

# Module III

## **Module-3 (Neural Networks (NN) and Support Vector Machines (SVM))**

Perceptron, Neural Network - Multilayer feed forward network, Activation functions (Sigmoid, ReLU, Tanh), Backpropagation algorithm.

SVM - Introduction, Maximum Margin Classification, Mathematics behind Maximum Margin Classification, Maximum Margin linear separators, soft margin SVM classifier, non-linear SVM, Kernels for learning non-linear functions, polynomial kernel, Radial Basis Function(RBF).

# APPLICATIONS OF SOFT COMPUTING

- Handwriting Recognition
- Image Processing and Data Compression
- Automotive Systems and Manufacturing
- Soft Computing to Architecture
- Decision-support Systems
- Soft Computing to Power Systems
- Neuro Fuzzy systems
- Fuzzy Logic Control
- Machine Learning Applications
- Speech and Vision Recognition Systems
- Process Control and So on

TRACE KTU

# DEFINITION OF NEURAL NETWORKS

**According to the DARPA Neural Network Study (1988, AFCEA International Press, p. 60):**

- ... a neural network is a system composed of many simple processing elements operating in parallel whose function is determined by network structure, connection strengths, and the processing performed at computing elements or nodes.

**According to Haykin (1994), p. 2:**

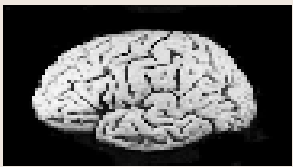
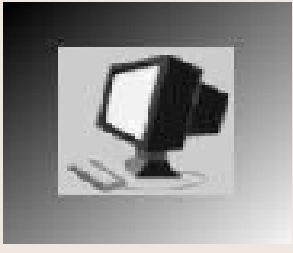
A neural network is a massively parallel distributed processor that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

- Knowledge is acquired by the network through a learning process.
- Interneuron connection strengths known as synaptic weights are used to store the knowledge.

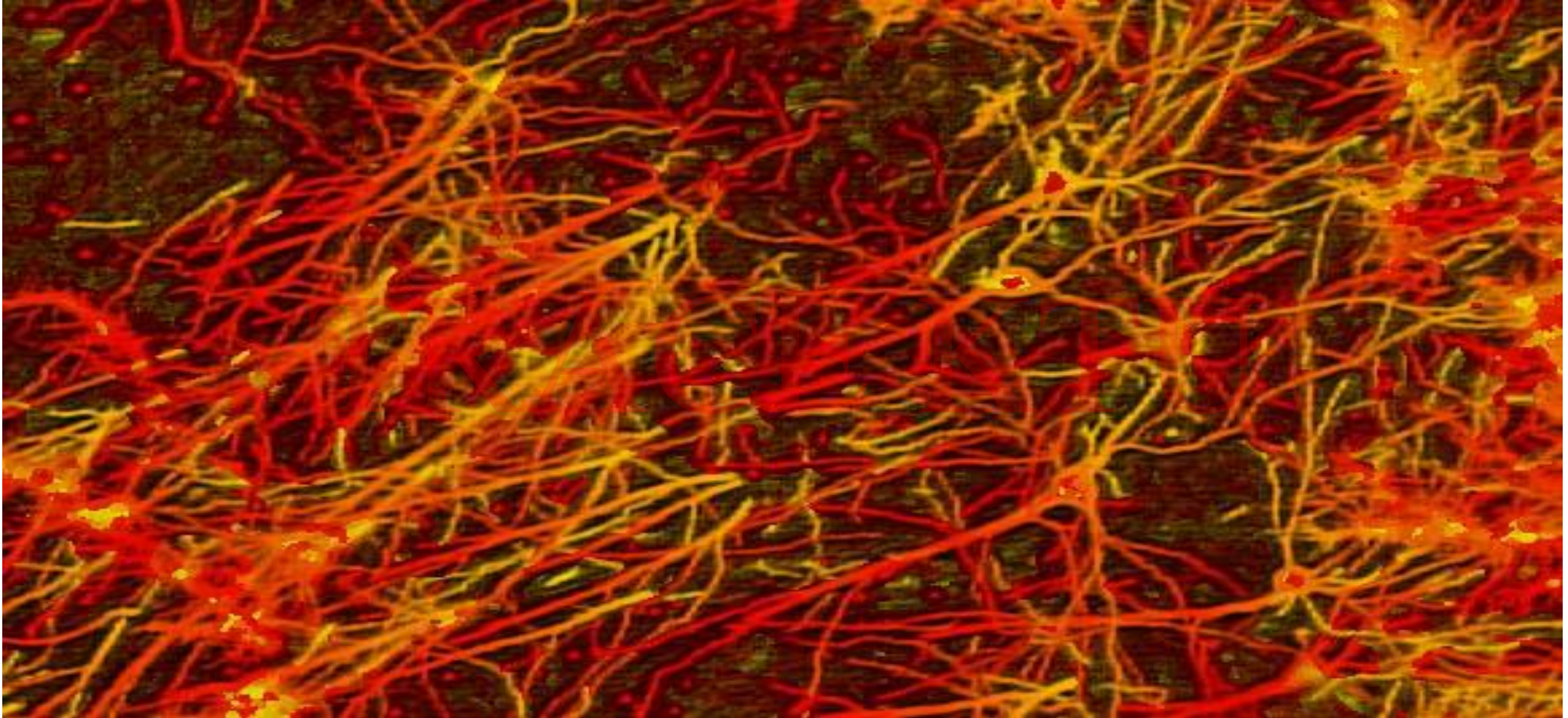
# BRAIN COMPUTATION



The **human brain** contains about 10 billion nerve cells, or neurons. On average, each neuron is connected to other neurons through approximately 10,000 synapses.

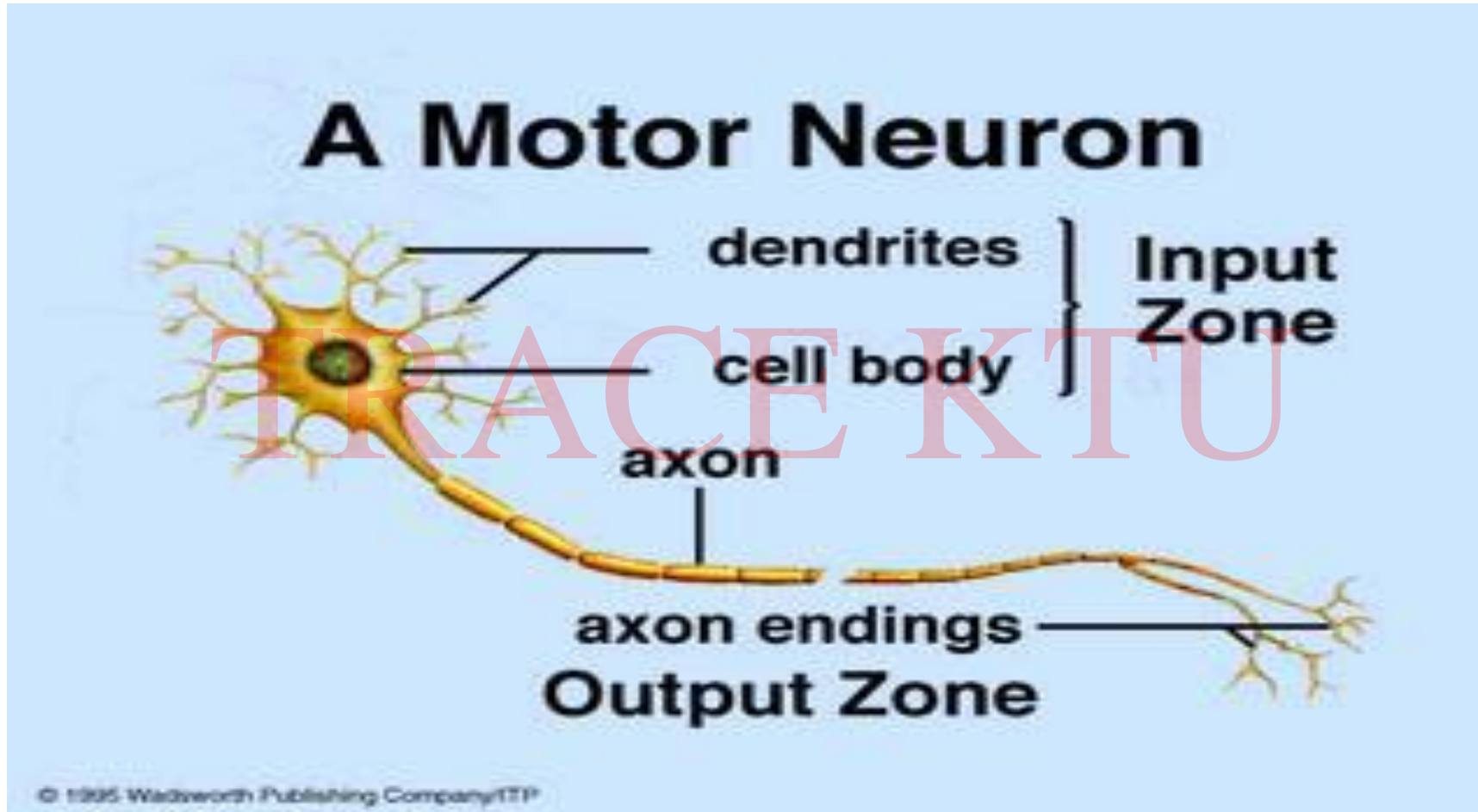
	processing elements	element size	energy use	processing speed	style of computation	fault tolerant	learns	intelligent, conscious
	$10^{14}$ synapses	$10^{-6}$ m	30 W	100 Hz	parallel, distributed	yes	yes	usually
	$10^8$ transistors	$10^{-6}$ m	30 W (CPU)	$10^9$ Hz	serial, centralized	no	a little	not (yet)

# INTERCONNECTIONS IN BRAIN



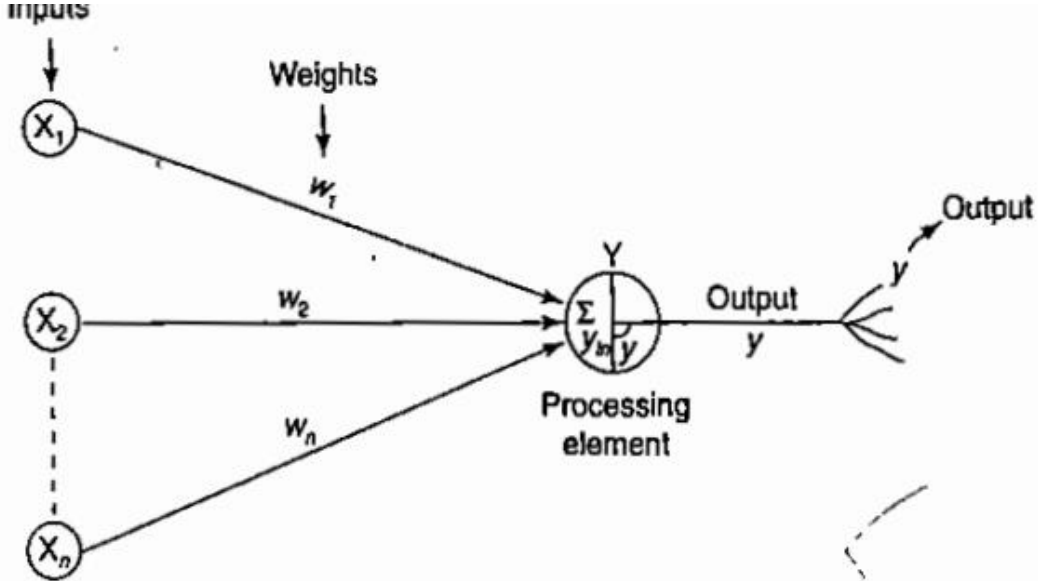


# BIOLOGICAL (MOTOR) NEURON



# ARTIFICIAL NEURAL NET

- Information-processing system.
- Neurons process the information.
- The signals are transmitted by means of connection links.
- The links possess an associated weight.
- The output signal is obtained by applying activations to the net input.



