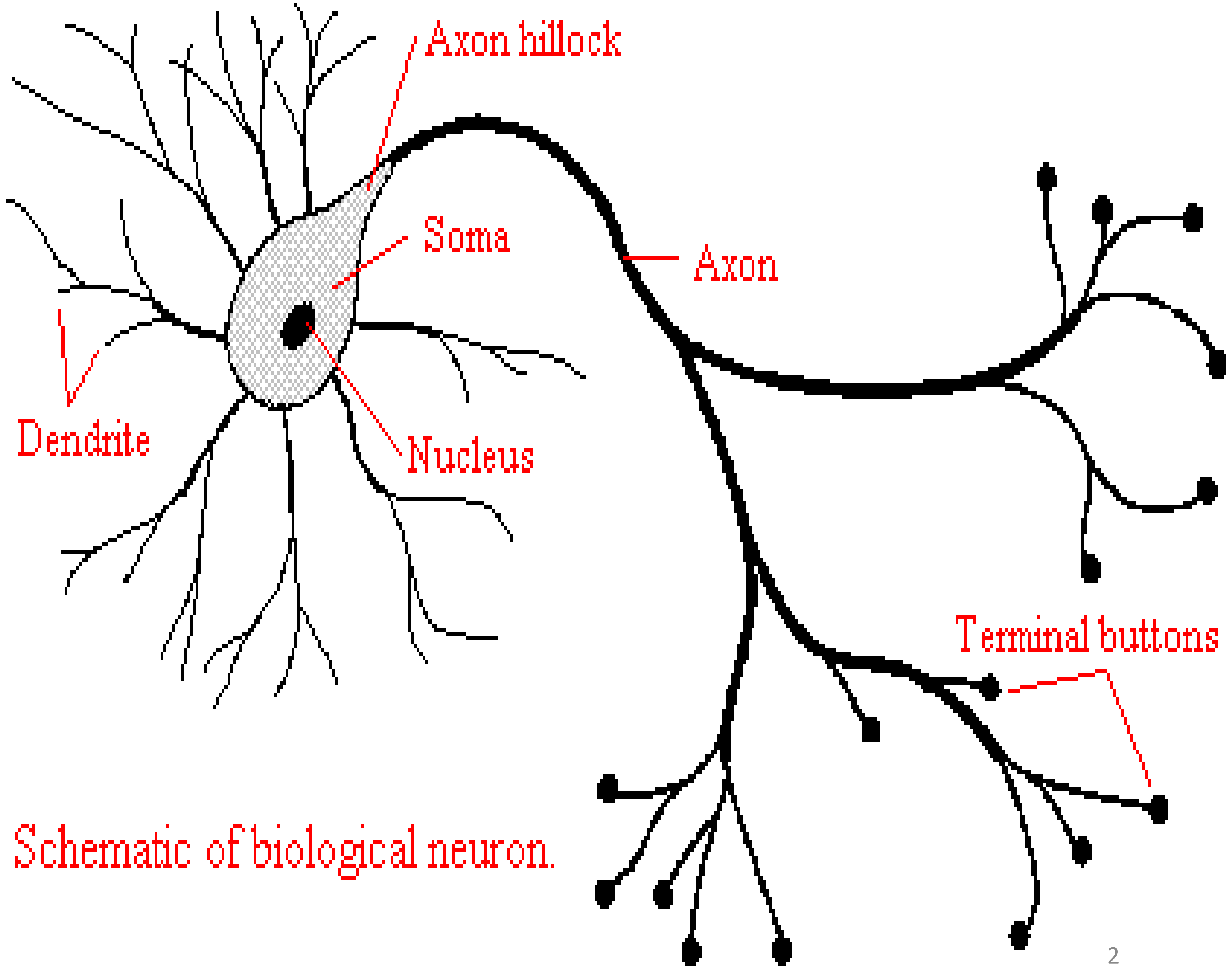
Chapter -7
Neural Networks

Compiled By: Bal Krishna Nyaupane
balkrishnanyaupane@gmail.com

Basic Components of Biological Neurons

- The brain is a collection of about 10 billion interconnected **neurons**.
- Each neuron is a cell that uses biochemical reactions to receive, process and transmit information.
- The majority of neurons encode their activation or outputs as a series of brief electrical pulses.
- A neuron's **dendritic tree** is connected to a thousand neighbouring neurons. When one of those neurons fire, a positive or negative charge is received by one of the dendrites. The strengths of all the received charges are added together through the processes of spatial and temporal summation.

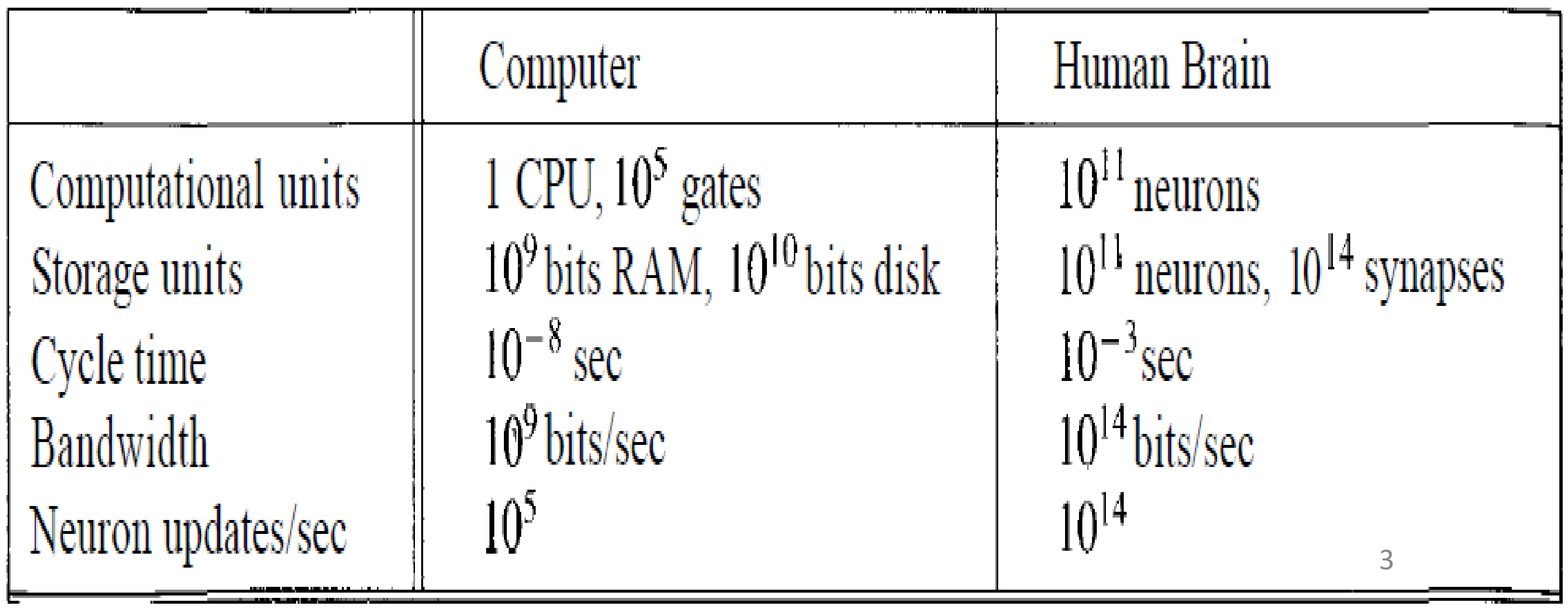


Schematic of biological neuron.

Basic Components of Biological Neurons

- The neuron’s cell body **(soma)** processes the incoming activations and converts them into output activations.
- The neuron’s nucleus contains the genetic material (DNA)
- **Dendrites** are fibers which emanate from the cell body and provide the receptive zone that receive activation from other neurons.
- **Axons** are fibers acting as transmission lines that send action potentials to other neurons.
- Each terminal button is connected to other neurons across a small gap called a **synapse**.The synapses allow signal transmission between the axons and the dendrites.

Biological NN	**Artificial NN**
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	weight



	Computer	Human Brain
Computational units	1 CPU, 10^5 gates	10^{11} neurons
Storage units	10^9 bits RAM, 10^{10} bits disk	10^{11} neurons, 10^{14} synapses
Cycle time	10^{-8} sec	10^{-3}sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec
Neuron updates/sec	10^5	10^{14}

Introduction to Neural Networks

- **McCulloch & Pitts (1943)** are generally recognised as the designers of the first neural network.
- The inventor of the first neurocomputer, ***Dr. Robert Hecht-Nielsen***, defines a neural network as: a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.
- An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the biological nervous systems, such as the human brain's information processing mechanism.
- An artificial network consists of a pool of simple processing units which communicate by sending signals to each other over a large number of weighted connections.
- Artificial Neural Networks (ANNs) are networks of Artificial Neurons and hence constitute crude approximations to parts of real brains.
- Computers point of view, an ANN is just a parallel computational system consisting of many simple processing elements connected together in a specific way in order to perform a particular task.
- An Artificial Neural Network is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. NNs, like people, learn by example.

Introduction to Neural Networks

- Neural networks are a powerful technique to solve many real world problems. They have the ability to learn from experience in order to improve their performance and to adapt themselves to changes in the environment. In addition to that they are able to deal with incomplete information or noisy data and can be very effective especially in situations where it is not possible to define the rules or steps that lead to the solution of a problem.
- **Why are Artificial Neural Networks?**
  - They are extremely powerful computational devices.
  - Massive parallelism makes them very efficient
  - They can learn and generalize from training data, so there is no need for enormous feats of programming
  - They are particularly fault tolerant.
  - They are very noise tolerant, so they can cope with situations where normal symbolic systems would have difficulty
  - In principle, they can do anything a symbolic/logic system can do, and more.
  - They can perform tasks that a linear program cannot perform.
  - Widely applied in data classification, clustering, pattern recognition.

Neural Network Applications

- **Brain modelling**
  - Aid our understanding of how the brain works, how behavior emerges from the interaction of networks of neurons, what needs to "get fixed" in brain damaged patients.
- **Real world applications**
  - ***Financial modelling*** – predicting the stock market
  - ***Time series prediction*** – climate, weather
  - ***Computer games*** – intelligent agents, chess, backgammon (A board game for two players; pieces move according to throws of the dice)
  - ***Robotics*** – autonomous adaptable robots
  - ***Pattern recognition*** – speech recognition, seismic activity, sonar signals (acoustic pulse in water and measures distances in terms of the time for the echo of the pulse to return)
  - ***Data analysis*** – data compression, data mining
  - ***Bioinformatics*** – DNA sequencing, alignment

Learning Processes in Neural Networks

- Neural network has the ability to learn from its environment, and to improve its performance through learning. The improvement in performance takes place over time in accordance with some prescribed measure.
- A neural network learns about its environment through an iterative process of adjustments applied to its synaptic weights and thresholds. The network becomes more knowledgeable about its environment after each iteration of the learning process.
- There are three broad types of learning:
    1. Supervised learning (i.e. learning with an external teacher)
    2. Unsupervised learning (i.e. learning with no help)
    3. Reinforcement learning (i.e. learning with limited feedback)

Learning Processes in Neural Networks

- **Supervised learning**
  - Supervised learning is the machine learning task of inferring a function from training data and the training data consist of a set of training examples i.e. a supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples.
  - In supervised the variables under investigation can be split into two groups: ***explanatory variables and one (or more) dependent variables.*** The target of the analysis is to specify a relationship between the explanatory variables and the dependent variable as it is done in regression analysis.
  - In supervised training, both the inputs and the outputs are provided. The network then processes the inputs and compares its resulting outputs against the desired outputs.
  - Errors are then propagated back through the system, causing the system to adjust the weights which control the network. This process occurs over and over as the weights are continually tweaked.
  - The set of data which enables the training is called ***the training set***. During the training of a network the same set of data is processed many times as the connection weights are ever refined.
  - Used for: classification, regression

Learning Processes in Neural Networks

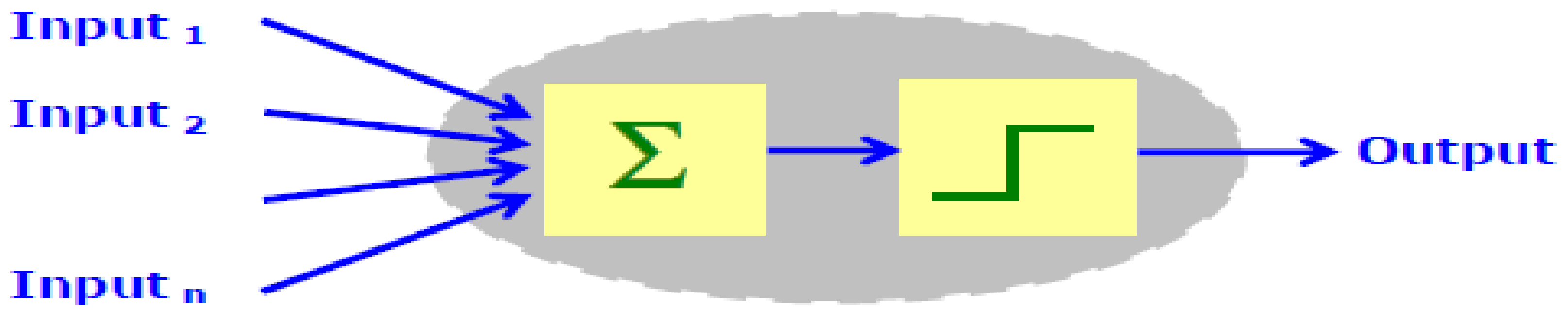
- **Unsupervised learning**
  - Unsupervised learning is the task of finding hidden structure in unlabeled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution.
  - In unsupervised learning situations all variables are treated in the same way, there is no distinction between explanatory and dependent variables.
  - In unsupervised training, the network is provided with inputs but not with desired outputs. The system itself must then decide what features it will use to group the input data. This is often referred to as self-organization or adaption.
  - The most common unsupervised learning method is cluster analysis, which is used for exploratory data analysis to find hidden patterns or grouping in data.
  - Used for: clustering

Learning Processes in Neural Networks

- **Reinforcement learning**
  - Reinforcement learning: in the case of the agent acts on its environment, it receives some evaluation of its action (reinforcement), but is not told of which action is the correct one to achieve its goal
  - It allows machines and software agents to automatically determine the ideal behaviour within a specific context, in order to maximize its performance. Simple reward feedback is required for the agent to learn its behaviour; this is known as the reinforcement signal.
  - In the reinforcement learning, the learner receives feedback about the appropriateness of its response. For correct responses, reinforcement learning resembles supervised learning.
  - However, the two forms of learning differ significantly for errors, situations in which the learner's behavior is in some way inappropriate. In these situations, supervised learning lets the learner know exactly what it should have done, whereas reinforcement learning only says that the behavior was inappropriate and (usually) how inappropriate it was.
  - Consider an animal that has to learn some aspects of how to walk. It tries out various movements. Some work -- it moves forward -- and it is rewarded. Others fail -- it stumbles or falls down -- and it is punished with pain.

