_j + w_0\right)$$



(a)



(b)

**Figure 5.6** Mathematical model of a neuron (perceptron)

# 1. Binary step function

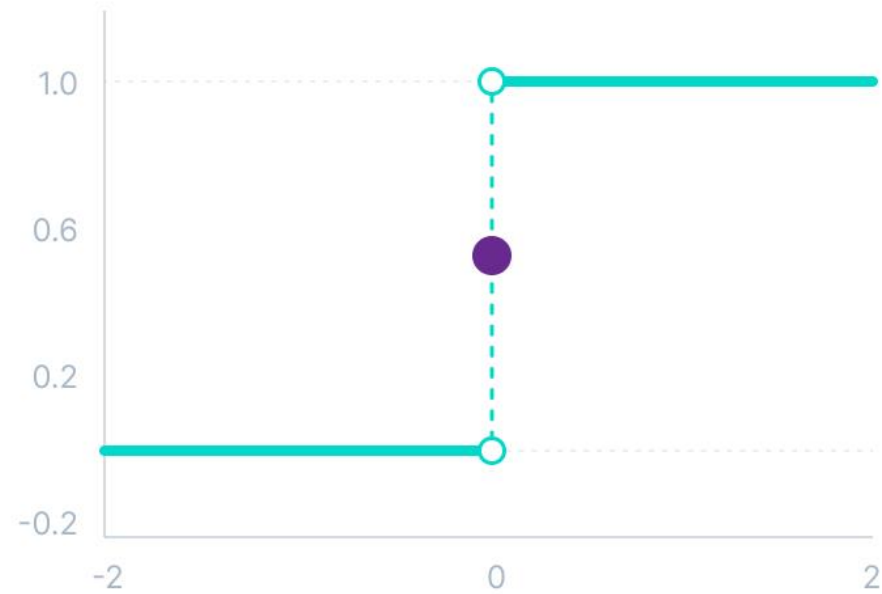
- Binary step function depends on a threshold value that decides whether a neuron should be activated or not.
- The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer.
- It is very simple and useful to classify binary problems or classifier.

Mathematically represented with:

*Binary step*

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

**Binary Step Function**

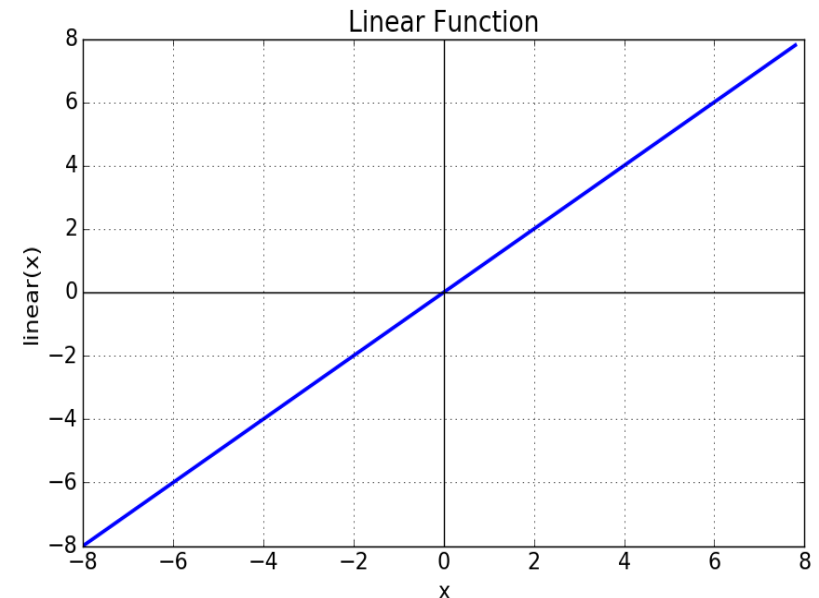


## 2. Linear function

- The linear activation function, also known as "no activation," or "identity function" (multiplied x1.0), is where the activation is proportional to the input.
- The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.
- Linear function has the equation similar to as of a straight line i.e.

$$y = ax.$$

The range is  $[-\infty \text{ to } +\infty]$



# Limitations of linear function

A linear activation function has two major problems :

- It's not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input  $x$ .
- All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

# 3. Non-Linear Activation Functions

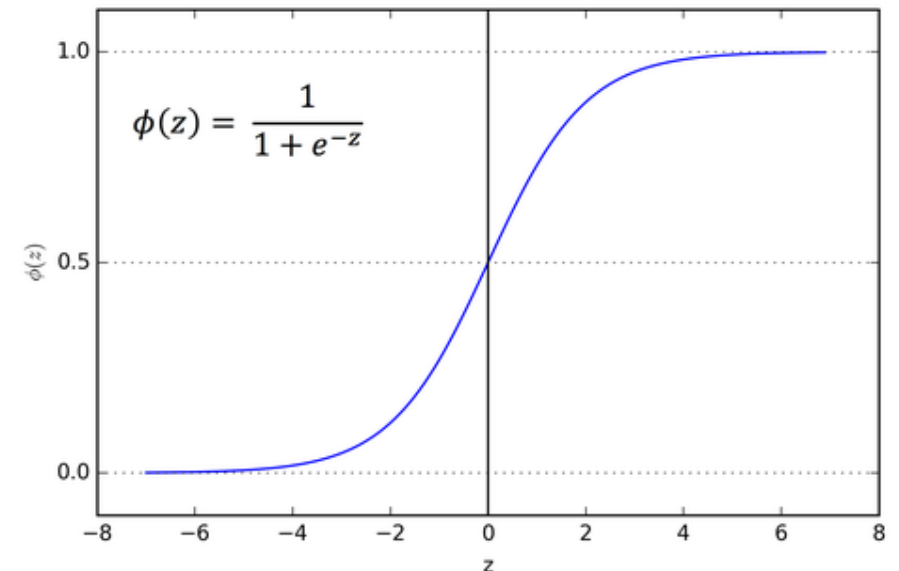
- The linear activation function is simply a linear regression model because of its limited power, this does not allow the model to create complex mappings between the network's inputs and outputs.
- Non-linear activation functions solve the following limitation of linear activation functions:
  - They allow backpropagation because now the derivative function would be related to the input, and it's possible to go back and understand which weights in the input neurons can provide a better prediction

## 3.1 Sigmoid / Logistic Activation Function

- This function takes any real value as input and outputs values in the range of 0 to 1.
- The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0.
- Mathematically it can be represented as:

*Sigmoid / Logistic*

$$f(x) = \frac{1}{1 + e^{-x}}$$





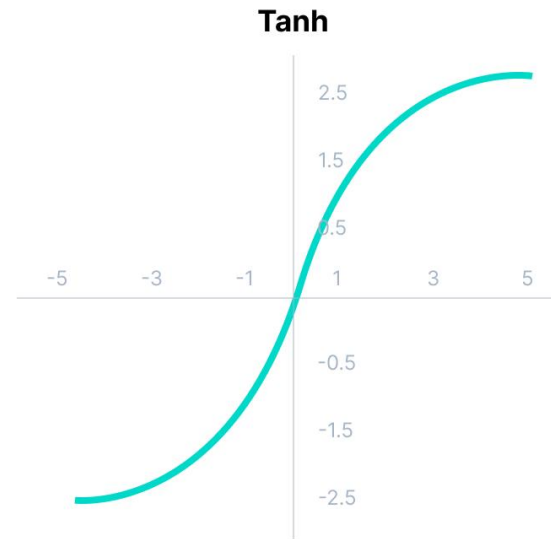
Here's why sigmoid/logistic activation function is one of the most widely used functions:

- It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
- The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

## 3.2 Tanh Function (Hyperbolic Tangent)

- Tanh function is very similar to the sigmoid/logistic activation function, and even has the same S-shape with the difference in output range of -1 to 1.
- In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.
- Mathematically it can be represented as:

$$\text{Tanh}$$
$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$



## Advantages of using this activation function are:

- The output of the tanh activation function is Zero centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.
- Usually used in hidden layers of a neural network as its values lie between -1 to; therefore, the mean for the hidden layer comes out to be 0 or very close to it. It helps in centering the data and makes learning for the next layer much easier.

