

Introduction to Neural Networks

Vijay Khare
2013

Vijay Khare

Chapter 1

Introduction

1. Knowledge is acquired by the network from its environment through the learning process.
2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

Introduction

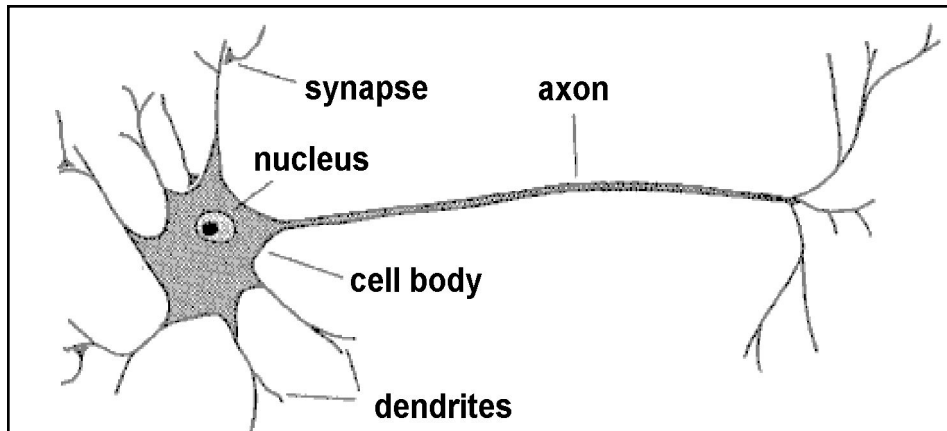
● Why ANN

- Some tasks can be done easily (effortlessly) by humans but are hard by conventional paradigms on Von Neumann machine with algorithmic approach
 - Pattern recognition (old friends, hand-written characters)
 - Content addressable recall
 - Approximate, common sense reasoning (driving, playing piano, baseball player)
- These tasks are often ill-defined, experience based, hard to apply logic

Artificial Neural network

- Inspired from human brain , AS human is complex nonlinear processor and have million of neuron and trillion of interconnection of neuron.
 - 1.Knowledge is acquire by the brain from its environment through the learning process.
 2. Interneuron connection strengths, know as synaptic weights are used to store the acquired knowledge.

Biological neuron

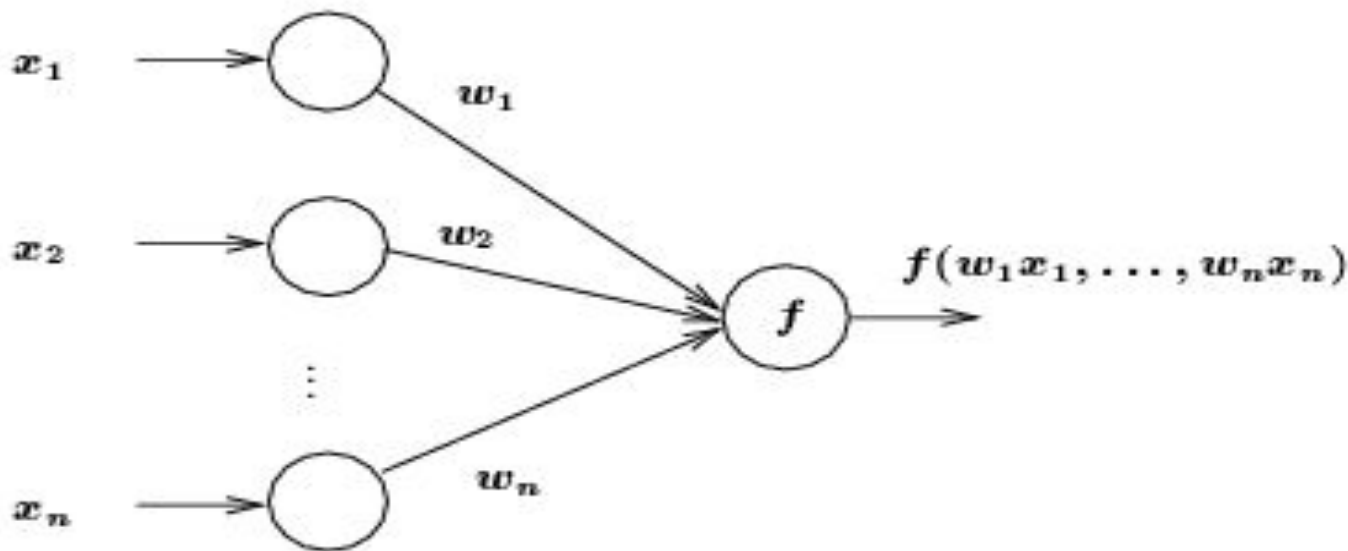


- A neuron has
 - A branching input (dendrites)
 - A branching output (the axon)
- The information circulates from the dendrites to the axon via the cell body
- Axon connects to dendrites via synapses
 - Synapses vary in strength
 - Synapses may be excitatory or inhibitory

ANN

- Artificial neural networks nonlinear information processing devices which are built from interconnection elementary processing unit called neuron.
- Artificial neural network is information processing device which is inspired from human brain . The key elements is the novel structure of information processing system.
- It is composed of large number of highly interconnection processing elements called neuron working in union to solve the specific problem.

Artificial neural network is information processing device. In this information processing system the element called as neuron process the information .the signal is transmitted by mean of connection link .



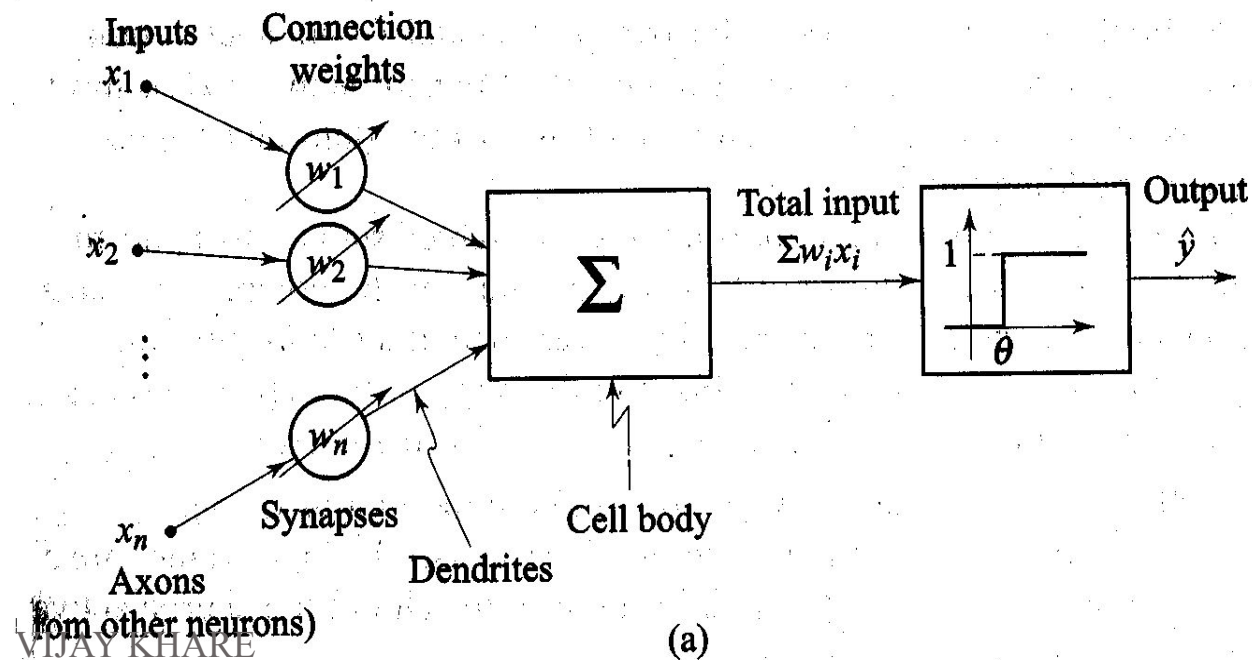
Neuron is characterized by :

1. Architecture (connection between the neuron)
- 2 Training or learning (determine weights on the connection)
3. Activation function

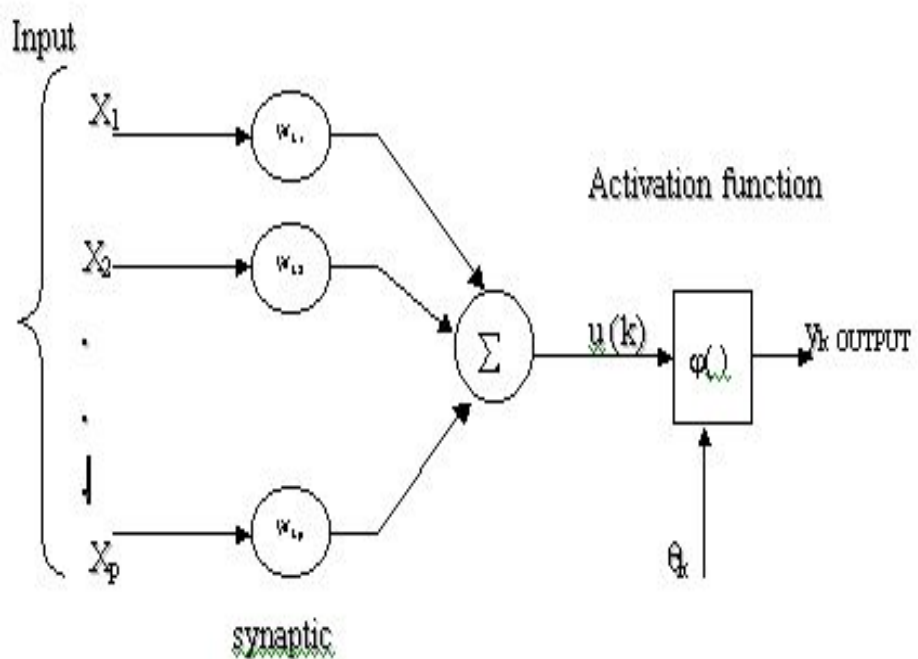
Introduction

- **What is an (artificial) neural network**
 - A set of **nodes** (units, neurons, processing elements)
 - Each node has input and output
 - Each node performs a simple computation by its **node function**
 - **Weighted connections** between nodes
 - Connectivity gives the structure/architecture of the net
 - What can be computed by a NN is primarily determined by the connections and their weights
 - A very much simplified version of networks of neurons in animal nerve systems

Similarities of human brain and neuron model



Neuron Model

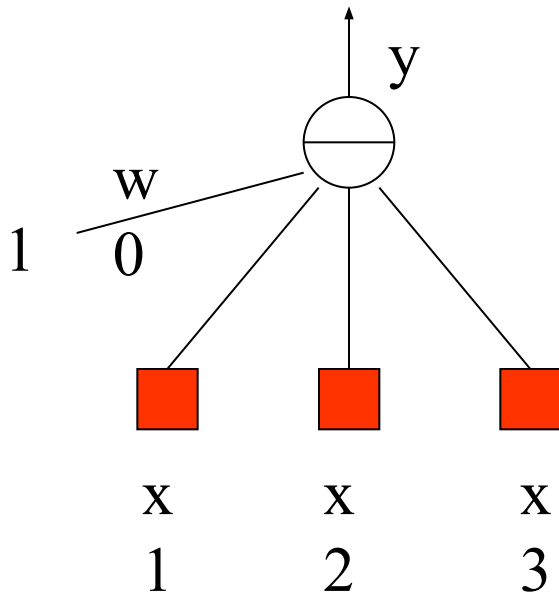


$$u_k = \sum_{j=1}^m w_{kj} x_j$$

Mc-Culloch and Pitts Model

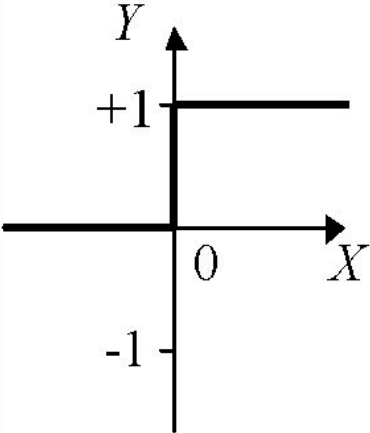
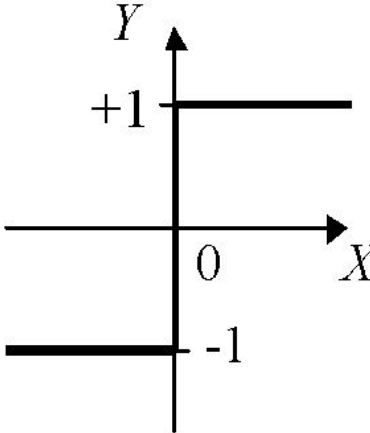
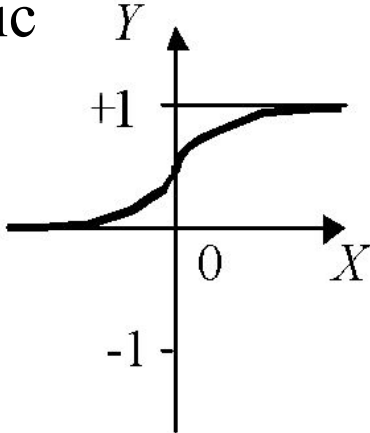
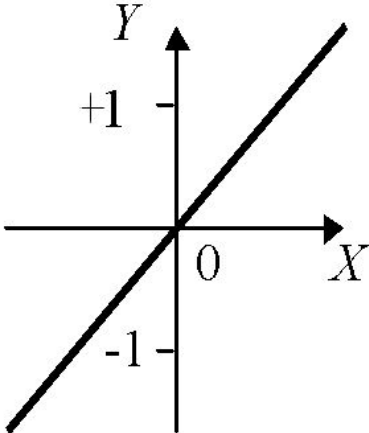
What is an artificial neuron