McCulloch-Pitts (M-P) Neurons Equation

McCulloch-Pitts neuron is a simplified model of real biological neuron.



Simplified Model of Real Neuron
(Threshold Logic Unit)

The equation for the output of a McCulloch-Pitts neuron as a function of **1** to **n** inputs is written as

$$\text{Output} = \text{sgn}\left(\sum_{i=1}^{n} \text{Input}_i - \Phi \right)$$

where Φ is the neuron's activation threshold.

If $\sum_{i=1}^{n} \text{Input}_i \geq \Phi$ then Output = 1

If $\sum_{i=1}^{n} \text{Input}_i < \Phi$ then Output = 0

Artificial Neuron- Basic Elements

Neuron consists of three basic components - weights, thresholds, and a single activation function.

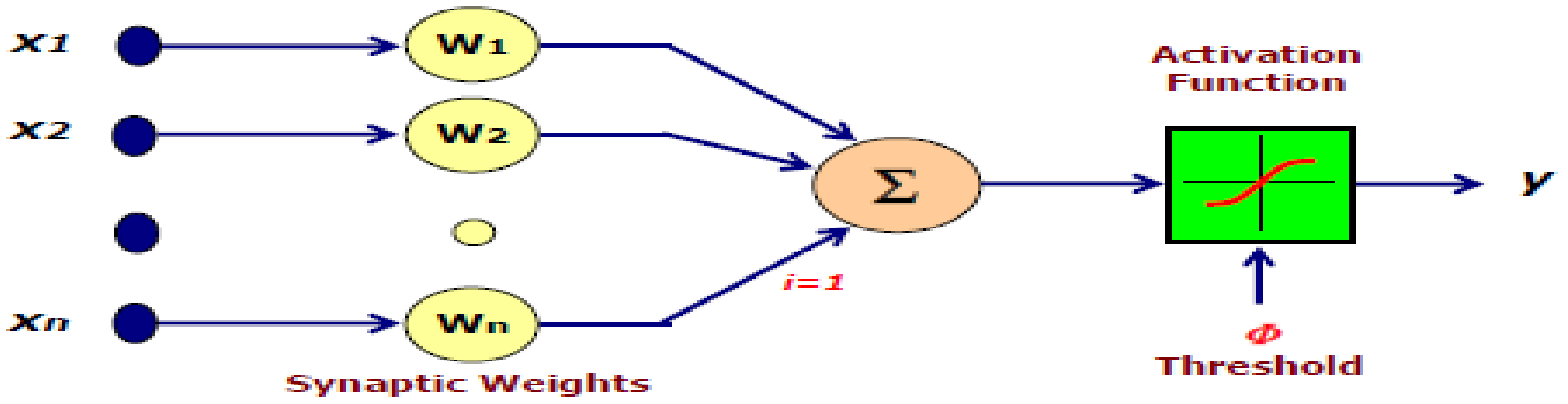


Fig Basic Elements of an Artificial Linear Neuron

- **Weighting Factors w**

  The values $w_1, w_2, \ldots w_n$ are weights to determine the strength of input vector $X = [x_1, x_2, \ldots, x_n]^T$. Each input is multiplied by the associated weight of the neuron connection $X^T W$. The +ve weight excites and the -ve weight inhibits the node output.

$$I = X^T . W = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n = \sum_{i=1}^{n} x_i w_i$$

Artificial Neuron- Basic Elements

- **Threshold Φ**

  The node's internal threshold Φ is the magnitude offset. It affects the activation of the node output y as:

  $$Y = f(I) = f\{\sum_{i=1}^{n} x_i w_i - \Phi_k\}$$

  To generate the final output Y , the sum is passed on to a non-linear filter f called Activation Function or Transfer function or Squash function which releases the output Y.

- **Activation Function**

  An activation function f performs a mathematical operation on the signal output. The most common activation functions are:

  - Linear Function,
  - Threshold Function,
  - Piecewise Linear Function,
  - Sigmoidal (S shaped) function,
  - Tangent hyperbolic function

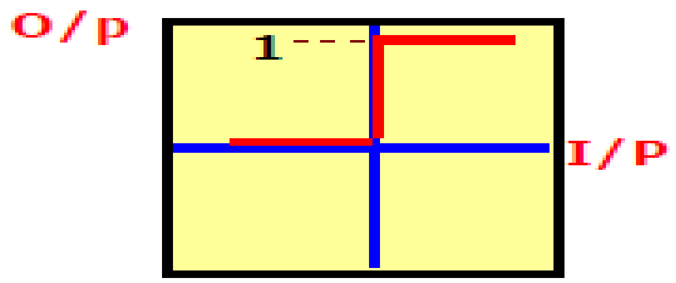
  The activation functions are chosen depending upon the type of problem to be solved by the network.

Activation function

Threshold Function

A threshold (hard-limiter) activation function is either a binary type or a bipolar type as shown below.

binary threshold



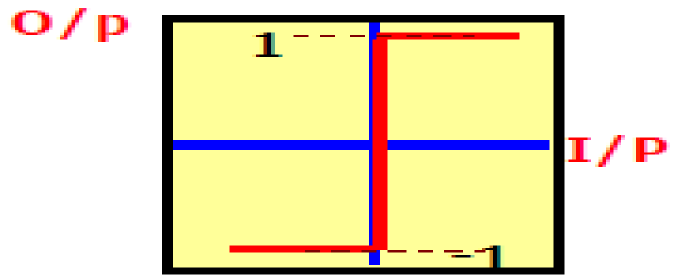
Output of a binary threshold function produces :

1 if the weighted sum of the inputs is +ve,

0 if the weighted sum of the inputs is -ve.

$$Y = f(I) = \begin{cases} 1 & \text{if } I \geq 0 \\ 0 & \text{if } I < 0 \end{cases}$$

bipolar threshold



Output of a bipolar threshold function produces :

1 if the weighted sum of the inputs is +ve,

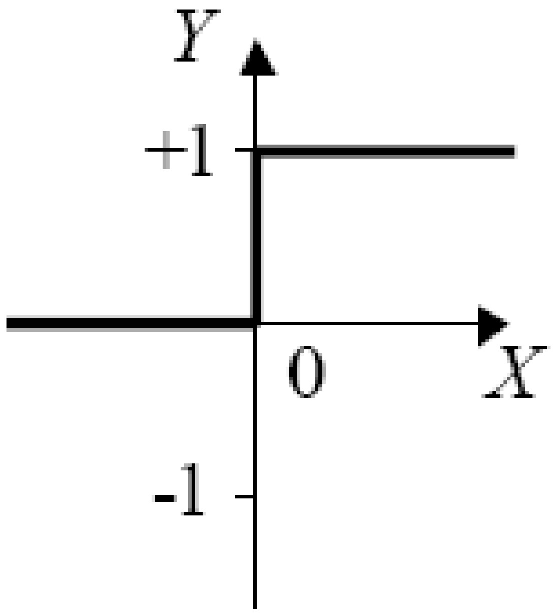
-1 if the weighted sum of the inputs is -ve.

$$Y = f(I) = \begin{cases} 1 & \text{if } I \geq 0 \\ -1 & \text{if } I < 0 \end{cases}$$

Neuron with hard limiter activation function is called McCulloch-Pitts model.

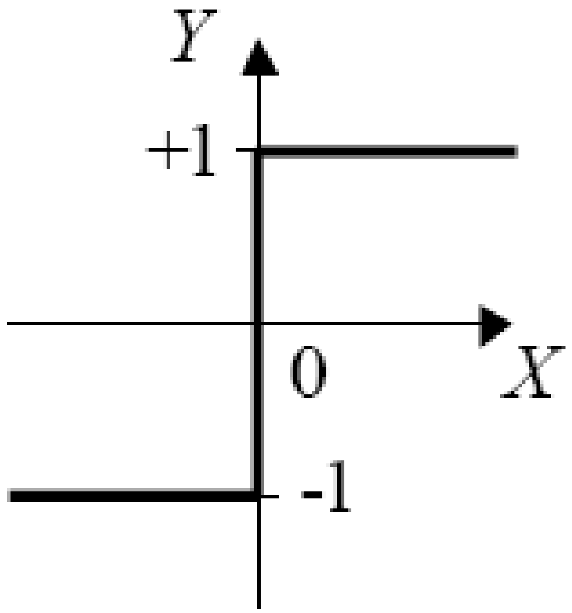
Activation function

Step function



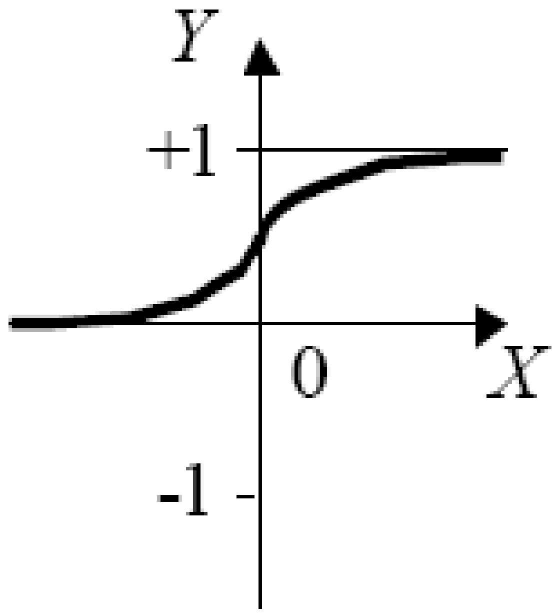
$$Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{ if } X < 0 \end{cases}$$

Sign function



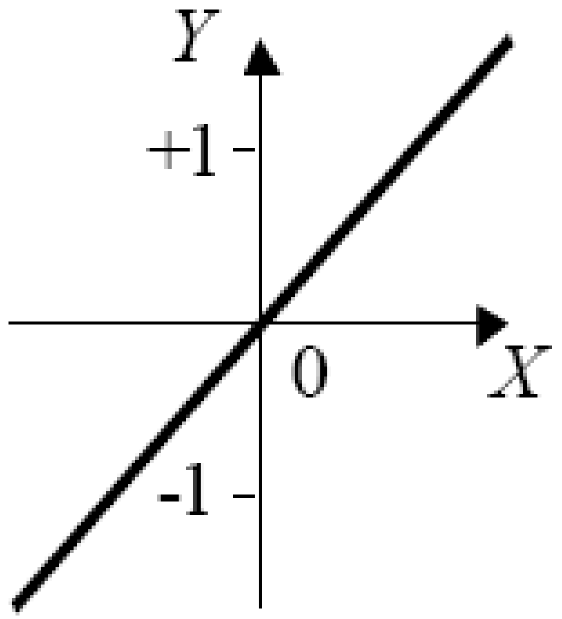
$$Y^{sign} = \begin{cases} +1, \text{ if } X \geq 0 \\ -1, \text{ if } X < 0 \end{cases}$$

Sigmoid function



$$Y^{sigmoid} = \frac{1}{1 + e^{-X}}$$

Linear function



$$Y^{linear} = X$$

Example

The neuron shown consists of four inputs with the weights.

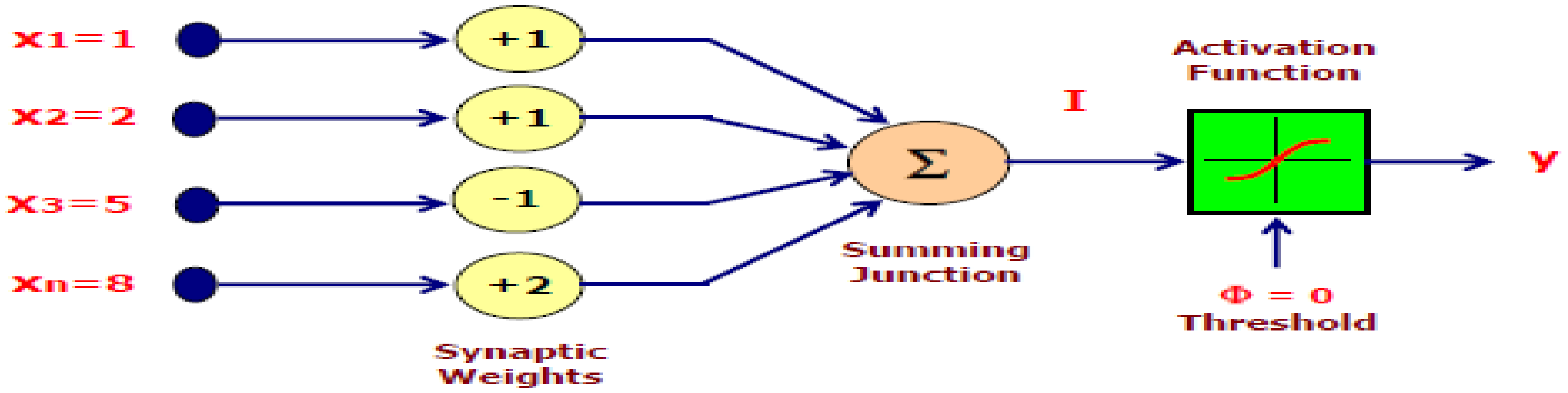


Fig Neuron Structure of Example

The output I of the network, prior to the activation function stage, is

$$I = X^T . W = \begin{bmatrix} 1 & 2 & 5 & 8 \end{bmatrix} \bullet \begin{pmatrix} +1 \\ +1 \\ -1 \\ +2 \end{pmatrix} = 14$$

$$= (1 \times 1) + (2 \times 1) + (5 \times -1) + (8 \times 2) = 14$$

With a binary activation function the outputs of the neuron is:

y (threshold) = 1;

Types of Layers in ANN

- **The input layer**
    - Introduces input values into the network.
    - No activation function or other processing.
- **The hidden layer(s)**
    - Perform classification of features
    - Two hidden layers are sufficient to solve any problem
    - Features imply more layers may be better
- **The output layer**
    - Functionally just like the hidden layers
    - Outputs are passed on to the world outside the neural network.