## 3.3 ReLU activation function

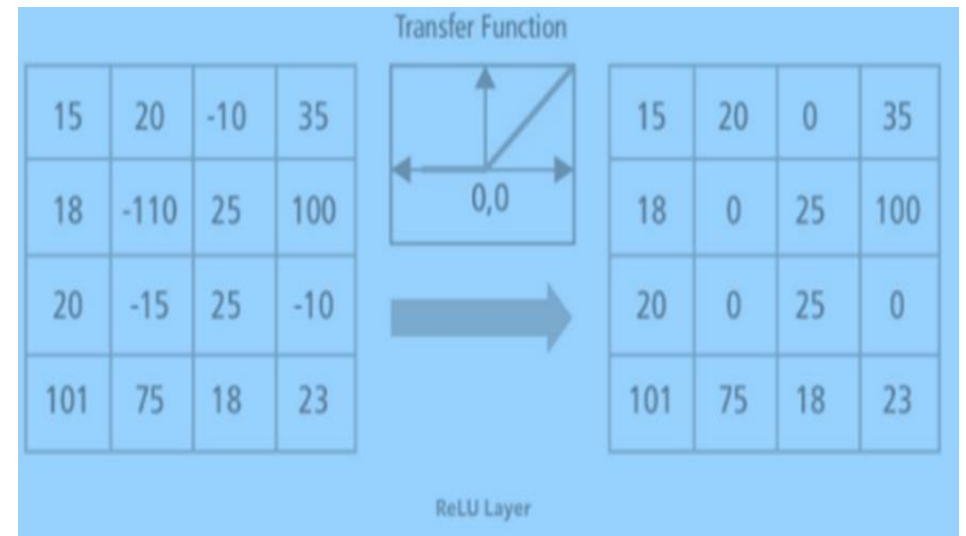
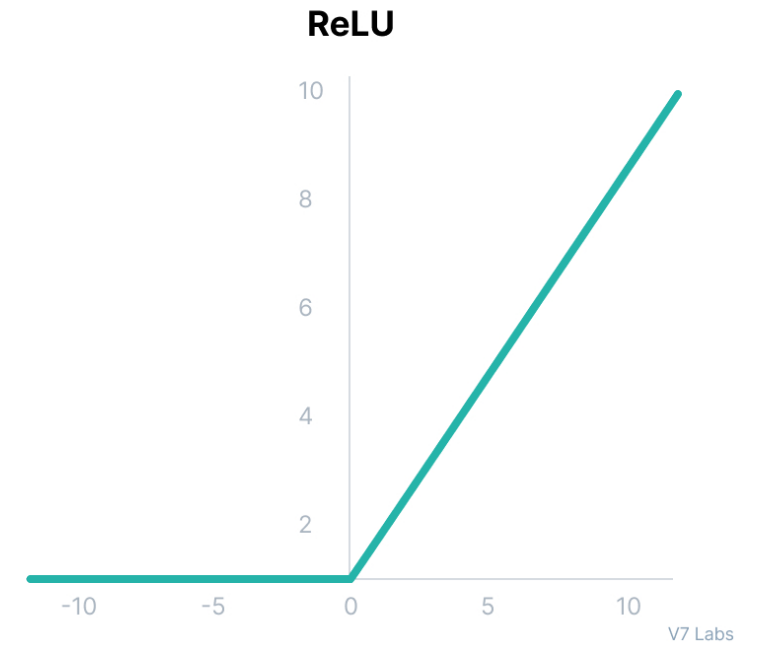
- A **Rectified Linear Unit (ReLU)** is a non-linear activation function that performs on multi layer neural network (ex:  $f(x)=\max(0,x)$  where  $x$  is input). In this layer we remove every negative value from the filtered image and replace it with zero.
- This function only activates when the node input is above a certain quantity. So, when the input is below zero the output is zero
- The purpose of applying the rectifier function is to increase the non-linearity in our images. This function causes dying ReLU problem and the solution is Leaky ReLU
- The main catch here is that the ReLU function does not activate all the neurons at the same time.
- The neurons will only be deactivated if the output of the linear transformation is less than 0.

- Mathematically it can be represented as:

*ReLU*

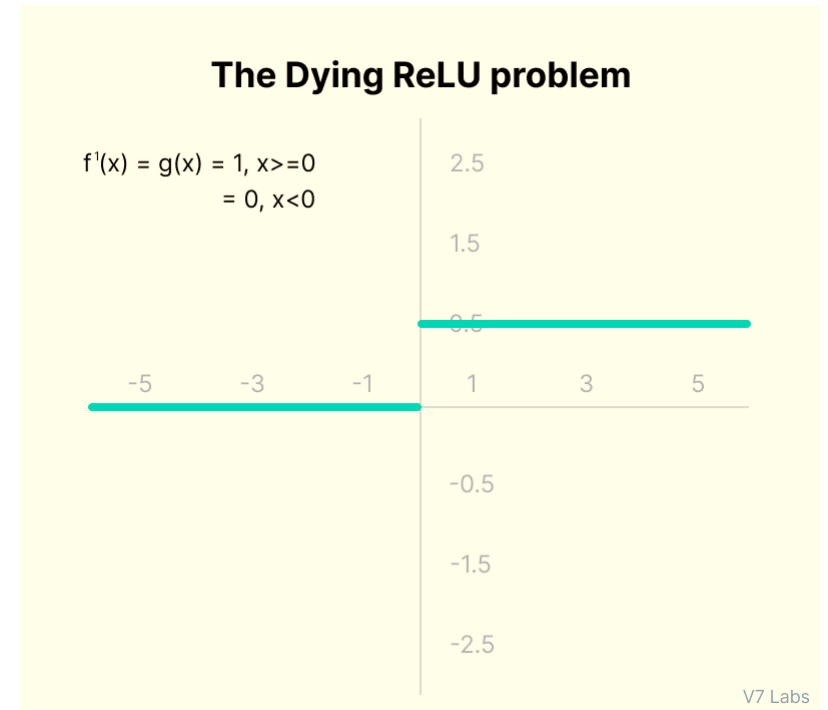
$$f(x) = \max(0, x)$$

x	f(x)=x	F(X)
-110	f(-110)=0	0
-15	f(-15)=0	0
15	f(15)=15	15
25	f(25)=25	25



# Dying ReLU problem

- The negative side of the graph makes the gradient value zero.
- Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated.
- This can create dead neurons which never get activated.
- All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.

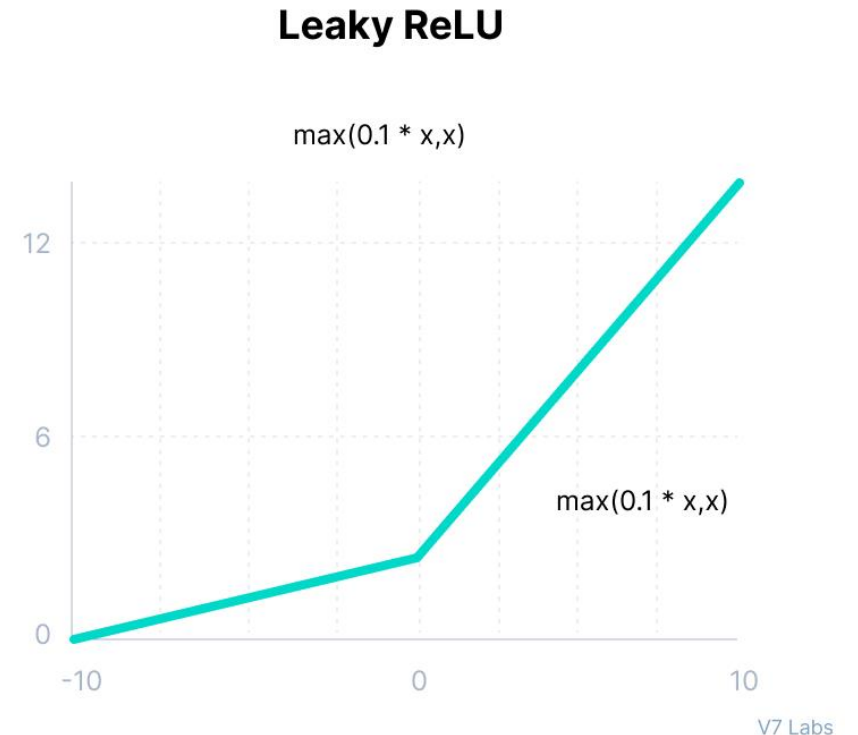


## 3.4 Leaky ReLU Function

- Leaky ReLU is an improved version of ReLU function to solve the Dying ReLU problem as it has a small positive slope in the negative area.
- Mathematically it can be represented as:

*Leaky ReLU*

$$f(x) = \max(0.1x, x)$$



- The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values.
- By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value. Therefore, we would no longer encounter dead neurons in that region.

### **The limitations that this function faces include:**

- The predictions may not be consistent for negative input values.
- The gradient for negative values is a small value that makes the learning of model parameters time-consuming.