- Bias



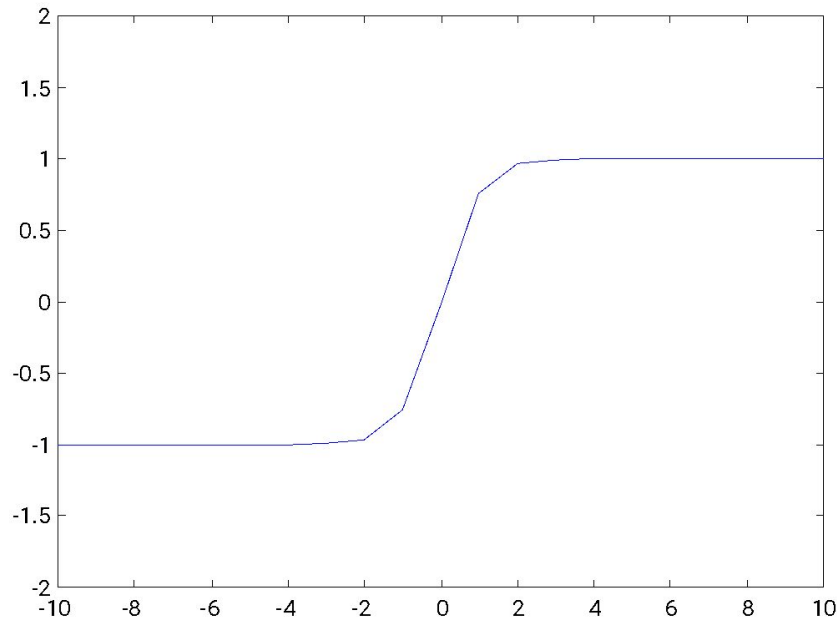
$$y = f\left(w_0 + \sum_{i=1}^{n-1} w_i x_i\right)$$

Activation functions of a neuron

Step function	Sign function	Sigmoid function Logistic	Linear function
			
$y^{step} = \begin{cases} 1, & \text{if } X > 0 \\ 0, & \text{if } X \leq 0 \end{cases}$	$y^{sign} = \begin{cases} 1, & \text{if } X > 0 \\ -1, & \text{if } X < 0 \end{cases}$	$y^{sigmoid} = \frac{1}{1 + e^{-X}}$	$y^{linear} = X$

Bipolar activation function=
Hyperbolic
tangent

$$o_k = 2 \left(\frac{1}{1 + \exp(-net_k)} - \frac{1}{2} \right)$$

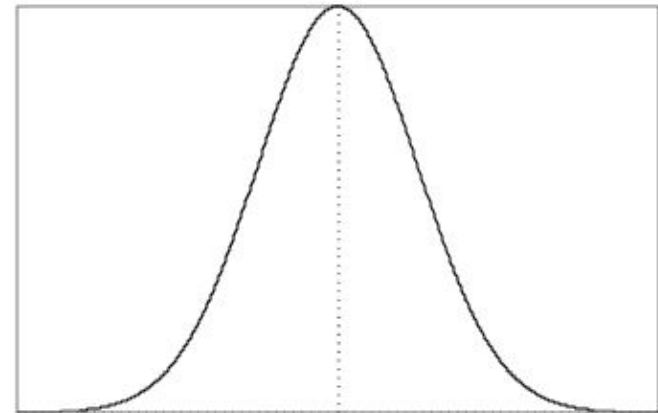


Node Function

- **Gaussian function**

- Bell-shaped (radial basis)
- Continuous
- $f(\text{net})$ asymptotically approaches 0 (or some constant) when $|\text{net}|$ is large
- Single maximum (when $\text{net} = \mu$)
- Example:

$$f(\text{net}) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left[-\frac{1}{2} \left(\frac{\text{net} - \mu}{\sigma} \right)^2 \right]$$



Gaussian function

Introduction

Von Neumann machine

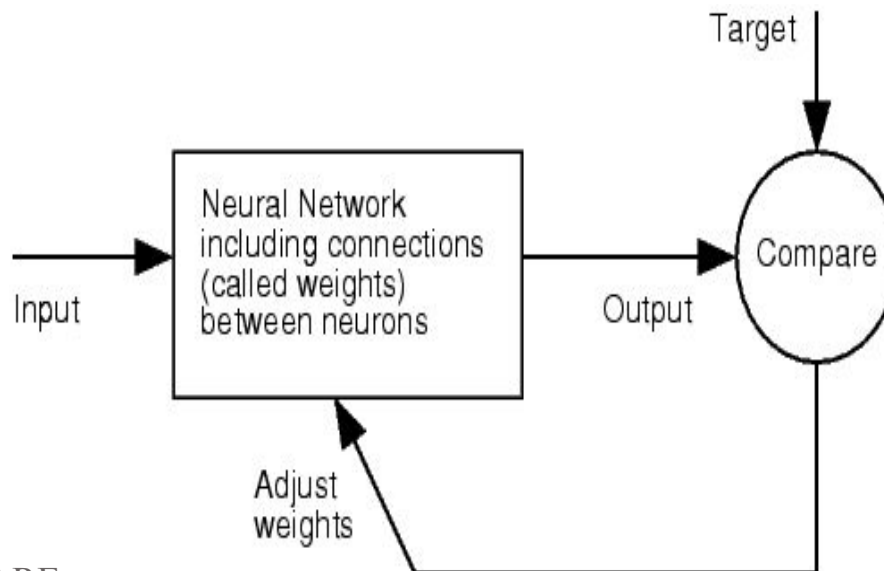
- One or a few high speed (ns) processors with considerable computing power
- One or a few shared high speed buses for communication
- Sequential memory access by address
- Problem-solving knowledge is separated from the computing component

Human Brain

- Large # (10^{11}) of low speed processors (ms) with limited computing power
- Large # (10^{15}) of low speed connections
- Content addressable recall (CAM)
- Problem-solving knowledge resides in the connectivity of neurons
- Adaptation by changing the connectivity

Neural Network

- Massively parallel distributed processor.
- Neural networks are composed to artificial neurons.
- Neural network can be trained to performed particular task by adjusting their weights.

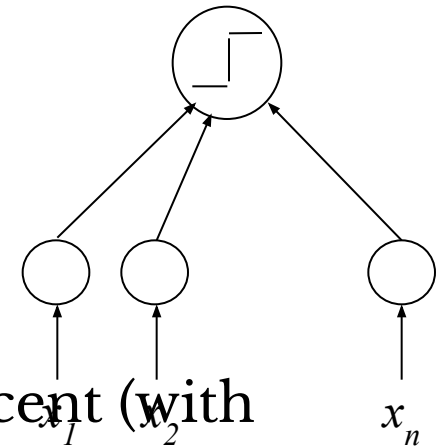


History of NN

- **Pitts & McCulloch (1943)**
 - First mathematical model of biological neurons
 - All Boolean operations can be implemented by these neuron-like nodes (with different threshold and excitatory/inhibitory connections).
 - Origin of automata theory.
- **Hebb (1949)**
 - Hebbian rule of learning: increase the connection strength between neurons i and j whenever both i and j are activated.

History of NN

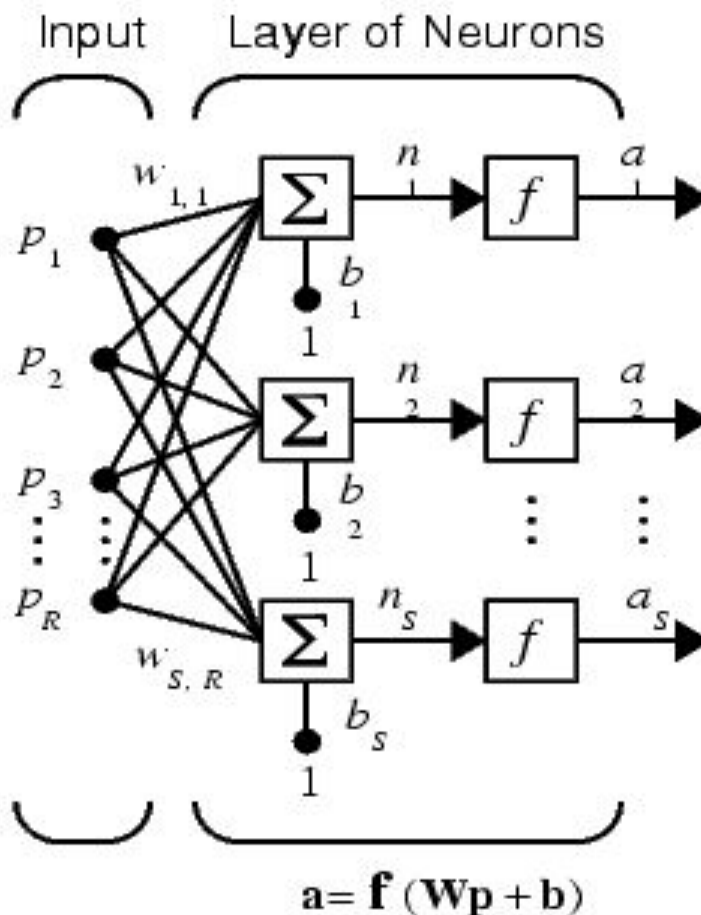
- **Early booming (50's – early 60's)**
 - Rosenblatt (1958)
 - Perceptron: network of threshold nodes for pattern classification
 - Widrow and Hoff (1960)
 - Learning rule based on gradient descent (with differentiable unit)
- **The setback (mid 60's – late 70's)**
 - Serious problems with perceptron model (Minsky's book 1969)
 - Single layer perceptrons cannot represent (learn) simple functions such as XOR.
- **Renewed enthusiasm and flourish (80's – present)**
 - New techniques
 - Backpropagation learning for multi-layer feed forward nets (with non-linear, differentiable node functions).
 - Impressive application (character recognition, speech recognition, text-to-speech)



Network architecture

- Single layer network
- Multilayer network
- Recurrent network

Single layer Neural Network

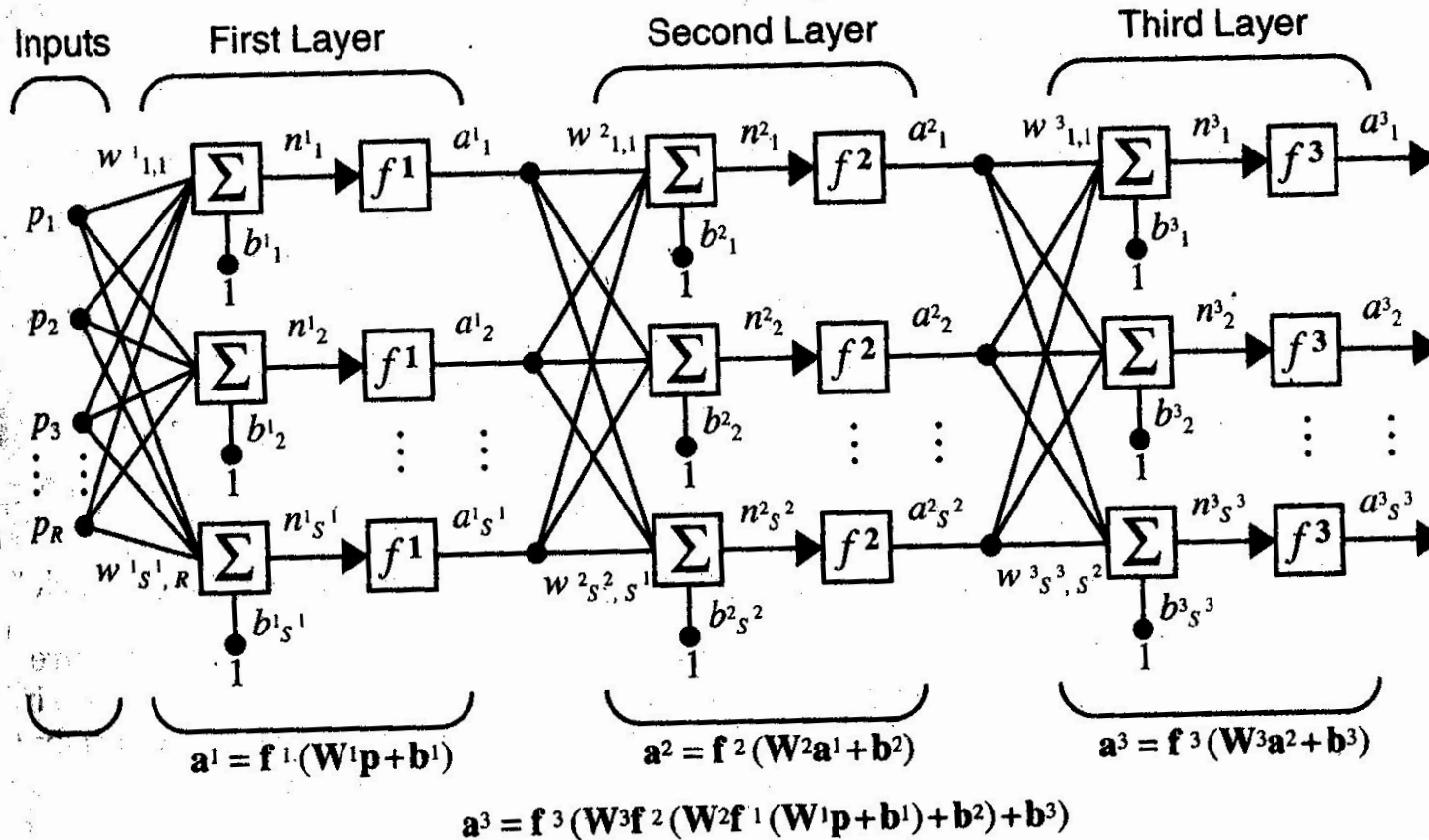


Where...

