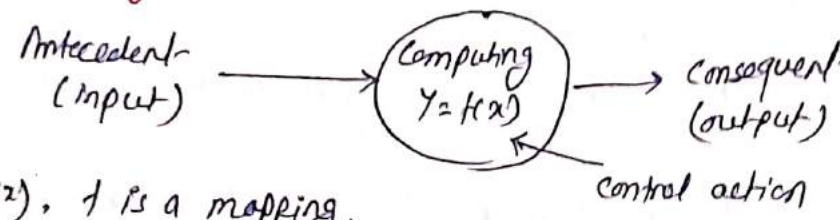
Soft computing.

Concept of computing:-



→ mapping method from input-set to output-set called as computing

$y = f(x)$, f is a mapping.

f is also called a formal method or an algorithm

Important characteristics of computing

→ Should provide precise solution.

→ Control action should be unambiguous & accurate.

→ suitable for problem, which is easy to model mathematically.

Hard computing:

→ In 1996, L.A. Zade (LAZ) introduced the term hard computing.

→ According to LAZ; we term a computing as Hard computing, if

- Precise result is guaranteed.
- Control action is unambiguous.
- control action is formally defined (i.e, with mathematical model or algorithm)

Ex of Hard computing:

- solving numerical problems (e.g. roots of polynomials, integration etc.)
- Searching & sorting techniques.
- Solving computational geometry problems (e.g. shortest tour in a graph, finding closest pairs of points given a set of points etc.)

Soft Computing:

- The term soft computing was proposed by the inventor of fuzzy logic

⇒ Soft computing is a collection of methodologies that aim to exploit the tolerance for imprecision & uncertainty to achieve tractability, robustness, & low solution cost.

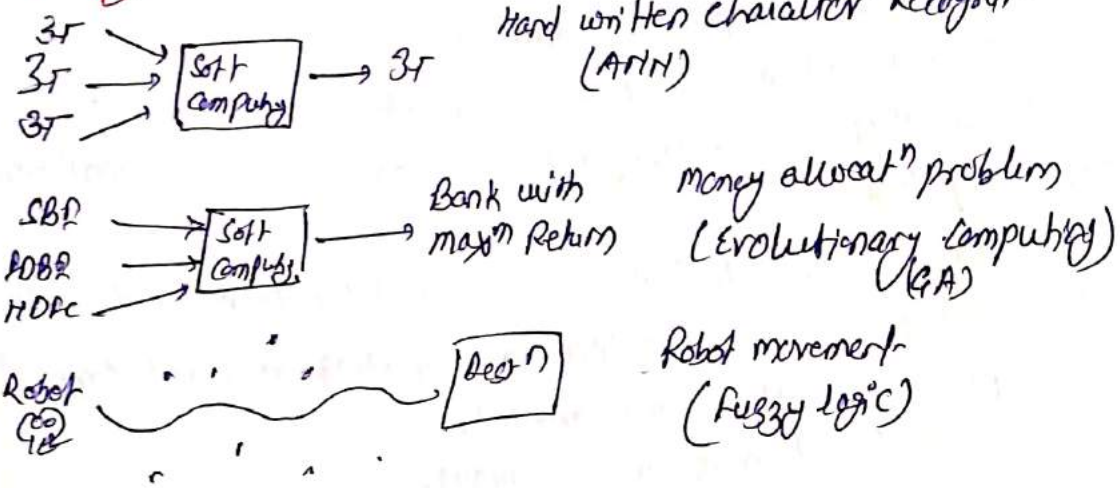
→ Its principal constituents are fuzzy logic, neurocomputing, and probabilistic reasoning

→ The role model for soft computing is the human mind.

Characteristics of soft computing:

- It does not require any mathematical modeling of problem solving.
- It may not yield the precise solution.
- Algorithms are adaptive (i.e. it can adjust to the change of dynamic environment).
- Use some biological inspired methodologies such as genetics, evolution, Ant's behaviors, particles swarming, human nervous system etc.

Ex of Softcomputing :-



How Soft computing ?

ex1 • How a student learns from his teacher?
  - Teacher asks questions & tell the answers them.
  - Teacher puts questions & hints answers & asks whether the answers are correct or not.
  - Student thus learn a topic & store in his memory.
  - Based on the knowledge he solve new problems.
- This is the way how human brain works.
- Based on this concept ANN is used to solve problems.

ex2:
- How world selects the best?
  - It starts with a population (random)
  - Reproduces another population (next generation)
  - Rank the population & selects the superior individuals.
- GA is based on this natural phenomena.
  - population is synonymous to solutions.
  - selection of superior solution is synonymous to exploring the optimal soln.