## 3.5 SoftMax function

- The SoftMax function is described as a combination of multiple Sigmoids.
- It calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class.
- It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.
- Mathematically it can be represented as:

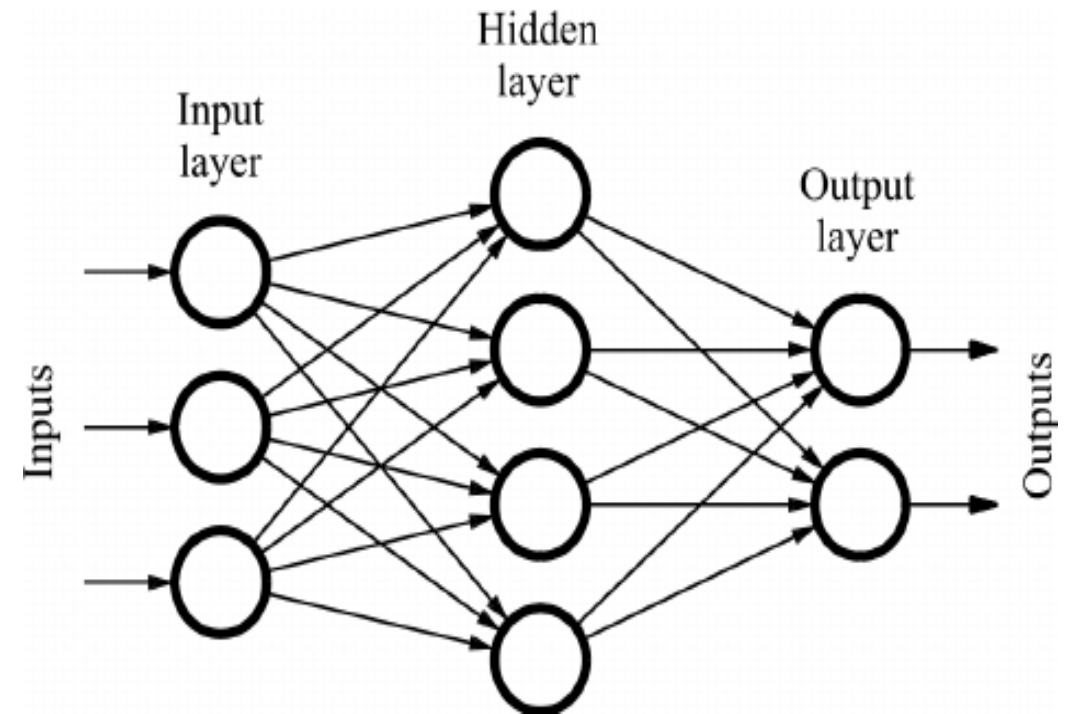
$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

# 4. Network Architectures

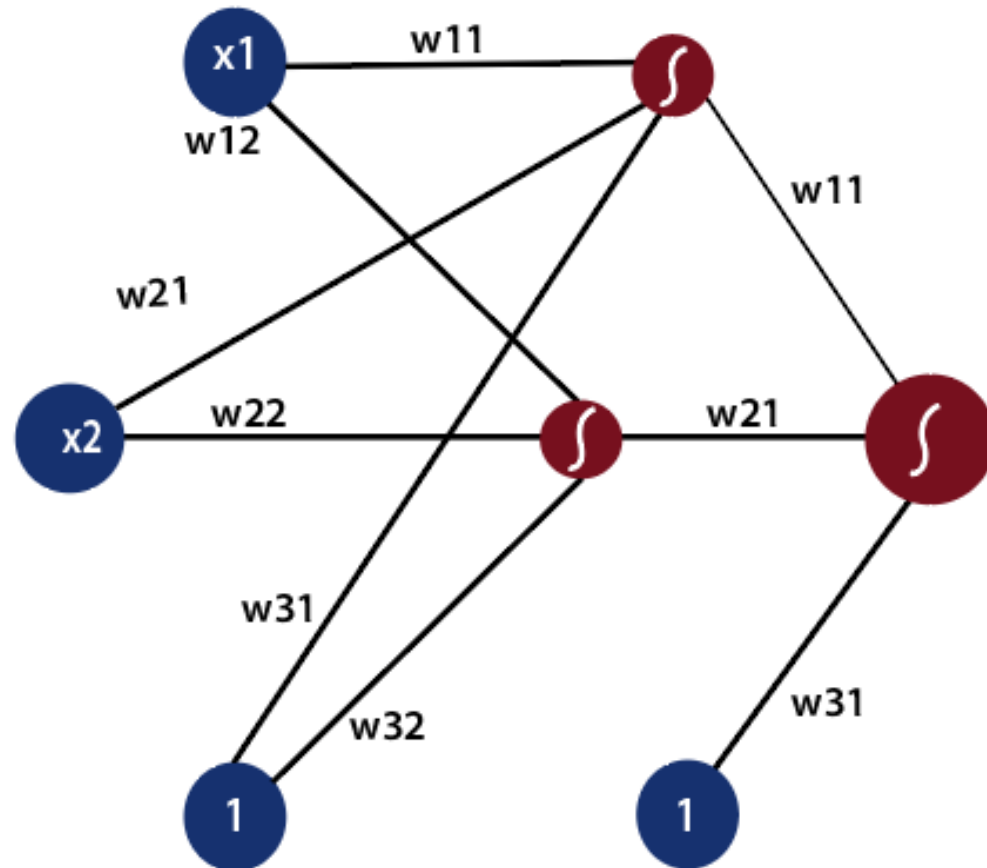
- The artificial neural network is built by neuron models.
- Many different types of artificial neural networks have been proposed, just as there are many theories on how biological neural processing works.
- We may classify the organization of the neural networks largely into two types: a feed-forward net and a recurrent net.
- The feed-forward net has a hierarchical structure that consists of several layers, without interconnection between neurons in each layer, and signals flow from input to output layer in one direction.
- In the recurrent net, multiple neurons in a layer are interconnected to organize the network.

## 4.1 Network Architecture – Feed Forward Neural Network

- Feed forward neural networks were among the first and most successful learning algorithms.
- They are also called deep networks, multi-layer perceptron (MLP), or simply neural networks.
- "The process of receiving an input to produce some kind of output to make some kind of prediction is known as Feed Forward."
- Feed Forward neural network is the core of many other important neural networks such as convolution neural network.



- In the feed-forward neural network, there are not any feedback loops or connections in the network. Here is simply an input layer, a hidden layer, and an output layer.



To gain a solid understanding of the feed-forward process, let's see this mathematically.

1) The first input is fed to the network, which is represented as matrix  $x_1$ ,  $x_2$ , and one where one is the bias value  $[x_1 \ x_2 \ 1]$

2) Each input is multiplied by weight with respect to the first and second model to obtain their probability of being in the positive region in each model.

- So, we will multiply our inputs by a matrix of weight using matrix multiplication.

$$[x_1 \ x_2 \ 1] = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = [\text{score} \ \text{score}]$$

- 3) After that, we will take the sigmoid of our scores and gives us the probability of the point being in the positive region in both models.