Types/Architectures/Structures/Topologies of Neural Network

- **Single layer feed forward Network**
  - The single layer feed forward Network consists of a single layer of weights, where the inputs are directly connected to the outputs ,via a series of weights. The synaptic links carrying weights connect every input to every output, but not other way. This way it is considered a network of feed-forward type.
  - The sum of the products of the weights and the inputs is calculated in each neuron node, and if the value is above some threshold (*typically 0*) the neuron fires and takes the activated value (*typically 1*); otherwise it takes the deactivated value(*typically -1*).
  - For example, a simple Perceptron.

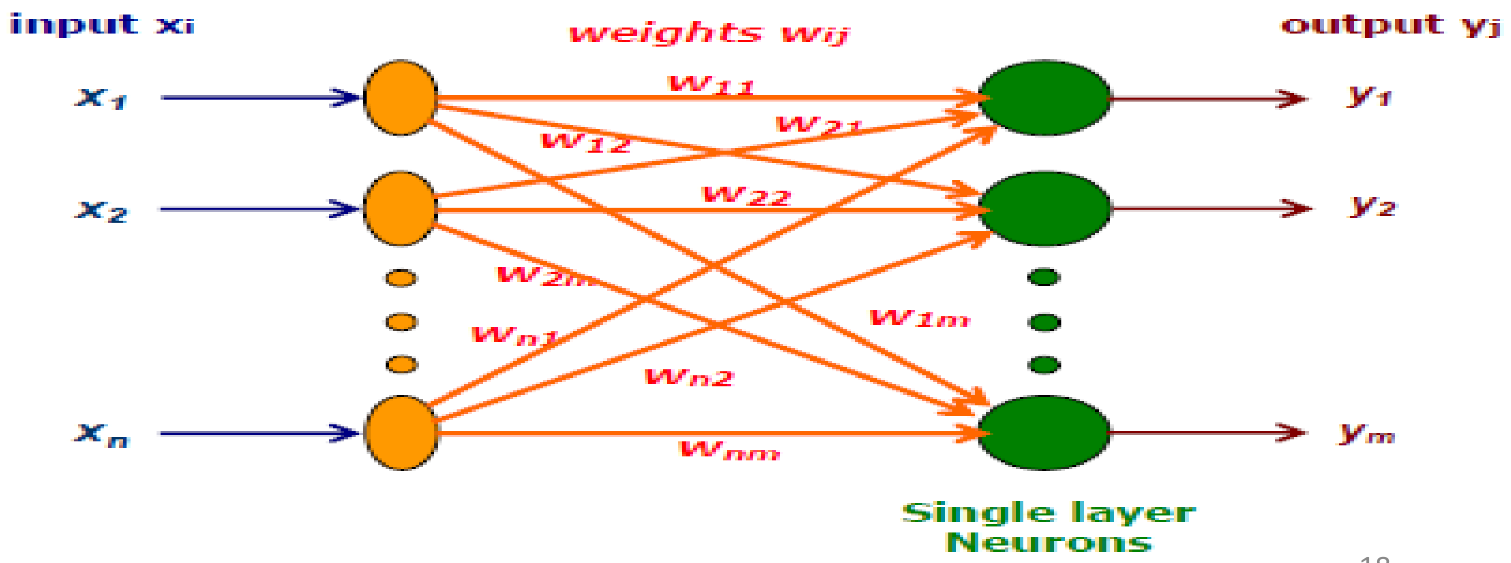


Fig. Single Layer Feed-forward Network

Types/Architectures/Structures/Topologies of Neural Network

- **Multi-layer feed forward Network**
  - One input layer, one output layer, and one or more hidden layers of processing units. The hidden layers sit in between the input and output layers, and are thus hidden from the outside world. The computational unit of hidden layers are known as hidden neurons.
  - The hidden layer does intermediate computation before directing the input to output layer. The input layer neurons are linked to the hidden layer neurons.
  - A multi-layer feedforward network with **l** input neurons, **m_1** neurons in first hidden layer, **m_2** neurons in second hidden layer, and **n** output layer is written as **(l - m_1 - m_2 – n)**
  - For example, a Multi-Layer Perceptron.

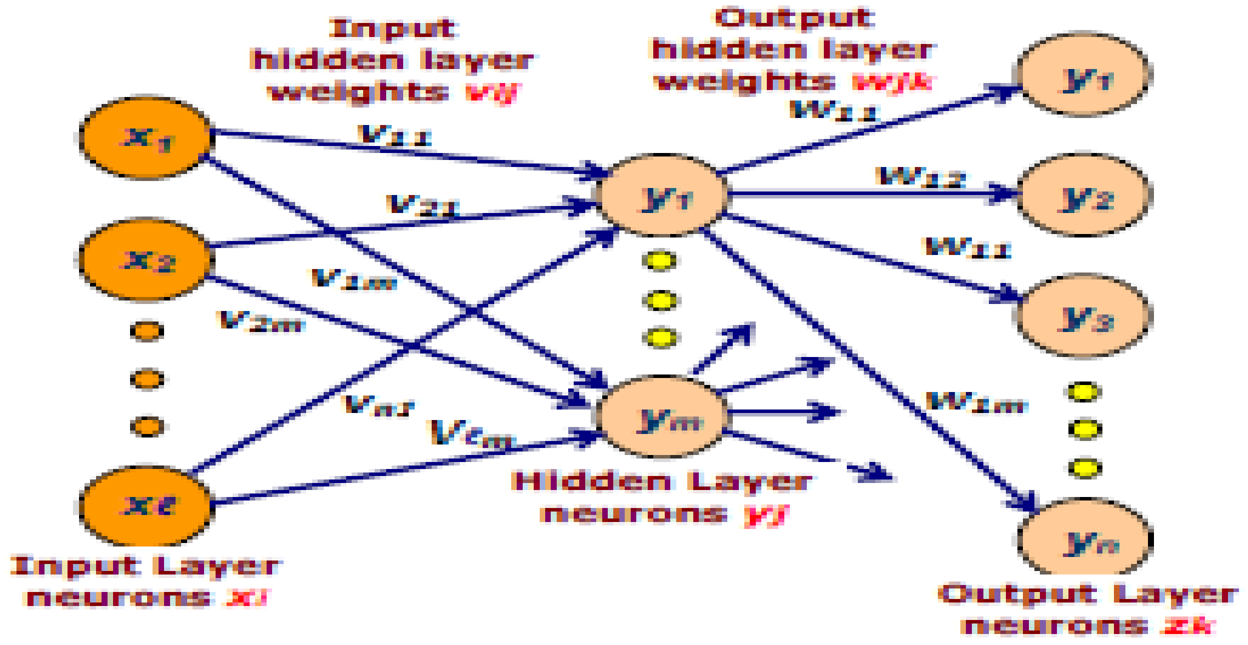


Fig. Multilayer feed-forward network in $(\ell - m - n)$ configuration.

Types/Architectures/Structures/Topologies of Neural Network

- **Recurrent Network**
  - A recurrent network has at least one feedback loop.
  - There could be neurons with self-feedback loop; that is the output of a neuron is feedback into itself as input.

Example :

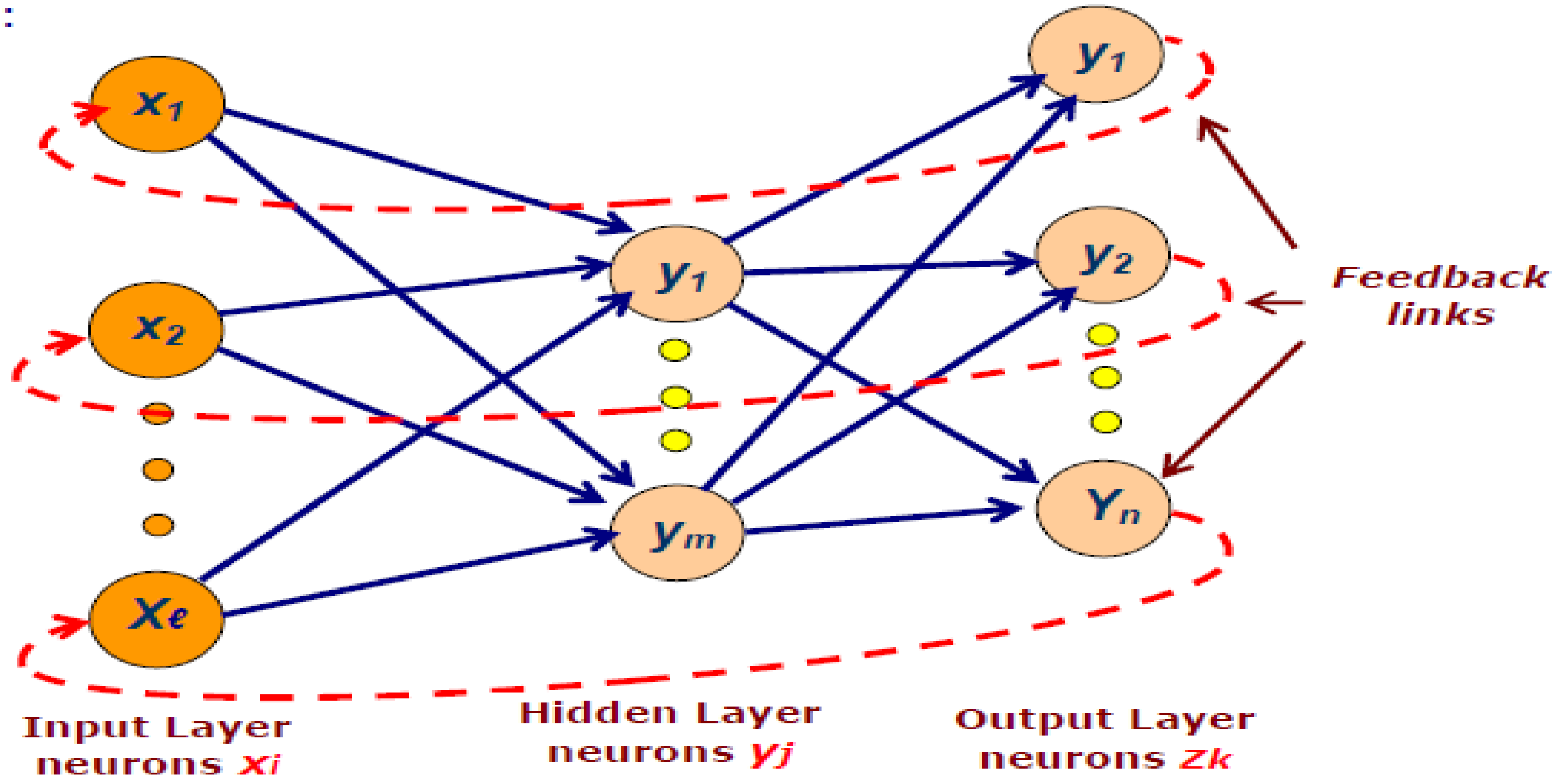
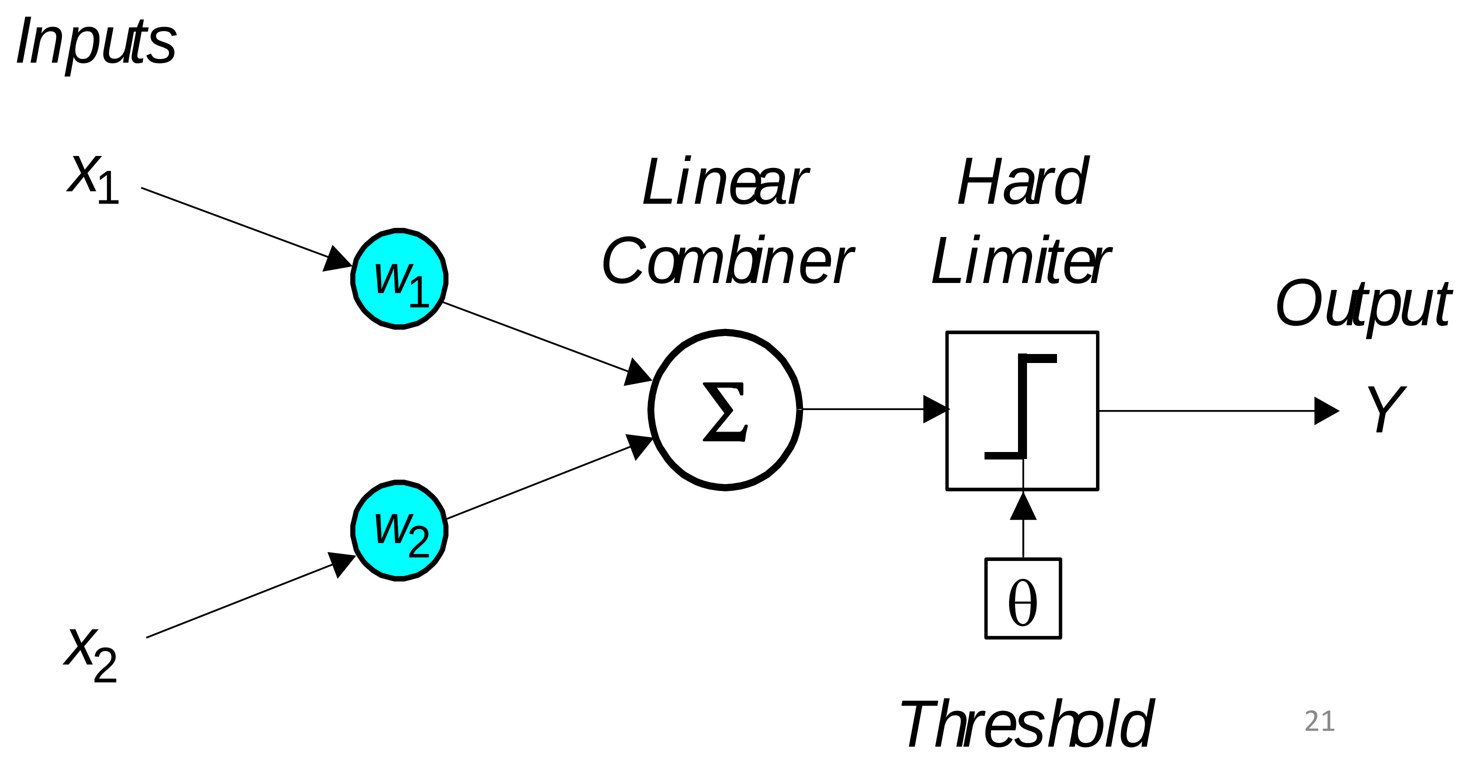

x_1
x_2
X_ℓ
y_1
y_m
y_1
y_2
Y_n
Feedback links
Input Layer neurons x_i
Hidden Layer neurons y_j
Output Layer neurons z_k

Fig. Recurrent neural network

The Perceptron

- First studied in the late 1950s.
- Also known as Layered Feed-Forward Networks.
- The operation of Rosenblatt's perceptron is based on the ***McCulloch and Pitts neuron model***. The model consists of a linear combiner followed by a hard limiter. The weighted sum of the inputs is applied to the hard limiter, which produces an output equal to +1 if its input is positive and -1 if it is negative.
- ***Single-layer two-input perceptron***



The perceptron learning rule

$$w_i(p+1) = w_i(p) + \alpha \cdot x_i(p) \cdot e(p)$$

- where p = 1, 2, 3, . . .