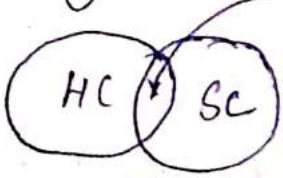
Eg:-

- How a doctor treats his patient?
  - Doctor asks the patient about suffering.
  - Doctor find the symptoms of diseases.
  - Doctor prescribed tests & medicines.
- This is exactly the way fuzzy logic works.
  - Symptoms are correlated with diseases with uncertainty.
  - Doctor prescribes tests/medicines fuzzify.

Hard computing Vs.	Soft computing
• It requires a precisely stated analytical model & often a lot of computation time.	• It is tolerant of imprecision, uncertainty, partial truth, & approximation.
• It is based on binary logic, crisp system, numerical analysis & crisp software.	• It is based on fuzzy logic, neural nets & probabilistic reasoning.
• It has the characteristics of precision & categoricity.	• It has the characteristics of approximation & dispositionality.
• It is deterministic	• It incorporates stochasticity.
• It requires exact input data	• It can deal with ambiguous & noisy data.
• It is strictly sequential.	• It allows parallel computations.
• It produces precise answers.	• It can yield approximate answers.

Hybrid computing

→ It is a combination of the conventional hard computing & emerging soft computing.



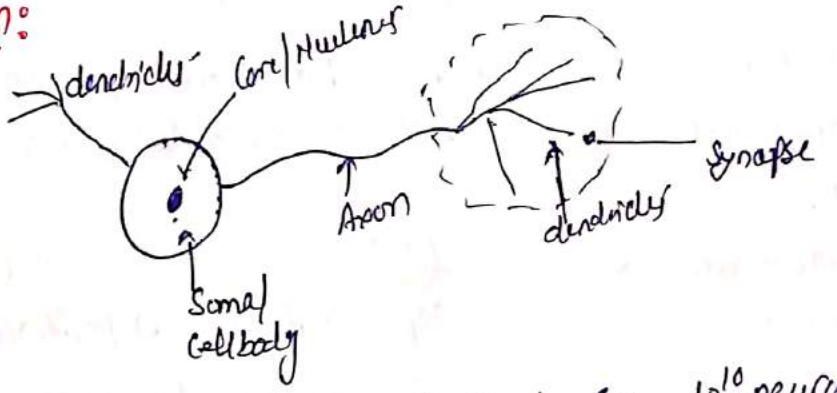
Biological nervous system

→ Biological nervous system is the most important part of many living things, in particular, human beings.

- There is a part called brain at the centre of human nervous system.
- In fact, any biological nervous system consists of a large number of interconnected processing units called neurons.
- Each neuron is approximately 10 μ long & they can operate in μ!
- Typically, a human brain consists of approximately 10^{11} neurons communicating with each other with the help of electric impulses.

→ Atomic unit in brain called neuron.

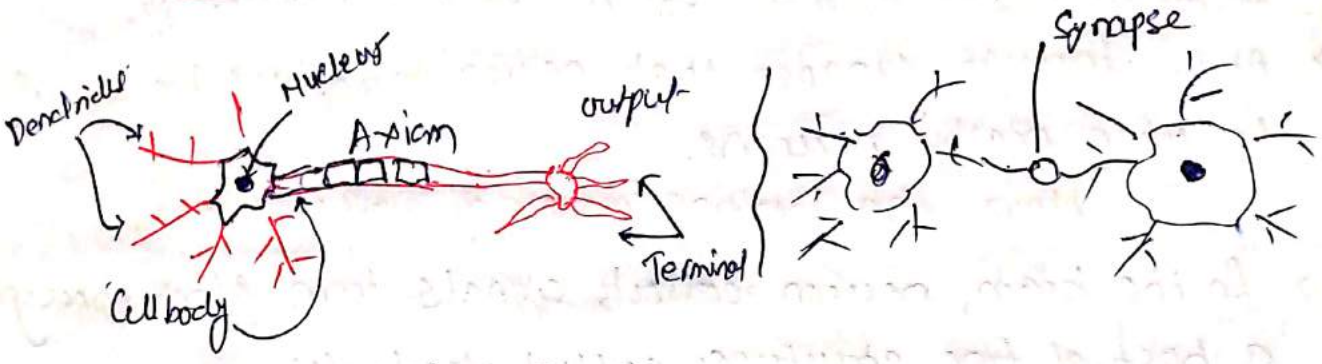
Neuron:



→ fundamental constituent of the brain, 10^{10} neurons in brain.
→ Each neuron is connected to about 10^4 other neurons.
→ It receives electro-chemical signals from its various sources & in turn responds by transmitting electrical impulses to other neurons.
→ A neuron is composed of a nucleus - a cell body known as soma.
→ Long irregularly shaped filaments or (hairy thin) attached to the soma called as dendrites.
→ Dendrites behave as input channels, all input from other neurons arrive through the dendrites.
→ Axon is electrically active & serves as an output channel.
→ The axon terminates in a specialised contact called synapse that connects the axon with the dendritic link of another neuron.