$$\frac{1}{1 + e^{-x}} [\text{score} \ \text{score}] = \text{probability}$$

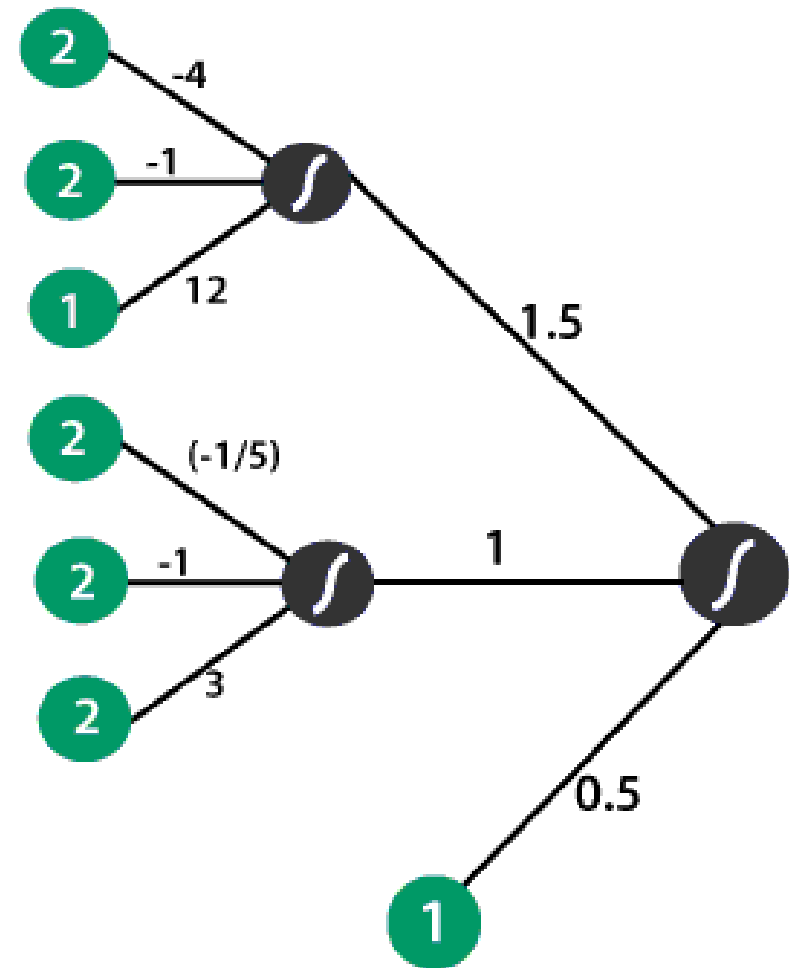
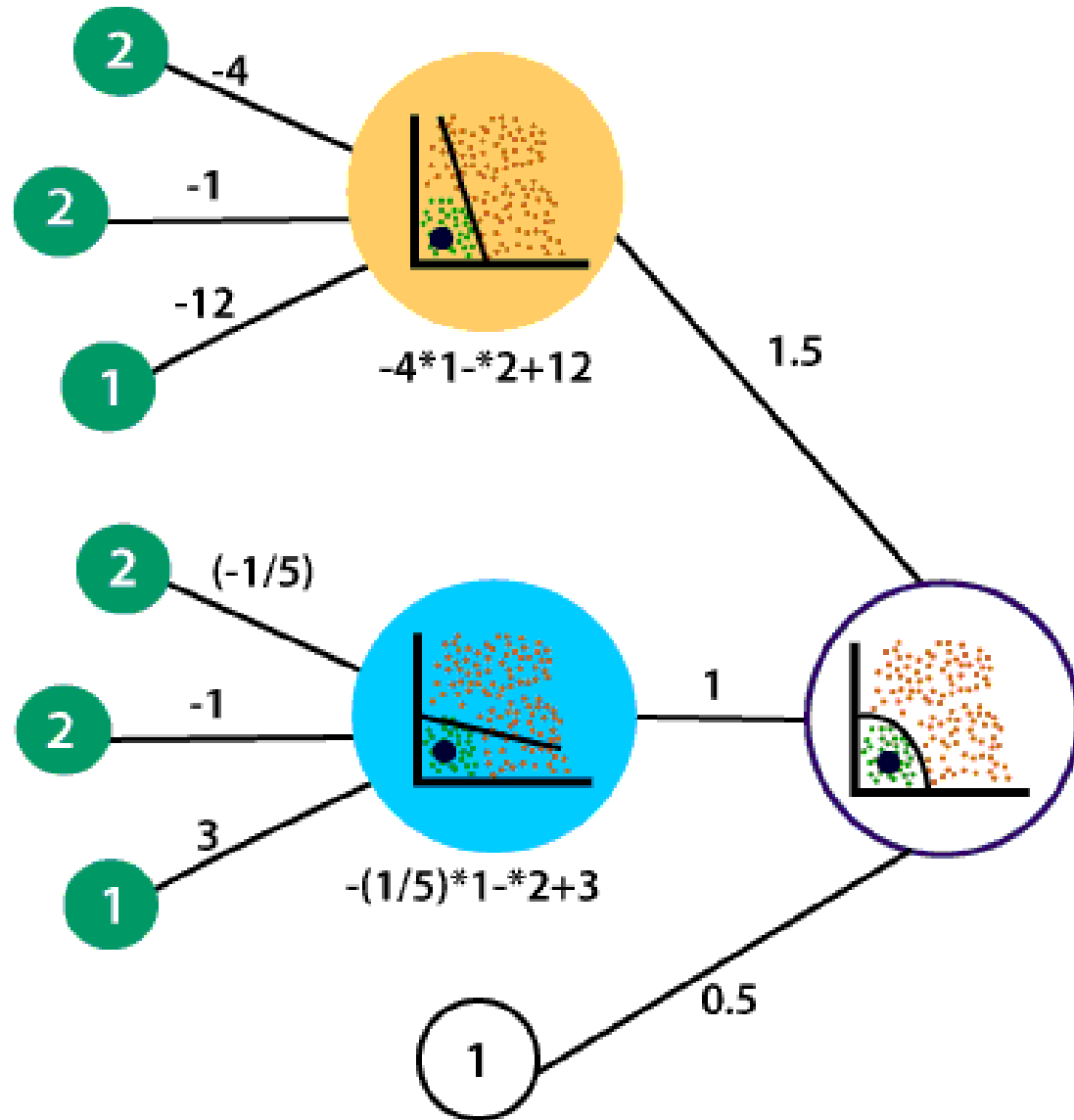
4) We multiply the probability which we have obtained from the previous step with the second set of weights. We always include a bias of one whenever taking a combination of inputs.

$$[\text{probability} \quad \text{probability} \quad 1] \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} = [\text{score}]$$

And as we know to obtain the probability of the point being in the positive region of this model, we take the sigmoid and thus producing our final output in a feed-forward process.

$$\frac{1}{1 + e^{-x}} [\text{score}] = [\text{probability}]$$

# Example on Single Layer Network



So, what we will do we use our non-linear model to produce an output that describes the probability of the point being in the positive region. The point was represented by 2 and 2. Along with bias, we will represent the input as

$$[2 \quad 2 \quad 1]$$

The first linear model in the hidden layer recall and the equation defined it

$$-4x_1 - x_2 + 12$$

Which means in the first layer to obtain the linear combination the inputs are multiplied by -4, -1 and the bias value is multiplied by twelve.

$$[2 \quad 2 \quad 1] \times \begin{bmatrix} -4 & w_{12} \\ -1 & w_{22} \\ 12 & w_{32} \end{bmatrix}$$

The weight of the inputs are multiplied by -1/5, 1, and the bias is multiplied by three to obtain the linear combination of that same point in our second model.

$$[2 \quad 2 \quad 1] \times \begin{bmatrix} -4 & -1/5 \\ -1 & -1 \\ 12 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 2(-4) + 2(-1) + 1(12) \\ 2(-4) + 2(-1) + 1(12) \end{bmatrix}$$

$$[2 \quad 0.6]$$



Now, to obtain the probability of the point is in the positive region relative to both models we apply sigmoid to both points as

$$\left[ \frac{1}{1+\frac{1}{e^x}} \quad \frac{1}{1+\frac{1}{e^x}} \right] = \left[ \frac{1}{1+\frac{1}{e^2}} \quad \frac{1}{1+\frac{1}{e^{0.6}}} \right] = [0.88 \quad 0.64]$$

The second layer contains the weights which dictated the combination of the linear models in the first layer to obtain the non-linear model in the second layer. The weights are 1.5, 1, and a bias value of 0.5. Now, we have to multiply our probabilities from the first layer with the second set of weights as

$$[0.88 \quad 0.64 \quad 1] \times \begin{bmatrix} 1.5 \\ 1 \\ 0.5 \end{bmatrix} = [0.88(1.5) + (0.64)(1) + 1(0.5)] = 2.46$$

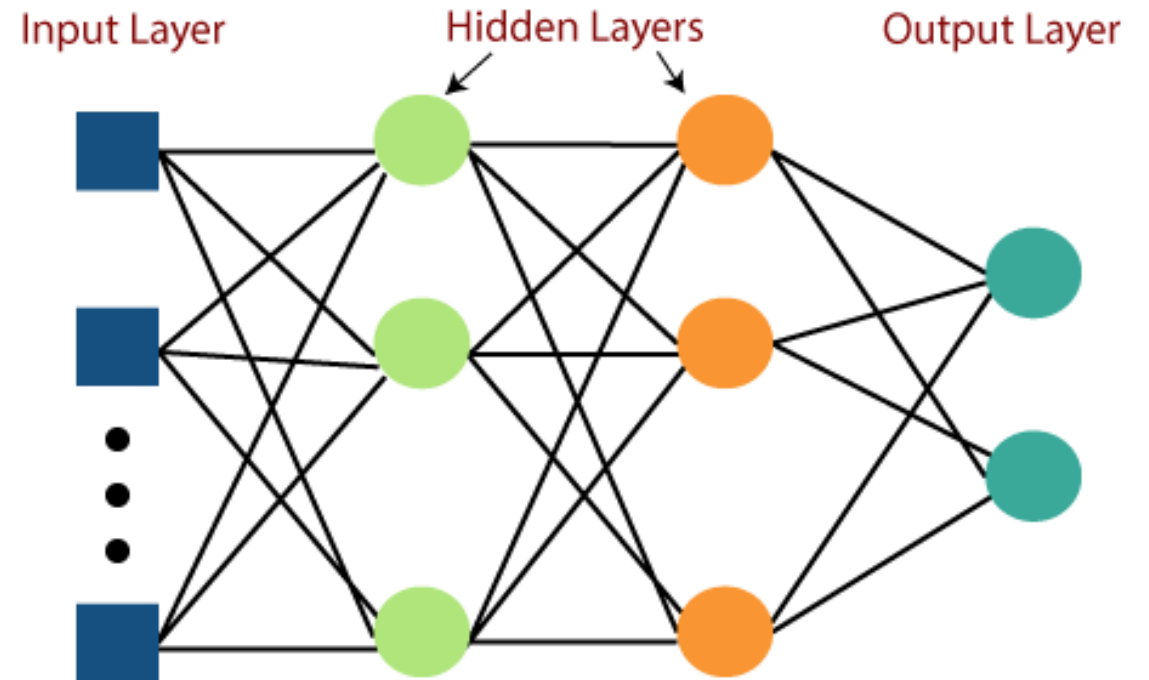
Now, we will take the sigmoid of our final score

$$\frac{1}{1+\frac{1}{e^{2.46}}} = [0.92]$$

It is complete math behind the feed forward process where the inputs from the input traverse the entire depth of the neural network. In this example, there is only one hidden layer. Whether there is one hidden layer or twenty, the computational processes are the same for all hidden layers.