**Figure 2-5** Mathematical model of artificial neuron.

**Table 2-1** Terminology relationships between biological and artificial neurons

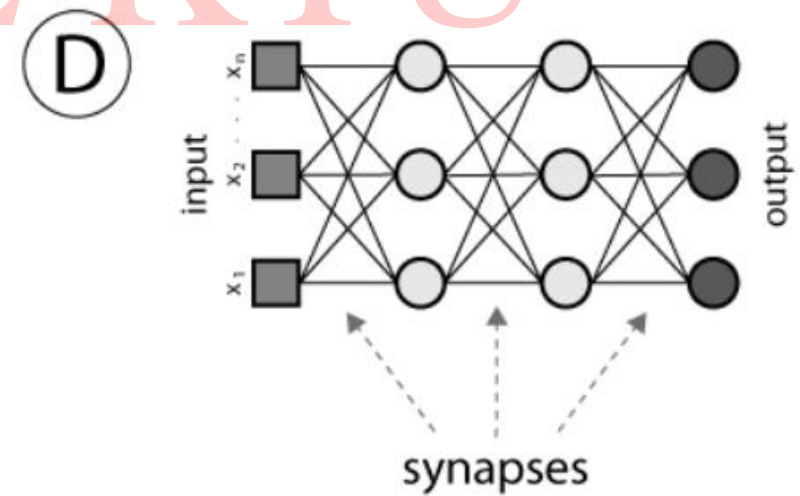
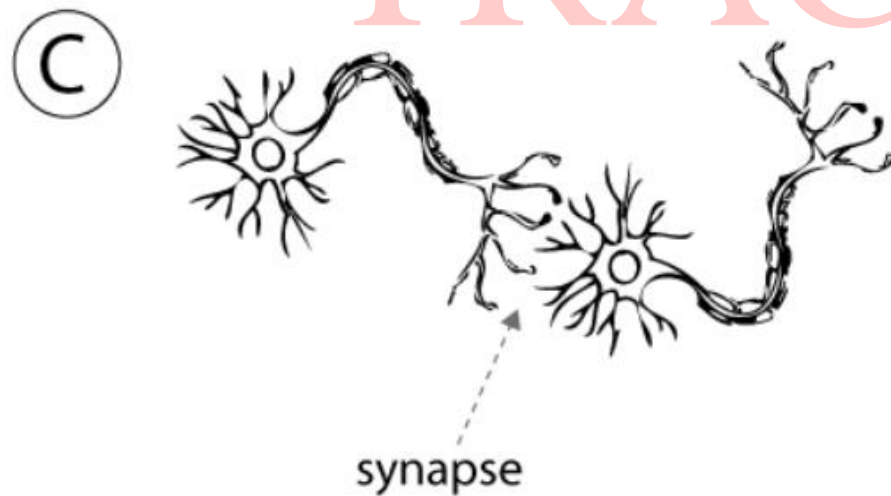
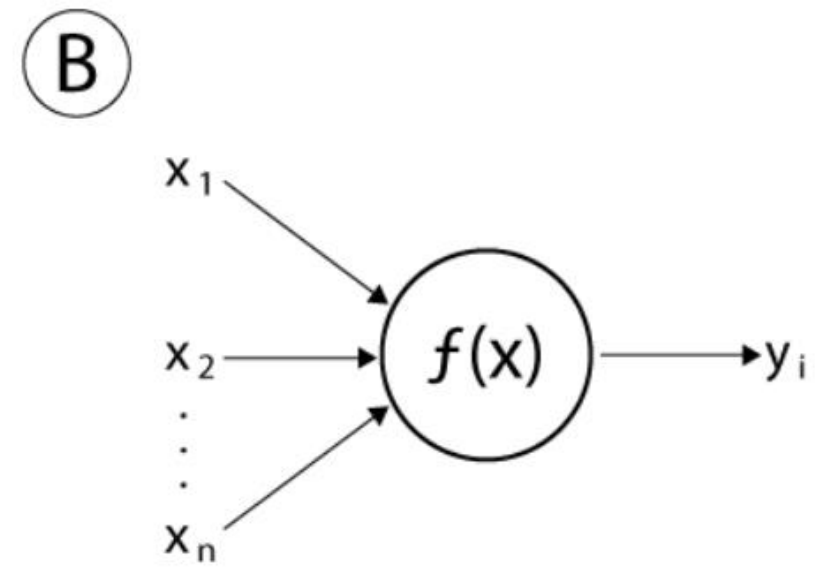
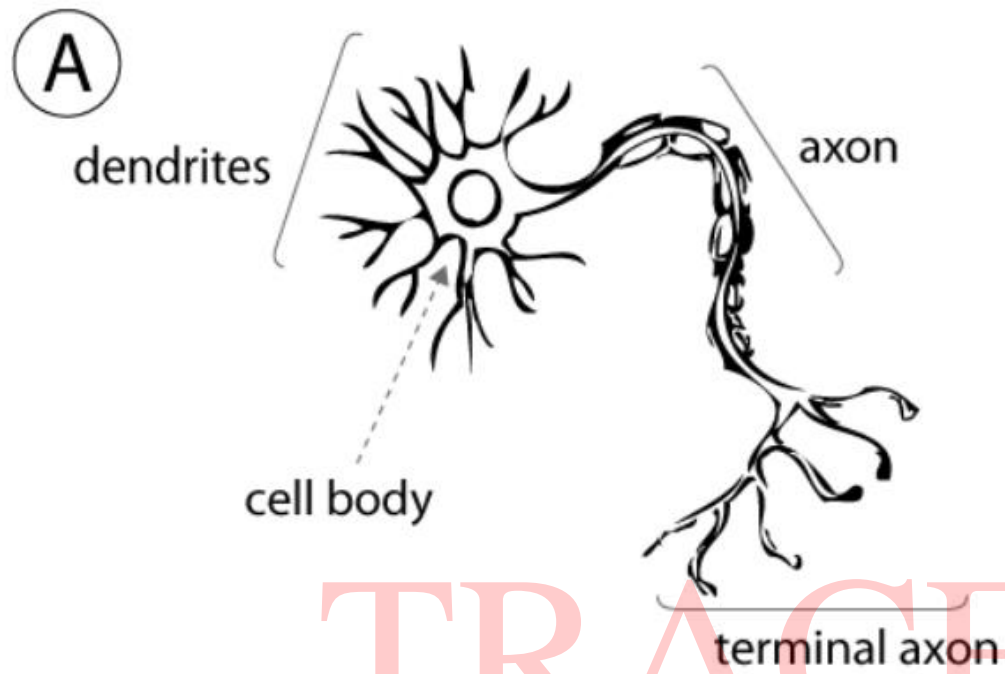
Biological neuron	Artificial neuron
Cell	Neuron
Dendrites	Weights or interconnections
Soma	Net input
Axon	Output

Figure 2-5 shows a mathematical representation of the above-discussed chemical processing taking place in an artificial neuron.

In this model, the net input is elucidated as

$$y_{in} = x_1 w_1 + x_2 w_2 + \dots + x_n w_n = \sum_{i=1}^n x_i w_i$$





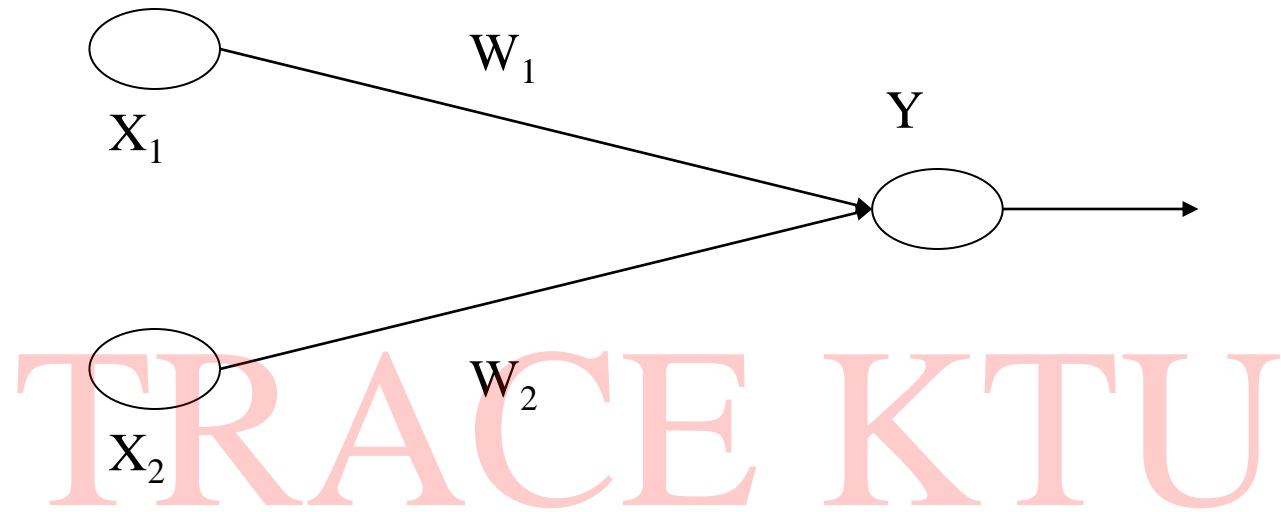
Synapses and Neurons in Neural Networks both Biological and Computational

## **The major areas being:**

- Massive parallelism
- Distributed representation and computation
- Learning ability
- Generalization ability
- Adaptivity
- Inherent contextual information processing
- Fault tolerance
- Low energy consumption.

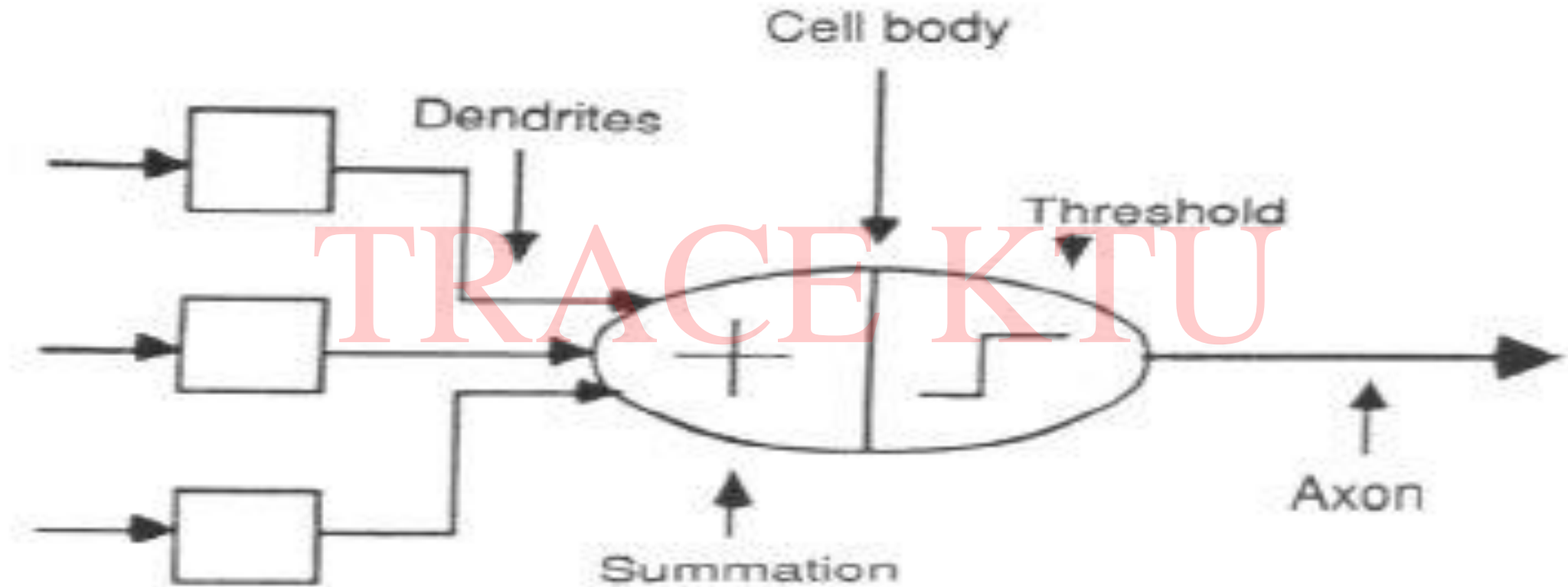
TRACE KTU

# ARTIFICIAL NEURAL NET



The figure shows a simple artificial neural net with two input neurons ( $X_1$ ,  $X_2$ ) and one output neuron ( $Y$ ). The inter connected weights are given by  $W_1$  and  $W_2$ .