Unit II

A Biological Neural Networks



1) Neurons are the fundamental unit of the brain
2) A neural network is made up of number of processing elements called as neurons.
3) A neural networks consists of many interconnected neurons.
4) The brain consists of hundreds of billion of cells called neurons.
5) Human brain consists of a huge number of neurons, approx 10^{11} with numerous interconnections.
6) Neurons are connected together by snapses which are the connection across which a neuron can send an impulse to another neuron. Dendrites receives i/p through the synapses of other neuron.
7) There are approx 10^4 synapses per neuron in the human brain.
8) The synapses have a processing value or weight.
9) Each neuron accepts input from the outside world via dendrides or from outputs from other neurons, process it & gives the output through Axons.
10) A neuron is composed of nucleus, a cell body (or soma), dendrides are tree-like networks made of nerve fibre connected to the cell body.
11) An axon is a single, long interconnection extending from the cell body & carrying signals from the neuron.

12) The end of the axon splits into fine strands & each strand terminates into a small bulb like organ called synapse

13) It is through synapse that neuron introduces its signal to other nearby neurons.

How the thinking process of works in brain -

14) In the brain, neuron collects signals from other through a host of fine structures called dendrites.

The neuron sends out spikes of electrical activity through the axon which can split into thousands of branches. At the end of each branch, a snapses converts that activity from the axon into electrical effects that inhibit or excite activity on the contracted (traget) neuron.

15) When a neuron sends or excitatory signal to another neuron, then this signal will be enabled to all of the inputs of that neuron.

16) if the cumulative input recieved by the cell body raises the internal elutric potential of a cell to threshold then the receiving cell fires a pulse or action potential of fixed strength & duration is sent out through the axon to other neurons via synapses.

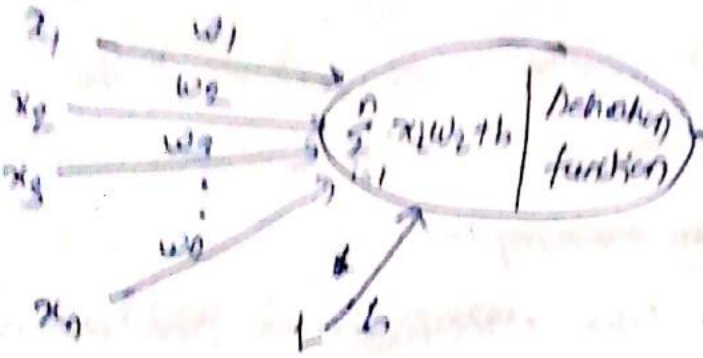
- This is how the thinking prouss works internally.

16

Artificial Neural Network (ANN)

- ANN is a computational nonlinear model that is inspired from brain (i.e. neurons)
- Like people, it learn by examples.
- ANN can perform tasks like classification, prediction, decision making, visualizn. & other just by considering examples.
- ANN consists of large collection of artificial neurons as processing elements which operates in parallel.
- Every neuron is connected with other neuron through connection link.
- Every connection link is associated with a weight that has information about the input signal.
- weight is the most useful information for network to solve a particular problem because of it usually excites or inhibits the signal that is being communicated.
- Every neuron has weighted inputs (synapses), an activation function (that defines the output given an input) & one output
- ANN can be viewed as weighted directed graphs in which artificial neurons are nodes & directed edges with weights are connections between neuron outputs & neuron inputs.
- The neurons/nodes can take input data & perform simple operations on the data & the the results of these operations is passed to other neuron.
- ANN are capable of learning, which takes place by altering weight value.

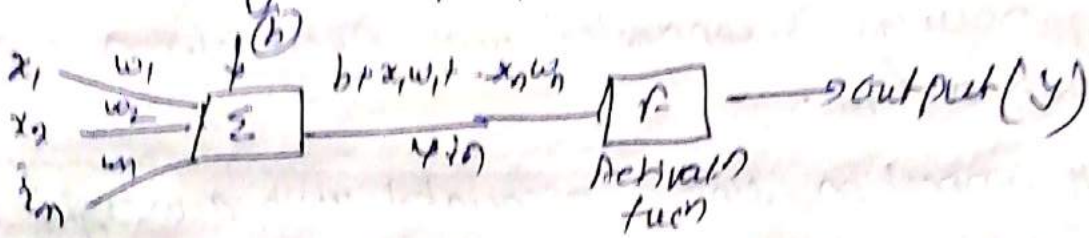
b - bias



$x_1, x_2 \ldots x_n$ - inputs
$w_1, w_2, \ldots w_n$ - weights of connection

Perceptron (Single layer ANN)

- A single layer neural networks is called perceptron. It gives a single output.