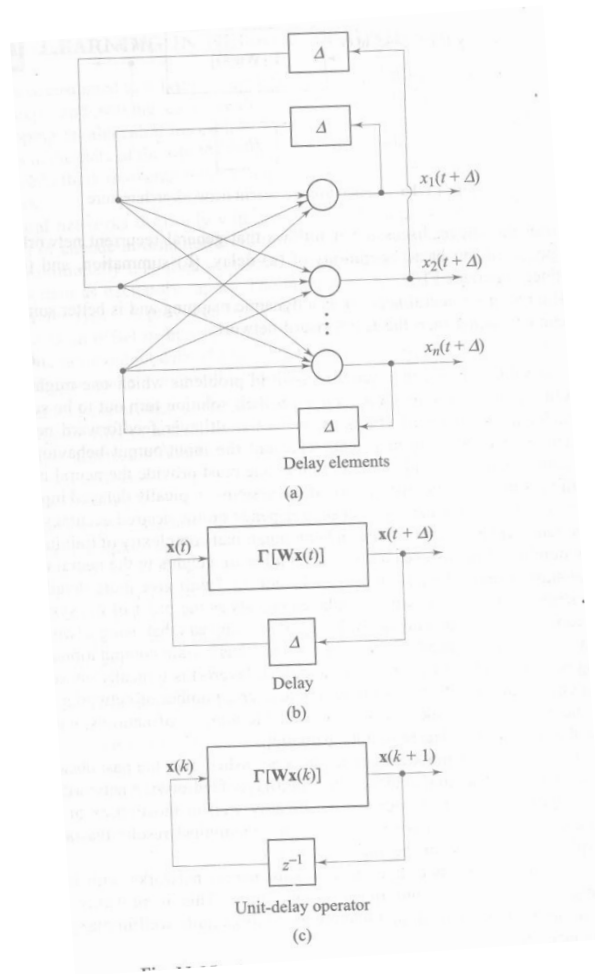
R = number of
elements in
input vector

S = number of
neurons in layer

Multilayer Neural Network

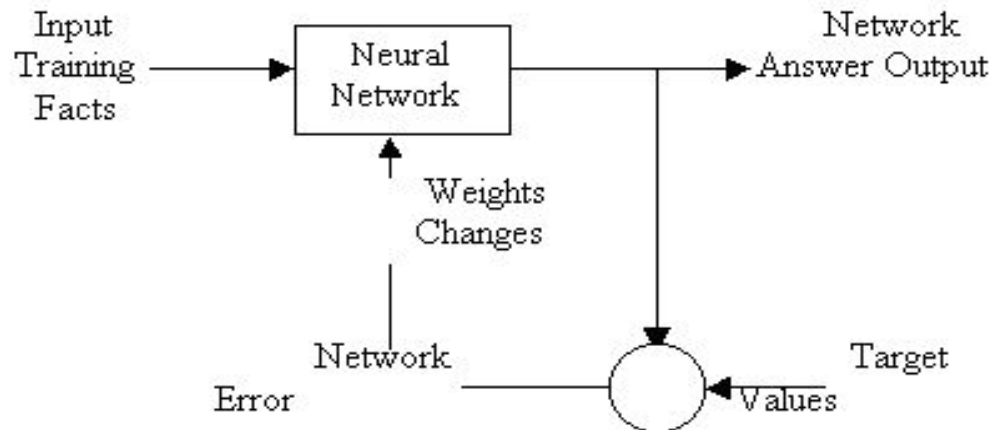


Recurrent Neural Network



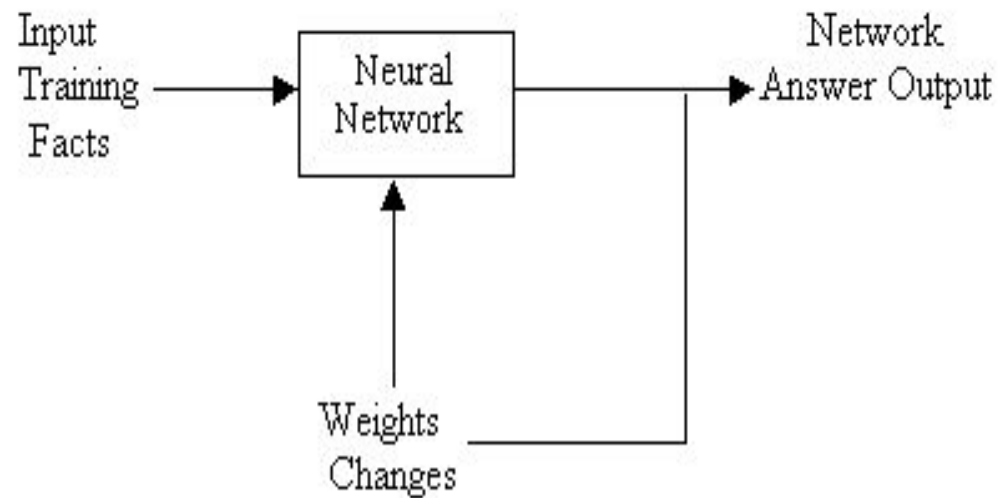
Learning paradigm

- Supervised learning rule



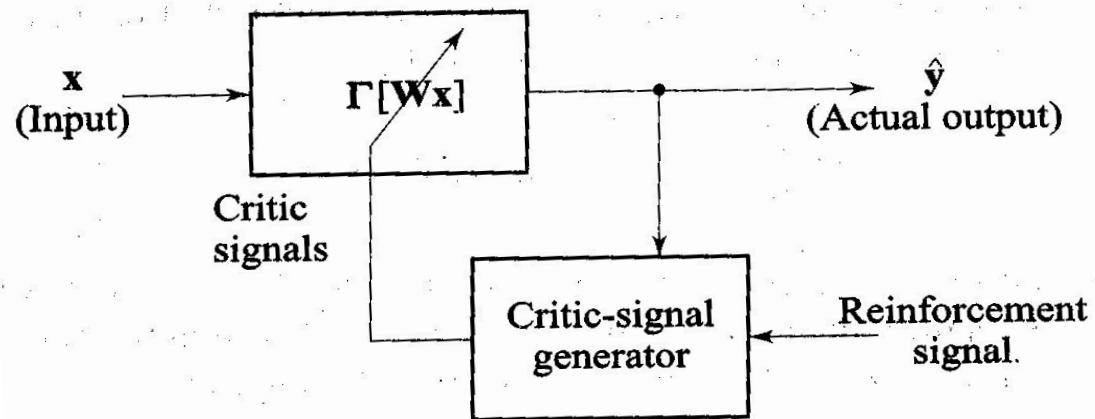
Supervised learning

Unsupervised learning rule



Unsupervised learning

Reinforcement learning



(b) Reinforcement learning

Advantages

- Massive parallel computing system
- Adaptive learning : able to adaptive the change in environment.
- Learning ability: Ability to learning to how to do tasks based on data given for training or past experience.
- Self organization: Ann can create its own organization or represent of information it receive during learning time.
- Fault-tolerant/graceful degrading
- Generalization ability.

Applications of Neural Network

- **Banking**

Check and other document reading, credit application evaluation .

Telecommunications

- Image and data compression, automated information services.

- **Defense**

- target tracking, object discrimination, facial recognition, **Electronics**

- integrated circuit chip layout, automated process control, **Entertainment**

- Animation, share market, forecasting

Continue.....

- **Industrial**

Neural networks are being trained to predict the output gases of furnaces and other industrial processes. They then replace complex and costly equipment used for this purpose in the past.

- **Robotics**

Trajectory control, manipulator controllers

- **Speech**

Speech recognition, speech compression, vowel classification

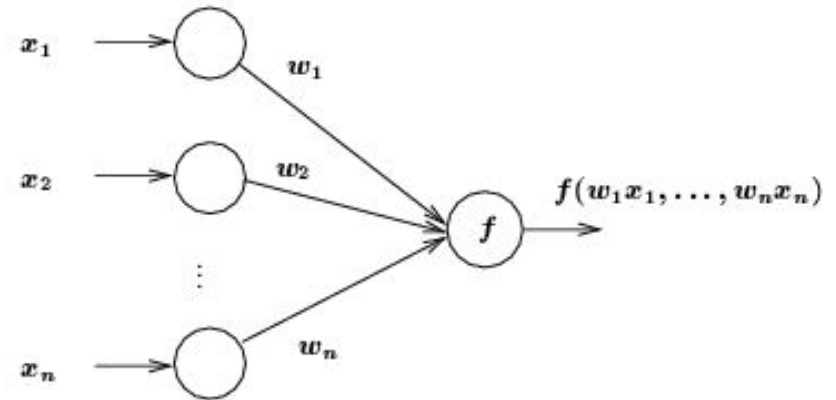
ANN Neuron Models

- Each node has one or more inputs from other nodes, and one output to other nodes
- Input/output values can be

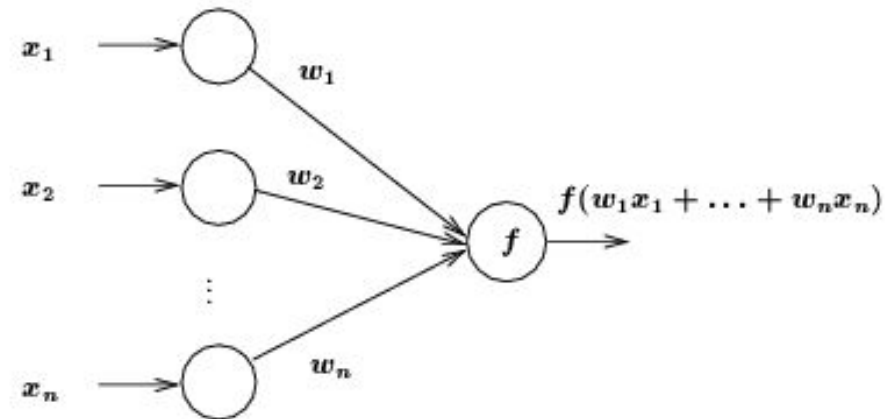
- Binary $\{0, 1\}$
- Bipolar $\{-1, 1\}$
- Continuous

- All inputs to one node come in at the same time and remain activated until the output is produced

- Weights associated with links

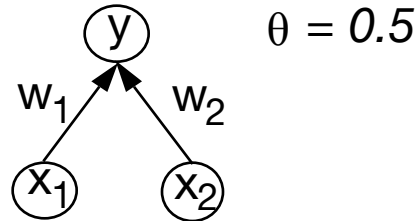


General neuron model



Weighted input summation

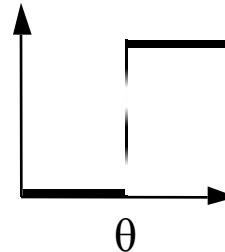
AND



input	output
00	0
01	0
10	0
11	1

$$f(x_1 w_1 + x_2 w_2) = y$$

$$\begin{aligned} \longrightarrow f(0w_1 + 0w_2) &= 0 \\ \longrightarrow f(0w_1 + 1w_2) &= 0 \\ \longrightarrow f(1w_1 + 0w_2) &= 0 \\ \longrightarrow f(1w_1 + 1w_2) &= 1 \end{aligned}$$