  α is the **learning rate**, α positive constant less than unity.

- The perceptron learning rule was first proposed by **Rosenblatt** in 1960. Using this rule we can derive the perceptron training algorithm for classification tasks.

- **Perceptron's training algorithm**

  <u>Step 1: Initialization</u>

  - Set initial weights w_1, w_2,…, w_n and threshold θ to random numbers in the range [-0.5, 0.5].
  - If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

Perceptron's training algorithm

<u>Step 2: Activation</u>

- Activate the perceptron by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired output $Y_d\ (p)$.
- Calculate the actual output at iteration $p = 1$

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p)\, w_i(p) - \theta\right]$$

  - where n is the number of the perceptron inputs, and step is a step activation function.
  - If at iteration p, the actual output is Y(p) and the desired output is Y_d (p), then the error is given by:

$$e(p) = Y_d(p) - Y(p)$$ where $p = 1, 2, 3, \ldots$

Perceptron's training algorithm

Step 3: Weight training

Update the weights of the perceptron

$$w_i(p+1) = w_i(p) + \Delta w_i(p)$$

where $\Delta w_i(p)$ is the weight correction at iteration p.

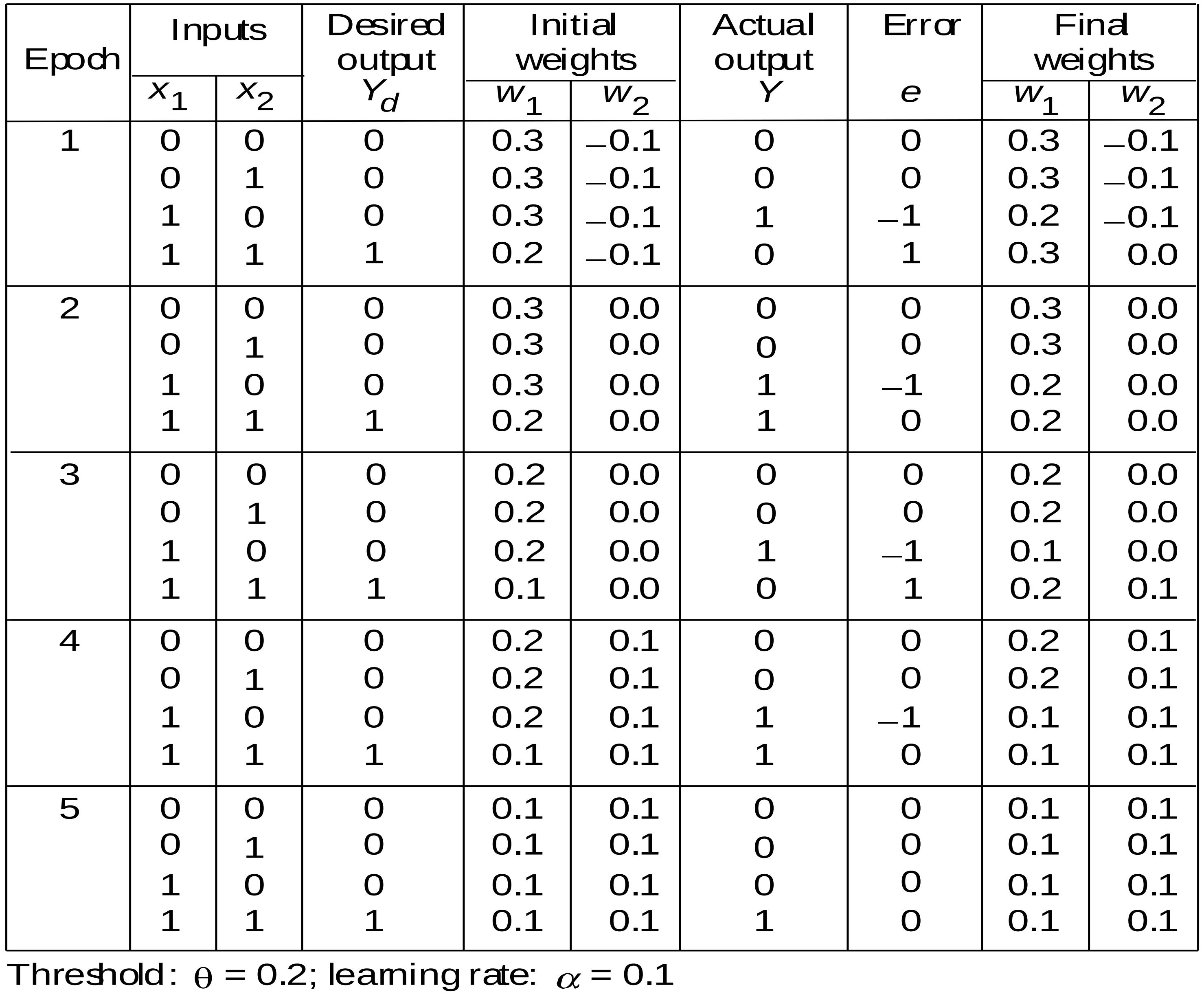
The weight correction is computed by the **delta rule**:

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

Step 4: Iteration

Increase iteration p by one, go back to *Step 2* and repeat the process until convergence.

Example of perceptron learning: the logical operation *AND*

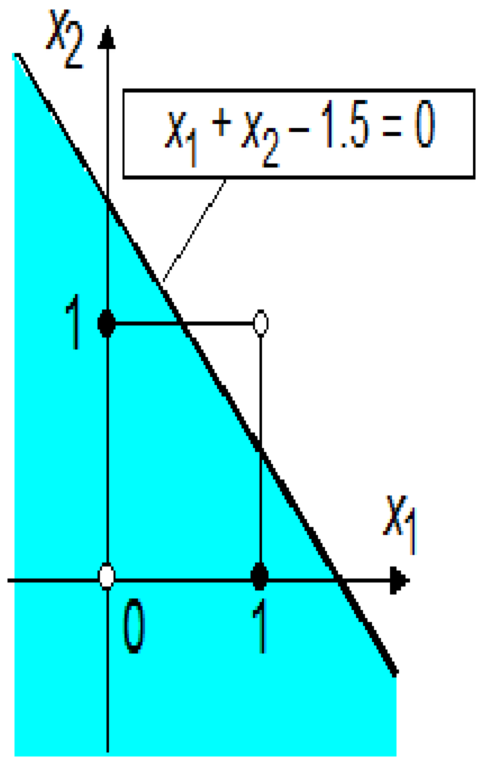


Epoch	Inputs		Desired output	Initial weights		Actual output	Error	Final weights	
	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	w_2
1	0	0	0	0.3	–0.1	0	0	0.3	–0.1
	0	1	0	0.3	–0.1	0	0	0.3	–0.1
	1	0	0	0.3	–0.1	1	–1	0.2	–0.1
	1	1	1	0.2	–0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	–1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	–1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	–1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

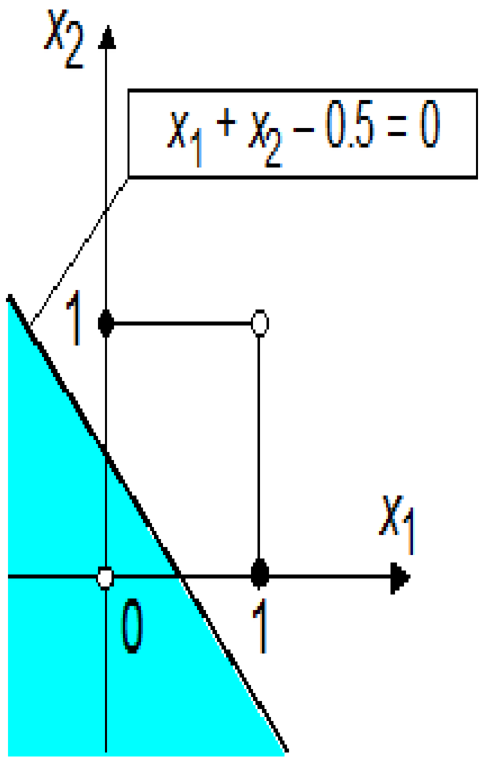
Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$

Perceptron: Linear separability

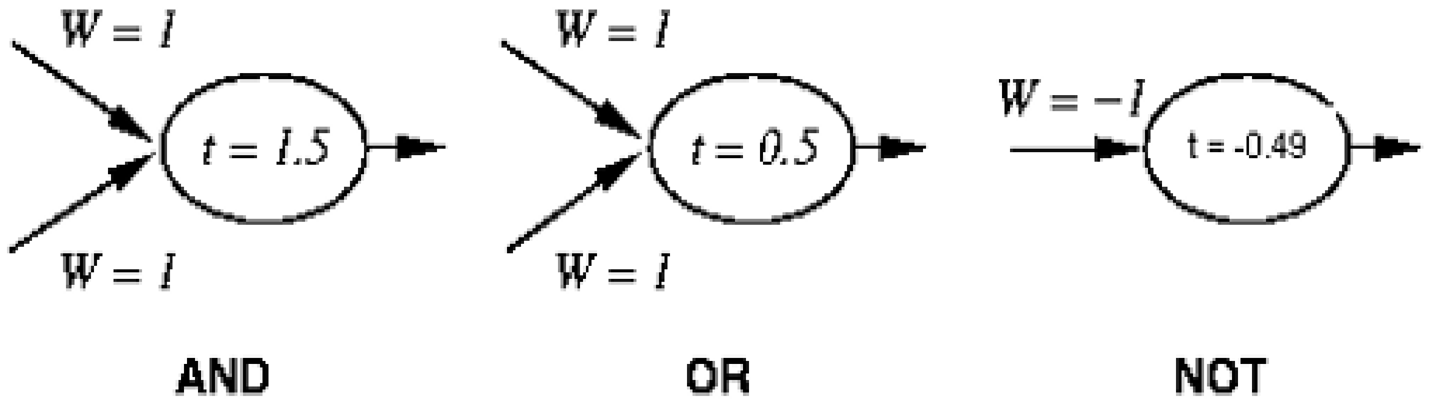
- The single layer perceptron algorithm converges if examples are linearly separable.
- A single layer perceptron can only learn linearly separable concepts.
- A single layer perceptron can learn the operations AND, OR, and NOT , but not Exclusive-OR.



AND



OR



$$\text{Output}=\begin{cases}1 \text{ if } \sum_{i=0} w_i x_i > t \\ 0 \text{ otherwise}\end{cases}$$

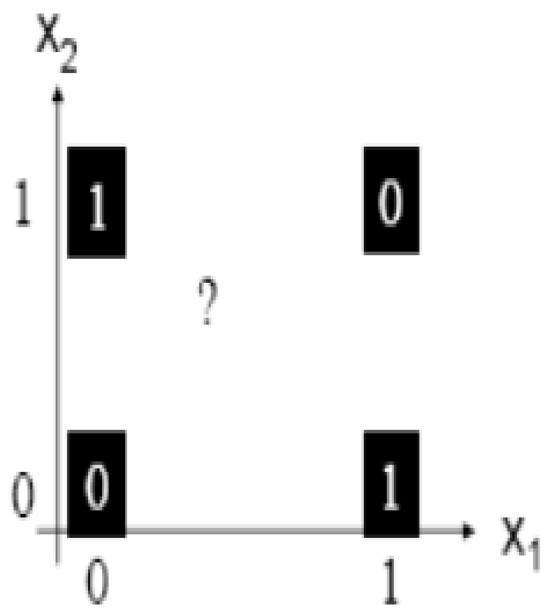
t=Linear threshold value

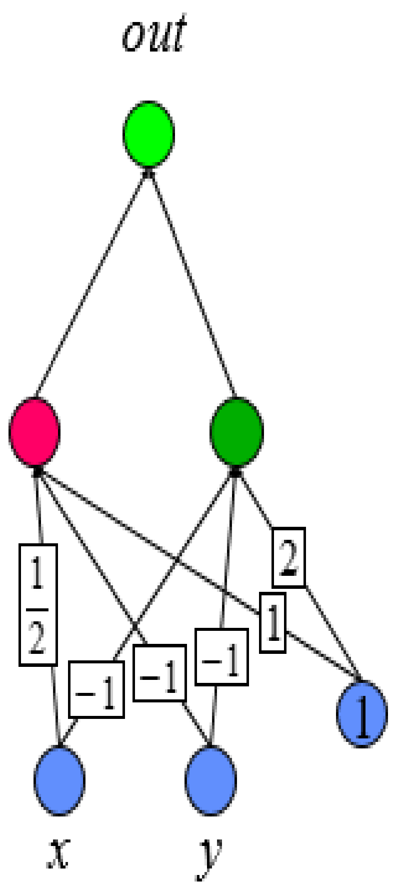
Example: Linear separability

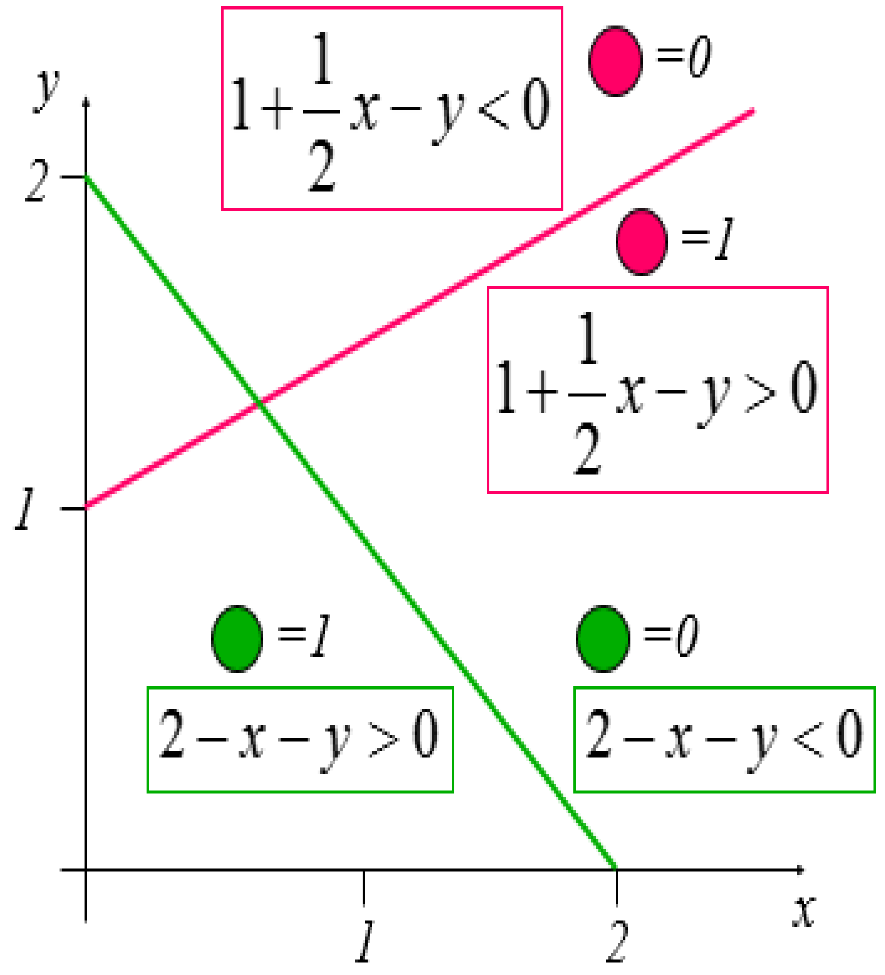
- **The Exclusive OR problem**: A single layer Perceptron cannot represent Exclusive OR since it is not linearly separable.