Perceptron (General model of ANN)

- In above figure $x_1, x_2, \ldots x_n$ represents various inputs. (independent variables) to the network.
- Each of these inputs is multiplied by a connection weight or synapses.
- weight $w_1, w_2 \ldots w_n$ shows the strength of a particular node.
- b. bias value, it allows you to shift the activation function up or down.
- Product of input & ~~output~~ weight Summed & feed to a activation function to generate a result. & then result is sent as an output.
- ANN receives input from the external world in the form of pattern & image (in vector form).
- Each input is multiplied by its corresponding weights and all weighted input are summed up inside computing unit (artificial neuron)

- Incase the weighted sum is zero; bias (z_i) is added to make the output non-zero or to scale up the system response
- weighted sum corresponds to any numerical value ranging from 0 to ∞.
- ANN helps you to conduct image understanding human learning, computer speech etc.
- The main purpose of the activation function is to convert an input signal of a node in an ANN to an output signal. This output signal is used as input to the next layer in the network.
- From the above general model of ANN, the net input can be calculated as

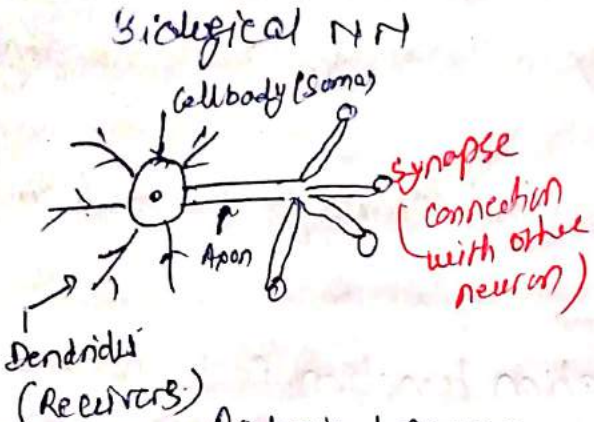
$$y_{in} = x_1 w_1 + w_2 x_2 + \dots x_n x_n = \sum_{i=1}^{n} x_i w_i$$

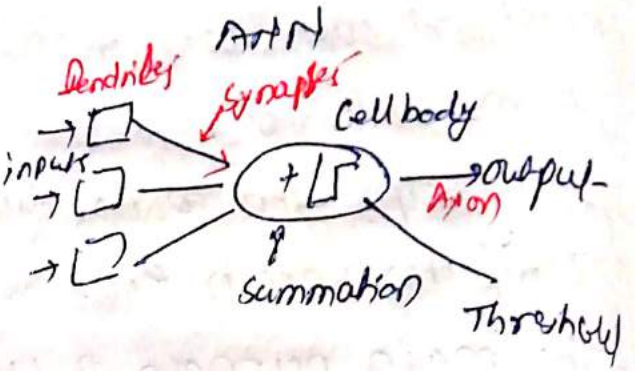
- The output can be calculated by applying the activation function over the net input

$$Y = F(\Sigma x_i w_i + \text{bias})$$

- In order to limit the response arrive at desired value; the threshold value is set up.
- For this, the sum is passed through activation function.
- The activation function is set of the transfer function used to get desired output. Range [0 1] [-1 1]
- Activation function can be linear or non linear.

17 Comparision b/w biological neural network & ANN





Biological neuron	Artificial neuron
- Slower in processing info^n	- Faster in processing info^n.
- Stores the info^n in the synapse	- Stores the info^n in continuous memory locat^n.
- There is no control unit for processing info^n in the brain.	- In ANN, control unit monitors all the activities.
- Big size & more complex.	- small size & less complex.
- 10^{11} neurons & 10^5 interconnections	- Depends in the type of applicat^n & network designer
- No loss of memory.	- Sometimes memory may loss.
- It has fault tolerance.	- Limited fault tolerence bcoz info^n gets distrupted when interconnect^n are disconnected
- Control mechanism is complicated in BNN as it involves chemical in biological neuron.	- control mechanism is simpler in ANN.
- Massively parallel, slow but superior than ANN.	- Massively parallel, fast but inferior than BNN.

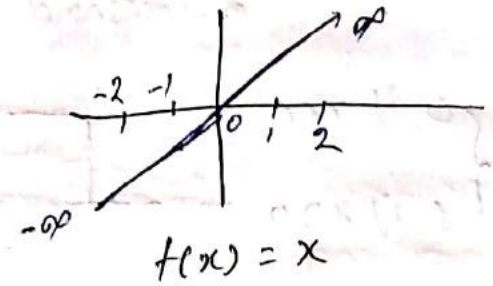
18 Activation Function

- Activation function is used to get the output of a node.
- It convert a input signal of a node in ANN to an o/p signal.
- Activation fun^n is used to maps the resulting values in b/w 0 to 1 or -1 to 1 depending upon the type of fun^n used.