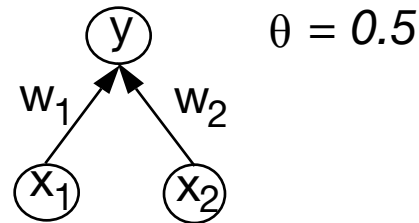


$$f(a) = \begin{cases} 1, & \text{for } a > \theta \\ 0, & \text{for } a \leq \theta \end{cases}$$

some possible values for w_1 and w_2

w_1	w_2
0.20	0.35
0.20	0.40
0.25	0.30
0.40	0.20

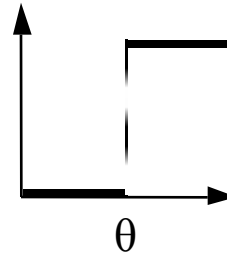
XOR



input	output
00	0
01	1
10	1
11	0

$$f(x_1w_1 + x_2w_2) = y$$

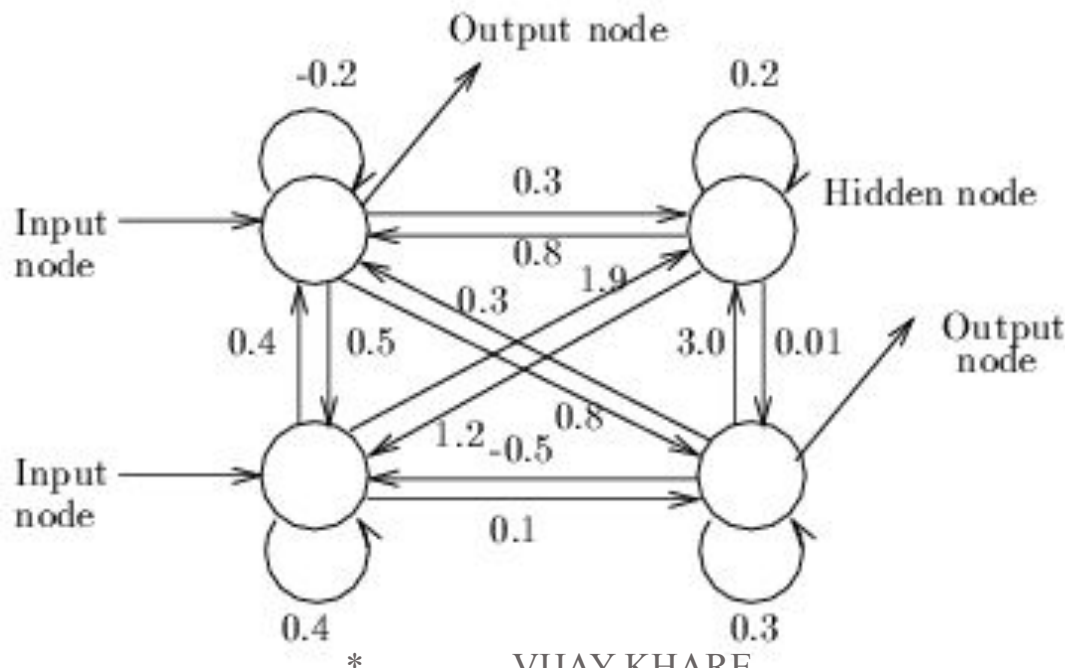
$$\begin{aligned} \longrightarrow f(0w_1 + 0w_2) &= 0 \\ \longrightarrow f(0w_1 + 1w_2) &= 1 \\ \longrightarrow f(1w_1 + 0w_2) &= 1 \\ \longrightarrow f(1w_1 + 1w_2) &= 0 \end{aligned}$$



$$f(a) = \begin{cases} 1, & \text{for } a > \theta \\ 0, & \text{for } a \leq \theta \end{cases}$$

Network Architecture

- (Asymmetric) Fully Connected Networks
 - Every node is connected to every other node
 - Connection may be excitatory (positive), inhibitory (negative), or irrelevant (≈ 0).
 - Most general
 - Symmetric fully connected nets: weights are symmetric ($w_{ij} = w_{ji}$)



Input nodes: receive input from the environment

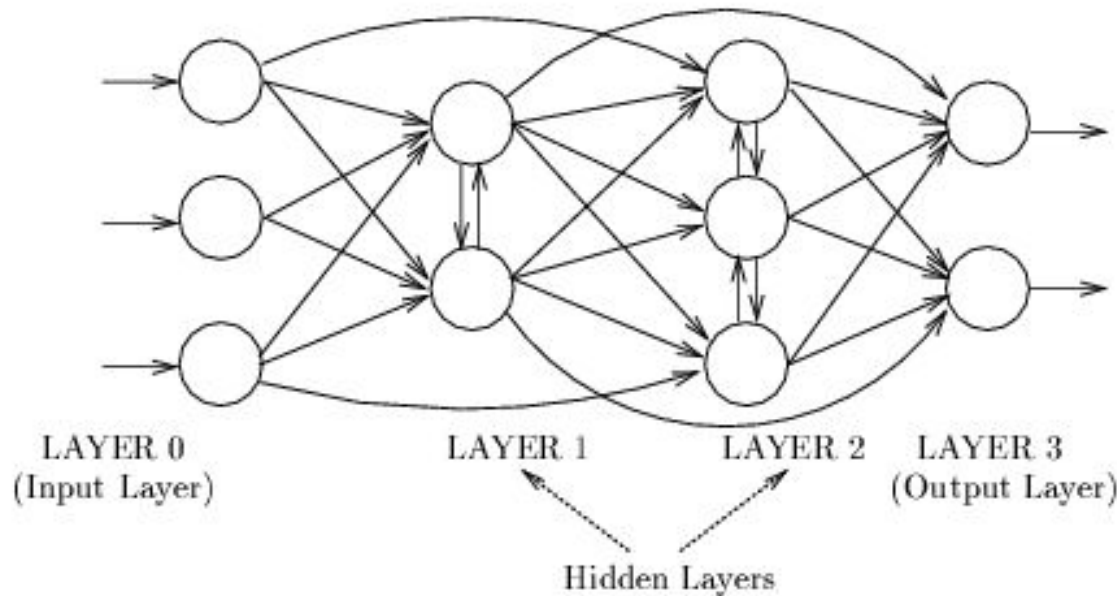
Output nodes: send signals to the environment

Hidden nodes: no direct interaction to the environment

Network Architecture

● Layered Networks

- Nodes are partitioned into subsets, called layers.
- No connections that lead from nodes in layer i to those in layer k if $i > k$.

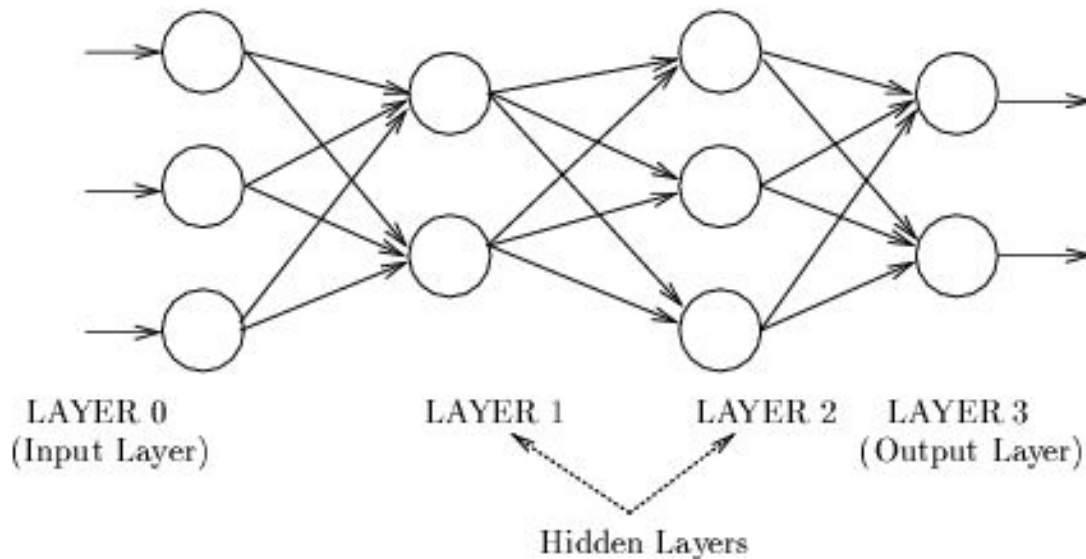


- Inputs from the environment are applied to nodes in layer 0 (**input layer**).
- Nodes in input layer are place holders with no computation occurring (i.e., their node functions are identity function)

Network Architecture

● Feedforward Networks

- A connection is allowed from a node in layer i only to nodes in layer $i + 1$.
- Most widely used architecture.



Conceptually, nodes at higher levels successively abstract features from preceding layers

Network Architecture

- **Acyclic Networks**

- Connections do not form directed cycles.
- Multi-layered feedforward nets are acyclic

- **Recurrent Networks**

- Nets with directed cycles.
- Much harder to analyze than acyclic nets.

- **Modular nets**

- Consists of several modules, each of which is itself a neural net for a particular sub-problem
- Sparse connections between modules

Delta Learning Rule

The delta learning rule is only valid for continuous activation functions as defined in (2.3a), (2.4a), and in the supervised training mode. The learning signal for this rule is called *delta* and is defined as follows

$$r \triangleq [d_i - f(\mathbf{w}_i^t \mathbf{x})] f'(\mathbf{w}_i^t \mathbf{x}) \quad (2.36)$$

The term $f'(\mathbf{w}_i^t \mathbf{x})$ is the derivative of the activation function $f(net)$ computed for $net = \mathbf{w}_i^t \mathbf{x}$. The explanation of the delta learning rule is shown in Figure 2.24. This learning rule can be readily derived from the condition of least squared error between o_i and d_i . Calculating the gradient vector with respect to \mathbf{w}_i of the squared error defined as

$$E \triangleq \frac{1}{2} (d_i - o_i)^2 \quad (2.37a)$$

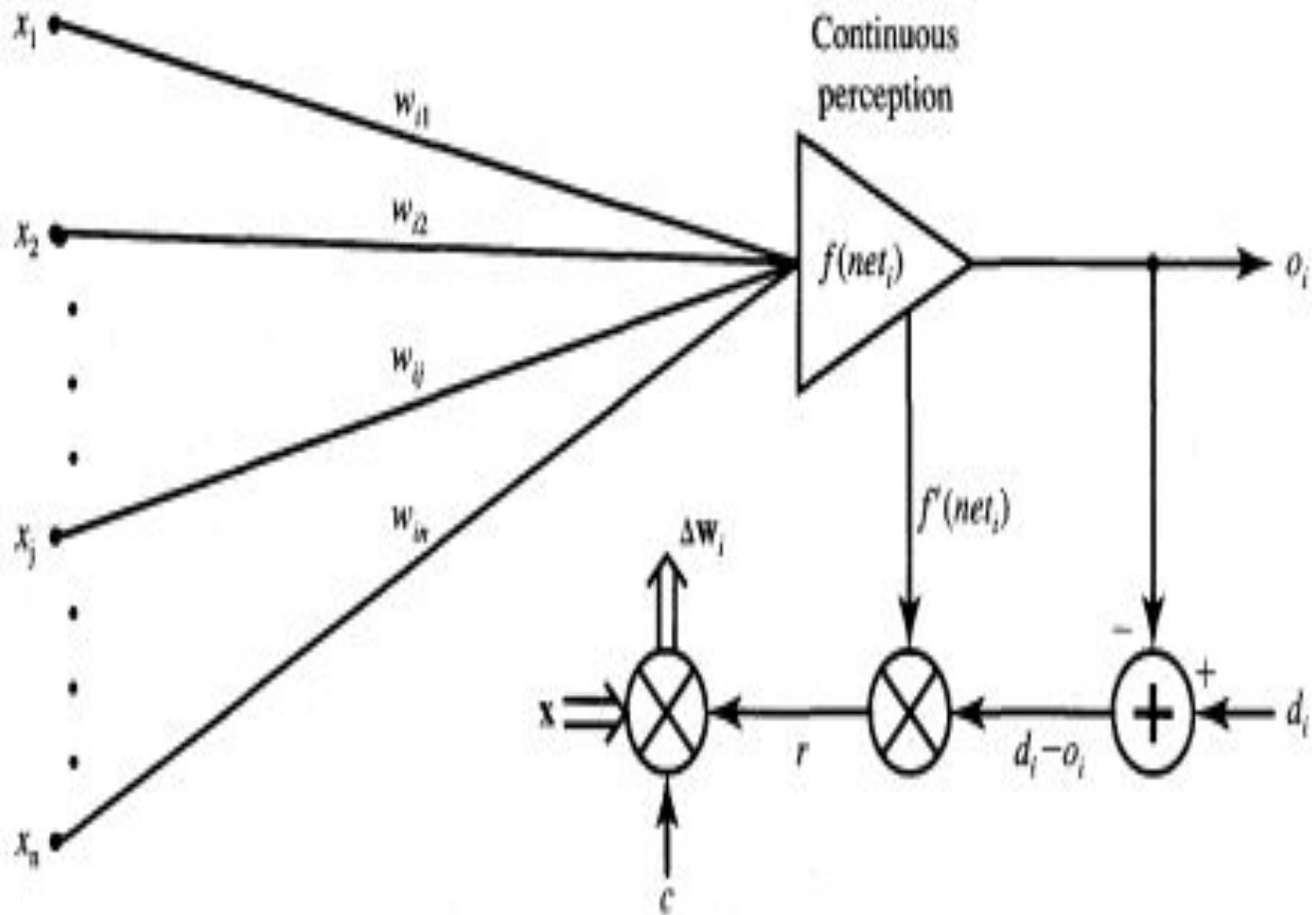


Figure 2.24 Delta learning rule.