XOR function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

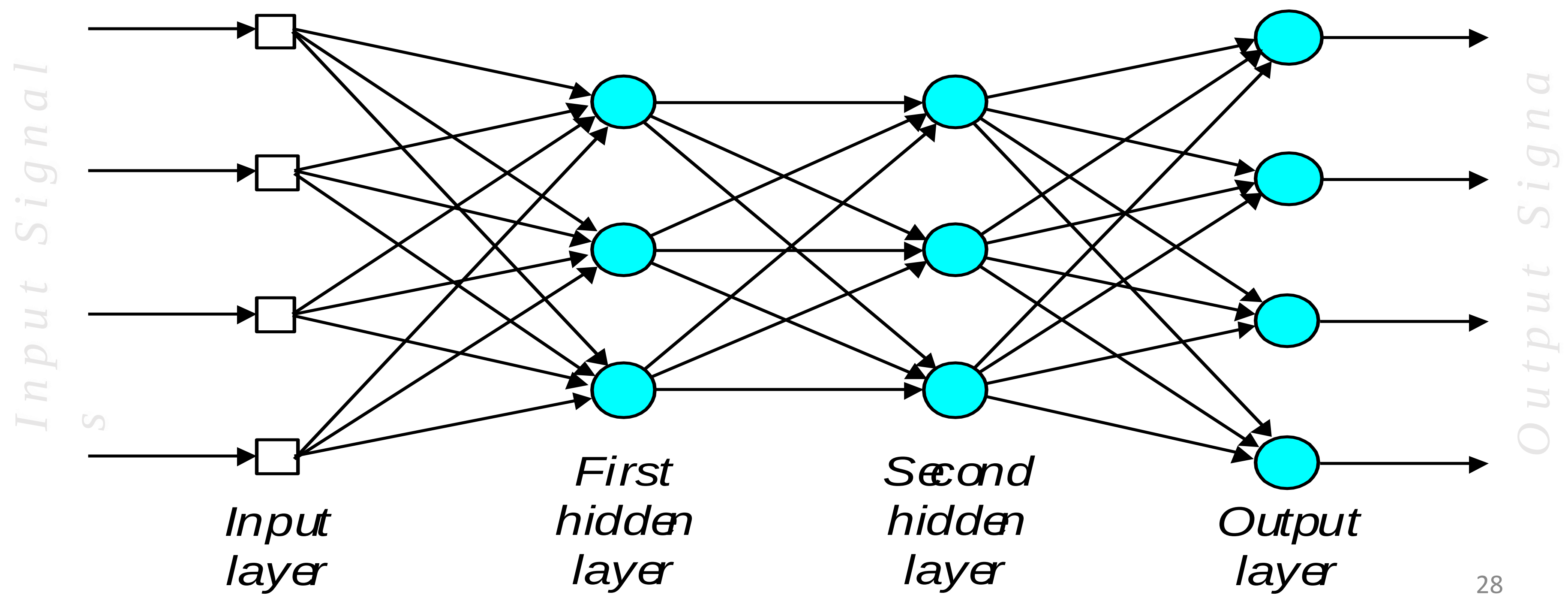






Multilayer Perceptron Neural Networks

- A multilayer perceptron is a feedforward neural network with one or more hidden layers.
- The network consists of an **input layer** of source neurons, at least one middle or **hidden layer** of computational neurons, and an **output layer** of computational neurons.
- The input signals are propagated in a forward direction on a layer-by-layer basis.
- ***Figure: Multilayer perceptron with two hidden layers***



Backpropagation Algorithm

- In a back-propagation neural network, the learning algorithm has **two phases**.
- First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer.
- If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

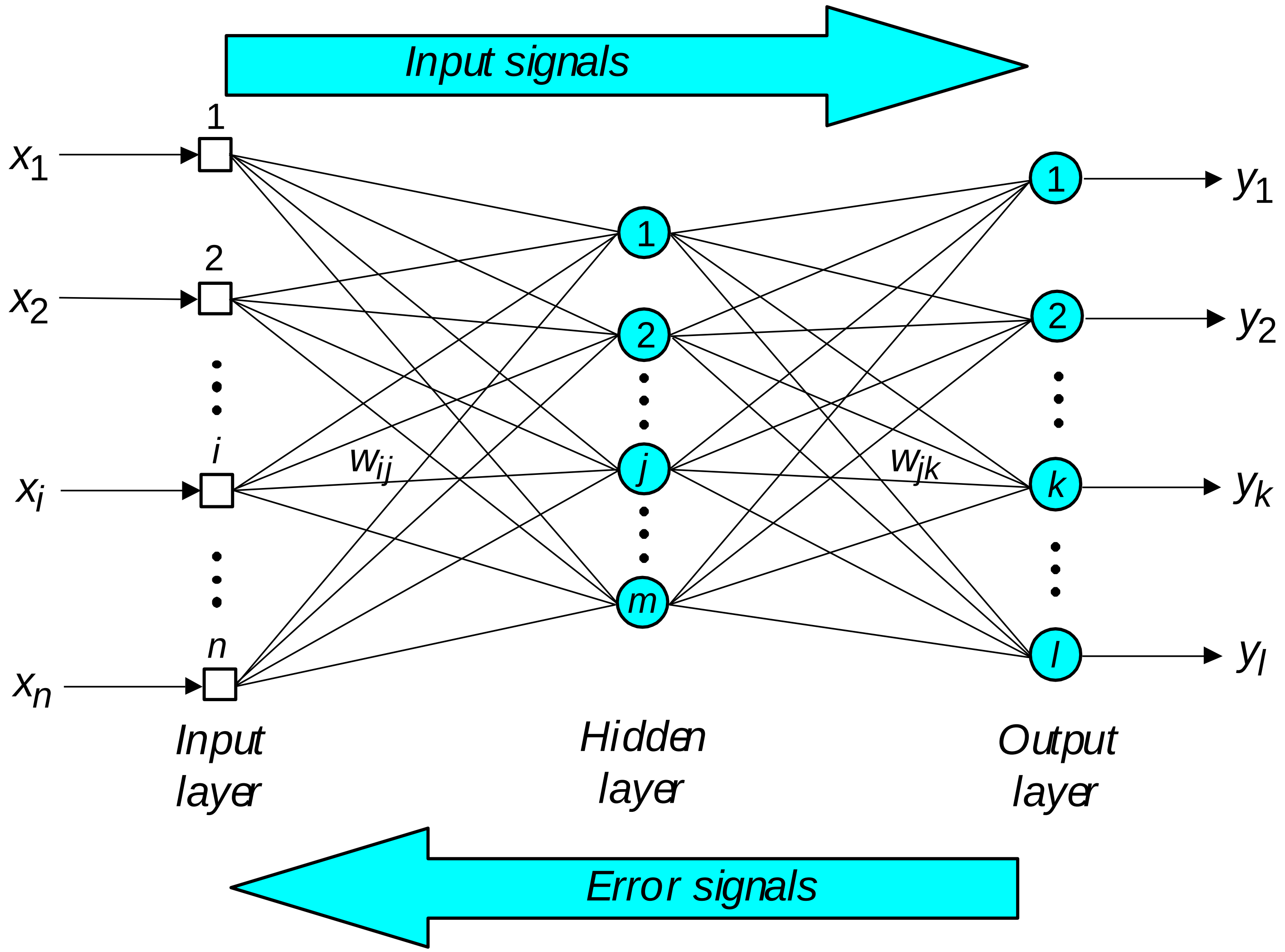


Figure: Three-layer back-propagation neural network

The Back-Propagation Algorithm

<u>Step 1: Initialization</u>

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range:

$$\left(-\frac{2.4}{F_i}, \quad +\frac{2.4}{F_i} \right)$$

where F_i is the total number of inputs of neuron i in the network. The weight initialization is done on a neuron-by-neuron basis.

Step 2: Activation

Activate the back-propagation neural network by applying inputs $x_1(p)$, $x_2(p)$,…, $x_n(p)$ and desired outputs $y_{d,1}(p)$, $y_{d,2}(p)$,…, $y_{d,n}(p)$.

(*a*) Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p) \cdot w_{ij}(p) - \theta_j\right]$$

where *n* is the number of inputs of neuron *j* in the hidden layer, and *sigmoid* is the *sigmoid* activation function.

(b) Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) \cdot w_{jk}(p) - \theta_k\right]$$

where m is the number of inputs of neuron k in the output layer.

Step 3: Weight training

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(*a*) Calculate the error gradient for the neurons in the output layer:

$$\delta_k(p) = y_k(p) \cdot [1 - y_k(p)] \cdot e_k(p)$$

where $e_k(p) = y_{d,k}(p) - y_k(p)$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \cdot y_j(p) \cdot \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

(*b*) Calculate the error gradient for the neurons in the hidden layer:

$$\delta_j(p) = y_j(p) \cdot [1 - y_j(p)] \cdot \sum_{k=1}^{l} \delta_k(p)\, w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \cdot x_i(p) \cdot \delta_j(p)$$

Update the weights at the hidden neurons:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

<u>Step 4: Iteration</u>

Increase iteration *p* by one, go back to *Step 2* and repeat the process until the selected error criterion is satisfied.

Example: The Back-Propagation Algorithm

- As an example, we may consider the three-layer back-propagation network. Suppose that the network is required to perform logical operation *Exclusive-OR*.

- The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ, connected to a fixed input equal to -1.

- The initial weights and threshold levels are set randomly as follows: $w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$, $w_{35} = -1.2$, $w_{45} = 1.1$, $\theta_3 = 0.8$, $\theta_4 = -0.1$ and $\theta_5 = 0.3$.

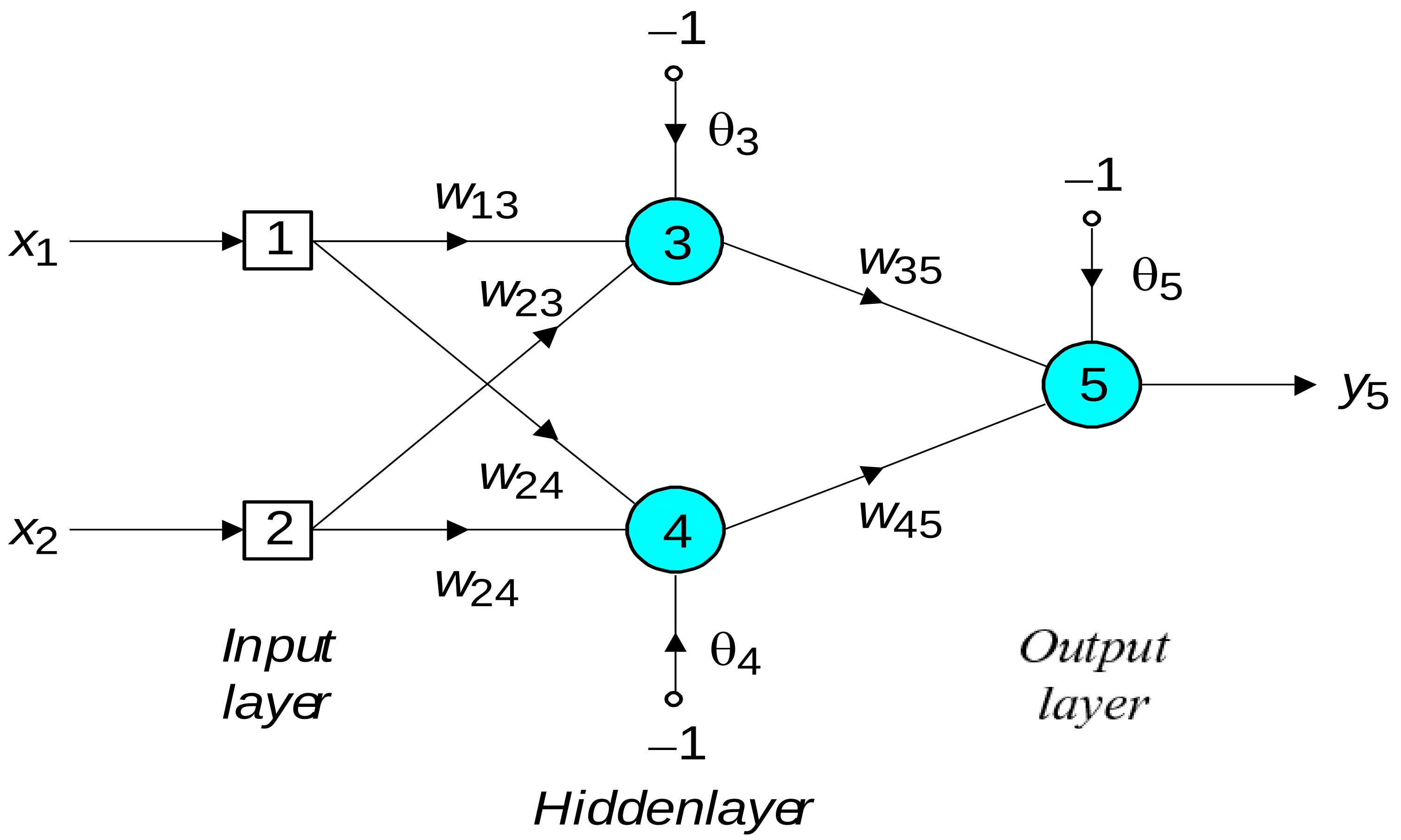


Figure: Three-layer network for solving the Exclusive-OR operation.