Activation fun^n → Linear Activat^n fun^n
→ Non-linear activat^n fun^n

Linear Activat^n fun^n

Ex. Linear or identity activat^n fun^n



$f(x) = x$

- The output of the func^n will not be confined b/w any range.
- NN would not be able to learn.

- It makes it easy for the model to generate or adopt with variety of data & to differentiate b/w the o/p
- It make the n/w more powerful & add ability to it to learn sometimes complex & complicated from data.

ex- Logistic/Sigmoid, Tanh ReLU etc.

Activation function :- Linear AF → AF → non linear

- The AF help the n/w use the important info^n & supress the irrelevant data points.
- Activation function decides whether a neuron should be activated or not by calculating the weighted sum & further adding bias to it.
- It introduces non-linearity into the op of a neuron.
- If we don't apply AF then the o/p signal would be simply linear function (1 degree polynomial)

- Linear func^n is easy to solve but they are limited in their complexity & have less power.
- without AF our model cannot learn & model complicated data such as image, video, audio, speed etc.
- We need ANN to learn & represent almost anything & any arbitrary complex function that maps an i/p to o/p. (non linearity)

* Neural Nlw without activation func^n - linear regression model

[19] Types of Activation function

1) Threshold AF (Binary step func^n)

o - Threshold

- A Binary Activation function is a threshold based AF.
- Also know as Heaviside func^n.

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

- If the input to the binary step func^n is greater than a then the neuron is activated else it is deactivated.
- Binary Step AF are useful for binary classification schemes
- This AF is useful when the i/p pattern can only belong to one or two groups, that is binary classification.

Limitation:

This function will not be useful when there are multiple classes in the target variable.

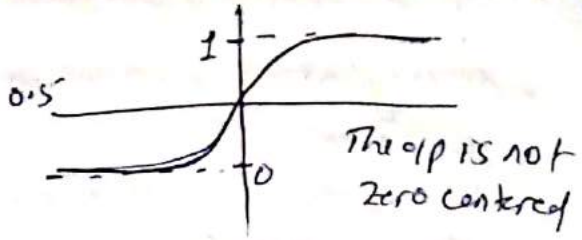
2) Sigmoid - (logistic - func^n)

- It is one of the most widely used AF
- It transform the values b/w 0 & 1.
- Mathematical expression for sigmoid is

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

$x \to -\infty$ $\quad f(x) = 0$
$x \to \infty$ $\quad f(x) = 1$
$x = 0$ $\quad f(x) = 0.5$

- A sigmoid function is a mathematical func^{n} having S shaped curve or Sigmoid curve which rapes blw 0 & 1.



- It is used for models where we need to predict the possibility as an output.
- It is mostly used for ~~binary~~ multiclass classification.
- It is used in major ML algos such as logistic regress & NN.

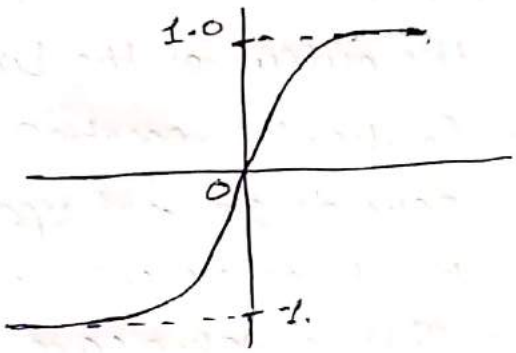
$$\text{Probability of event } \sigma(x) = \frac{1}{1+e^{-x}}$$

- It converts a value x into a probability of something happing. such as, the probability P of rainfall given the weather cond^{n}.
- If $P >= 0.5$ the o/p is marked as true.
- if $P < 0.5$, output is false.

3) Tanh - Hyperbolic Tangent func^{n}

- It is similar to sigmoid but better in performance.
- It is also a non linear Af.
- The func^{n} ranges blw $(-1, 1)$
- It is symmetric around the origion
- formula used

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



- The advantages is that the negative inputs will be mapped strongly negative & the zero inputs will be mapped near zero in the tanh graph.
- The tanh function is mainly, used classification blw two classes.

4) ReLU (Rectified Linear Unit) AF

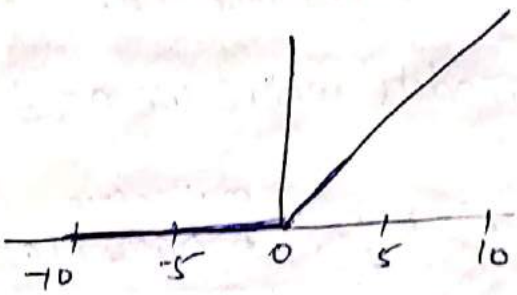
- Mostly used activation function, since it is used in almost all the CNN & DL tasks.