# Multi layer perceptron's:

- A multilayer perceptron (MLP) is a feed forward artificial neural network that generates a set of outputs from a set of inputs.
- An MLP is characterized by several layers of input nodes connected as a directed graph between the input and output layers.
- MLP is a deep learning method



# 5. PERCEPTRONS

- Classical NN systems are based on units called PERCEPTRON and ADALINE (ADaptive Linear Element).
- Perceptron was developed in 1958 by Frank Rosenblatt, a researcher in neurophysiology, to perform a kind of pattern recognition tasks. In mathematical terms, it resulted from the solution of classification problem.
- ADALINE was developed by Bernard Widrow and Marcian Hoff; it originated from the field of signal processing, or more specifically from the adaptive noise cancellation problem.
- It resulted from the solution of the regression problem; the regressor having the properties of noise canceller (linear filter).

- The perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs; then outputs  $+1$  if the result is greater than the threshold and  $-1$  otherwise.
- The ADALINE in its early stage consisted of a neuron with a linear activation function, a hard limiter (a thresholding device with a signum activation function) and the Least Mean Square (LMS) learning algorithm.
- We focus on the two most important parts of ADALINE—its linear activation function and the LMS learning rule.
- The roots of both the perceptron and the ADALINE were in the linear domain.

## 5.1 Limitations of Perceptron Algorithm for Linear Classification Tasks

- The roots of the perceptron were in the linear domain. It was developed as the classifier providing the *linear separability of class patterns*.
- It was observed that there is a major problem associated with this algorithm for real-world solutions: datasets are almost certainly not linearly separable, while the algorithm finds a separating hyper plane only for linearly separable data.
- When the dataset is linearly inseparable, the test of the decision surface will always fail for some subset of training points regardless of the adjustments we make to the free parameters, and the algorithm will loop forever.

- The limitations of perceptrons can be overcome by neural networks.
- The perceptron criterion function is based on misclassification error (number of samples misclassified) and the gradient procedures for minimization are not applicable.
- The neural networks primarily solve the regression problems, are based on minimum squared-error criterion and employ gradient procedures for minimization.
- The algorithms for separable—as well as inseparable—data classification are first developed in the context of regression problems and then adapted for classification problems

## 5.2 Linear Classification using Regression Techniques

- The regression techniques can be used for linear classification with a careful choice of the target values associated with classes.
- Let the set of training (data) examples  $D$  be:

$$\begin{aligned} \mathcal{D} &= \{x_j^{(i)}, y^{(i)}\}; i = 1, \dots, N; j = 1, \dots, n \\ &= \{\mathbf{x}^{(i)}, y^{(i)}\} \end{aligned}$$

where  $\mathbf{x}^{(i)} = [x_1^{(i)} \ x_2^{(i)} \ \dots \ x_n^{(i)}]^T$  is an  $n$ -dimensional *input vector* (pattern with  $n$ -features) for the  $i$ th example in a real-valued space;  $y^{(i)}$  is its *class label* (output value), and  $y^{(i)} \in [+1, -1]$ ,  $+1$  denotes Class 1 and  $-1$  denotes Class 2. To build a linear classifier, we need a linear function of the form

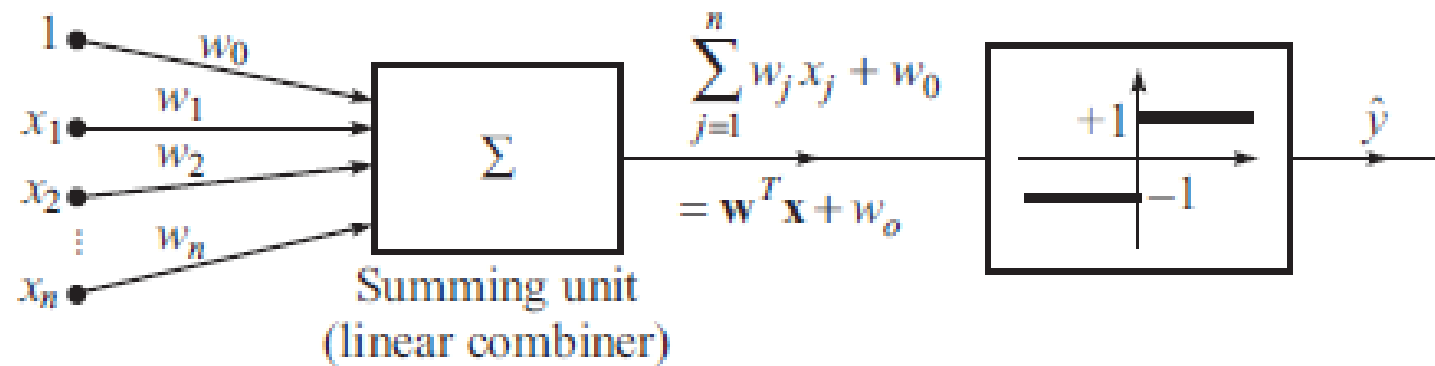
$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 \tag{5.14}$$

so that the input vector  $\mathbf{x}^{(i)}$  is assigned to Class 1 if  $g(\mathbf{x}^{(i)}) > 0$ , and to Class 2 if  $g(\mathbf{x}^{(i)}) < 0$ , i.e.,

$$\hat{y}^{(i)} = \begin{cases} +1 & \text{if } \mathbf{w}^T \mathbf{x}^{(i)} + w_0 > 0 \\ -1 & \text{if } \mathbf{w}^T \mathbf{x}^{(i)} + w_0 < 0 \end{cases} \quad (5.15)$$

$\mathbf{w} = [w_1 \ w_2 \ \dots w_n]^T$  is the *weight vector* and  $w_0$  is the *bias*. In terms of regression, we can view this classification problem as follows.

Given a vector  $\mathbf{x}^{(i)}$ , the output of the *summing unit* (linear combiner) will be  $\mathbf{w}^T \mathbf{x}^{(i)} + w_0$  (decision hyperplane) and thresholding the output through a *sgn* function gives us perceptron output  $\hat{y}^{(i)} = \pm 1$  (refer to Fig. 5.11).



**Figure 5.11** Linear classification using regression technique



The sum of error squares for the classifier becomes (Eqns (3.71))

$$E = \sum_{i=1}^N (e^{(i)})^2 = \sum_{i=1}^N (y^{(i)} - \hat{y}^{(i)})^2 \quad (5.16)$$

We require  $E$  to be a function of  $(\mathbf{w}, w_0)$  to design the linear function  $g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$  that minimizes  $E$ . To obtain  $E(\mathbf{w}, w_0)$ , we replace the perceptron outputs  $\hat{y}^{(i)}$  by the linear combiner outputs  $\mathbf{w}^T \mathbf{x}^{(i)} + w_0$ ; this gives us the error function

$$E(\mathbf{w}, w_0) = \sum_{i=1}^N (y^{(i)} - (\mathbf{w}^T \mathbf{x}^{(i)} + w_0))^2 \quad (5.17)$$

Consider pattern  $i$ . If  $\mathbf{x}^{(i)} \in$  Class 1, the desired output  $y^{(i)} = +1$  with summing unit output  $\mathbf{w}^T \mathbf{x}^{(i)} + w_0 > 0$  ( $\hat{y}^{(i)} = +1$ , refer to Fig. 5.11), the contribution of correctly classified pattern  $i$  to  $E(\mathbf{w}, w_0)$  is small when compared with wrongly classified pattern ( $\mathbf{w}^T \mathbf{x}^{(i)} + w_0 < 0$ ;  $\hat{y}^{(i)} = -1$ ).

## 5.3 Standard Gradient Descent Optimization Scheme: Steepest Descent

What is Gradient?

- The derivative of error function with respect to weights is called the gradient of that layer.
- At the output layer, the network must calculate the total error for all data points and take its derivative with respect to weights at that layer.
- The weights for that layer are then updated based on the gradient. This update can be the gradient itself or a factor of it. This factor is known as the learning rate.
- The gradient descent training rule for a single neuron is important because it provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units.

- Gradient Descent is an optimization algorithm used for minimizing the cost function in various machine learning algorithms. It is basically used for updating the parameters of the learning model.

## **Types of gradient Descent:**

### **1. Batch Gradient Descent:**

- This is a type of gradient descent which processes all the training examples for each iteration of gradient descent.
- But if the number of training examples is large, then batch gradient descent is computationally very expensive.
- Hence if the number of training examples is large, then batch gradient descent is not preferred.
- Instead, we prefer to use stochastic gradient descent or mini-batch gradient descent.

## 2. Stochastic Gradient Descent:

- This is a type of gradient descent which processes 1 training example per iteration.
- Hence, the parameters are being updated even after one iteration in which only a single example has been processed. Hence this is quite faster than batch gradient descent.
- But again, when the number of training examples is large, even then it processes only one example which can be additional overhead for the system as the number of iterations will be quite large.

## 3. Mini Batch gradient descent:

- This is a type of gradient descent which works faster than both batch gradient descent and stochastic gradient descent. Here  $b$  examples where  $b < m$  are processed per iteration.
- So even if the number of training examples is large, it is processed in batches of  $b$  training examples in one go. Thus, it works for larger training examples and that too with lesser number of iterations.