- We consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/\left[1 + e^{-(1\cdot 0.5 + 1\cdot 0.4 - 1\cdot 0.8)}\right] = 0.5250$$

$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/\left[1 + e^{-(1\cdot 0.9 + 1\cdot 1.0 + 1\cdot 0.1)}\right] = 0.8808$$

- Now the actual output of neuron 5 in the output layer is determined as:

$$y_5 = sigmoid(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/\left[1 + e^{-(-0.5250\cdot 1.2 + 0.8808\cdot 1.1 - 1\cdot 0.3)}\right] = 0.5097$$

- Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

- The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, *e*, from the output layer backward to the input layer.

- First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5 (1 - y_5)\, e = 0.5097 \cdot (1 - 0.5097) \cdot (-0.5097) = -0.1274$$

- Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha \cdot y_3 \cdot \delta_5 = 0.1 \cdot 0.5250 \cdot (-0.1274) = -0.0067$$
$$\Delta w_{45} = \alpha \cdot y_4 \cdot \delta_5 = 0.1 \cdot 0.8808 \cdot (-0.1274) = -0.0112$$
$$\Delta \theta_5 = \alpha \cdot (-1) \cdot \delta_5 = 0.1 \cdot (-1) \cdot (-0.1274) = -0.0127$$

- Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1 - y_3) \cdot \delta_5 \cdot w_{35} = 0.5250 \cdot (1 - 0.5250) \cdot (-0.1274) \cdot (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \cdot \delta_5 \cdot w_{45} = 0.8808 \cdot (1 - 0.8808) \cdot (-0.1274) \cdot 1.1 = -0.0147$$

- We then determine the weight corrections:

$$\Delta w_{13} = \alpha \cdot x_1 \cdot \delta_3 = 0.1 \cdot 1 \cdot 0.0381 = 0.0038$$

$$\Delta w_{23} = \alpha \cdot x_2 \cdot \delta_3 = 0.1 \cdot 1 \cdot 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \cdot (-1) \cdot \delta_3 = 0.1 \cdot (-1) \cdot 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \cdot x_1 \cdot \delta_4 = 0.1 \cdot 1 \cdot (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \cdot x_2 \cdot \delta_4 = 0.1 \cdot 1 \cdot (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \cdot (-1) \cdot \delta_4 = 0.1 \cdot (-1) \cdot (-0.0147) = 0.0015$$

- At last, we update all weights and threshold:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

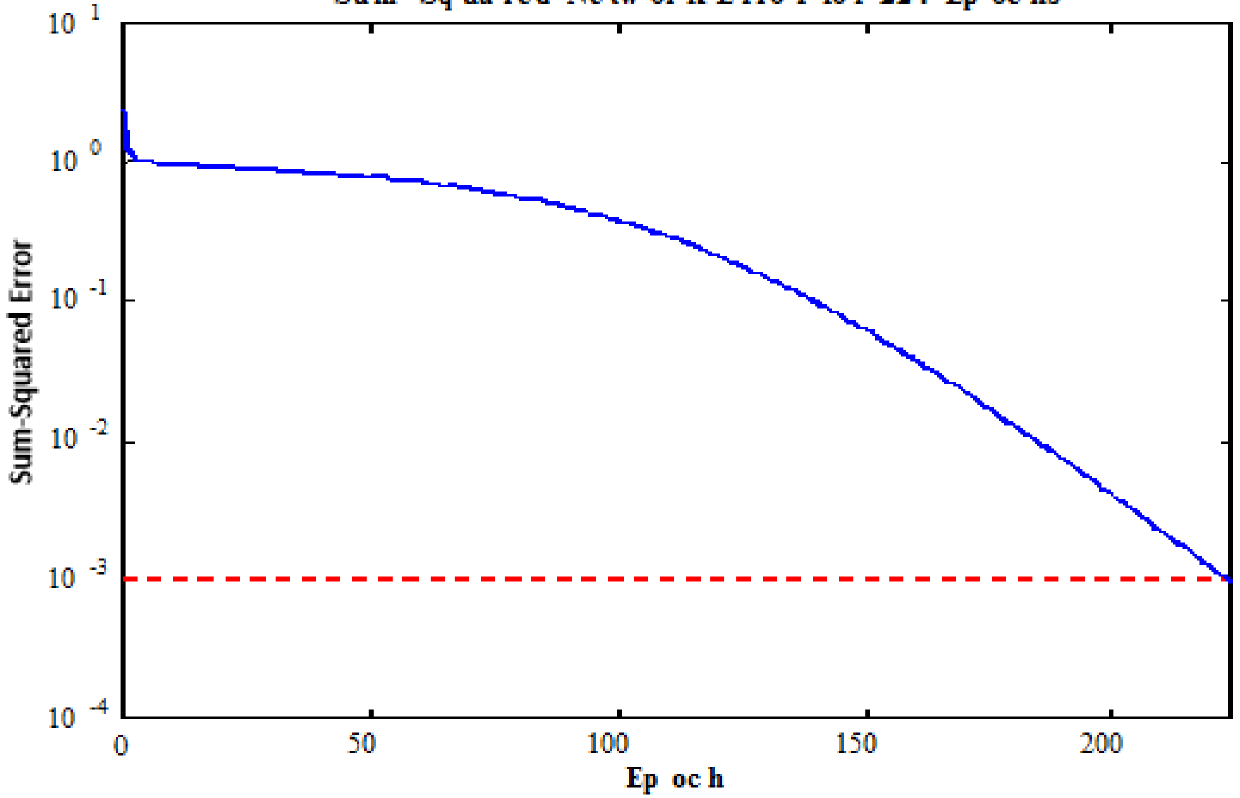
$$\theta_3 = \theta_3 + \Delta\theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta\theta_4 = -0.1 + 0.0015 = -0.0985$$

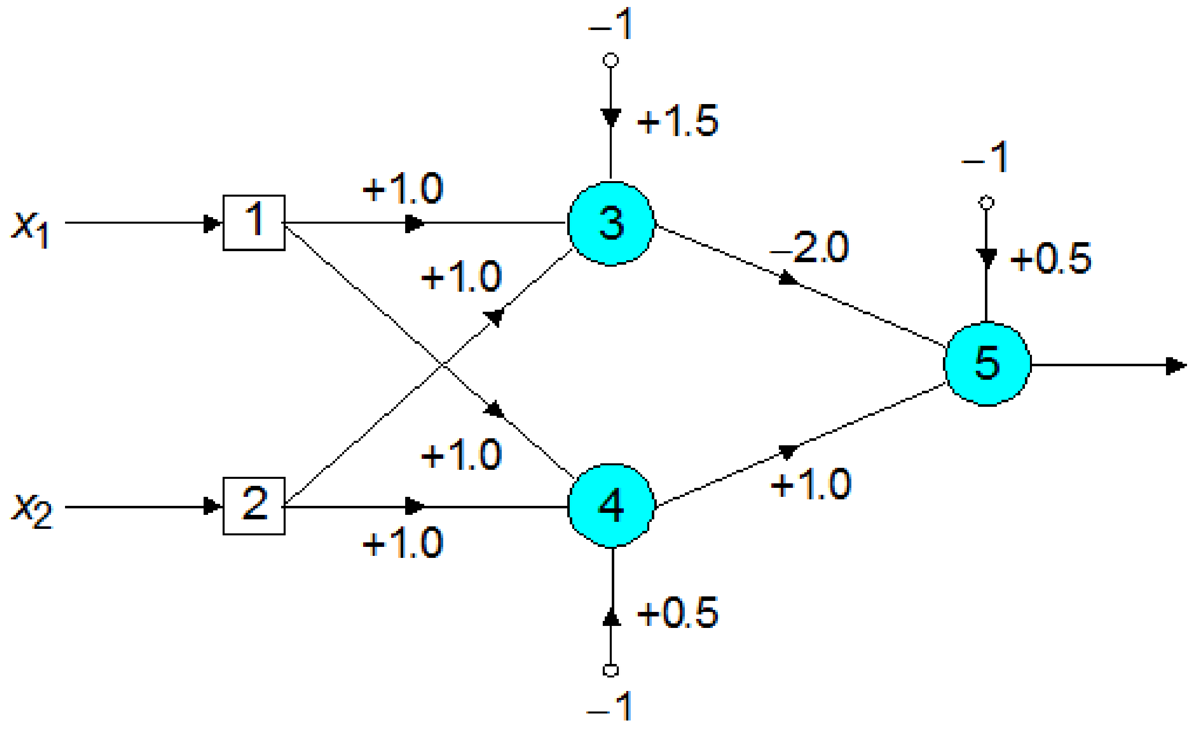
$$\theta_5 = \theta_5 + \Delta\theta_5 = 0.3 + 0.0127 = 0.3127$$

- **The training process is repeated until the sum of squared errors is less than 0.001.**

Learning curve for operation *Exclusive-OR*



Network represented by McCulloch-Pitts model for solving the *Exclusive-OR* operation



Final results of three-layer network learning

Inputs		Desired output	Actual output	Error	Sum of squared errors
x_1	x_2	y_d	y_5	e	
1	1	0	0.0155	−0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	−0.0175	

Hebbian learning

- In 1949, Donald Hebb proposed one of the key ideas in biological learning, commonly known as Hebb's Law. Hebb's Law provides the basis for learning without a teacher.
- Hebb's Law states that **"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B is increased"**
- Hebb's Law can be represented in the form of two rules:

1. If two neurons on either side of a connection are activated synchronously, then the weight of that connection is increased.
2. If two neurons on either side of a connection are activated asynchronously, then the weight of that connection is decreased.

Hebbian learning algorithm

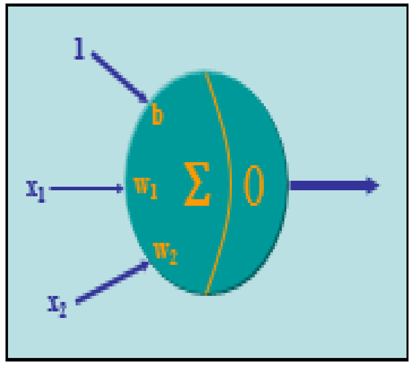
- **Step 0**: initialize all weights to 0
- **Step 1**: Given a training input “s”, with its target output “t”, set the activations of the input units: $x_i = s_i$
- **Step 2**: Set the activation of the output unit to the target value: $y = t$
- **Step 3**: Adjust the weights: $w_i(new) = w_i(old) + x_i y$
- **Step 4**: Adjust the bias (just like the weights): $b(new) = b(old) + y$

Example

- **PROBLEM**:
  Construct a Hebb Net which performs like an AND function, that is, only when both features are "active" will the data be in the target class.

x1	x2	bias	Target
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

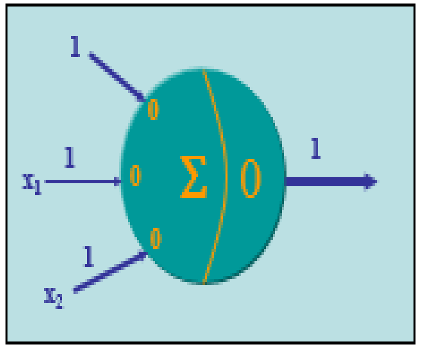
- **TRAINING SET** (with the bias input always at 1):



Training – First Input

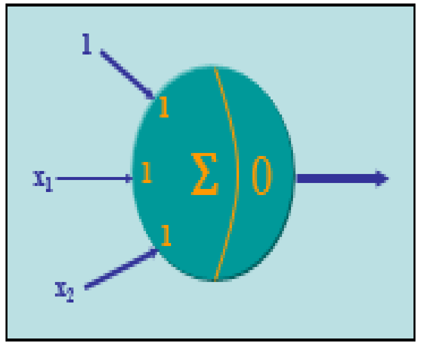
- Initialize the weights to 0

Present the first input (1 1 1) with a target of 1



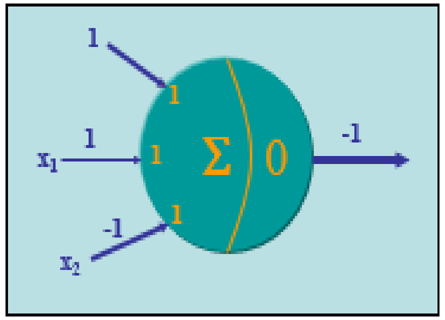
Update the weights:

$$w_1(\text{new}) = w_1(\text{old}) + x_1 t$$
$$= \quad 0 + 1 = 1$$
$$w_2(\text{new}) = w_2(\text{old}) + x_2 t$$
$$= \quad 0 + 1 = 1$$
$$b(\text{new}) \quad = b(\text{old}) + t$$
$$= \quad 0 + 1 = 1$$



Training – Second Input

- Present the second input: (1 -1 1) with a target of -1



Update the weights:

$w_1(\text{new}) = w_1(\text{old}) + x_1 t$

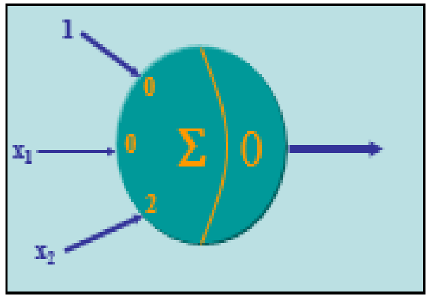
$= \quad 1 + 1(-1) = 0$

$w_2(\text{new}) = w_2(\text{old}) + x_2 t$

$= \quad 1 + (-1)(-1) = 2$

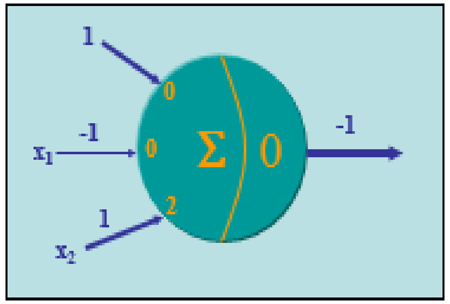
$b(\text{new}) \quad = b(\text{old}) + t$