# ASSOCIATION OF BIOLOGICAL NET WITH ARTIFICIAL NET



# PROCESSING OF AN ARTIFICIAL NET

The neuron is the basic information processing unit of a NN. It consists of:

1. A set of links, describing the neuron inputs, with weights  $W_1, W_2, \dots, W_m$ .
2. An adder function (linear combiner) for computing the weighted sum of the inputs (real numbers):

$$u = \sum_{j=1}^m W_j X_j$$

3. Activation function for limiting the amplitude of the neuron output.

# BIAS OF AN ARTIFICIAL NEURON

The bias value is added to the weighted sum

$\sum w_i x_i$  so that we can transform it from the origin.

$$Y_{in} = \sum w_i x_i + b, \text{ where } b \text{ is the bias}$$

TRACE KTU



# MULTI LAYER ARTIFICIAL NEURAL NET

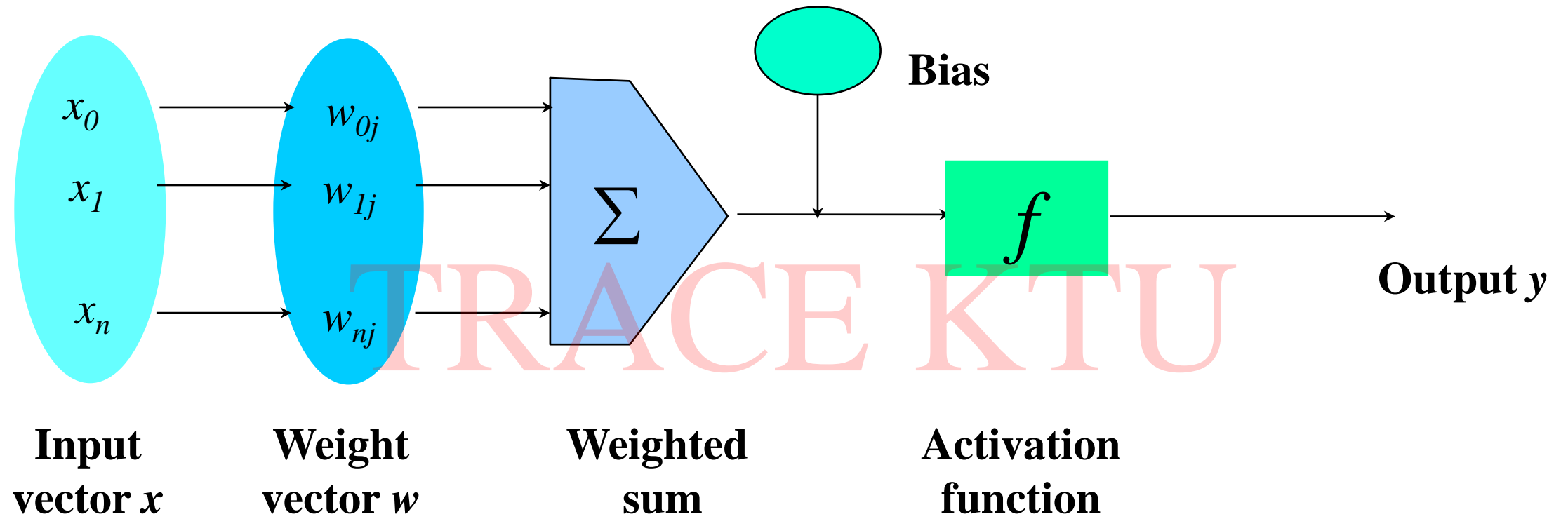
**INPUT:** records without class attribute with normalized attributes values.

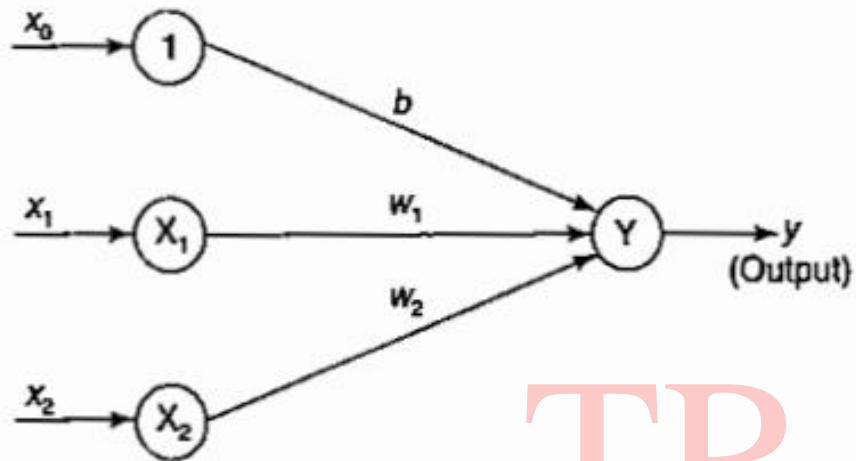
**INPUT VECTOR:**  $X = \{x_1, x_2, \dots, x_n\}$  where  $n$  is the number of (non-class) attributes.

**INPUT LAYER:** there are as many nodes as non-class attributes, i.e. as the length of the input vector.

**HIDDEN LAYER:** the number of nodes in the hidden layer and the number of hidden layers depends on implementation.

# OPERATION OF A NEURAL NET





**Figure 2-19** A single-layer neural net.

$$y_{inj} = \sum_{i=0}^n x_i w_{ij} = x_0 w_{0j} + x_1 w_{1j} + x_2 w_{2j} + \cdots + x_n w_{nj}$$

$$= w_{0j} + \sum_{i=1}^n x_i w_{ij}$$

$$y_{inj} = b_j + \sum_{i=1}^n x_i w_{ij}$$

The bias included in the network has its impact in calculating the net input. The bias is included by adding a component  $x_0 = 1$  to the input vector  $X$ . Thus, the input vector becomes

$$X = (1, X_1, \dots, X_i, \dots, X_n)$$

# WEIGHT AND BIAS UPDATION

## Per Sample Updating

- updating weights and biases after the presentation of each sample.

## Per Training Set Updating (Epoch or Iteration)

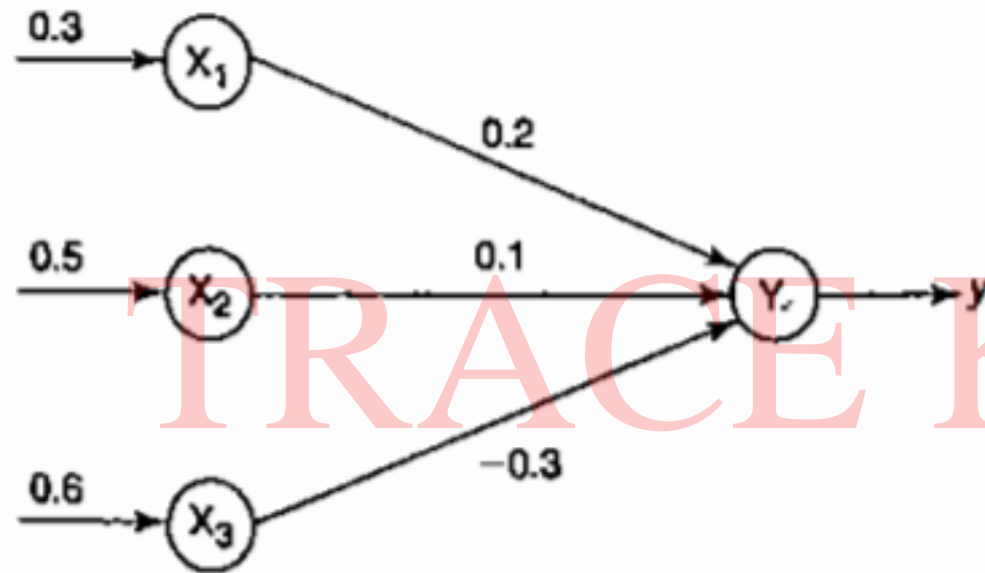
- weight and bias increments could be accumulated in variables and the weights and biases updated after all the samples of the training set have been presented.

# STOPPING CONDITION

- All change in weights ( $\Delta w_{ij}$ ) in the previous epoch are below some threshold, or
- The percentage of samples misclassified in the previous epoch is below some threshold, or
- A pre-specified number of epochs has expired.
- In practice, several hundreds of thousands of epochs may be required before the weights will converge.

TRACE KTU

1. For the network shown in Figure 1, calculate the net input to the output neuron.



**Figure 1** Neural net.



# BUILDING BLOCKS OF ARTIFICIAL NEURAL NET

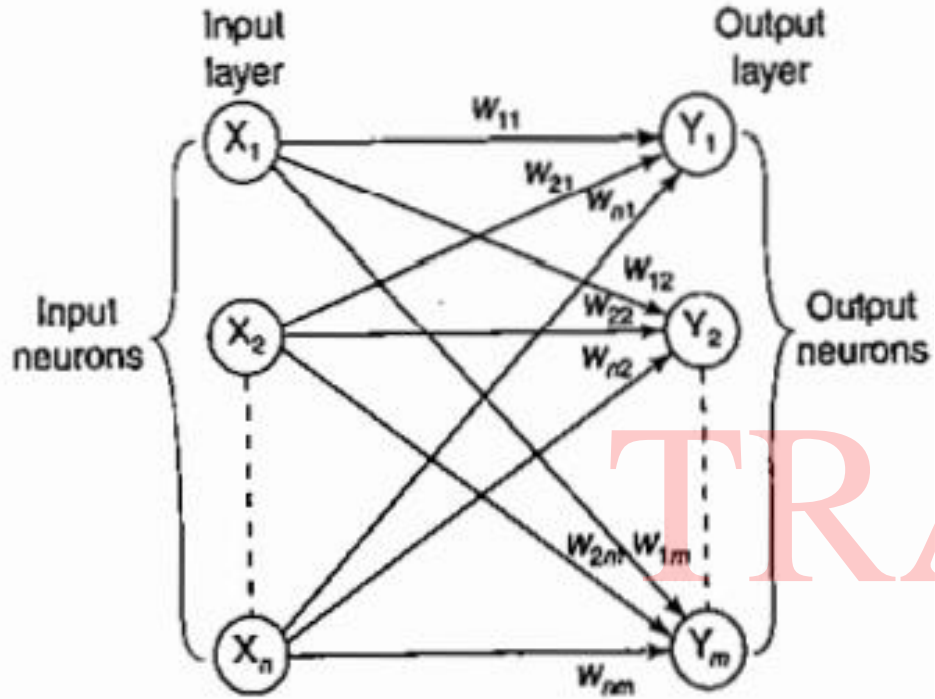
## ➤ Network Architecture (Connection between Neurons)

- 1. Single-layer feed-forward network
- 2. Multilayer feed-forward network
- 3. Single node with its own feedback
- 4. Single-layer Recurrent network
- 5. Multilayer Recurrent network

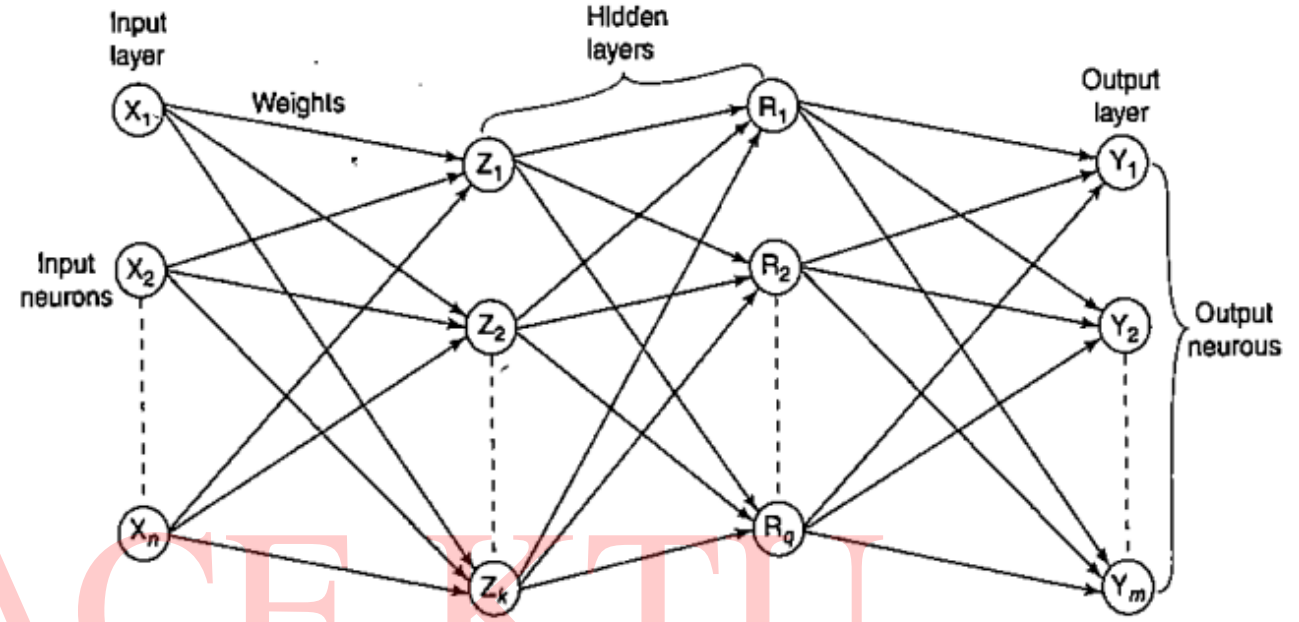
## ➤ Setting the Weights (Training)