# Challenges with Gradient descent

## Vanishing and Exploding Gradient:

- ***Vanishing Gradients:*** Vanishing Gradient occurs when the gradient is smaller than expected. During back-propagation, this gradient becomes smaller that causing the decrease in the learning rate of earlier layers than the later layer of the network. Once this happens, the weight parameters update until they become insignificant.
- ***Exploding Gradient:*** Exploding gradient is just opposite to the vanishing gradient as it occurs when the Gradient is too large and creates a stable model. Further, in this scenario, model weight increases, and they will be represented as NaN. This problem can be solved using the dimensionality reduction technique, which helps to minimize complexity within the model.

## 6. Linear Neuron and the Widrow-Hoff Learning Rule

- The WIDROW-HOFF Learning rule is very similar to the perception Learning rule. However the origins are different.
- The units with linear activation functions are called linear units. A network with a single linear unit is called as adaline (adaptive linear neuron). That is in an ADALINE, the input-output relationship is linear.
- Adaline uses bipolar activation for its input signals and its target output. The weights between the input and the output are adjustable. Adaline is a net which has only one output unit.
- The adaline network may be trained using the delta learning rule. The delta learning rule may also be called as **least mean square (LMS)** rule or **Widrow-Hoff rule**. This learning rule is found to minimize the mean-squared error between the activation and the target value.

# 7. The Error-correction Delta Rule

- Delta Learning Rule:
  - The updation of weight matrix or updation of weights and bias in artificial neural network is known as learning. This is also known as training.
  - The purpose of the learning rule is to train the network to perform some task to reduce error.
  - Delta learning rule is a gradient descent learning rule for updating the weights of the inputs. It uses an error function to perform gradient descent learning.
  - It is introduced by Widrow and Hoff. This rule is applicable only for continuous activation function.
  - Partial derivatives of error with respect to weights is used for training Neural network.
  - The aim of applying delta rule is to minimize the mean squared error between activation (net input) and target value.

# Learning Perceptron by using "Delta Learning Rule"

Input			Target	Net i/p	$t - Y_{in}$	change in weight and bias			New weights			Error $(t - Y_{in})^2$
$(x_1)$	$(x_2)$	$(b)$	$(t)$	$Y_{in}$		$\Delta w_1$	$\Delta w_2$	$\Delta b$	$w_1$	$w_2$	$b$	
Epoch 1												
1	1	1	1	0.3	0.7	0.07	0.07	0.07	0.17	0.17	0.17	0.49
1	-1	1	1	0.17	0.83	0.083	-0.083	0.083	0.253	0.087	0.253	0.69
-1	1	1	1	0.087	0.913	-0.0913	0.0913	0.0913	0.1617	0.1783	0.3443	0.83
-1	-1	1	-1	0.0043	-1.0043	0.10043	0.10043	-0.10043	0.262	0.278	0.243	1.01

## Epoch 2

1	1	1	1	0.7847	0.2153	0.0215	0.0215	0.0215	0.2837	0.3003	0.2654	0.046
1	-1	1	1	0.2488	0.7512	0.07512	-0.0751	0.0751	0.3588	0.251	0.3405	0.564
-1	1	1	1	0.2069	0.7931	-0.793	0.0793	0.0793	0.2795	0.3044	0.4198	0.629
-1	-1	1	-1	-0.1641	-0.8359	0.0836	0.0836	-0.083	0.3631	0.388	0.366	0.699

$$= 0.4845$$

$$MSE = 0.75465$$



Example:-

①

Implement Logic OR operation using Delta Learning Rule, Use bipolar input and Targets. Consider learning rate ( $\alpha = 0.1$ ), Perform 2 epoch full network training.

Sol:-

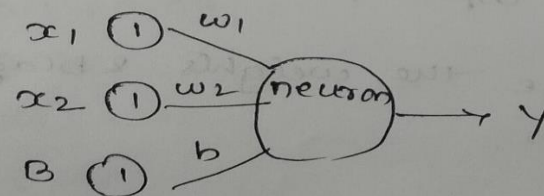
Logic OR Truth Table

Input		Output
$x_1$	$x_2$	$y$
1	1	1
1	0	1
0	1	1
0	0	0

Bipolar i/p & O/p representation

$= -1 \text{ \& } 1$

Input			Target O/p
$x_1$	$x_2$	$B$	$t$
1	1	1	1
1	-1(0)	1	1
-1(0)	1	1	1
-1(0)	-1(0)	1	-1



Step 1:-

Initialize weights and bias

Let us take  $w_1 = w_2 = b = 0.1$

Learning rate ( $\alpha$ ) = 0.1

Step a: -

From truth table take first pattern.

$$x_1 = 1 \quad x_2 = 1 \quad B = 1, t = 1$$

$$\begin{aligned} \text{Net input } (Y_{in}) &= w_1 x_1 + w_2 x_2 + b \\ &= (0.1)(1) + (0.1)(1) + 0.1 \end{aligned}$$

$$Y_{in} = 0.3$$

$$t - Y_{in} = 1 - 0.3 = 0.7$$

$$\begin{aligned} \text{Error } (t - Y_{in})^2 &= (0.7)^2 \\ &= 0.49 \end{aligned}$$

→ Now we need to reduce the error rate after taking average of all input patterns in Epoch 1

→ we have change the weights & bias in order to get new weights

New weight formula :-

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

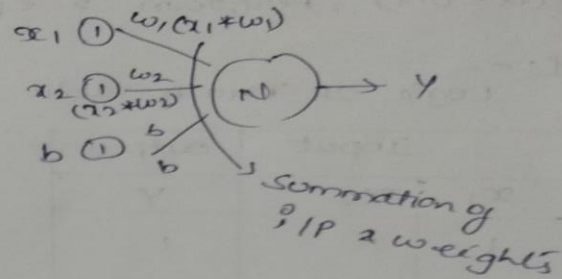
$$\Delta w_i = \alpha (t - Y_{in}) x_i$$

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

New bias: -

$$b_i(\text{new}) = b_i(\text{old}) + \Delta b_i$$

$$\Delta b_i = \alpha (t - Y_{in}); \quad b_i(\text{old}) + \alpha (t - Y_{in}); \quad \Delta b_i = \alpha (t - Y_{in})$$



$$\Delta w_1 = 0.1 (0.7) * 1$$

$$= 0.07$$

$$\Delta w_2 = 0.1 (0.7) * 1$$

$$= 0.07$$

$$\Delta b_i = \alpha (t - y_{in})$$

$$= 0.1 (0.7)$$

$$= 0.07$$

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1$$

$$= 0.1 + 0.07$$

$$= 0.17$$

$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2$$

$$= 0.1 + 0.07$$

$$= 0.17$$

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

$$= 0.1 + 0.07$$

$$= 0.17$$

Now consider second patterns:-

$$x_1 = 1 ; x_2 = -1 ; t = 1$$

$$\text{Net i/p} = (\text{old weights}) = [w_1 = 0.17, w_2 = 0.17, b = 0.17]$$

$$y_{in} = w_1 x_1 + w_2 x_2 + b$$

$$= (0.17)(1) + (0.17)(-1) + 0.17$$

$$y_{in} = 0.17$$

(2)



$$t - y_{in} = 1 - 0.17 = 0.83$$

$$\text{Error } (t - y_{in})^2 = (0.83)^2 = 0.69$$

$$\Delta w_1 = \alpha (t - y_{in}) x_1$$

$$= (0.1) (0.83) (1)$$

$$= 0.083$$

$$\Delta w_2 = \alpha (t - y_{in}) x_2$$

$$= (0.1) (0.83) (-1)$$

$$= -0.083$$

$$\Delta b = \alpha (t - y_{in})$$

$$= (0.1) (0.83)$$

$$= 0.083$$

New weights :

$$w_1 = w_1(\text{old}) + \Delta w_1$$

$$= 0.17 + 0.083$$

$$w_1 = 0.253$$

$$w_2 = w_2(\text{old}) + \Delta w_2$$

$$= 0.17 + (-0.083)$$

$$w_2 = 0.087$$

$$b = b(\text{old}) + \Delta b$$

$$= 0.17 + 0.083$$

$$= 0.253$$



Take Third pattern.