$= \quad 1 + (-1) = 0$



Training – Third Input

- Present the third input: (-1 1 1) with a target of -1



Update the weights:

$w_1(\text{new}) = w_1(\text{old}) + x_1 t$

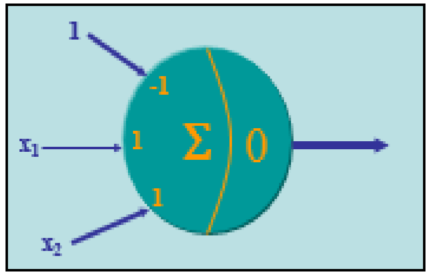
$= \quad 0 + (-1)(-1) = 1$

$w_2(\text{new}) = w_2(\text{old}) + x_2 t$

$= \quad 2 + 1(-1) = 1$

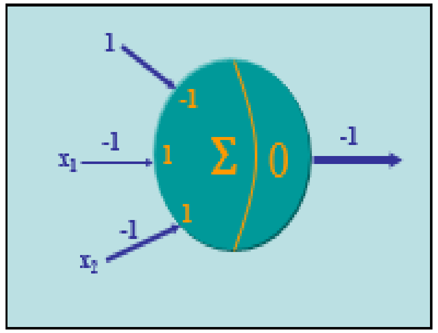
$b(\text{new}) \quad = b(\text{old}) + t$

$= \quad 0 + (-1) = -1$



Training – Fourth Input

- Present the fourth input: (-1 -1 1) with a target of -1



Update the weights:

$w_1(\text{new}) = w_1(\text{old}) + x_1 t$

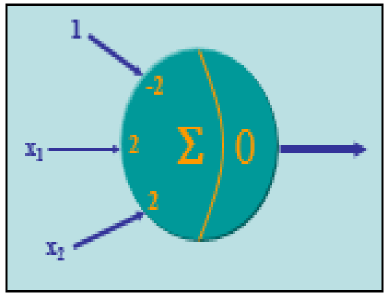
$= \quad 1 + (-1)(-1) = 2$

$w_2(\text{new}) = w_2(\text{old}) + x_2 t$

$= \quad 1 + (-1)(-1) = 2$

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

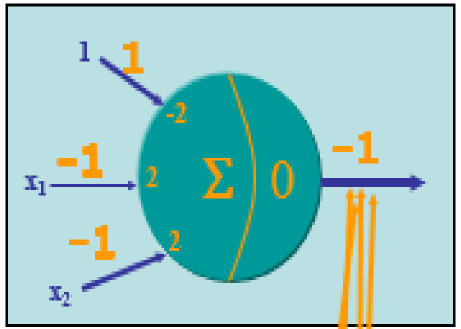
$= \quad -1 + (-1) = -2$



Final Neuron

- This neuron works:

x1	x2	bias	Target
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1



$1*2 + 1*2 + 1*(-2) = 2 > 0$

$(-1)*2 + 1*2 + 1*(-2) = -2 < 0$

$1*2 + (-1)*2 + 1*(-2) = -2 < 0$

$(-1)*2 + (-1)*2 + 1*(-2) = -6 < 0$

Adaline Network

- **ADALINE: Adaptive Linear Neuron** or later **Adaptive Linear Element**
- It was developed by Professor *Bernard Widrow* and his graduate student *Ted Hoff* at Stanford University in 1960.
- It is based on the McCulloch–Pitts neuron. It consists of a weight, a bias and a summation function.
- The difference between Adaline and the perceptron is that in the learning phase the weights are adjusted according to the weighted sum of the inputs. In the standard perceptron, the net is passed to the activation function and the function's output is used for adjusting the weights.
- Variation on the Perceptron Network
  - inputs are +1 or -1
  - outputs are +1 or -1
  - uses a bias input
- It is trained using the Delta Rule which is also known as the least mean squares (LMS) or Widrow-Hoff rule. The activation function, during training is the identity function. After training the activation is a threshold function.

Algorithm of Adaline Network

- Step 0: initialize the weights to small random values and select a learning rate, α
- Step 1: for each input vector “s”, with target output “t”; set the inputs to s
- Step 2: compute the neuron inputs

Neuron input

$$\mathbf{y\_in = b + \sum x_i w_i}$$

- Step 3: use the delta rule to update the bias and weights

Delta rule

$$\mathbf{b(new) = b(old) + \alpha(t - y\_in)}$$
$$\mathbf{w_i(new) = w_i(old) + \alpha(t - y\_in)x_i}$$

- Step 4: stop if the largest weight change across all the training samples is less than a specified tolerance, otherwise cycle through the training set again

The Learning Rate, α

- The performance of an ADALINE neuron depends heavily on the choice of the learning rate
  - if it is too large the system will not converge
  - if it is too small the convergence will take to long
- Typically, α is selected by trial and error
  - typical range: $0.01 < \alpha < 10.0$
  - often start at 0.1
  - sometimes it is suggested that:
    $0.1 < n\,\alpha < 1.0$
    where n is the number of inputs
- **Example:** Construct an AND function for a ADALINE neuron with $\alpha=0.1$.

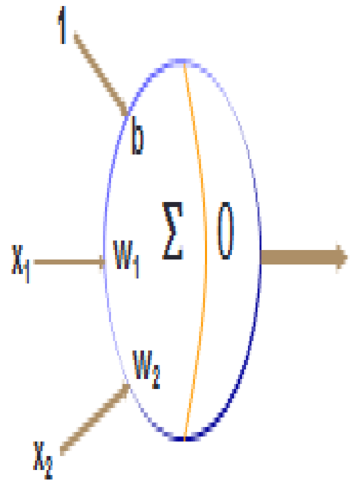
Neuron input

$$\mathbf{y\_in = b + \sum x_i w_i}$$

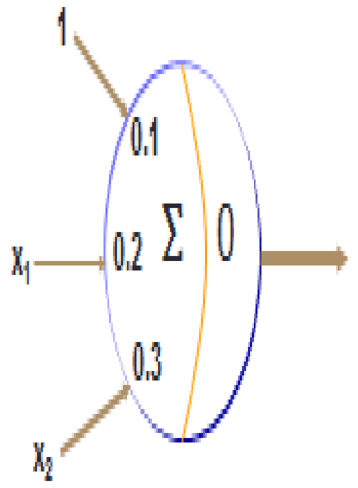
Activation Function

$$\mathbf{y = \begin{cases} 1 \text{ if } y\_in >= 0 \\ -1 \text{ if } y\_in < 0 \end{cases}}$$

x1	x2	bias	Target
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

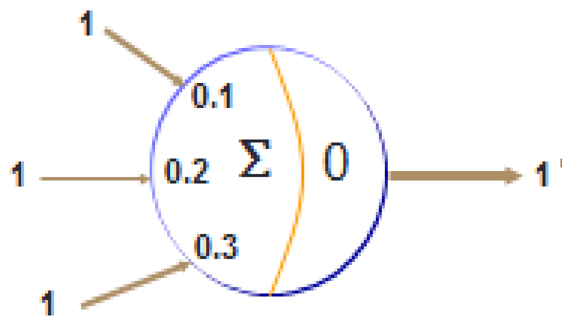


Initial Conditions: Set the weights to small random values:



First Training Run

- Apply the input (1,1) with output 1



Neuron input

$$y\_in = b + \sum x_i w_i$$

The net input is:

$$y\_in = 0.1 + 0.2*1 + 0.3*1 = 0.6$$

Delta rule

$$b(new) = b(old) + \alpha(t - y\_in)$$
$$w_i(new) = w_i(old) + \alpha(t - y\_in)x_i$$

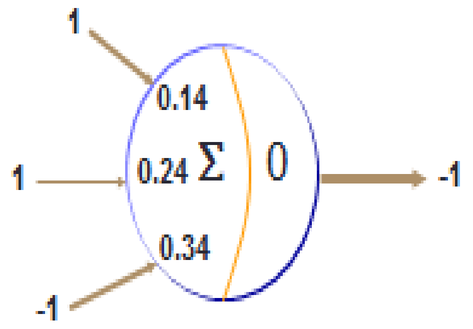
The new weights are:

$$b = 0.1 + 0.1(1-0.6) = 0.14$$
$$w_1 = 0.2 + 0.1(1-0.6)1 = 0.24$$
$$w_2 = 0.3 + 0.1(1-0.6)1 = 0.34$$

The largest weight change is 0.04

Second Training Run

- Apply the second training set (1 -1) with output -1



The net input is:

$y\_in = 0.14 + 0.24*1 + 0.34*(-1) = 0.04$

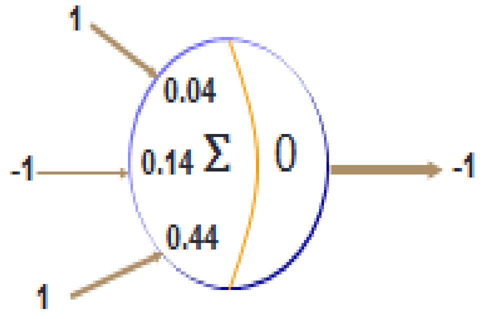
The new weights are:

$b = 0.14 - 0.1(1+0.04) = 0.04$
$w_1 = 0.24 - 0.1(1+0.04)1 = 0.14$
$w_2 = 0.34 + 0.1(1+0.04)1 = 0.44$

The largest weight change is 0.1

Third Training Run

- Apply the third training set (-1 1) with output -1



The net input is:

$y\_in = 0.04 - 0.14*1 + 0.44*1 = 0.34$

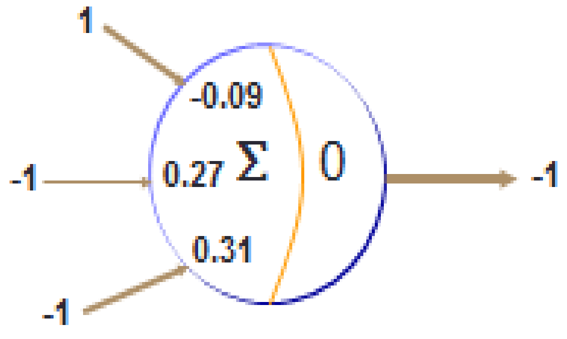
The new weights are:

$b = 0.04 - 0.1(1+0.34) = -0.09$
$w_1 = 0.14 + 0.1(1+0.34)1 = 0.27$
$w_2 = 0.44 - 0.1(1+0.34)1 = 0.31$

The largest weight change is 0.13

Fourth Training Run

- Apply the fourth training set (-1 -1) with output -1



The net input is:

y_in = -0.09 - 0.27*1 - 0.31*1 = -0.67

The new weights are:

$b = -0.09 - 0.1(1+0.67) = -0.27$
$w_1 = 0.27 + 0.1(1+0.67)1 = 0.43$
$w_2 = 0.31 + 0.1(1+0.67)1 = 0.47$

The largest weight change is 0.16

Continue to cycle through the four training inputs until the largest change in the weights over a complete cycle is less than some small number (say 0.01)

In this case, the solution becomes:
$b = -0.5$,
$w_1 = 0.5$,
$w_2 = 0.5$

Hopfield Neural Network

- A Hopfield network is a form of recurrent artificial neural network popularized by John Hopfield in 1982, but described earlier by Little in 1974.
- A recurrent neural network has feedback loops from its outputs to its inputs. The presence of such loops has a profound impact on the learning capability of the network.

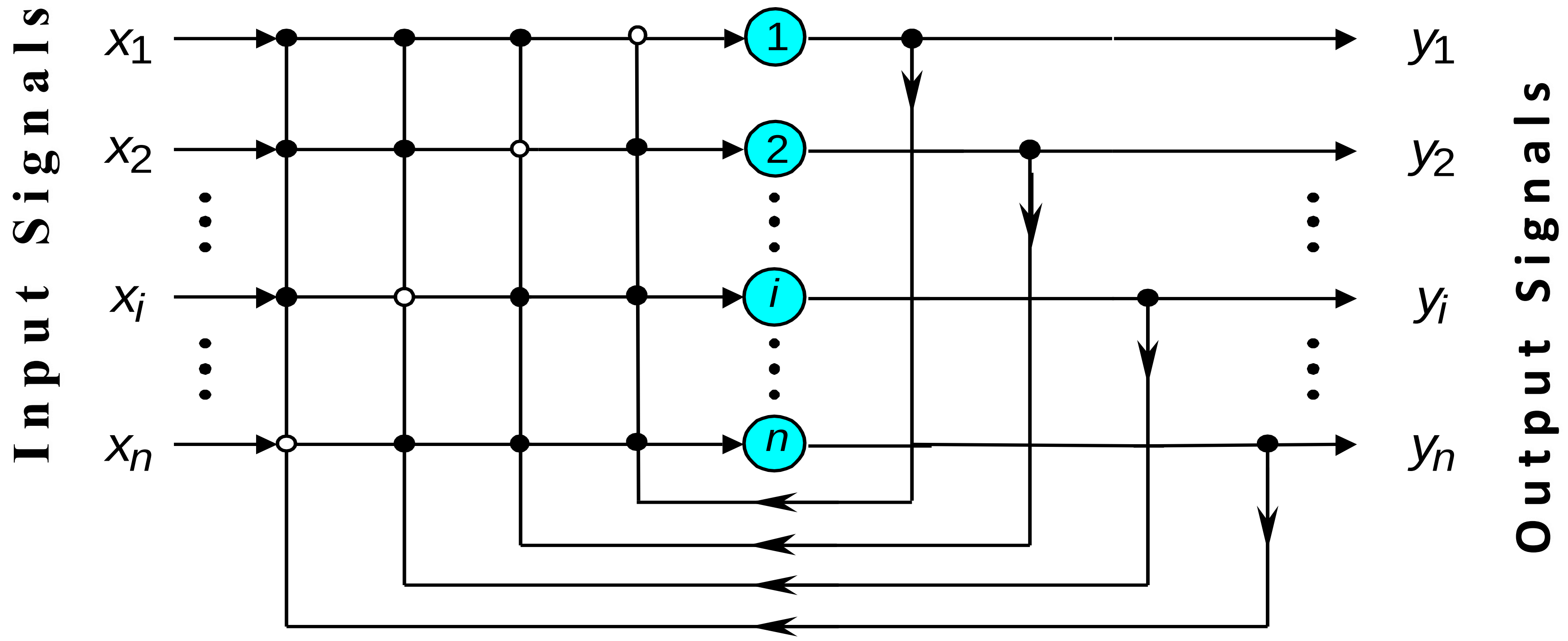


Figure: Single-layer *n*-neuron Hopfield network

Hopfield Neural Network

- The Hopfield network uses McCulloch and Pitts neurons with the **sign activation function** as its computing element.
- The current state of the Hopfield network is determined by the current outputs of all neurons, $y_1, y_2, \ldots, y_n$. Thus, for a single-layer n-neuron network, the state can be defined by the ***state vector.***

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \qquad Y^{sign} = \begin{cases} +1, & \text{if } X > 0 \\ -1, & \text{if } X < 0 \\ Y, & \text{if } X = 0 \end{cases} \qquad \mathbf{W} = \sum_{m=1}^{M} \mathbf{Y}_m \mathbf{Y}_m^T - M\,\mathbf{I}$$

state vector ***sign activation*** **function** ***matrix form***

- In the Hopfield network, synaptic weights between neurons are usually represented in ***matrix form*** as shown in above.
- where ***M*** is the number of states to be memorized by the network, $\mathbf{Y}_m$ is the n-dimensional binary vector, $\mathbf{I}$ is $n \times n$ identity matrix, and superscript ***T*** denotes matrix transposition.