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

or

$f(x) = x, \quad x \geq 0$

$0, \quad x < 0$



It gives an output x if x is +ve & 0 otherwise.

- It ranges from 0 to ∞
- It is Half rectified.

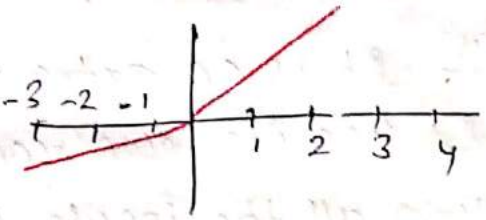
Limitations: All negative values becomes zero immediately which decreases the ability of the model to fit or train from the data properly.

Adv:
- It is less computationally expensive than tanh & Sigmoid because it involves simple mathematical operations.
- The main advantage of using the ReLU function over other AFs is that it does not activate all the neuron's at the same time.
- This means that the neurons will only be deactivated; if the the output of the Linear transformation is less than 0.
- In backpropagation process, the weight & biases for some neuron's are not updated. This can create dead neurons, which never gets activated.
- This is taken care by 'Leaky' ReLU function.

(5) Leaky ReLU -

- It is an improved version of the ReLU function.
- For ReLU function the gradient is 0 for $x < 0$, which would deactivate the neurons in that region, Leaky ReLU is defined to address this problem.
- Instead of defining the ReLU function as 0 for negative values of x, we define it as an extremly small linear component of x.
- The mathematical expression for Leaky ReLU function is

$$f(x) = 0.01x, \quad x < 0$$
$$= x, \quad x \geq 0$$



- By making this small modification, the gradient of the left side of graph comes out to be a non-zero value.
- So there would no layer encounter dead neuron in that region.

23) Terms used in ANN

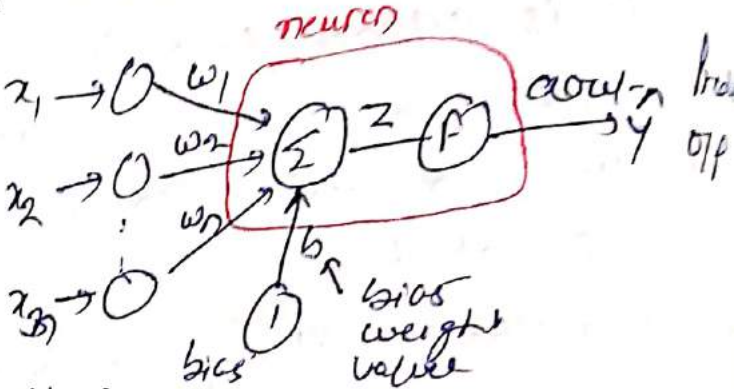
Neuron / Node:

- It is the basic unit of a neural network.
- It gets certain no. of inputs & a bias value

$$z = x_1 w_1 + x_2 w_2 + \dots + x_n w_n + b \times 1$$

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

$$a_{out} = f(z) = \frac{1}{1+e^{-z}}$$

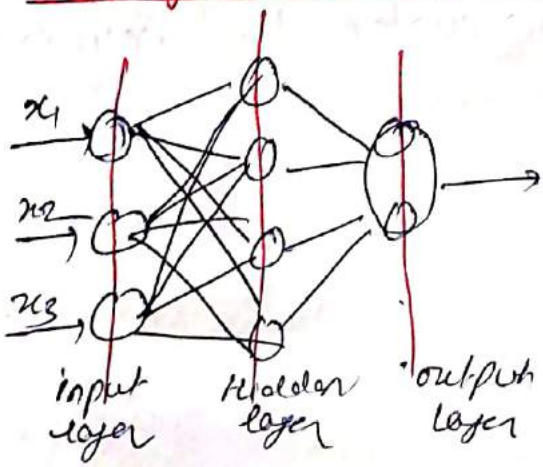


$\hat{y} = a_{out} = f(z) = \frac{1}{1+e^{-z}}$ (for sigmoid AF)

Bias: It is an extra input to neurons & it is always 1 & has its own connection weight b.

- when all the inputs are zeros, value of bias inputed to an AF.

Multi layer Neural Network ^



^ There may be single or multiple outputs

^ There can be multiple no. of hidden layers

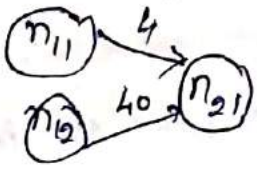
(multi layer ANN with single o/p)

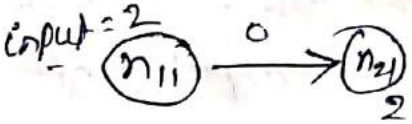
Connection: It connects one neuron in one layer to another neuron in another layer.