which is equivalent to

$$E = \frac{1}{2} [d_i - f(\mathbf{w}_i^t \mathbf{x})]^2 \quad (2.37b)$$

we obtain the error gradient vector value

$$\nabla E = -(d_i - o_i) f'(\mathbf{w}_i^t \mathbf{x}) \mathbf{x} \quad (2.38a)$$

The components of the gradient vector are

$$\frac{\partial E}{\partial w_{ij}} = -(d_i - o_i) f'(\mathbf{w}_i^t \mathbf{x}) x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.38b)$$

Since the minimization of the error requires the weight changes to be in the negative gradient direction, we take

$$\Delta \mathbf{w}_i = -\eta \nabla E \quad (2.39)$$

where η is a positive constant. We then obtain from Eqs. (2.38a) and (2.39)

$$\Delta \mathbf{w}_i = \eta (d_i - o_i) f'(\text{net}_i) \mathbf{x} \quad (2.40a)$$

or, for the single weight the adjustment becomes

$$\Delta w_{ij} = \eta (d_i - o_i) f'(\text{net}_i) x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.40b)$$

Note that the weight adjustment as in (2.40) is computed based on minimization of the squared error. Considering the use of the general learning rule (2.28) and plugging in the learning signal as defined in (2.36), the weight adjustment becomes

$$\Delta \mathbf{w}_i = c (d_i - o_i) f'(\text{net}_i) \mathbf{x} \quad (2.41)$$

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$$

and the initial weight vector \mathbf{w}^1 is assumed identical as in Example 2.4. The learning constant is assumed to be $c = 0.1$. The teacher's desired responses for $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ are $d_1 = -1, d_2 = -1$, and $d_3 = 1$, respectively. The learning

Delta learning and Bipolar continuous activation function

initial weight vector

$$\mathbf{w}^1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$$

Step 1 Input is vector \mathbf{x}_1 , initial weight vector is \mathbf{w}^1 :

$$net^1 = \mathbf{w}^{1t} \mathbf{x}_1 = 2.5$$

$$o^1 = f(net^1) = 0.848$$

$$f'(net^1) = \frac{1}{2}[1 - (o^1)^2] = 0.140$$

$$\begin{aligned} \mathbf{w}^2 &= c(d_1 - o^1)f'(net^1)\mathbf{x}_1 + \mathbf{w}^1 \\ &= [0.974 \quad -0.948 \quad 0 \quad 0.526]^t \end{aligned}$$

Step 2 Input is vector \mathbf{x}_2 , weight vector is \mathbf{w}^2 :

$$net^2 = \mathbf{w}^{2t} \mathbf{x}_2 = -1.948$$

$$o^2 = f(net^2) = -0.75$$

$$f'(net^2) = \frac{1}{2}[1 - (o^2)^2] = 0.218$$

$$\begin{aligned} \mathbf{w}^3 &= c(d_2 - o^2)f'(net^2)\mathbf{x}_2 + \mathbf{w}^2 \\ &= [0.974 \quad -0.956 \quad 0.002 \quad 0.531]^t \end{aligned}$$

Step 3 Input is x_3 , weight vector is \mathbf{w}^3 :

$$net^3 = \mathbf{w}^{3t} \mathbf{x}_3 = -2.46$$

$$o^3 = f(net^3) = -0.842$$

$$f'(net^3) = \frac{1}{2}[1 - (o^3)^2] = 0.145$$

$$\begin{aligned} \mathbf{w}^4 &= c(d_3 - o^3)f'(net^3)\mathbf{x}_3 + \mathbf{w}^3 \\ &= [0.947 \quad -0.929 \quad 0.016 \quad 0.505]^t \end{aligned}$$

Widrow-Hoff Learning Rule

The Widrow-Hoff learning rule (Widrow 1962) is applicable for the supervised training of neural networks. It is independent of the activation function of neurons used since it minimizes the squared error between the desired output value d_i and the neuron's activation value $net_i = \mathbf{w}_i^t \mathbf{x}$. The learning signal for this rule is defined as follows

$$r \triangleq d_i - \mathbf{w}_i^t \mathbf{x} \quad (2.42)$$

The weight vector increment under this learning rule is

$$\Delta \mathbf{w}_i = c(d_i - \mathbf{w}_i^t \mathbf{x}) \mathbf{x} \quad (2.43a)$$

or, for the single weight the adjustment is

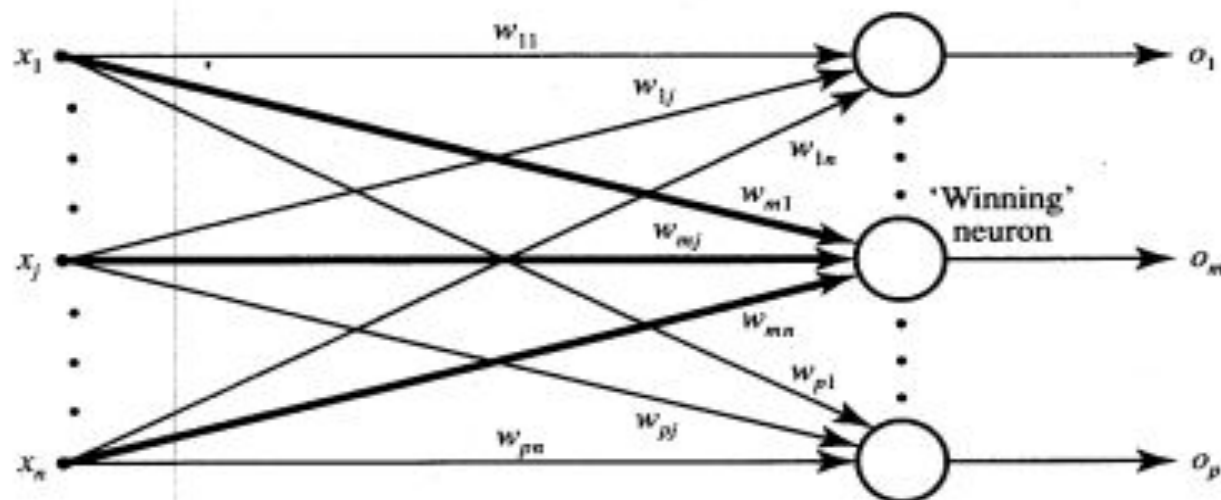
$$\Delta w_{ij} = c(d_i - \mathbf{w}_i^t \mathbf{x}) x_j, \quad \text{for } j = 1, 2, \dots, n \quad (2.43b)$$

This rule can be considered a special case of the delta learning rule. Indeed, assuming in (2.36) that $f(\mathbf{w}_i^t \mathbf{x}) = \mathbf{w}_i^t \mathbf{x}$, or the activation function is simply the identity function $f(net) = net$, we obtain $f'(net) = 1$, and (2.36) becomes identical to (2.42). This rule is sometimes called the *LMS (least mean square) learning rule*. Weights are initialized at any values in this method.

Winner-Take-All Learning Rule

This learning rule differs substantially from any of the rules discussed so far in this section. It can only be demonstrated and explained for an ensemble of neurons, preferably arranged in a layer of p units. This rule is an example of competitive learning, and it is used for unsupervised network training. Typically, winner-take-all learning is used for learning statistical properties of inputs (Hecht-Nielsen 1987). The learning is based on the premise that one of the neurons in the layer, say the m 'th, has the maximum response due to input \mathbf{x} , as shown in Figure 2.25. This neuron is declared the *winner*. As a result of this winning event, the weight vector \mathbf{w}_m

$$\mathbf{w}_m = [w_{m1} \quad w_{m2} \quad \cdots \quad w_{mn}]^t \quad (2.45)$$



(adjusted weights are highlighted)

containing weights highlighted in the figure is the only one ^{میں} adjusted in the given unsupervised learning step. Its increment is computed as follows

$$\Delta \mathbf{w}_m = \alpha(\mathbf{x} - \mathbf{w}_m) \quad (2.46a)$$

or, the individual weight adjustment becomes

$$\Delta w_{mj} = \alpha(x_j - w_{mj}), \quad \text{for } j = 1, 2, \dots, n \quad (2.46b)$$

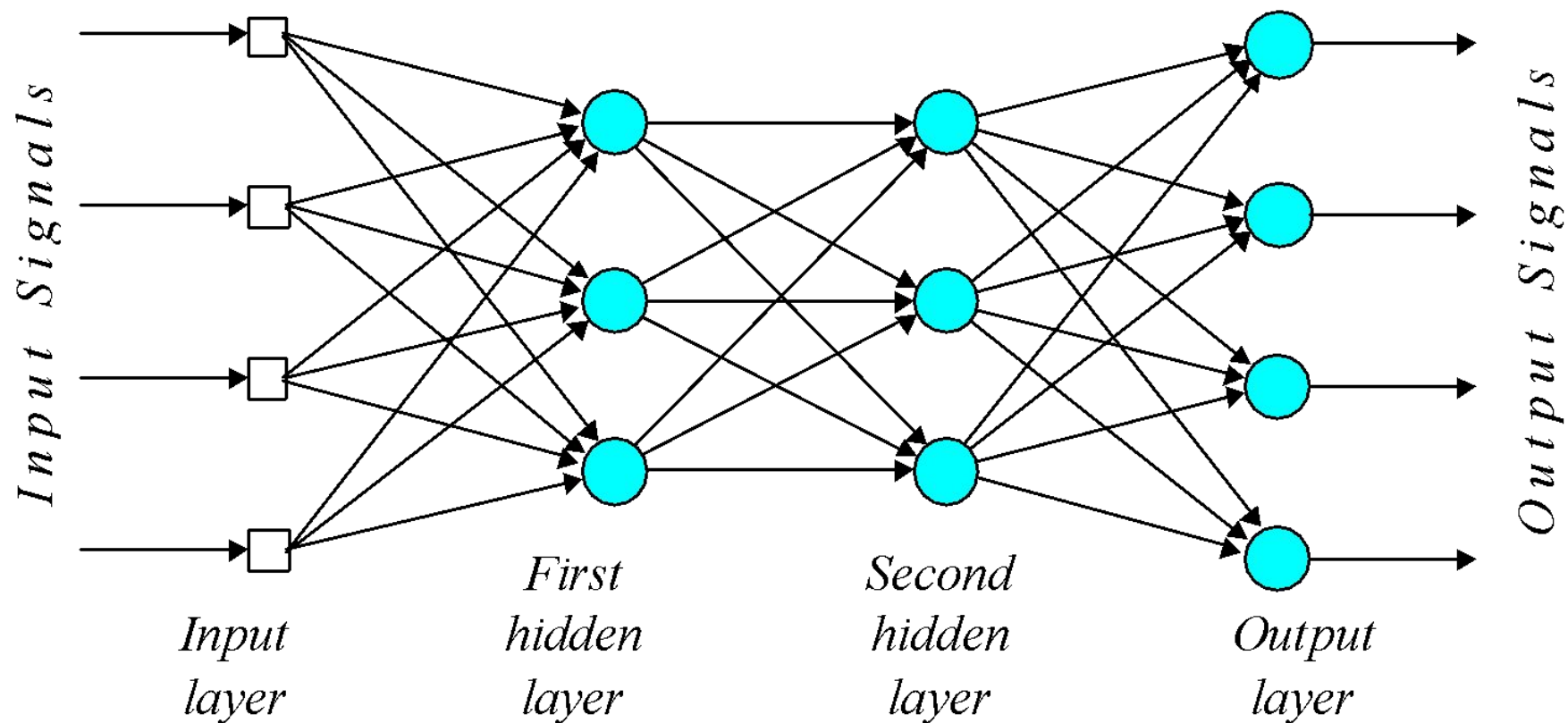
where $\alpha > 0$ is a small learning constant, typically decreasing as learning progresses. The winner selection is based on the following criterion of maximum activation among all p neurons participating in a competition:

$$\mathbf{w}_m^t \mathbf{x} = \max_{i=1,2,\dots,p} (\mathbf{w}_i^t \mathbf{x}) \quad (2.47)$$

Multilayer 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.