## ➤ Activation Function

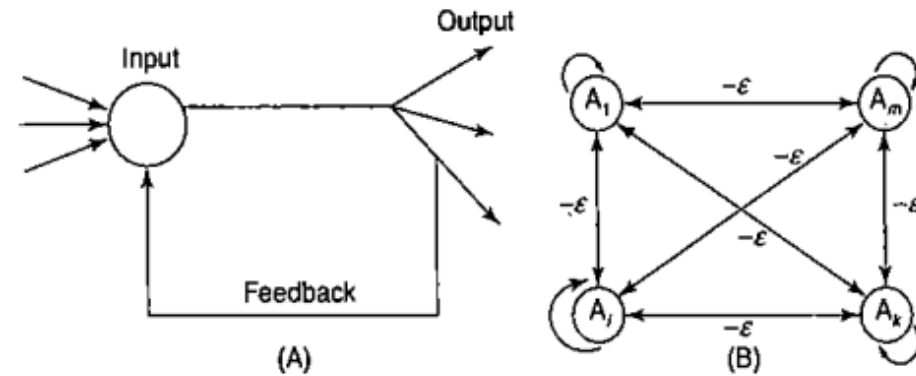
TRACE KTU



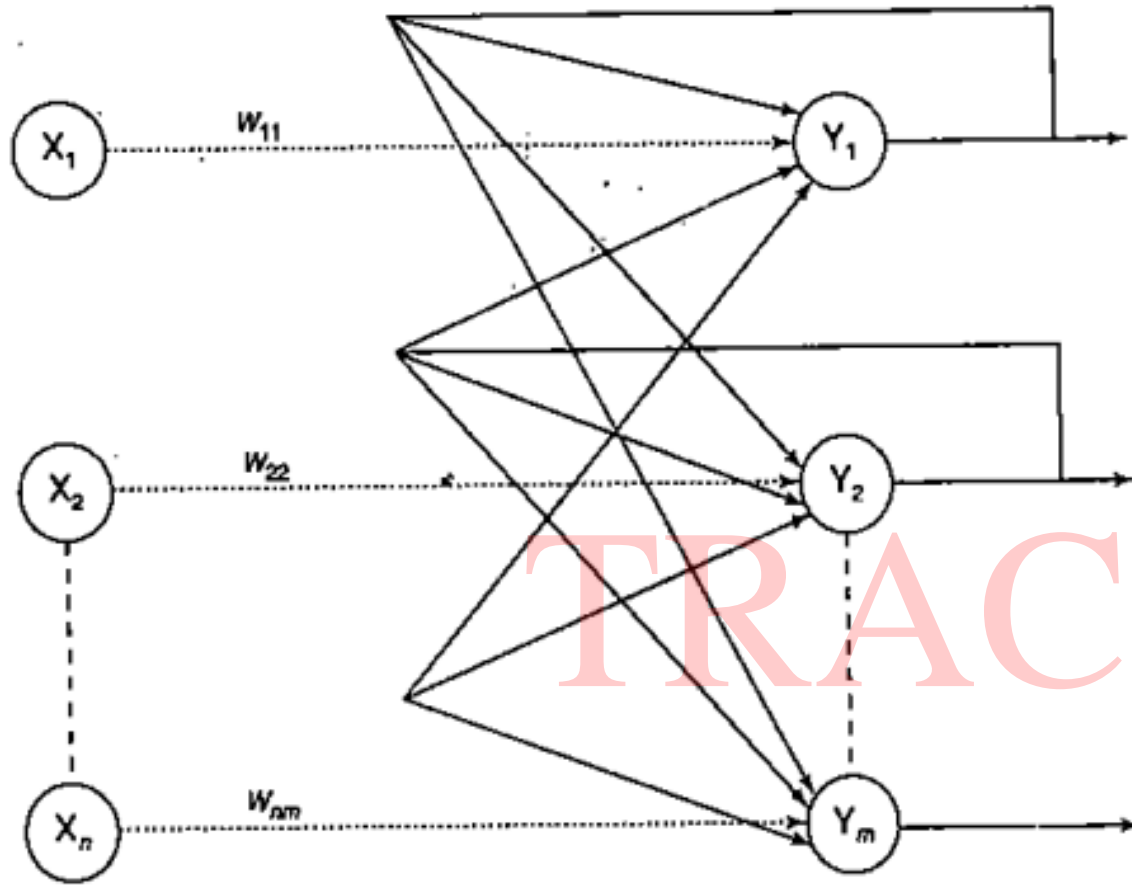
**Figure 2-6** Single-layer feed-forward network.



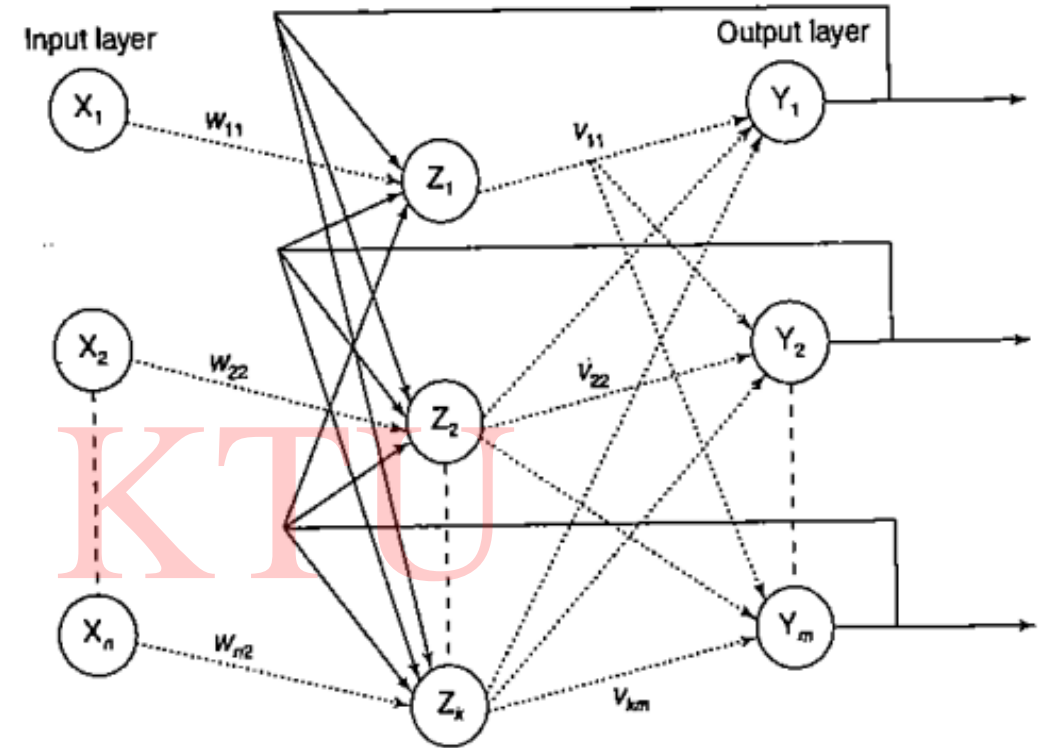
**Figure 2-7** Multilayer feed-forward network.



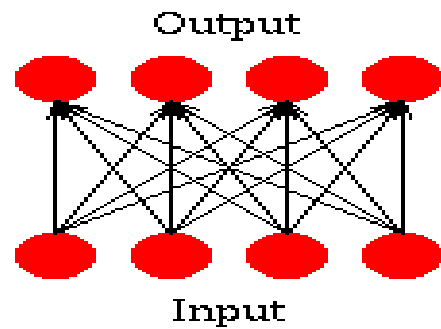
**Figure 2-8** (A) Single node with own feedback. (B) Competitive nets.



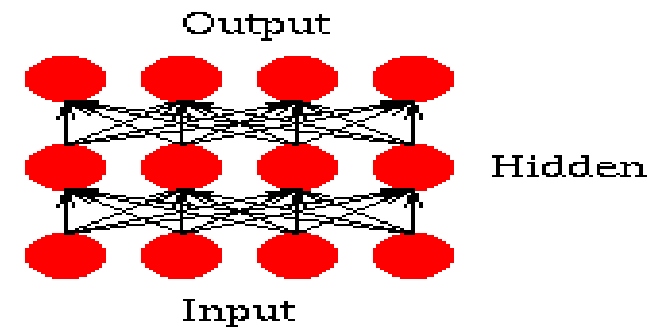
**Figure 2-9** Single-layer recurrent network.



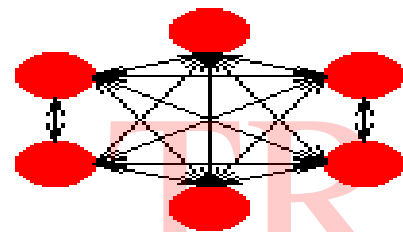
**Figure 2-10** Multilayer recurrent network.



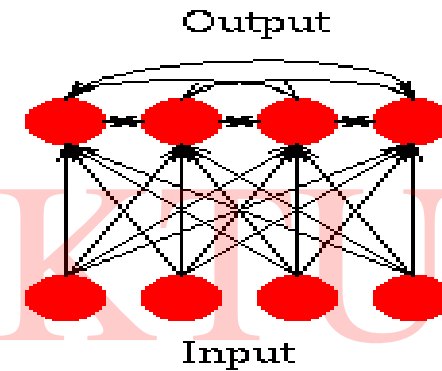
Single Layer Feedforward



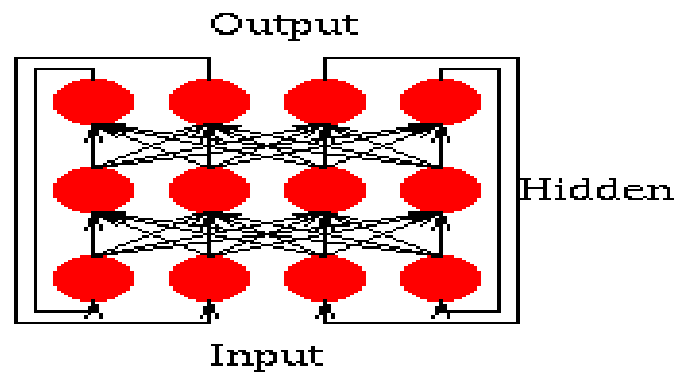
Multi Layer Feedforward



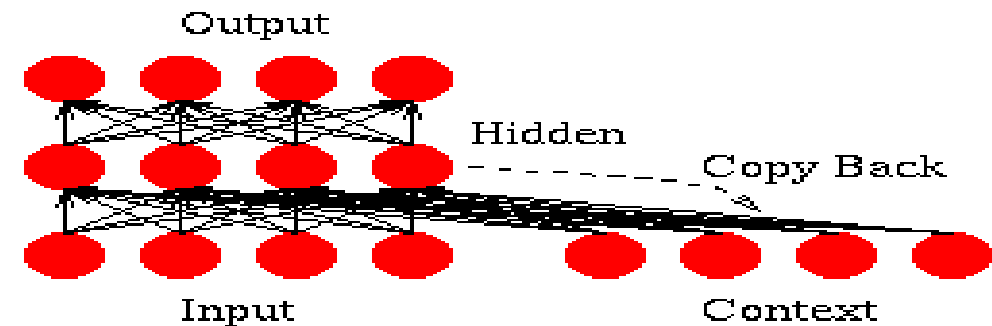
Fully Recurrent Network



Competitive Network



Jordan Network



Simple Recurrent Network

# LAYER PROPERTIES

- **Input Layer:** Each input unit may be designated by an attribute value possessed by the instance.
- **Hidden Layer:** Not directly observable, provides nonlinearities for the network.
- **Output Layer:** Encodes possible values.

TRACE KTU

# TRAINING PROCESS

- **Supervised Training** - Providing the network with a series of sample inputs and comparing the output with the expected responses.
- **Unsupervised Training** - Most similar input vector is assigned to the same output unit.
- **Reinforcement Training** - Right answer is not provided but indication of whether 'right' or 'wrong' is provided.