- A connection has always a weight value associated with it
- Goal of ANN training is to update the connection weight value to decrease the loss/error

weights/parameters

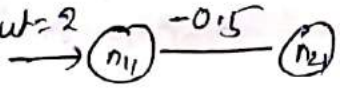
A weight represents the strength of the connection b/w nodes/neu





If the weight from node1 to node2 has greater magnitude, it mean that neuron 1 has greater influence over neuron 2.

- A weight brings down the importance of the input value.
- weight near zero means changing this input will not change the output.



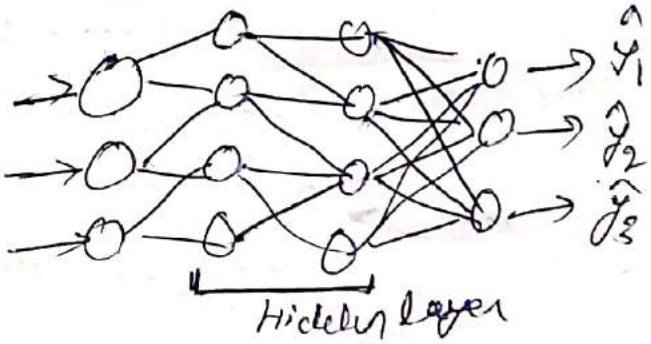
$2 \times (-0.5) = -1$

- Negative weights means increasing this input will decrese the output.
- So weight decides how much influence the input will have on the output.

24) Multilayer ANN

An ANN consists of artifical neurons or processing elements & organized in three interconnected layers -

1) input layer (single)
2) Hidden layer (one or more)
3) output layer (single)



i) Input layer:

- First layer in NN.
- This layer contains neurons which receives i/p from the outside world & passes them on to the next layer (Hidden layer).
- It doesn't apply any operations on the input values & has no weight & biases values associated.

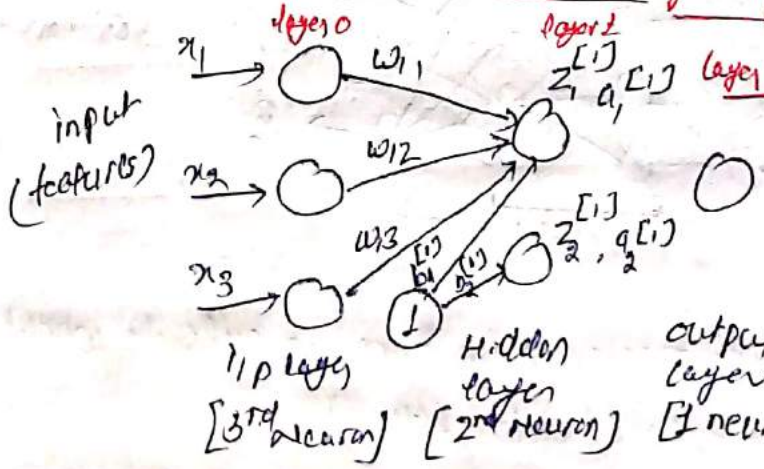
(2) Hidden Layer

- These units (neurons) are in blw i/p & o/p layers.
- The job of hidden layer is to transform the i/p into some that output unit can use in some way.
- These neurons are hidden from the people who are interfa with the system & act as a black box to them.
- On increasing the hidden layers with neurons, the systems computational & processing power can be increas but the training process of the system gets more comple at the same time.
- Hidden layer neuron send information to o/p layer.

(3) Output layer

- It is the last layer in the neural network & it recie input from the last hidden layer.
- With this layer we can get desired number of value (o/ps) & in a desired range.
- Most ANN are fully connected that means each hidde neuron is fully connected to every neuron in its previou layer (input) & to the next layer (output) layer.

Representation of weights & Bias in ANN

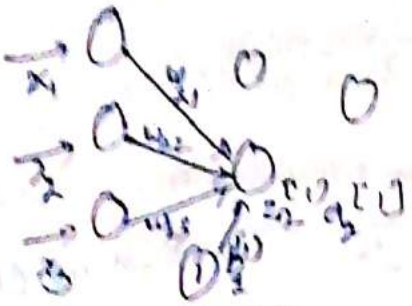


$$z_1^{[1]} = x_1 w_{11}^{[1]} + x_2 w_{12}^{[1]} + x_3 w_{13}^{[1]} + 1 b_1^{[1]}$$

(Layer no ↑, Neuron no ↑)