Multilayer perceptron with two hidden layers

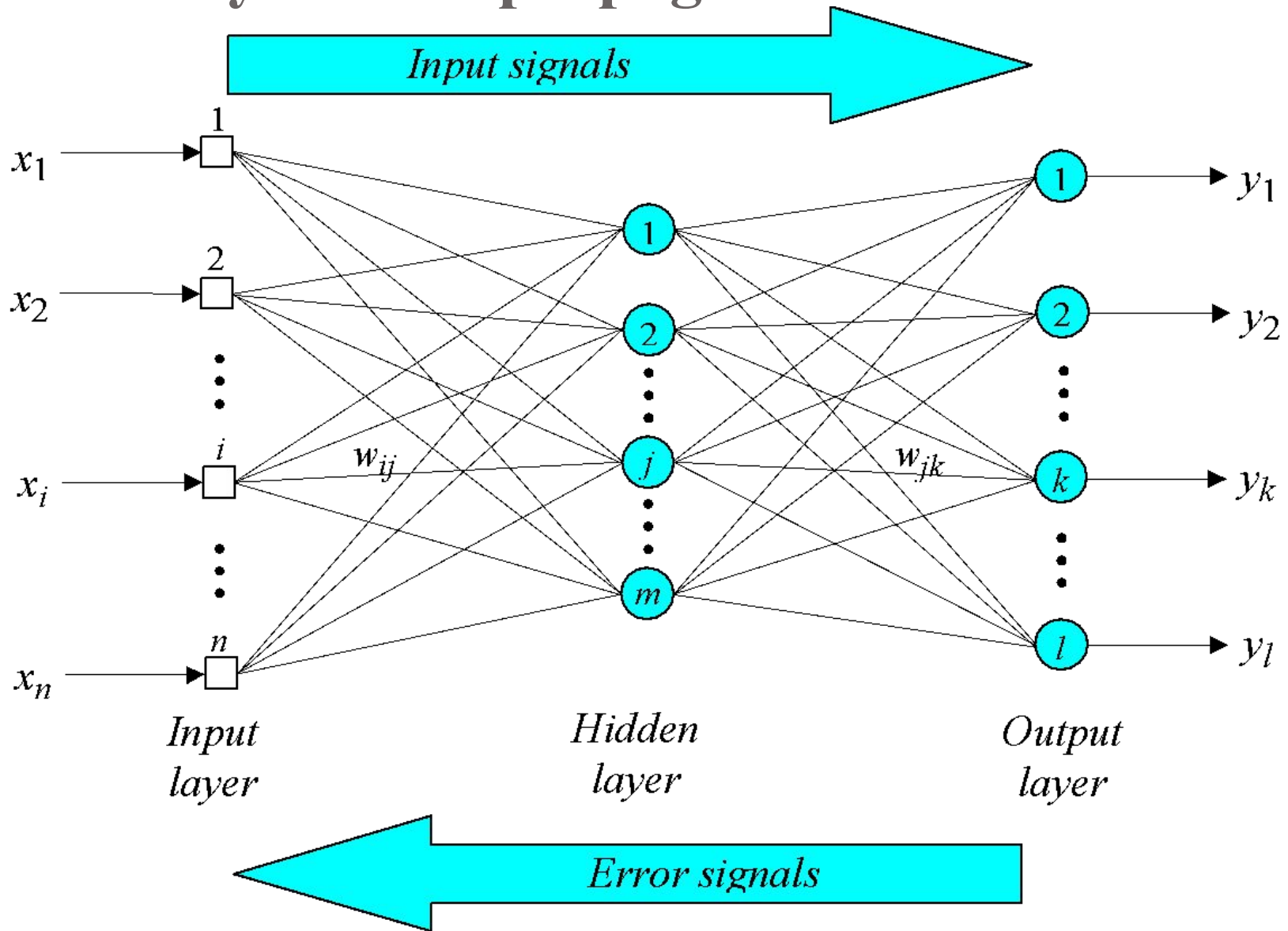


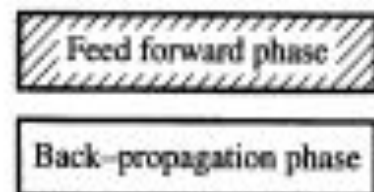
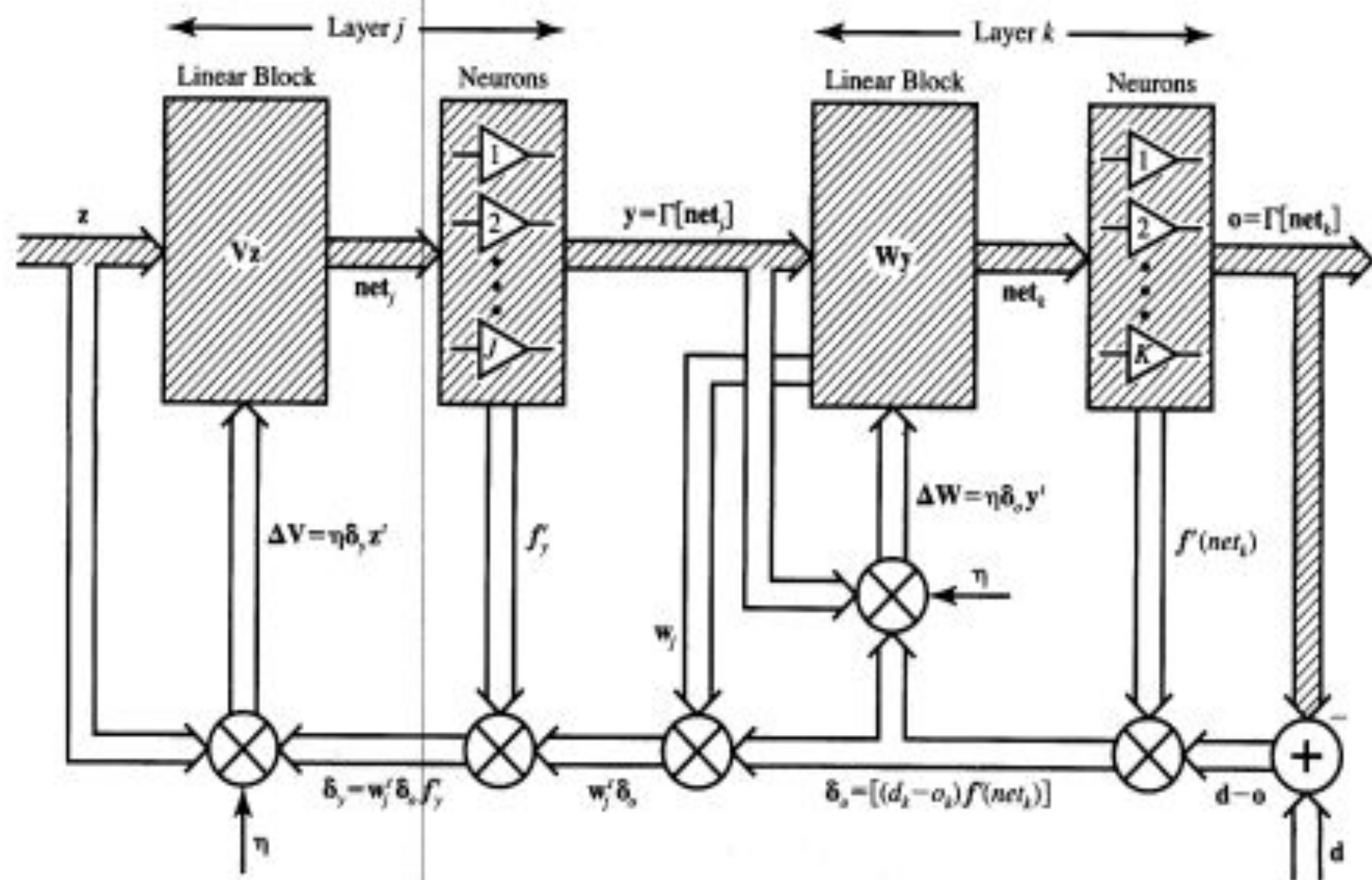
Back-propagation neural network

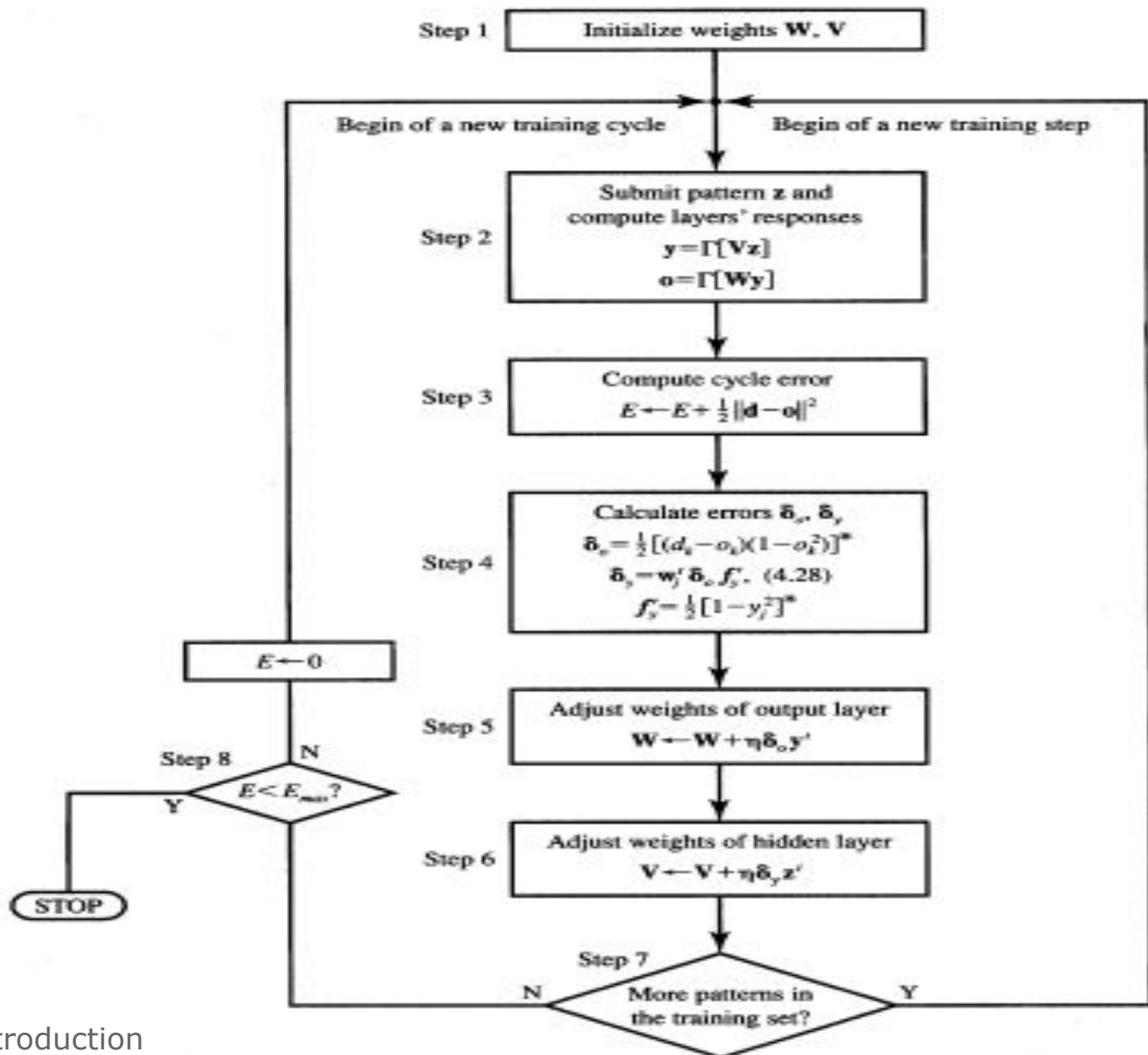
- Learning in a multilayer network proceeds the same way as for a delta learning.
- A training set of input patterns is presented to the network.
- The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are adjusted to reduce this error.

- 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.

Three-layer back-propagation neural network







Backpropagation Learning

● Notations:

- Weights: two weight matrices:

$w^{(1,0)}$ from input layer (0) to hidden layer (1)

$w^{(2,1)}$ from hidden layer (1) to output layer (2)

$w_{2,1}^{(1,0)}$ weight from node 1 at layer 0 to node 2 in layer 1

- Training samples: pair of $\{(x_p, d_p) | p = 1, \dots, P\}$

so it is supervised learning

- Input pattern: $x_p = (x_{p,1}, \dots, x_{p,n})$

- Output pattern: $o_p = (o_{p,1}, \dots, o_{p,k})$

- Desired output: $d_p = (d_{p,1}, \dots, d_{p,k})$

- Error: $l_{p,j} = o_{p,j} - d_{p,j}$ error for output j when x_p is applied

sum square error $= \sum_{p=1}^P \sum_{j=1}^K (l_{p,j})^2$

$w^{(1,0)}$ $w^{(2,1)}$

This error drives learning (change and

Backpropagation Learning

- Sigmoid function again:

- Differentiable:

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

$$S'(x) = -\frac{1}{(1 + e^{-x})^2} \cdot (1 + e^{-x})'$$

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

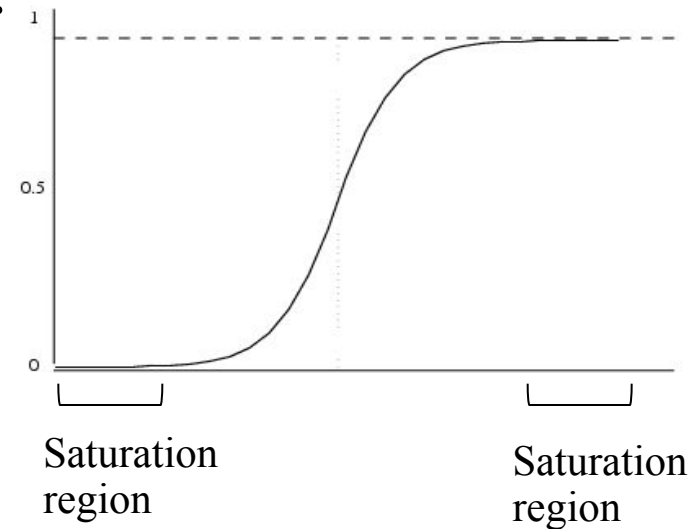
$$= \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}}$$

$$= S(x)(1 - S(x))$$

- When $|net|$ is sufficiently large, it moves into one of the two saturation regions, behaving like a threshold or ramp function.