TRACE KTU



# ACTIVATION FUNCTION

## ➤ ACTIVATION LEVEL – DISCRETE OR CONTINUOUS

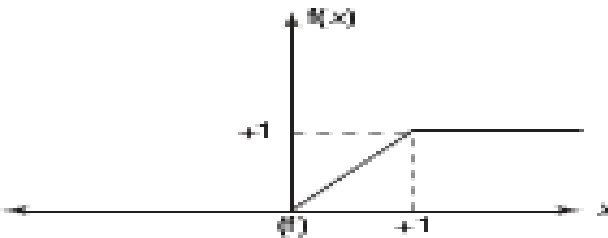
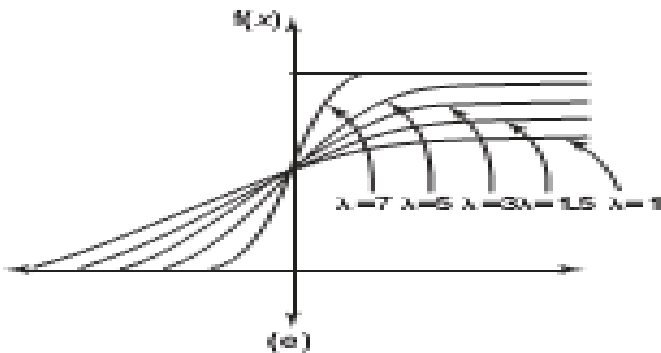
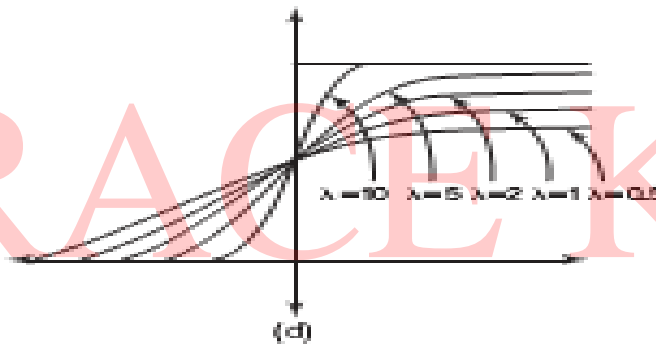
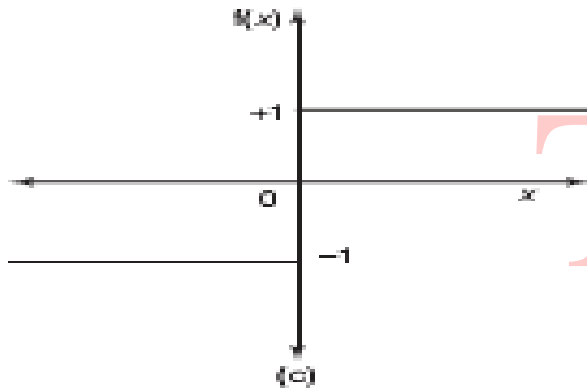
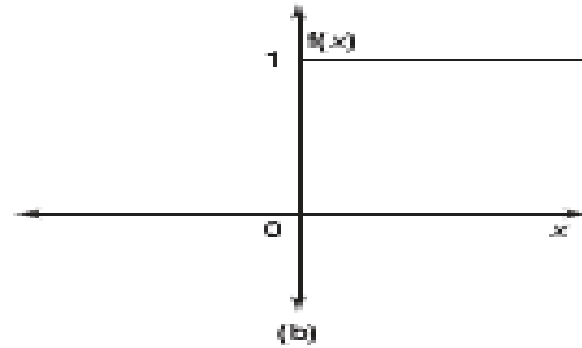
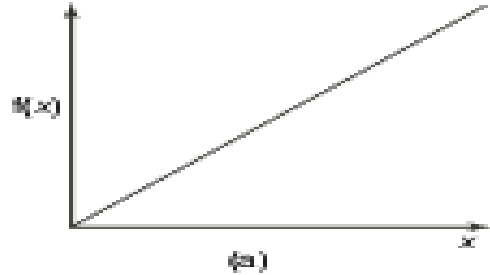
### ➤ HARD LIMIT FUNCTION (DISCRETE)

- Binary Activation function
- Bipolar activation function
- Identity function

### ➤ SIGMOIDAL ACTIVATION FUNCTION (CONTINUOUS)

- Binary Sigmoidal activation function
- Bipolar Sigmoidal activation function

# ACTIVATION FUNCTION



## Activation functions:

(A) Identity

(B) Binary step

(C) Bipolar step

(D) Binary sigmoidal

(E) Bipolar sigmoidal

(F) Ramp

1. *Identity function*: It is a linear function and can be defined as

$$f(x) = x \text{ for all } x$$

2. *Binary step function*: This function can be defined as

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

where  $\theta$  represents the threshold value. This function is most widely used in single-layer nets to convert the net input to an output that is a binary (1 or 0).

3. *Bipolar step function*: This function can be defined as

$$f(x) = \begin{cases} 1 & \text{if } x \geq \theta \\ -1 & \text{if } x < \theta \end{cases}$$

where  $\theta$  represents the threshold value. This function is also used in single-layer nets to convert the net input to an output that is bipolar (+1 or -1).

4. *Sigmoidal functions*: The sigmoidal functions are widely used in back-propagation nets because of the relationship between the value of the functions at a point and the value of the derivative at that point which reduces the computational burden during training.

Sigmoidal functions are of two types:

- *Binary sigmoid function*: It is also termed as logistic sigmoid function or unipolar sigmoid function. It can be defined as

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

where  $\lambda$  is the steepness parameter. The derivative of this function is

$$f'(x) = \lambda f(x)[1 - f(x)]$$

Here the range of the sigmoid function is from 0 to 1.

- *Bipolar sigmoid function*: This function is defined as

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

where  $\lambda$  is the steepness parameter and the sigmoid function range is between  $-1$  and  $+1$ . The derivative of this function can be

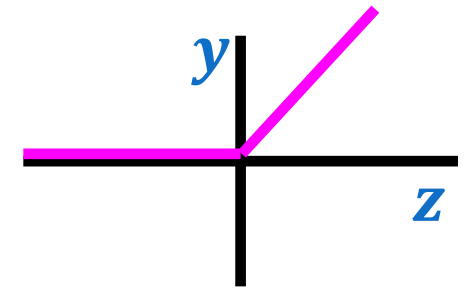
$$f'(x) = \frac{\lambda}{2}[1 + f(x)][1 - f(x)]$$

5. *Ramp function:* The ramp function is defined as

$$f(x) = \begin{cases} 1 & \text{if } x > 1 \\ x & \text{if } 0 \leq x \leq 1 \\ 0 & \text{if } x < 0 \end{cases}$$

TRACE KTU

# Rectified Linear Unit (ReLU)



- Activation function

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

- Derivative

$$\frac{\partial y}{\partial z} = \begin{cases} 0 & z \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

- Advantages

- fast to compute activation and derivatives
  - no squashing of back propagated error signal as long as unit is activated
  - discontinuity in derivative at  $z=0$
  - sparsity ?

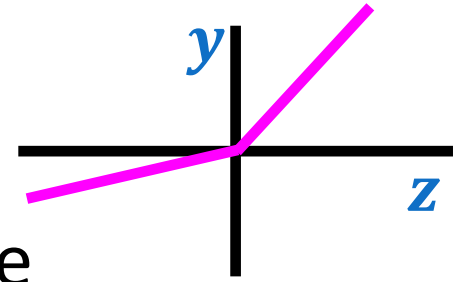
- Disadvantages

- can potentially lead to exploding gradients and activations
  - may waste units: units that are never activated above threshold won't learn



- This is one of the most widely used activation function.
- The benefits of ReLU is the sparsity, it allows only values which are positive and negative values are not passed which will speed up the process and it will negate or bring down possibility of occurrence of a dead neuron.
- $f(x) = \max(0, x)$
- Derivative:  $f'(x) = \begin{cases} 1; x > 0, \\ 0; x < 0 \end{cases}$
- This function will allow only the maximum values to pass during the front propagation

# Leaky ReLU



- Activation function

Derivative

$$y = \begin{cases} z & z > 0 \\ \alpha z & \text{otherwise} \end{cases}$$

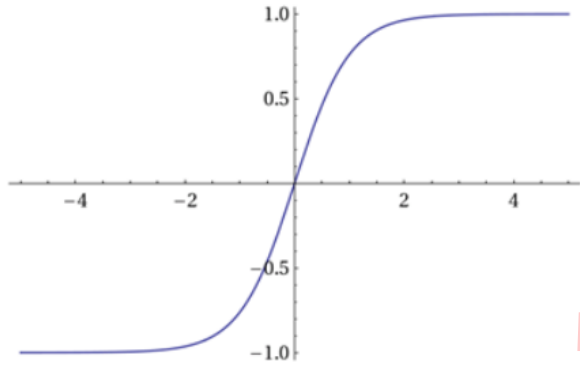
$$\frac{\partial y}{\partial z} = \begin{cases} 1 & z > 0 \\ \alpha & \text{otherwise} \end{cases}$$

- Reduces to standard ReLU if  $\alpha = 0$
- Trade off
  - $\alpha = 0$  leads to inefficient use of resources (underutilized units)
  - $\alpha = 1$  lose nonlinearity essential for interesting computation

**Leaky ReLU** where slope is changed left of  $x=0$  in above figure and thus causing a **leak** and *extending* the *range* of ReLU.

## I. Tanh / Hyperbolic Tangent Activation Function [🔗](#)

The activation that almost always works better than sigmoid function is Tanh function also known as Tangent Hyperbolic function. It's actually a mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.



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

Equation:  $f(x) = a = \tanh(x) = 2 / (1 + e^{-2x}) - 1 = 2 * \text{sigmoid}(2x) - 1$

Derivative:  $(1 - a^2)$

Value Range: -1 to +1

### Advantages:

If you compare it to sigmoid, it solves just one problem of being Zero centred — it's easier to model inputs that have strongly negative, neutral, and strongly positive values.

Works better than sigmoid function

### Disadvantage:

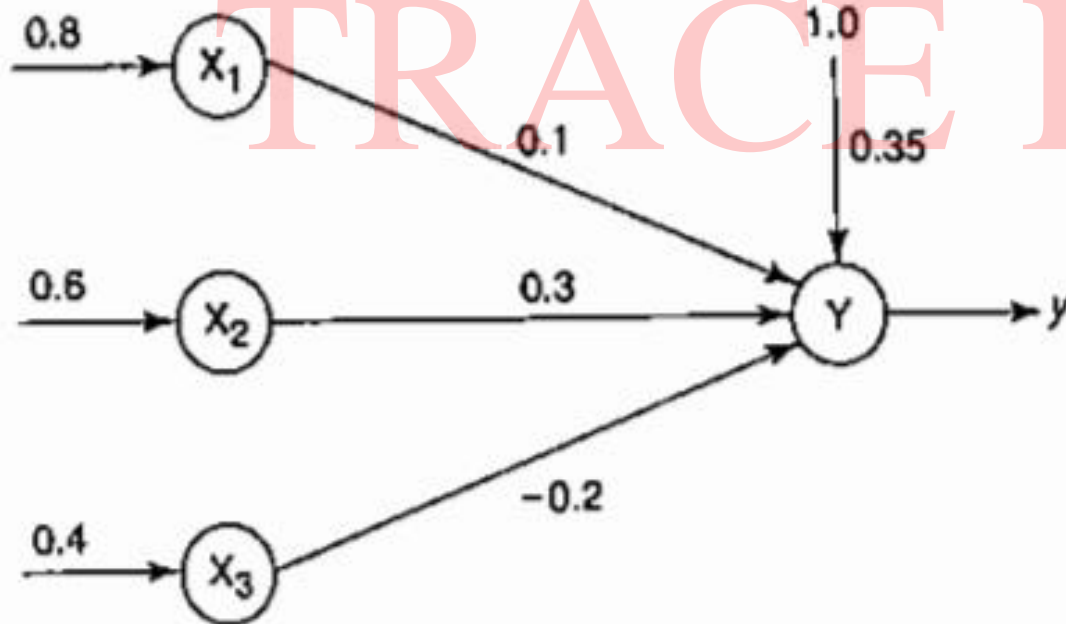
It also suffers from the vanishing gradient problem and hence slow convergence.

### Uses:

Can be used in hidden layers of a neural network as its values lie between -1 to 1 . hence the mean for the hidden layer is centered and comes out to be 0 or very close to it. This makes learning for the next layer much easier.

# Problems

3. Obtain the output of the neuron Y for the network shown in Figure 3 using activation functions as: (i) binary sigmoidal and (ii) bipolar sigmoidal.