③

$$x_1 = -1, x_2 = 1, t = 1$$

$$w(\text{old}) = [w_1 = 0.253, w_2 = 0.087, b = 0.253]$$

$$\text{Net i/p } (Y_{in}) = w_1 x_1 + w_2 x_2 + b$$

$$= (0.253)(-1) + (0.087)(1) + 0.253$$

$$Y_{in} = (-0.253) + 0.087 + 0.253$$

$$Y_{in} = 0.087$$

$$t - Y_{in} = (1 - 0.087)$$

$$= 0.913$$

$$\text{Error } (t - Y_{in})^2 = (0.913)^2$$

$$= 0.83$$

$$\Delta w_1 = \alpha (t - Y_{in}) x_1$$

$$= (0.1)(0.913)(-1)$$

$$= -0.0913$$

$$\Delta w_2 = \alpha (t - Y_{in}) x_2$$

$$= (0.1)(0.913)(1)$$

$$= 0.0913$$

$$\Delta b = \alpha (t - Y_{in})$$

$$= 0.1(0.913)$$

$$= 0.0913$$

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1$$

$$= 0.253 + (-0.0913)$$

$$= 0.1617$$

$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2$$

$$= 0.087 + 0.0913$$

$$= 0.1783$$

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

$$= 0.253 + 0.0913$$

$$= 0.3443$$

Take 4th pattern

$$x_1 = -1; x_2 = -1; t = -1$$

$$w_1(\text{old}) = 0.1617, w_2(\text{old}) = 0.1783, b(\text{old}) = 0.3443$$

$$\text{NetIP}(y_{\text{in}}) = w_1 x_1 + w_2 x_2 + b$$

$$= (0.1617)(-1) + (0.1783)(-1) + 0.3443$$

$$= -0.1617 - 0.1783 + 0.3443$$

$$y_{\text{in}} = 0.0043$$

$$t - y_{\text{in}} = -1 - 0.0043$$

$$= -1.0043$$

$$\text{Error}(t - y_{\text{in}})^2 = 1.00$$

$$= 1.01$$

$$\Delta w_1 = \alpha (t - y_{\text{in}}) x_1$$

$$= (0.1)(-1.0043)(-1)$$

$$= 0.10043$$

$$\Delta w_2 = (0.1)(-1.0043)(-1)$$

$$= 0.10043$$

$$\Delta b = (0.1)(-1.0043)$$

$$= -0.10043$$

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1$$

$$= 0.1617 + 0.10043$$

$$= 0.262$$

$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2$$

$$= 0.1783 + 0.10043$$

$$= 0.278$$

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

$$= 0.3443 - 0.10043$$

$$= 0.24387$$

So for first Epoch, Mean Square Error =  $\frac{\sum (t - y_{in})^2}{4}$

$$MSE = \frac{0.49 + 0.69 + 0.83 + 1.01}{4}$$

$$\text{Epoch 1} = 0.75465$$

\* Epoch 2: -

$$x_1 = 1, x_2 = 1; t = 1$$

$$w_1(\text{old}) = 0.262; w_2(\text{old}) = 0.278; b = 0.243$$

$$\begin{aligned} \text{Netip}(y_{in}) &= w_1 x_1 + w_2 x_2 + b \\ &= (0.262)(1) + (0.278)(1) + 0.243 \\ &= 0.783 \end{aligned}$$

$$\begin{aligned} t - y_{in} &= (1 - 0.783) \\ &= 0.217 \end{aligned}$$

$$\begin{aligned} \text{Error}(t - y_{in})^2 &= (0.217)^2 \\ &= 0.046 \\ &= \end{aligned}$$

# Multi-Layer Perceptron (MLP) Networks and the Error Backpropagation Algorithm

- A multi-layer perceptron (MLP) network is a feedforward neural network with one or more hidden layers.
- Each hidden layer has its own specific function. Input terminals accept input signals from the outside world and redistribute these signals to all neurons in a hidden layer.
- The output layer accepts a stimulus pattern from a hidden layer and establishes the output pattern of the entire network.
- Neurons in the hidden layers perform transformation of input attributes; the weights of the neurons represent the features in the transformed domain.
- These features are then used by the output layer in determining the output pattern. A hidden layer 'hides' its desired output.



The derivation of error-backpropagation algorithm will be given here for two MLP structures

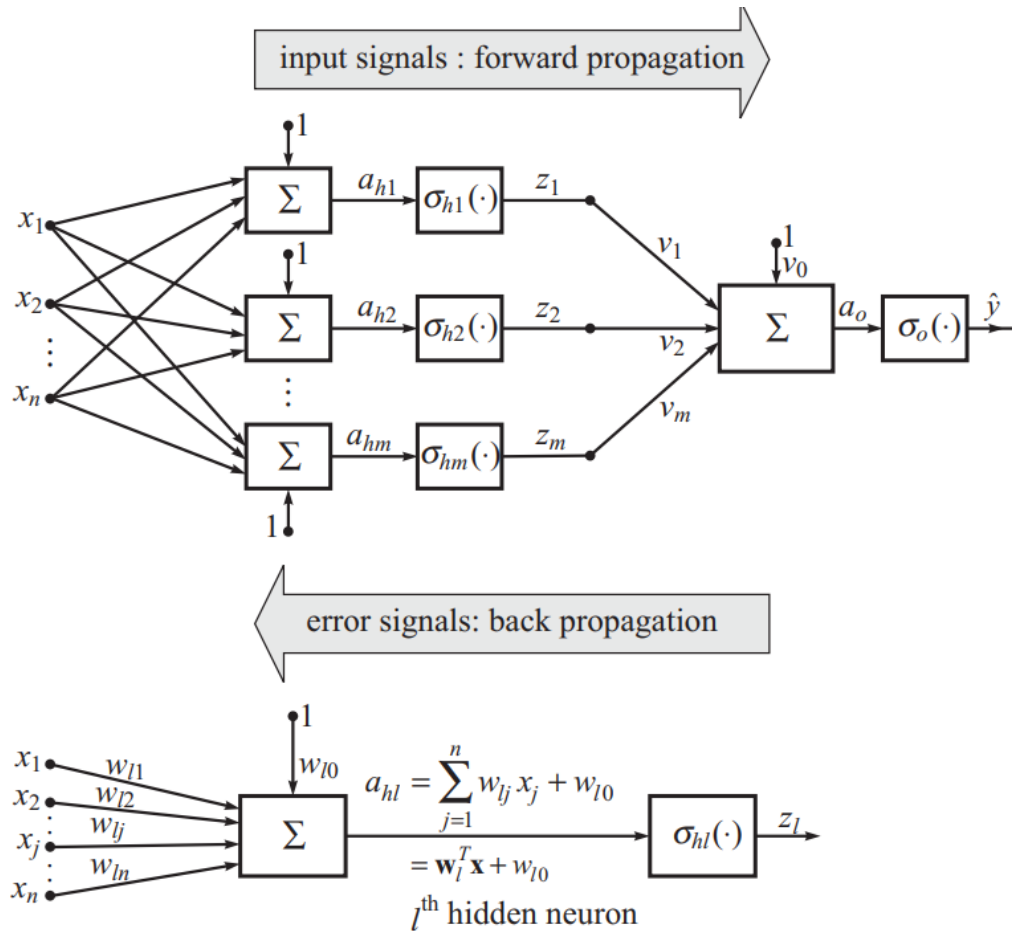


Figure 5.14 Scalar-output MLP network

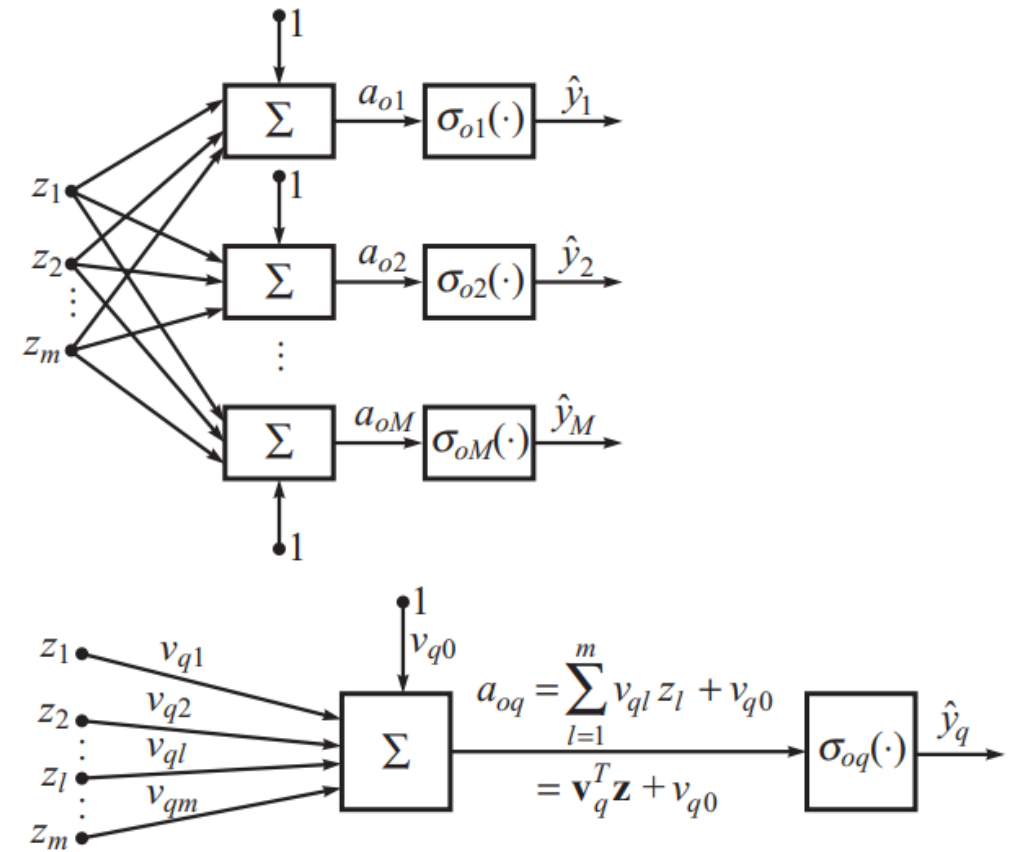


Figure 5.15 Output layer of vector-output MLP network

# Training Protocols

In the steepest descent mode, the error function  $E$  is given by,

$$E(k) = \frac{1}{2} \sum_{i=1}^N (y^{(i)} - \hat{y}(k))^2 = \frac{1}{2} \sum_{i=1}^N [e(k)]^2$$
$$\hat{y}(k) = \sum_{j=1}^n w_j(k) x_j^{(i)} + w_0(k)$$

All the patterns are presented to the network before a step of weight-update takes place.