- Chain rule of differentiation

$$\text{if } z = f(y), y = g(x), x = h(t) \text{ then } \frac{dz}{dt} = \frac{dz}{dy} \cdot \frac{dy}{dx} \cdot \frac{dx}{dt} = f'(y)g'(x)h'(t)$$



Backpropagation Learning

- **Forward computing:**

- Apply an input vector \mathbf{x} to input nodes
- Computing output vector $\mathbf{x}^{(1)}$ on hidden layer

$$x_j^{(1)} = S(\text{net}_j^{(1)}) = S(\sum w_{j,i}^{(1,0)} x_i)$$

- Computing the output vector \mathbf{o} on output layer

$$o_k = S(\text{net}_k^{(2)}) = S(\sum w_{k,j}^{(2,1)} x_j^{(1)})$$

- The net is said to be a map from input \mathbf{x} to output \mathbf{o}

- **Objective of learning:** $\sum_{p=1}^P \sum_{j=1}^K (l_{p,j})^2$

- reduce sum square error

for the given P training samples as much as possible (to zero if possible)

Backpropagation Learning

- **Idea of BP learning:**
 - Update of weights in $w^{(2,1)}$ (from hidden layer to output layer):
delta rule as in a single layer net using sum square error
 - Delta rule is not applicable to updating weights in $w^{(1,0)}$ (from input and hidden layer) because we don't know the desired values for hidden nodes
 - **Solution:** Propagating errors at output nodes down to hidden nodes, these computed errors on hidden nodes drives the update of weights in $w^{(1,0)}$ (again by delta rule), thus called error **BACKPROPAGATION (BP) learning**
 - How to compute errors on hidden nodes is the key
 - Error backpropagation can be continued downward if the net has more than one hidden

Backpropagation Learning

- Generalized delta rule:

- Consider sequential learning mode: for a given

$$\text{sample } (x_p, d_p)$$

$$E = \sum_k (l_{p,k})^2$$

- Update of weights by gradient descent

$$\Delta w_{k,j}^{(2,0)} \propto (-\partial E / \partial w_{k,j}^{(2,0)})$$

$$\text{For weight in } w^{(2,1)}: \Delta w_{j,i}^{(1,0)} \propto (-\partial E / \partial w_{j,i}^{(1,0)})$$

$$\text{For weight in } w^{(1,0)}:$$

- Derivation of update rule for $w^{(2,1)}$:

since E is a function of $net_k^{(2)}$ and $net_k^{(2)} = d_k - o_k$ is a function of $w_{k,j}^{(2,1)}$, by

$$\frac{\partial E}{\partial w_{k,j}^{(2,1)}} = \frac{\partial E}{\partial (d_k - o_k)} \frac{\partial (d_k - o_k)}{\partial net_k^{(2)}} \frac{\partial net_k^{(2)}}{\partial w_{k,j}^{(2,1)}}$$

$$= -2(d_k - o_k) \mathcal{S}'(net_k^{(2)}) x_j^{(1)}.$$

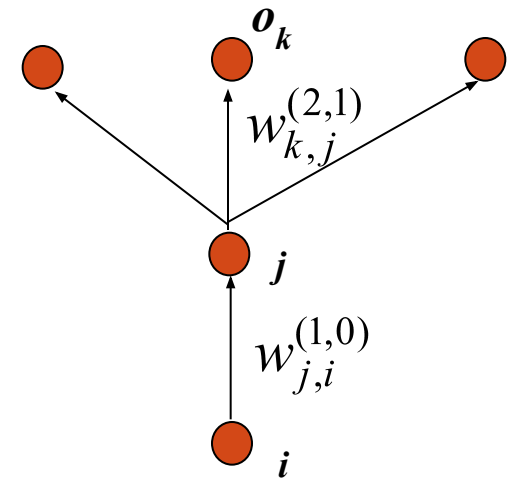
Backpropagation Learning

- Derivation of update rule for $w_{j,i}^{(1,0)}$
consider **hidden node j** :

weight $w_{j,i}^{(1,0)}$ influences $net_j^{(1)}$

it sends $S(net_j^{(1)})$ to all output nodes

\therefore all K terms in E are functions of $w_{j,i}^{(1,0)}$



$$E = \sum_k (d_k - o_k)^2, \quad o_k = S(net_k^{(2)}), \quad net_k^{(2)} = \sum_j x_j^{(1)} w_{k,j}^{(2,1)},$$

$$x_j^{(1)} = S(net_j^{(1)}), \quad net_j^{(1)} = \sum_i x_i w_{j,i}^{(1,0)}$$

by chain rule

$$\frac{\partial E}{\partial w_{j,i}^{(1,0)}} = \sum_{k=1}^K \left\{ \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial net_k^{(2)}} \frac{\partial net_k^{(2)}}{\partial x_j^{(1)}} \frac{\partial x_j^{(1)}}{\partial net_j^{(1)}} \frac{\partial net_j^{(1)}}{\partial w_{j,i}^{(1,0)}} \right\}$$

$$\frac{\partial E}{\partial w_{j,i}^{(1,0)}} = \sum_{k=1}^K \left\{ -2(d_k - o_k) \mathcal{S}'(net_k^{(2)}) w_{k,j}^{(2,1)} \mathcal{S}'(net_j^{(1)}) x_i \right\}$$

Backpropagation Learning

- Update rules:
for outer layer weights $w^{(2,1)}$:

$$\frac{\partial E}{\partial w_{k,j}^{(2,1)}} = -2(d_k - o_k) \mathcal{S}'(net_k^{(2)}) x_j^{(1)}$$

$$\Delta w_{k,j}^{(2,1)} = \eta \times \delta_k \times x_j^{(1)} \quad \text{where } \delta_k = (d_k - o_k) \mathcal{S}'(net_k^{(2)})$$

for inner layer weights $w^{(1,0)}$:

$$\frac{\partial E}{\partial w_{j,i}^{(1,0)}} = \sum_{k=1}^K \left\{ -2(d_k - o_k) \mathcal{S}'(net_k^{(2)}) w_{k,j}^{(2,1)} \mathcal{S}'(net_j^{(1)}) x_i \right\}$$

$$\Delta w_{j,i}^{(1,0)} = \eta \times \mu_j \times x_i \quad \text{where } \mu_j = \left(\sum_k \delta_k w_{k,j}^{(2,1)} \right) \mathcal{S}'(net_j^{(1)})$$

Weighted sum of errors
from output layer

Algorithm Backpropagation;

Start with randomly chosen weights;

while MSE is unsatisfactory

and computational bounds are not exceeded, do

for each input pattern x_p , $1 \leq p \leq P$,

 Compute hidden node inputs ($net_{p,j}^{(1)}$);

 Compute hidden node outputs ($x_{p,j}^{(1)}$);

 Compute inputs to the output nodes ($net_{p,k}^{(2)}$);

 Compute the network outputs ($o_{p,k}$);

 Modify outer layer weights:

$$\Delta w_{k,j}^{(2,1)} = \eta (d_{p,k} - o_{p,k}) \mathcal{S}'(net_{p,k}^{(2)}) x_{p,j}^{(1)}$$

 Modify weights between input & hidden nodes:

$$\Delta w_{j,i}^{(1,0)} = \eta \sum_k \left((d_{p,k} - o_{p,k}) \mathcal{S}'(net_{p,k}^{(2)}) w_{k,j}^{(2,1)} \right) \mathcal{S}'(net_{p,j}^{(1)}) x_{p,i}$$

 end-for

end-while.

Note: if S is a logistic function,
then $S'(x) = S(x)(1 - S(x))$

Backpropagation Learning

- **Function approximation:**

- For the given $w = (w^{(1,0)}, w^{(2,1)})$, $o = f(x)$: it realizes a functional mapping from f .
- Theoretically, feedforward nets with at least one hidden layer of non-linear nodes are able to approximate any L2 functions (all square-integral functions, including almost all commonly used math functions) to any given degree of accuracy, provided there are sufficient many hidden nodes

- Any L2 function $f(x)$ can be approximated by a

$$\hat{f}_N(x) = \sum_{k_1=-N}^N \cdots \sum_{k_n=-N}^N c_k e^{2\pi\sqrt{-1}(k \cdot x)}$$

$$\lim_{N \rightarrow \infty} \int_{[0,1]^n} |f(x) - \hat{f}_N(x)|^2 dx = 0$$

It has been shown the Fourier series approximation can be realized with Feedforward net with one hidden layer of cosine node function

- Reading for grad students (Sec 3.5) for more

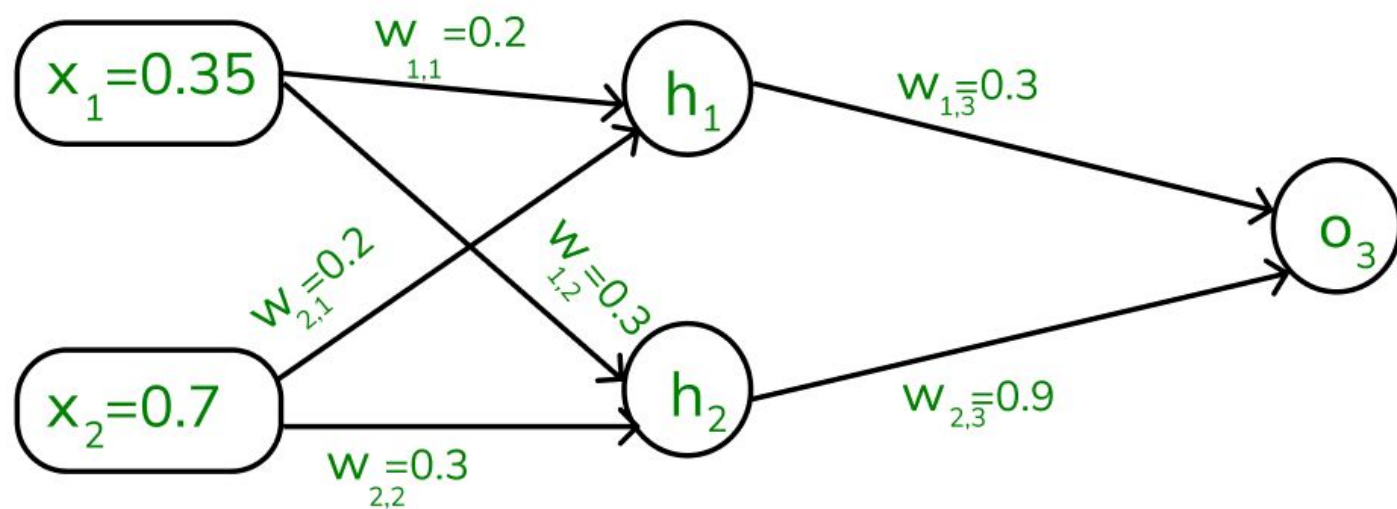
Strengths of BP Learning

- **Great representation power**
 - Any L2 function can be represented by a BP net
 - Many such functions can be approximated by BP learning (gradient descent approach)
- **Wide applicability of BP learning**
 - Only requires that a good set of training samples is available
 - Does not require substantial prior knowledge or deep understanding of the domain itself (ill structured problems)
 - Tolerates noise and missing data in training samples (graceful degrading)
- **Easy to implement the core of the learning algorithm**
- **Good generalization power**
 - Often produce accurate results for inputs

Deficiencies of BP Learning

- Learning often takes a **long time** to converge
 - Complex functions often need hundreds or thousands of epochs
- The net is essentially a **black box**
 - It may provide a desired mapping between input and output vectors (\mathbf{x} , \mathbf{o}) but does not have the information of why a particular \mathbf{x} is mapped to a particular \mathbf{o} .
 - It thus cannot provide an intuitive (e.g., causal) explanation for the computed result.
 - This is because the hidden nodes and the learned weights do not have clear semantics.
 - What can be learned are operational parameters, not general, abstract knowledge of a domain
 - Unlike many statistical methods, there is no theoretically well-founded way to **assess the quality** of BP learning
 - What is the confidence level one can have for a trained BP net, with the final E (which may or may not be close to zero)?

- **Example of Backpropagation in Machine Learning**
- Let us now take an example to explain backpropagation in Machine Learning,
- Assume that the neurons have the sigmoid activation function to perform forward and backward pass on the network. And also assume that the actual output of y is 0.5 and the learning rate is 1. Now perform the backpropagation using backpropagation algorithm.