# CONSTRUCTING ANN

- Determine the network properties:
  - Network topology
  - Types of connectivity
  - Order of connections
  - Weight range
- Determine the node properties:
  - Activation range
- Determine the system dynamics
  - Weight initialization scheme
  - Activation – calculating formula
  - Learning rule

# PROBLEM SOLVING

- Select a suitable NN model based on the nature of the problem.
- Construct a NN according to the characteristics of the application domain.
- Train the neural network with the learning procedure of the selected model.
- Use the trained network for making inference or solving problems.

TRACE KTU

# NEURAL NETWORKS

- **Neural Network** learns by adjusting the weights so as to be able to correctly classify the training data and hence, after testing phase, to classify unknown data.
- **Neural Network** needs long time for training.
- **Neural Network** has a high tolerance to noisy and incomplete data.

TRACE KTU

# SALIENT FEATURES OF ANN

- Adaptive learning
- Self-organization
- Real-time operation
- Fault tolerance via redundant information coding
- Massive parallelism
- Learning and generalizing ability
- Distributed representation

TRACE KTU

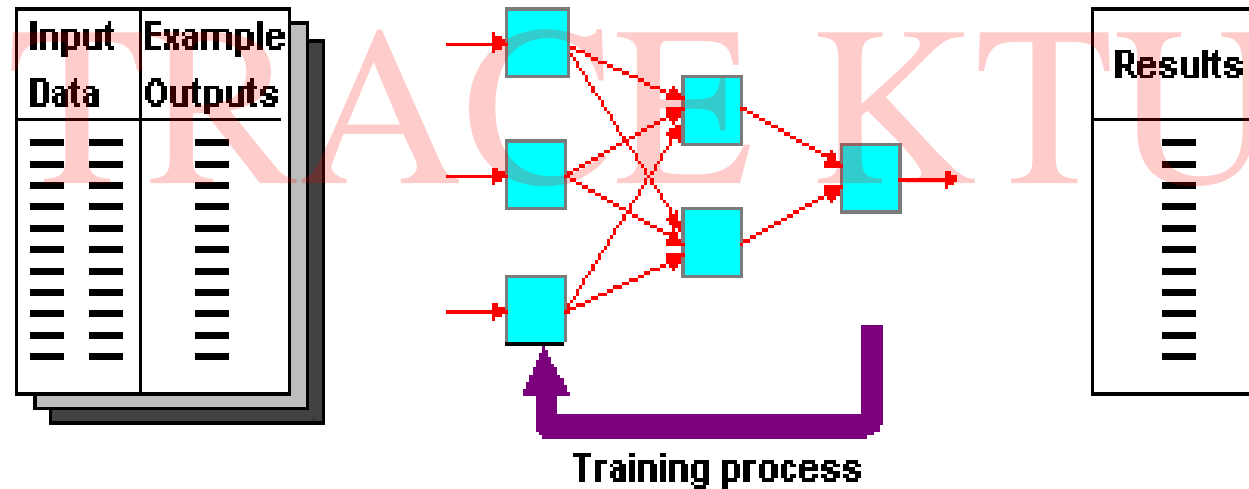


# FEW APPLICATIONS OF NEURAL NETWORKS

- Aerospace
- Automotive
- Banking
- Credit Card Activity Checking
- Defense
- Electronics
- Entertainment
- Financial
- Industrial
- Insurance
- Insurance
- Manufacturing
- Medical
- Oil and Gas
- Robotics
- Speech
- Securities
- Telecommunications
- Transportation

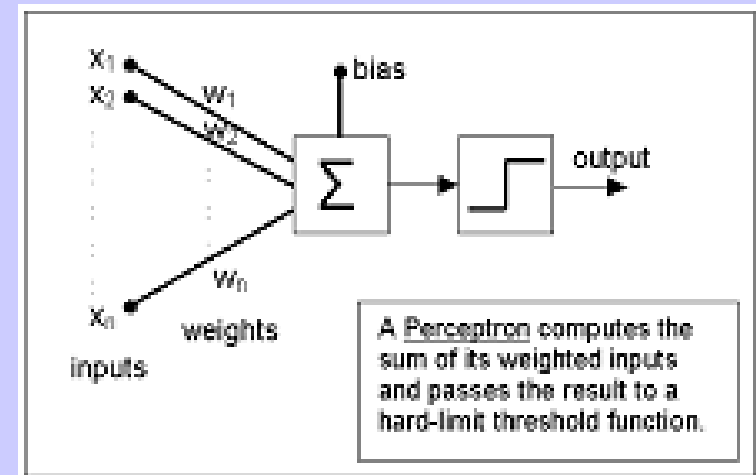
# DEFINITION OF SUPERVISED LEARNING NETWORKS

- Training and test data sets
- Training set; input & target are specified



# PERCEPTRON NETWORKS

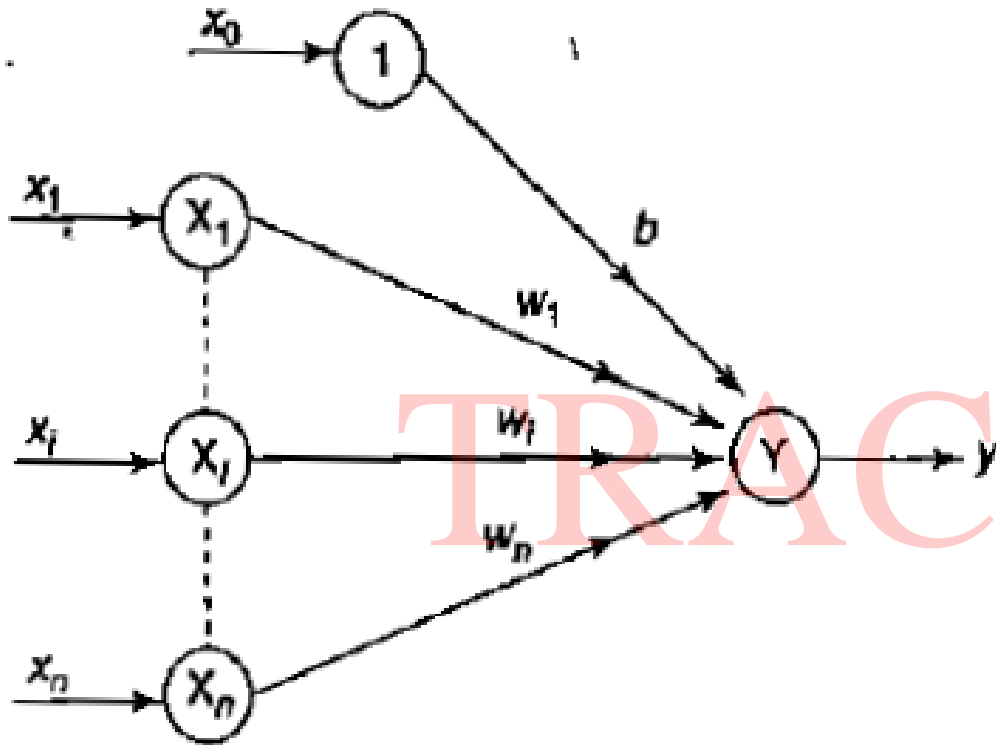
- One type of NN system is based on the “perceptron”.
- A perceptron computes a sum of weighted combination of its inputs, if the sum is greater than a certain threshold (bias), then it outputs a “1”, else a “-1”.



## Perceptron Training Algorithm For Single Output Class

- The perceptron algorithm can be used for
  - either binary or bipolar input vectors,
  - having bipolar targets, threshold being fixed and variable bias.
  - In the algorithm,
    - initially the inputs are assigned.
    - Then the net input is calculated.
    - The output of the network is obtained by applying the activation function over the calculated net input.
    - On performing comparison over the calculated and the desired output, the weight updation process is carried out.
    - The entire network is trained based on the mentioned stopping criterion.

## Architecture



**Figure 3-2** Single classification perceptron network.

- N input neurons
- 1 output Neuron and a bias

inputs and bipolar targets is shown in

**Table 3**

$x_1$	$x_2$	$t$
1	1	1
1	0	1
0	1	1
0	0	-1

**Table 1**

$x_1$	$x_2$	$t$
1	1	1
1	-1	-1
-1	1	-1
-1	-1	-1

## ALGORITHM

Step 0: Initialize the weights and the bias. (0) and learning rate  $\alpha$  ( $0 < \alpha \leq 1$ ).

*For simplicity  $\alpha$  is set to 1.*

Step 1: Perform Steps 2-6 until the final stopping condition *is false*.

Step 2: Perform Steps 3-5 for each training pair indicated by  $s:t$ .

Step 3: The input layer containing input units is applied with identity activation functions:

$$X_i = S_i$$

Step 4: Calculate the output of the network. To do so, first obtain the net input:

$$y_{in} = b + \sum_{i=1}^n x_i w_i$$

where " $n$ " is the *number of input neurons in the input layer*. Then apply *activations over the net* input calculated to obtain the output

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

Step 5: *Weight and bias adjustment: Compare the value of the actual (calculated) output and desired (target) output.*

– If  $y \neq t$ , then

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

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

• else, we have

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

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

Step 6: Train the network until there is no weight change. This is the stopping condition for the network.

If this condition is not met, then start again from Step 2.