Hopfield Neural Network

- The stable state-vertex is determined by the weight matrix **W**, the current input vector **X**, and the threshold matrix θ. If the input vector is partially incorrect or incomplete, the initial state will converge into the stable state-vertex after a few iterations.

- Suppose, for instance, that our network is ***required to memorize two opposite states, (1, 1, 1) and (-1, -1, -1).*** Thus,

$$\mathbf{Y}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{Y}_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} \quad \textbf{or} \quad \mathbf{Y}_1^T = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \quad \mathbf{Y}_2^T = \begin{bmatrix} -1 & -1 & -1 \end{bmatrix}$$

where $\mathbf{Y}_1$ and $\mathbf{Y}_2$ are the three-dimensional vectors.

- The 3×3 identity matrix **I** is

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Hopfield Neural Network

- Thus, we can now determine the weight matrix as follows:

$$\mathbf{W}=\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1 & 1 & 1\end{bmatrix}+\begin{bmatrix}-1\\-1\\-1\end{bmatrix}\begin{bmatrix}-1 & -1 & -1\end{bmatrix}-2\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}=\begin{bmatrix}0&2&2\\2&0&2\\2&2&0\end{bmatrix}$$

2: is *required to memorize two opposite states.*

- Next, the network is tested by the sequence of input vectors, $\mathbf{X}_1$ and $\mathbf{X}_2$, which are equal to the output (or target) vectors $\mathbf{Y}_1$ and $\mathbf{Y}_2$, respectively.

- First, we activate the Hopfield network by applying the input vector **X**. Then, we calculate the actual output vector **Y**, and finally, we compare the result with the initial input vector **X**.

$$\mathbf{Y}_1=sign\left\{\begin{bmatrix}0&2&2\\2&0&2\\2&2&0\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}-\begin{bmatrix}0\\0\\0\end{bmatrix}\right\}=\begin{bmatrix}1\\1\\1\end{bmatrix}$$

$$\mathbf{Y}_2=sign\left\{\begin{bmatrix}0&2&2\\2&0&2\\2&2&0\end{bmatrix}\begin{bmatrix}-1\\-1\\-1\end{bmatrix}-\begin{bmatrix}0\\0\\0\end{bmatrix}\right\}=\begin{bmatrix}-1\\-1\\-1\end{bmatrix}$$

Hopfield Neural Network

- The remaining six states are all unstable. However, stable states (also called **fundamental memories**) are capable of attracting states that are close to them. The fundamental memory (1, 1, 1) attracts unstable states (–1, 1, 1), (1, –1, 1) and (1, 1, –1). Each of these unstable states represents a single error, compared to the fundamental memory (1, 1, 1). The fundamental memory (–1, –1, –1) attracts unstable states (–1, –1, 1), (–1, 1, –1) and (1, –1, –1).
- Thus, the Hopfield network can act as an **error correction network**.

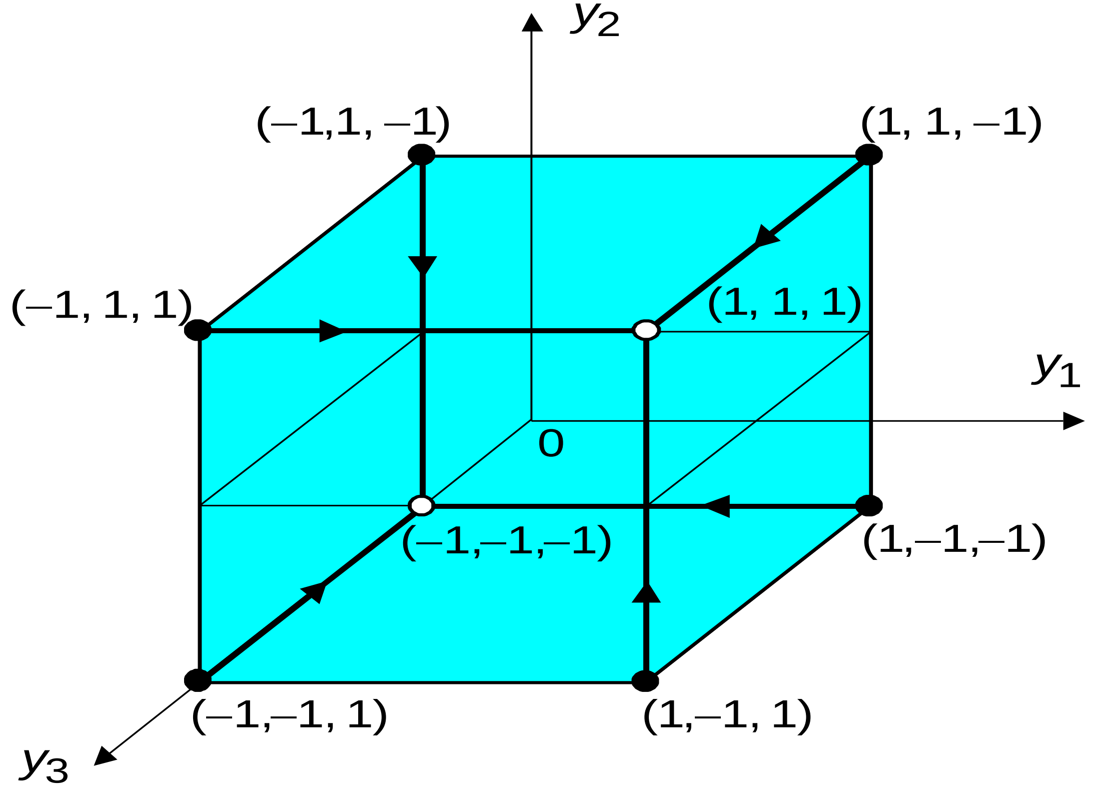


Figure: Possible states for the three-neuron Hopfield network

The Kohonen network

- The Kohonen model provides a topological mapping. It places a fixed number of input patterns from the input layer into a higher-dimensional output or Kohonen layer.
- Training in the Kohonen network begins with the winner's neighbourhood of a fairly large size. Then, as training proceeds, the neighbourhood size gradually decreases.

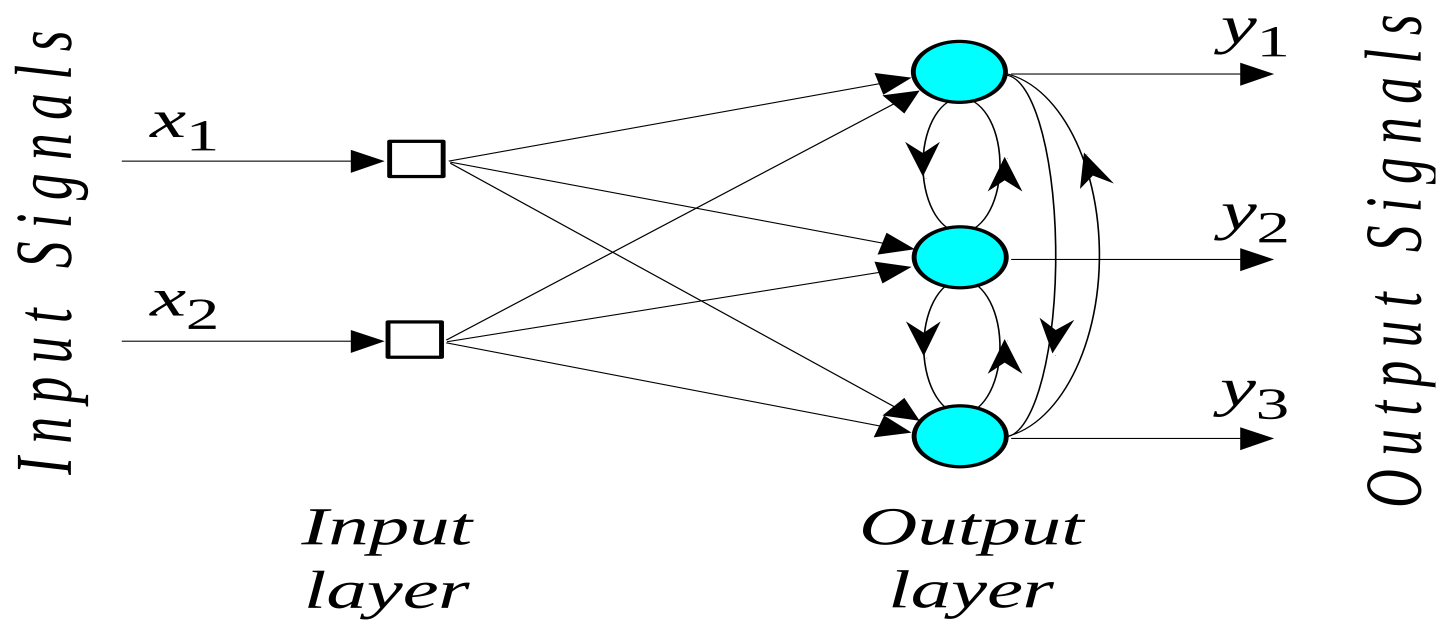


Figure: Architecture of the Kohonen Network

The Kohonen network

- The lateral connections are used to create a competition between neurons. The neuron with the largest activation level among all neurons in the output layer becomes the winner. This neuron is the only neuron that produces an output signal. The activity of all other neurons is suppressed in the competition.
- In the Kohonen network, a neuron learns by shifting its weights from inactive connections to active ones. Only the winning neuron and its neighbourhood are allowed to learn. If a neuron does not respond to a given input pattern, then learning cannot occur in that particular neuron.
- The **competitive learning rule** defines the change Δw_{ij} applied to synaptic weight w_{ij} as

$$\Delta w_{ij} = \begin{cases} \alpha\,(x_i - w_{ij}), & \text{if neuron } j \text{ wins the competition} \\ 0, & \text{if neuron } j \text{ loses the competition} \end{cases}$$

where x_i is the input signal and α is the learning rate parameter.

The Kohonen network

- The overall effect of the competitive learning rule resides in moving the synaptic weight vector $\mathbf{W}_j$ of the winning neuron j towards the input pattern $\mathbf{X}$. The matching criterion is equivalent to the minimum **Euclidean distance** between vectors.
- The Euclidean distance between a pair of n-by-1 vectors $\mathbf{X}$ and $\mathbf{W}_j$ is defined by

$$d = \left\| \mathbf{X} - \mathbf{W}_j \right\| = \left[\sum_{i=1}^{n} (x_i - w_{ij})^2 \right]^{1/2}$$

where x_i and w_{ij} are the ith elements of the vectors $\mathbf{X}$ and $\mathbf{W}_j$, respectively.

- To identify the winning neuron, jX, that best matches the input vector X, we may apply the following condition:

$$j_{\mathbf{X}} = \min_{j} \left\| \mathbf{X} - \mathbf{W}_j \right\|, \qquad j = 1, 2, \ldots, m$$

where **m** is the number of neurons in the Kohonen layer.

The Kohonen network

- Suppose, for instance, that the 2-dimensional input vector **X** is presented to the three-neuron Kohonen network,

$$\mathbf{X} = \begin{bmatrix} 0.52 \\ 0.12 \end{bmatrix}$$

- The initial weight vectors, $\mathbf{W}_j$, are given by

$$\mathbf{W}_1 = \begin{bmatrix} 0.27 \\ 0.81 \end{bmatrix} \qquad \mathbf{W}_2 = \begin{bmatrix} 0.42 \\ 0.70 \end{bmatrix} \qquad \mathbf{W}_3 = \begin{bmatrix} 0.43 \\ 0.21 \end{bmatrix}$$

- We find the winning (best-matching) neuron $j_{\mathbf{X}}$ using the minimum-distance Euclidean criterion:

$$d_1 = \sqrt{(x_1 - w_{11})^2 + (x_2 - w_{21})^2} = \sqrt{(0.52 - 0.27)^2 + (0.12 - 0.81)^2} = 0.73$$

$$d_2 = \sqrt{(x_1 - w_{12})^2 + (x_2 - w_{22})^2} = \sqrt{(0.52 - 0.42)^2 + (0.12 - 0.70)^2} = 0.59$$

$$d_3 = \sqrt{(x_1 - w_{13})^2 + (x_2 - w_{23})^2} = \sqrt{(0.52 - 0.43)^2 + (0.12 - 0.21)^2} = 0.13$$

The Kohonen network

- Neuron 3 is the winner and its weight vector $\mathbf{W}_3$ is updated according to the competitive learning rule.

$$\Delta w_{13} = \alpha\,(x_1 - w_{13}) = 0.1\,(0.52 - 0.43) = 0.01$$

$$\Delta w_{23} = \alpha\,(x_2 - w_{23}) = 0.1\,(0.12 - 0.21) = -0.01$$

- The updated weight vector $\mathbf{W}_3$ at iteration (p + 1) is determined as:

$$\mathbf{W}_3(p+1) = \mathbf{W}_3(p) + \Delta\mathbf{W}_3(p) = \begin{bmatrix} 0.43 \\ 0.21 \end{bmatrix} + \begin{bmatrix} 0.01 \\ -0.01 \end{bmatrix} = \begin{bmatrix} 0.44 \\ 0.20 \end{bmatrix}$$

- The weight vector $\mathbf{W}_3$ of the wining neuron 3 becomes closer to the input vector **X** with each iteration.