In the stochastic gradient descent, the error function is given by,

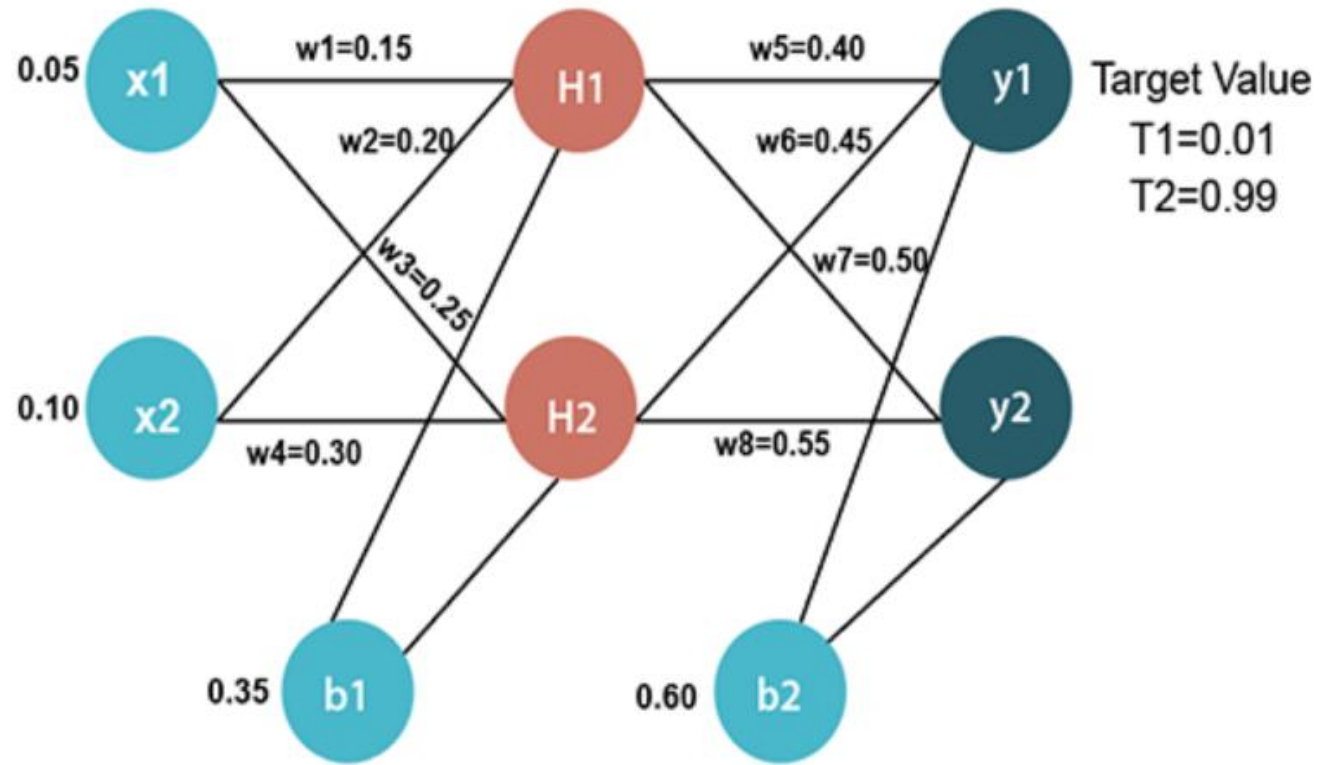
$$E(k) = \frac{1}{2} (y^{(i)} - \hat{y}(k))^2 = \frac{1}{2} [e(k)]^2$$
$$\hat{y}(k) = \sum_{j=1}^n w_j(k) x_j^{(i)} + w_0(k)$$

The weights are updated for each of the training pairs.

- These two schemes will be referred to as two protocols of training: batch training, and incremental training .
- Whereas the batch training rule computes the weight updates after summing errors over all the training examples (batch of training data), the idea behind incremental training rule is to update weights incrementally, following the calculation of error for each individual example (incremental training rule gives stochastic approximation of batch training which implements standard gradient descent (steepest descent)).
- The gradient descent weight-update rules for MLP networks are similar to the delta training rule described in the previous section for a single neuron.

# Error Back propagation

- The error backpropagation gives error terms for hidden units outputs. Using these error terms, the weights  $w_{lj}$  and  $w_{l0}$  between input terminals and hidden units are updated in a usual incremental/batch training mode.
- Thus, in a backpropagation network, the learning algorithm has two phases: the forward propagation to compute MLP outputs, and the back propagation to compute backpropagated errors.
- In the forward propagation phase, the input signals  $x$  are propagated through the network from input terminals to output layer, while in the backpropagation phase, the errors at the output nodes are propagated backwards from output layer to input terminals.
- This is an indirect way in which the weights  $w_{lj}$  and  $w_{l0}$  can influence the network outputs and hence the cost function  $E$



Here is the link for the solution

<https://www.javatpoint.com/pytorch-backpropagation-process-in-deep-neural-network>

# MULTI-CLASS DISCRIMINATION WITH MLP NETWORKS

- Multi-Layer Perceptron (MLP) networks can be used to realize nonlinear classifier. Most of the real-world problems require nonlinear classification.
- For a binary classification problem, the network is trained to fit the desired values of  $y(i) \in [0, 1]$  Learning with Neural Networks (NN) representing Classes 1 and 2, respectively. The network output  $\hat{y}$  approximates  $P(\text{Class 1}|\mathbf{x})$ ;  $P(\text{Class 2}|\mathbf{x}) = 1 - \hat{y}$
- When training results in perfect fitting, then the classification outcome is obvious. Even when target values cannot fit perfectly, we put some thresholds on the network output  $\hat{y}$ . For  $[0, 1]$  target values:

$$\text{Class} = \begin{cases} 1 & \text{if } 0 < \hat{y} \leq 0.5 \\ 2 & \text{if } \hat{y} > 0.5 \end{cases}$$

What about an example that is very close to the boundary  $\hat{y} = 0.5$ ?

- Ambiguous classification may result in near or exact ties.
- Instead of '0' and '1' values for classification, we may use values of 0.1 and 0.9. 0 and 1 is avoided because it is not possible for the sigmoidal units to produce these values, considering finite weights.
- If we try to train the network to fit target values of precisely 0 and 1, gradient descent will make the weights grow without limits.
- On the other hand, values of 0.1 and 0.9 are attainable with the use of the sigmoid unit with finite weights.

- Consider now a multi-class discrimination problem. The inputs to the MLP network are just the values of the feature measurements.
- We may use a single log-sigmoid node for the output. For the M-class discrimination problem, the network is trained to fit the desired values of  $y(i) = \{0.1, 0.2, \dots, 0.9\}$  for Classes 1, 2, ..., 9 (for  $M = 9$ ), respectively.
- We put some thresholds on the network output  $\hat{y}$ . For  $\{0.1, \dots, 0.9\}$  desired values

$$\text{Class} = \begin{cases} 1 & \text{if } 0 < \hat{y} \leq 0.1 \\ 2 & \text{if } 0.1 < \hat{y} \leq 0.2 \\ \vdots & \\ 9 & \text{if } \hat{y} > 0.8 \end{cases}$$



- Note that the difference between the desired network output for various classes is small; ambiguous classification may result in near or exact ties.
- A more realistic approach is **1-of-M encoding**. A separate node is used to represent each possible class and the target vectors consist of 0s everywhere except for the element that corresponds to the correct class.
- For a four-class ( $M = 4$ ) discrimination problem, the target vector

$$\mathbf{y}^{(i)} = [y_1^{(i)} y_2^{(i)} y_3^{(i)} y_4^{(i)}]^T$$

- To encode four possible classes, we use

$$\mathbf{y}^{(i)} = [1 \ 0 \ 0 \ 0]^T \text{ to encode Class 1}$$

$$\mathbf{y}^{(i)} = [0 \ 1 \ 0 \ 0]^T \text{ to encode Class 2}$$

$$\mathbf{y}^{(i)} = [0 \ 0 \ 1 \ 0]^T \text{ to encode Class 3}$$

$$\mathbf{y}^{(i)} = [0 \ 0 \ 0 \ 1]^T \text{ to encode Class 4}$$