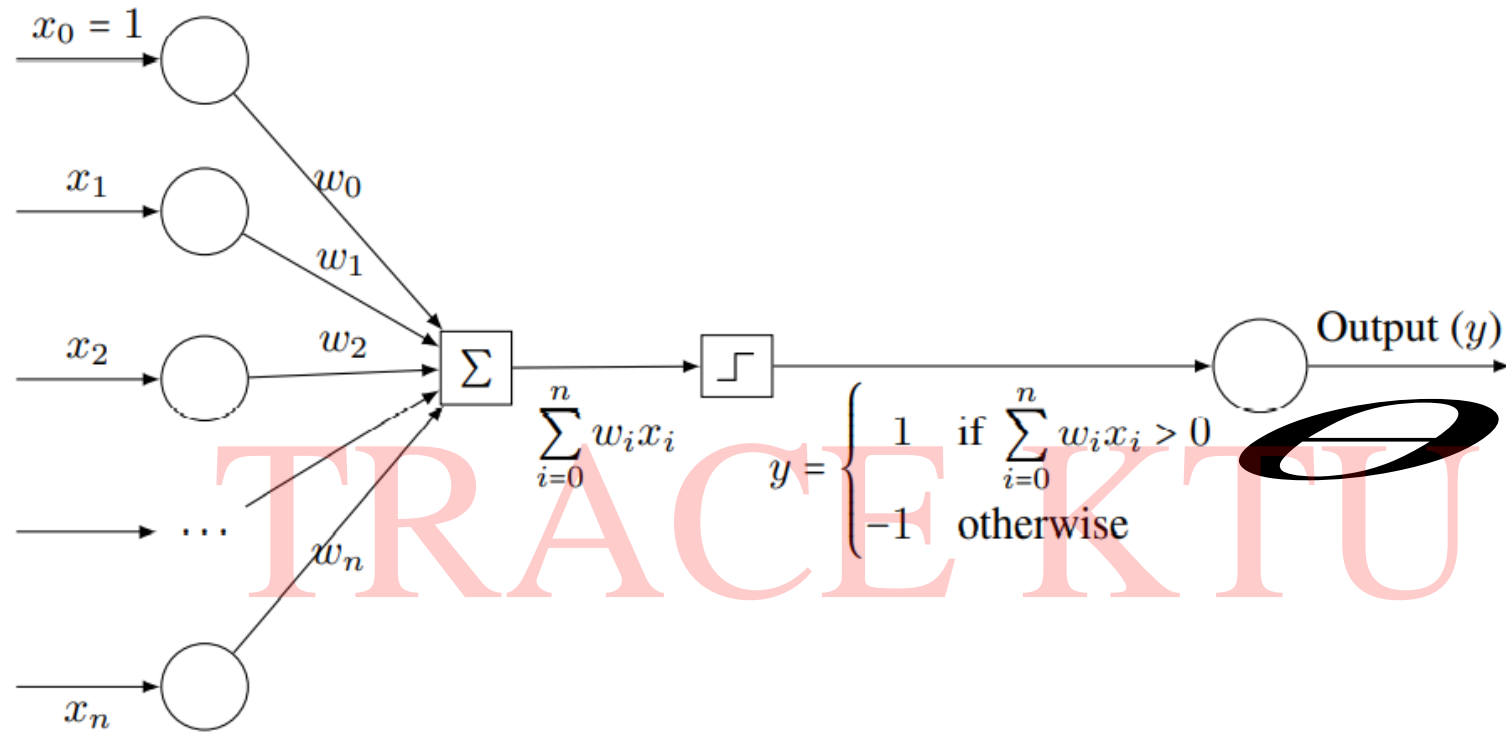


Figure 9.12: Schematic representation of a perceptron

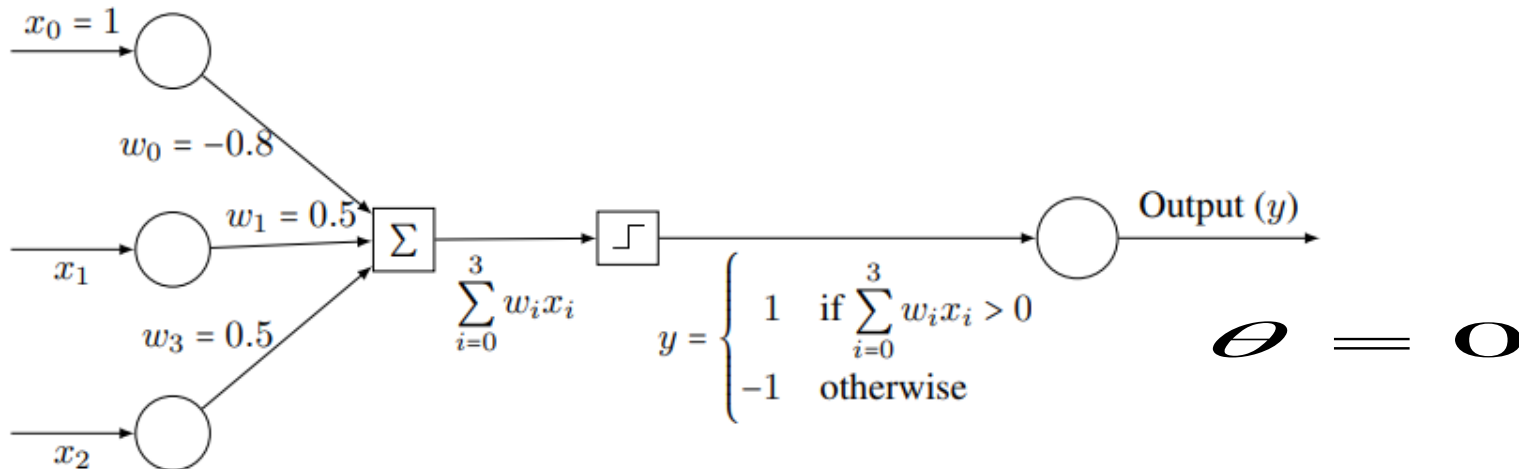
### 9.5.3 Representation of $x_1$ AND $x_2$

Let  $x_1$  and  $x_2$  be two boolean variables. Then the boolean function  $x_1$  AND  $x_2$  is represented by Table 9.1. It can be easily verified that the perceptron shown in Figure 9.13 represents the function

$x_1$	$x_2$	$x_1$ AND $x_2$
-1	-1	-1
-1	1	-1
1	-1	-1
1	1	1

Table 9.1: The boolean function  $x_1$  AND  $x_2$

$x_1$  AND  $x_2$ .



Suppose theta 0

Figure 9.13: Representation of  $x_1$  AND  $x_2$  by a perceptron

### 9.5.5 ~~Perceptron learning algorithm~~

#### Definitions

#### (Adaline Network Algorithm)

In the algorithm, we use the following notations:

$n$	:	Number of input variables
$y = f(\mathbf{z})$	:	Output from the perceptron for an input vector $\mathbf{z}$
$D = \{(\mathbf{x}_1, d_1), \dots, (\mathbf{x}_s, d_s)\}$	:	Training set of $s$ samples
$\mathbf{x}_j = (x_{j0}, x_{j1}, \dots, x_{jn})$	:	The $n$ -dimensional input vector
$d_j$	:	Desired output value of the perceptron for the input $\mathbf{x}_j$
$x_{ji}$	:	Value of the $i$ -th feature of the $j$ -th training input vector
$x_{j0}$	:	1
$w_i$	:	Weight of the $i$ -th input variable
$w_i(t)$	:	Weight $i$ at the $t$ -th iteration

## Algorithm

Step 1. Initialize the weights and the threshold. Weights may be initialized to 0 or to a small random value.

Step 2. For each example  $j$  in the training set  $D$ , perform the following steps over the input  $\mathbf{x}_j$  and desired output  $d_j$ :

a) Calculate the actual output:

$$y_j(t) = f[w_0(t)x_{j0} + w_1(t)x_{j1} + w_2(t)x_{j2} + \cdots + w_n(t)x_{jn}]$$

b) Update the weights:

$$w_i(t+1) = w_i(t) + (d_j - y_j(t))x_{ji}$$

for all features  $0 \leq i \leq n$ .

Step 3. Step 2 is repeated until the iteration error  $\frac{1}{s} \sum_{j=1}^s |d_j - y_j(t)|$  is less than a user-specified error threshold  $\gamma$ , or a predetermined number of iterations have been completed, where  $s$  is again the size of the sample set.

## Remarks

The above algorithm can be applied only if the training examples are *linearly separable*.

## PERCEPTRON LEARNING RULE

- Learning Signal
  - Difference between the desired and actual response of the pattern.
- Learning Rule:
  - Consider a finite " $n$ " *number of input training vectors, with their associated target values,  $x(n)$  and  $t(n)$ ,*
    - where " $n$ " ranges from 1 to  $N$ .
    - The target is either +1 or -1.
    - The output " $y$ " is obtained on the basis of the net input calculated and activation function being applied over the net input.

# Perceptron Learning

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = \alpha t x_i$$

where

$t = c(x)$  is the target value,

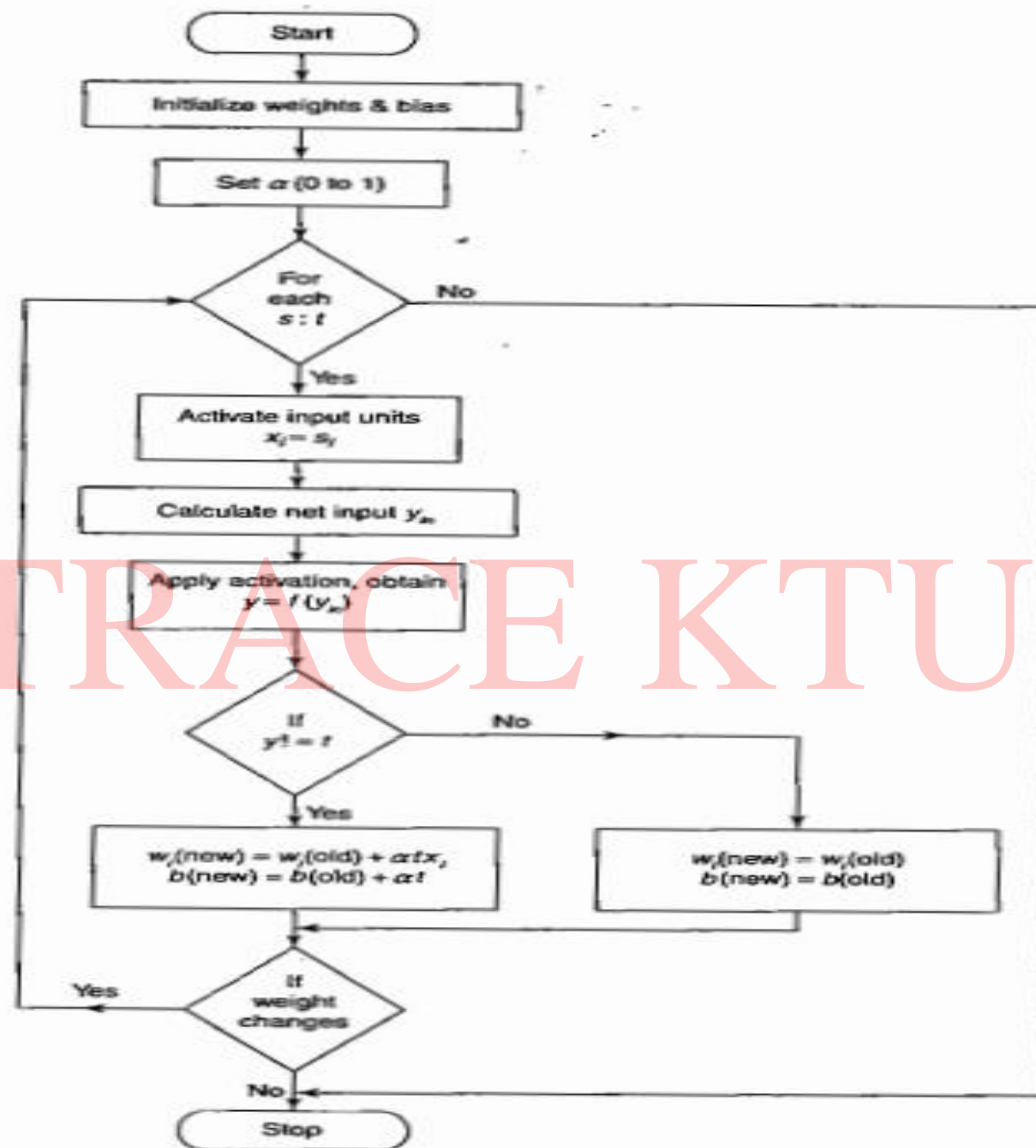
$o$  is the perceptron output,

$\alpha$  Is a small constant (e.g., 0.1) called learning rate.

- If the output is correct ( $t = o$ ) the weights  $w_i$  are not changed
- If the output is incorrect ( $t \neq o$ ) the weights  $w_i$  are changed such that the output of the perceptron for the new weights is closer to  $t$ .
- The algorithm converges to the correct classification
  - if the training data is linearly separable
  - $\alpha$  is sufficiently small

## FLOW CHART FOR TRAINING PROCESS

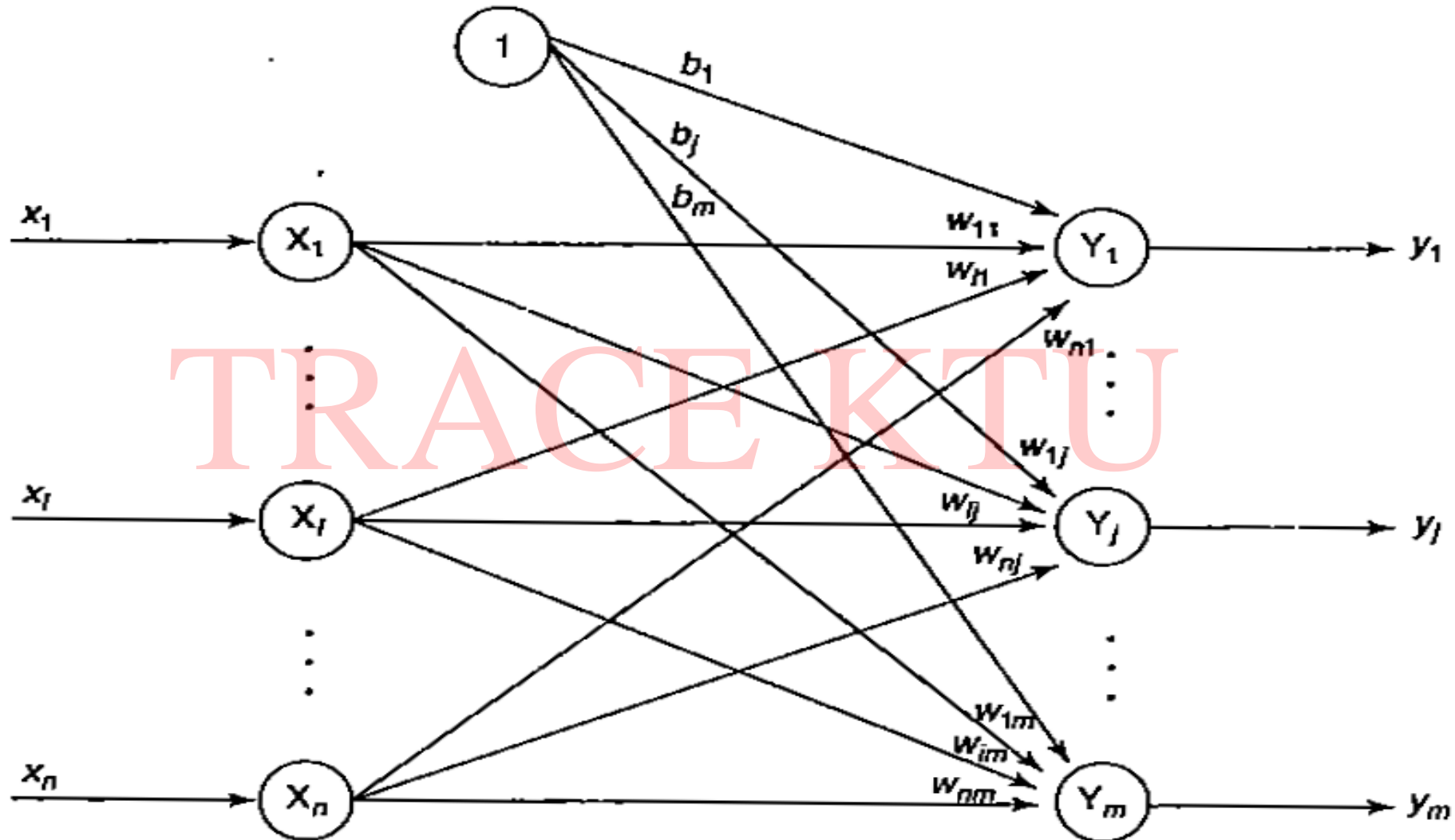
- The network has to be suitably trained to obtain the response.
- The entire loop of the training process continues until the training input pair is presented to the network.
- *The training (weight updation) is done on the basis of the comparison between the calculated and desired output.*
- The loop is terminated if there is no change in weight.