Step1: Before proceeding to calculating forward propagation, we need to know the two formulae:

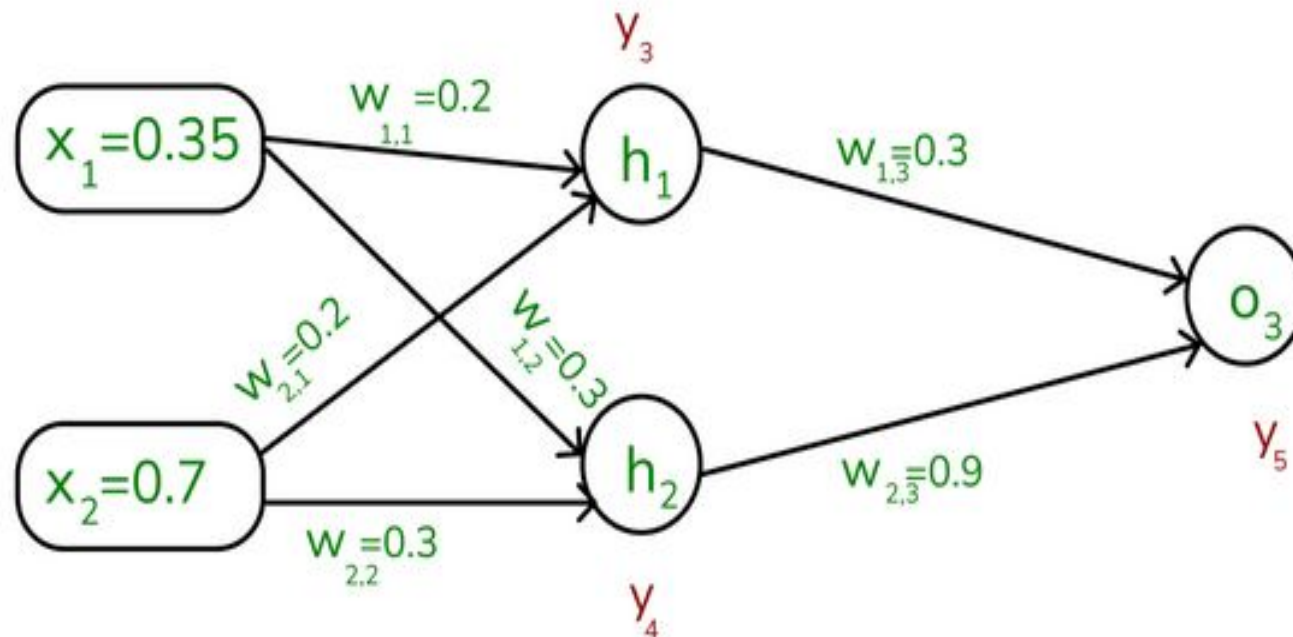
$$a_j = \sum (w_{i,j} * x_i)$$

Where,

- a_j is the weighted sum of all the inputs and weights at each node,
- $w_{i,j}$ – represents the weights associated with the j^{th} input to the i^{th} neuron,
- x_i – represents the value of the j^{th} input,

$y_j = F(a_j) = \frac{1}{1+e^{-a_j}}$, y_i – is the output value, F denotes the activation function [sigmoid activation function is used here), which transforms the weighted sum into the output value.

Step 2: To compute the forward pass, we need to compute the output for y_3 , y_4 , and y_5 .



To find the outputs of y_3 , y_4 and y_5

We start by calculating the weights and inputs by using the formula:

$a_j = \sum(w_{i,j} * x_i)$ To find y_3 , we need to consider its incoming edges along with its weight and input. Here the incoming edges are from X_1 and X_2 .

At h1 node,

$$\begin{aligned} a_1 &= (w_{1,1}x_1) + (w_{2,1}x_2) \\ &= (0.2 * 0.35) + (0.2 * 0.7) \\ &= 0.21 \end{aligned}$$

Once, we calculated the a_1 value, we can now proceed to find the y_3 value:

$$y_j = F(a_j) = \frac{1}{1+e^{-a_j}}$$

$$y_3 = F(0.21) = \frac{1}{1+e^{-0.21}}$$

$$y_3 = 0.56$$

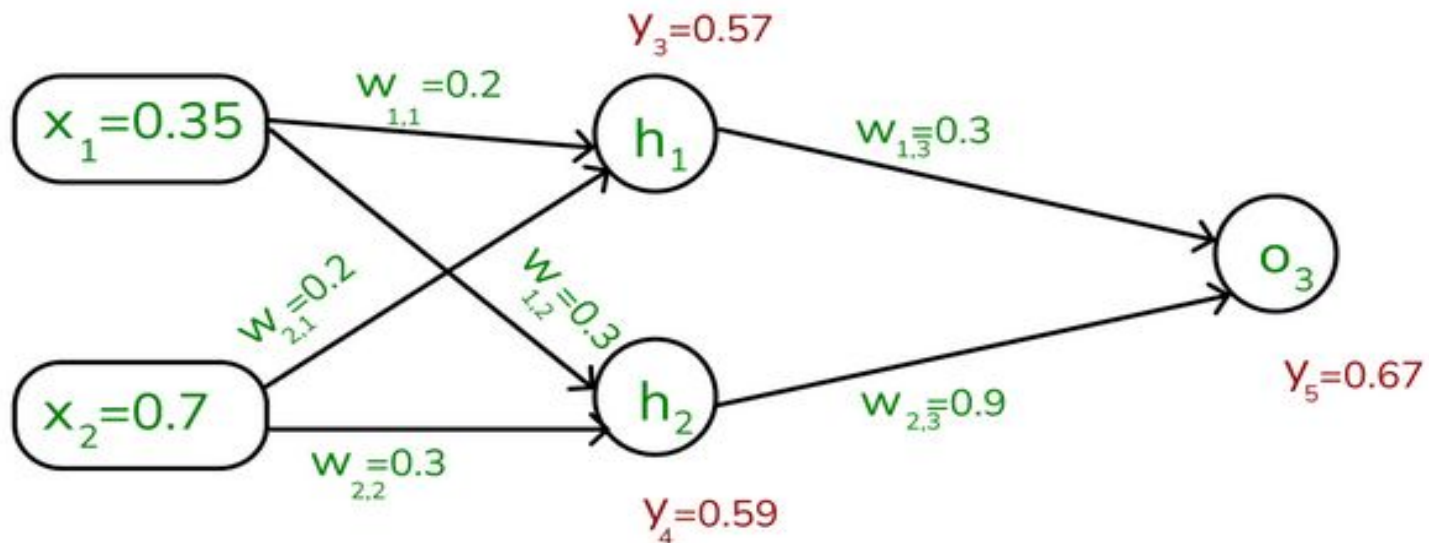
Similarly find the values of y_4 at h_2 and y_5 at O_3 ,

$$a_2 = (w_{1,2} * x_1) + (w_{2,2} * x_2) = (0.3 * 0.35) + (0.3 * 0.7) = 0.315$$

$$y_4 = F(0.315) = \frac{1}{1+e^{-0.315}}$$

$$a_3 = (w_{1,3} * y_3) + (w_{2,3} * y_4) = (0.3 * 0.57) + (0.9 * 0.59) = 0.702$$

$$y_5 = F(0.702) = \frac{1}{1+e^{-0.702}} = 0.67$$



Values of y_3 , y_4 and y_5

Note that, our actual output is 0.5 but we obtained 0.67. To calculate the error, we can use the below formula:

$$Error_j = y_{target} - y_5$$

$$Error = 0.5 - 0.67$$

$$= -0.17$$

Implementing Backward Propagation

Each weight in the network is changed by,

$$\nabla_{w_{ij}} = \eta \delta_j O_j$$

$$\delta_j = O_j (1 - O_j)(t_j - O_j) \text{ (if } j \text{ is an output unit)}$$

$$\delta_j = O_j (1 - O_j) \sum_k \delta_k w_{kj} \text{ (if } j \text{ is a hidden unit)}$$

where ,

η is the constant which is considered as learning rate,

t_j is the correct output for unit j

δ_j is the error measure for unit j

Step 3: To calculate the backpropagation, we need to start from the output unit:

To compute the δ_5 , we need to use the output of forward pass,

$$\delta_5 = y_5(1-y_5) (y_{\text{target}} - y_5)$$

$$= 0.67(1-0.67) (-0.17)$$

$$= -0.0376$$

For hidden unit,

To compute the hidden unit, we will take the value of δ_5

$$\delta_3 = y_3(1-y_3) (w_{1,3} * \delta_5)$$

$$= 0.56(1-0.56) (0.3 * -0.0376)$$

$$= -0.0027$$

$$\delta_4 = y_4 (1-y_5) (w_{2,3} * \delta_5)$$

$$= 0.59(1-0.59) (0.9 * -0.0376)$$

$$= -0.0819$$

Step 4: We need to update the weights, from output unit to hidden unit,

$$\nabla w_{j,i} = \eta \delta_j O_i$$

Note- Here our learning rate is 1

$$\nabla w_{2,3} = \eta \delta_5 O_4$$

$$= 1 * (-0.376) * 0.59$$

$$= -0.22184$$

We will be updating the weights based on the old weight of the network,

$$w_{2,3}(\text{new}) = \nabla w_{4,5} + w_{4,5} (\text{old})$$

$$= -0.22184 + 0.9$$

$$= 0.67816$$

From hidden unit to input unit,

For an hidden to input node, we need to do calculations by the following;

$$\begin{aligned}\nabla w_{1,1} &= \eta \delta_3 O_4 \\ &= 1 * (-0.0027) * 0.35 \\ &= 0.000945\end{aligned}$$

Similarly, we need to calculate the new weight value using the old one:

$$\begin{aligned}w_{1,1}(\text{new}) &= \nabla w_{1,1} + w_{1,1}(\text{old}) \\ &= 0.000945 + 0.2 \\ &= 0.200945\end{aligned}$$

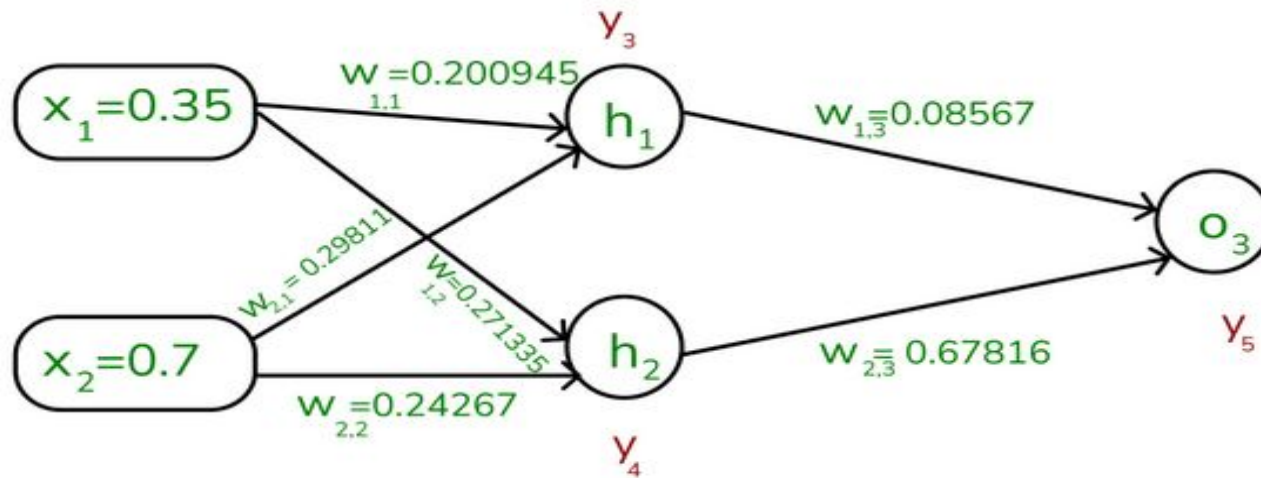
Similarly, we update the weights of the other neurons: The new weights are mentioned below

$$w_{1,2}(\text{new}) = 0.271335$$

$$w_{1,3}(\text{new}) = 0.08567$$

$$w_{2,1}(\text{new}) = 0.29811$$

The updated weights are illustrated below,



Through backward pass the weights are updated

Once, the above process is done, we again perform the forward pass to find if we obtain the actual output as 0.5.

While performing the forward pass again, we obtain the following values:

$$y_3 = 0.57$$

$$y_4 = 0.56$$

$$y_3 = 0.57$$

$$y_4 = 0.56$$

$$y_5 = 0.61$$

We can clearly see that our y_5 value is 0.61 which is not an expected actual output, So again we need to find the error and backpropagate through the network by updating the weights until the actual output is obtained.

$$Error = y_{target} - y_5$$

$$= 0.5 - 0.61$$

$$= -0.11$$

This is how the backpropagate works, it will be performing the forward pass first to see if we obtain the actual output, if not we will be finding the error rate and then backpropagating backwards through the layers in the network by adjusting the weights according to the error rate. This process is said to be continued until the actual output is gained by the neural network.