**Figure 3-3** Flowchart for perceptron network with single output.



## PERCEPTRON TRAINING ALGORITHM FOR MULTIPLE OUTPUT CLASSES



**Figure 3-4** Network architecture for perceptron network for several output classes.

# LAYERS IN NEURAL NETWORK

## ➤ The input layer:

- Introduces input values into the network.
- No activation function or other processing.

## ➤ The hidden layer(s):

- Performs classification of features.
- Two hidden layers are sufficient to solve any problem.
- Features imply more layers may be better.

## ➤ The output layer:

- Functionally is just like the hidden layers.
- Outputs are passed on to the world outside the neural network.

# PERCEPTRON TRAINING ALGORITHM FOR MULTIPLE OUTPUT CLASSES

Step 0: Initialize the weights, biases and learning rate suitably.

Step 1: Check for stopping condition; if it is false, perform Steps 2-6.

Step 2: Perform Steps 3-5 for each bipolar or binary training vector pair  $s:t$

Step 3: Set activation (identity) of each input unit  $i = 1$  to  $n$ :

$$x_i = s_i$$

Step 4 : Calculate output response of each output unit  $j=1$  to  $m$ .

First the net input is calculated as

$$y_{inj} = b_j + \sum_{i=1}^n x_i w_{ij}$$

Then activations are applied over the net input to calculate the output response.

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

Step 5: Make adjustments in weights and bias for  $j=1$  to  $m$  and  $i=1$  to  $n$

If  $t_j \neq y_j$ , then

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha t_j x_i$$

$$b_j(\text{new}) = b_j(\text{old}) + \alpha t_j$$

else, we have

$$w_{ij}(\text{new}) = w_{ij}(\text{old})$$

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

Step 6: Test for the stopping condition, i.e., if there is no change in weights then stop the training process, else start again from Step 2.

# BACK PROPAGATION NETWORK

- A training procedure which allows **multilayer feed forward Neural Networks to be trained.**
- Consisting of processing elements with continuous differentiable activation functions.
- networks associated with back-propagation learning algorithm are *so called back propagation network*. (BPNs)
- Can theoretically perform “any” input-output mapping.
- Can learn to solve linearly inseparable problems.

- *For a given set* of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given **input patterns correctly**.
- The aim of the neural network is to **train the** net to achieve a balance between the net's ability to respond (memorization) and its ability to give reasonable responses to the input that is similar but not identical to the one that is used in training (generalization).

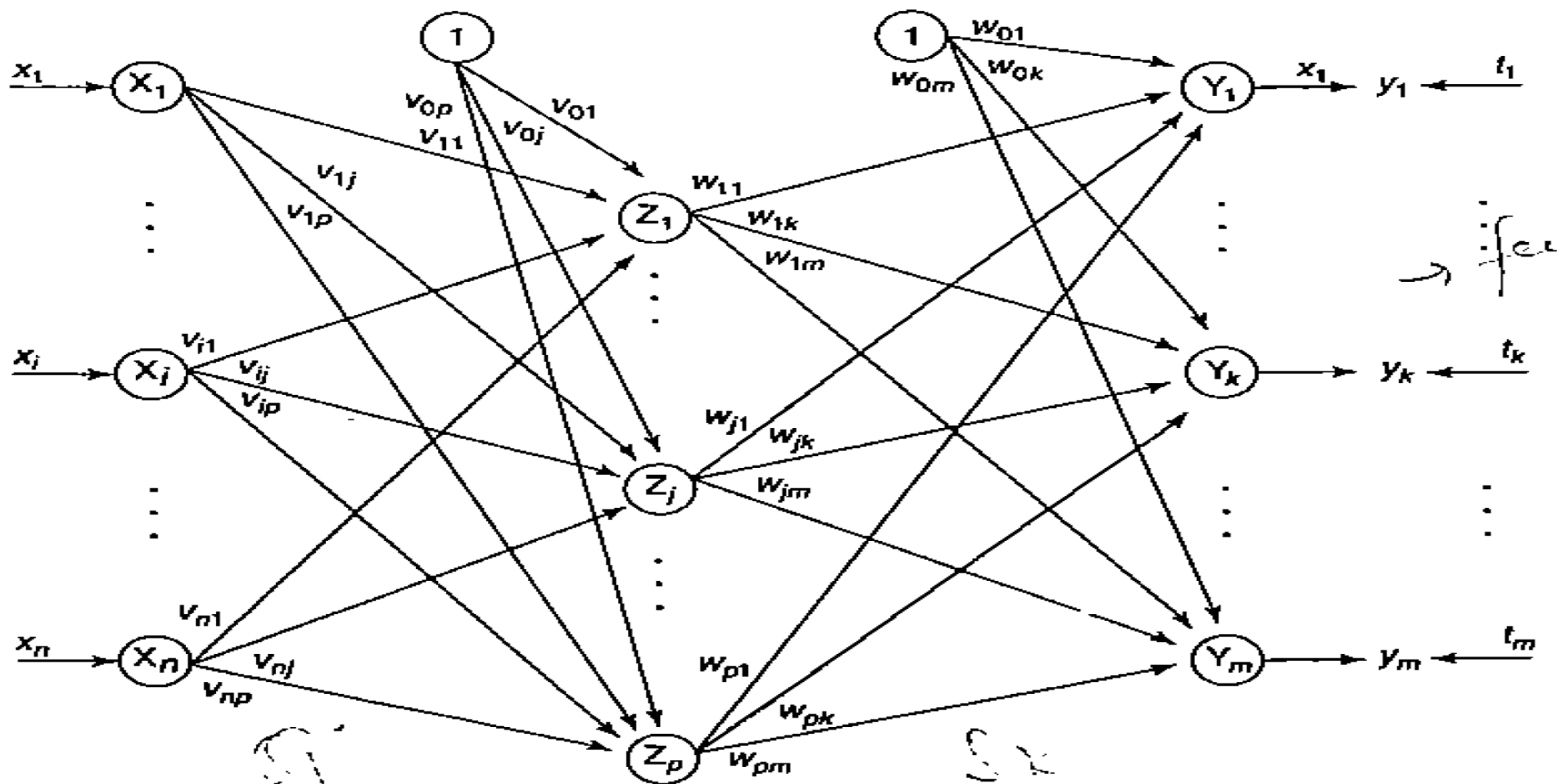
- ❑ The back-propagation algorithm is different from other networks in respect to the process by which weights are calculated during the learning period of the network.
- ❑ When the hidden layers are increased the network training becomes more complex.
  - ❑ To update weights, the error must be calculated.
    - ❑ The error, which is the difference between the actual (calculated) and the desired (target) output, is easily measured at the output layer.



- At the hidden layers, there is no direct information of the error.
- Therefore, other techniques should be used to calculate an error at the hidden layer, **which will cause minimization of the output error**, and this is the ultimate **goal**.
- The training of the BPN is done in three stages –
  - the feed-forward of the input training pattern,
  - The calculation and back-propagation of the error
  - updation of weights.

- The testing of the BPN involves the
  - computation of feed-forward phase only.
- There can be more than one hidden layer (more beneficial) but one hidden layer is sufficient.
- Even though the training is very slow, once the network is trained it can produce its outputs very rapidly.

# ARCHITECTURE



- $\delta_k$  =error correction weight adjustment for  $W_{jk}$  that is due to an error at output unit  $Y_k$ , which is back-propagated to the hidden units that feed into unit  $Y_k$
- $\delta_j$  error correction weight adjustment for  $V_{ij}$  that is due to the back-propagation of error to the hidden unit  $z_j$ .
- Commonly **used binary sigmoidal and bipolar sigmoidal activation functions** .
- These functions are used in the BPN because of the following characteristics: (i) Continuity ii) differentiability iii) non decreasing monotony.
- The range of binary sigmoid is from 0 to 1, and for bipolar sigmoid it is from -1 to+ 1.

## TRAINING ALGORITHM

Step 0: Initialize weights and learning rate. take some small random values

Step 1: Perform Steps 2-9 when stopping condition is false

Step 2: Perform Steps 3-8 for each training pair.

### *Feed Forward phase (Phase 1)*

Step 3: Each input unit receives input signal  $x_i$ ; and sends it to the hidden unit ( $i = 1$  to  $n$ ).

Step 4: Each hidden unit  $Z_j$  ( $j = 1$  to  $p$ ) sums its Weighted input Signals to calculate net input:

$$z_{inj} = v_{0j} + \sum_{i=1}^n x_i v_{ij}$$

Calculate output of the hidden unit by applying its activation over  $Z_{inj}$  (binary or bipolar sigmoidal activation function)

$$z_j = f(z_{inj})$$

and send the output signal from the hidden unit to the input of output layer units

Step 5: For each output unit  $y_k$  ( $k=1$  to  $m$ ) Calculate the net input:

$$y_{ink} = w_{0k} + \sum_{j=1}^p z_j w_{jk}$$

and apply the activation function to compute output signal

$$y_k = f(y_{ink})$$

## *Back propagation of error(Phase 2)*

- Step 6: Each output unit  $y_k$  ( $k=1$  to  $m$ ) receives a target pattern corresponding to the input training pattern and computes the error correction term:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

The derivative  $f'(y_{ink})$  can be calculated.

On the basis of the calculated error correction term, update the change in weights and bias:

$$\Delta w_{jk} = \alpha \delta_k z_j, \quad \Delta w_{0k} = \alpha \delta_k$$

Also, send  $\delta_k$  to the hidden layer backwards.

- Step 7: Each hidden unit ( $z_j$ ,  $j=1$  to  $p$ ) sums its delta inputs from the output units:

$$\delta_{inj} = \sum_{k=1}^n \delta_k w_{jk}$$

- The term  $\delta_{inj}$  gets multiplied with the derivative of  $f(Z_{inj})$  to calculate the error term:

$$\delta_j = \delta_{inj} f'(z_{inj})$$

- The  $f'(Z_{inj})$  can be calculated depending on whether binary or bipolar sigmoidal function is used.
- On the basis of the calculated  $\delta_j$ , update the change in weights and bias:

$$\Delta v_{ij} = \alpha \delta_j x_i; \quad \Delta v_{0j} = \alpha \delta_j$$



### *Weight and bias updation (Phase 3)*

**Step 8:** Each output unit ( $y_k$ ,  $k = 1$  to  $m$ ) updates the bias and weights:

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

$$w_{0k}(\text{new}) = w_{0k}(\text{old}) + \Delta w_{0k}$$

Each hidden unit ( $z_j$ ,  $j = 1$  to  $p$ ) updates its bias and weights:

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

$$v_{0j}(\text{new}) = v_{0j}(\text{old}) + \Delta v_{0j}$$

**Step 9:** Check for the stopping condition. The stopping condition may be certain number of epochs reached or when the actual output equals the target output.

- The above algorithm uses the incremental approach for updation of weights,
  - i.e., the weights are being changed immediately after a training pattern is presented.
- There is another way of training called batch-mode training, where the weights are changed only after all the training patterns are presented.
- The effectiveness of two approaches depends on the problem, but batch-mode training requires additional local storage for each connection to maintain the immediate weight changes.
- When a BPN is used as a classifier, it is equivalent to the optimal Bayesian discriminant function for asymptotically large sets of statistically independent training pattern

# APPLICATIONS OF BACKPROPAGATION NETWORK

- Load forecasting problems in power systems.
- Image processing.
- Fault diagnosis and fault detection.
- Gesture recognition, speech recognition.
- Signature verification.
- Bioinformatics.
- Structural engineering design (civil).