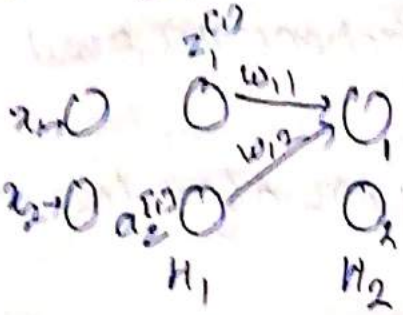
$$a_1^{[1]} = \sigma(z_1^{[1]}) = \frac{1}{1 - e^{-z^{[1]}}}$$

(for Sigmoid Af)



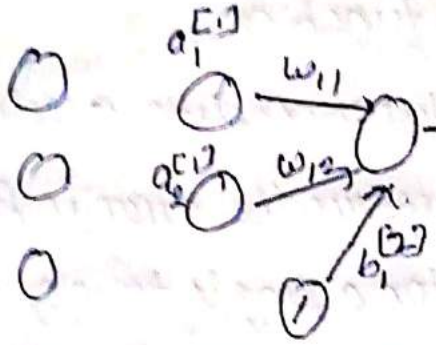
$$z_2^{[1]} = x_1 w_{21}^{[1]} + x_2 w_{22}^{[1]} + x_3 w_{23}^{[1]} + b_2^{[1]}$$

$$a_2^{[1]} = \sigma(z_2^{[1]}) = \frac{1}{1 - e^{-z_2^{[1]}}} \quad \text{(for Sigmoid AF)}$$



$$z_1^{[1]} = a_1^{[1]} w_{11} + a_2^{[1]} w_{12}$$

$$a_1^{[2]} = \sigma(z_1^{[2]})$$

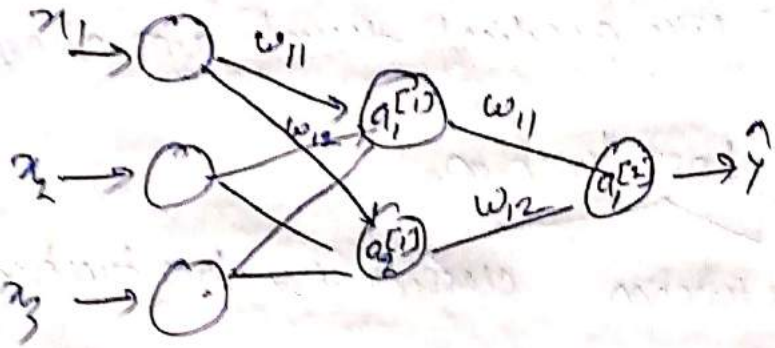


$\hat{y}$ (Predicted O/P)

$$z_1^{[2]} = a_1^{[1]} w_{11}^{[2]} + a_2^{[1]} w_{12}^{[2]} + 1 \times b_1^{[2]}$$

$$a_1^{[2]} = \sigma(z_1^{[2]})$$

$$\therefore \hat{y} = a_1^{[2]}$$



$$z^{[1]} = x_1 w_1^{[1]} + b^{[1]}$$

$$z^{[n]} = a^{[n-1]} w^{[n]} + b^{[n]}$$

$$a^{[n]} = \sigma(z^{[n]})$$

No of parameters in ANN

= no of weights + no of bias

= $(3 \times 2 + 2 \times 1) + 3 = 6 + 2 + 3 = 11$

no of bias = no of neurons - I/p layer neurons

by using ANN.

2.3.2 Learning

The main property of an ANN is its capability to learn. Learning or training is a process by means of which a neural network adapts itself to a stimulus by making proper parameter adjustments, resulting in the production of desired response. Broadly, there are two kinds of learning in ANNs:

1. *Parameter learning*: It updates the connecting weights in a neural net.
2. *Structure learning*: It focuses on the change in network structure (which includes the number of processing elements as well as their connection types).

The above two types of learning can be performed simultaneously or separately. Apart from these two categories of learning, the learning in an ANN can be generally classified into three categories as: supervised learning; unsupervised learning; reinforcement learning. Let us discuss these learning types in detail.

2.3.2.1 *Supervised Learning*

The learning here is performed with the help of a teacher. Let us take the example of the learning process of a small child. The child doesn't know how to read/write. He/she is being taught by the parents at home and by the teacher in school. The children are trained and molded to recognize the alphabets, numerals, etc. Their each and every action is supervised by a teacher. Actually, a child works on the basis of the output that he/she has to produce. All these real-time events involve supervised learning methodology. Similarly, in ANNs following the supervised learning, each input vector requires a corresponding target vector, which represents the desired output. The input vector along with the target vector is called *training pair*. The network here is informed precisely about what should be emitted as output. The block diagram of Figure 2-12 depicts the working of a supervised learning network.

During training, the input vector is presented to the network, which results in an output vector. This output vector is the actual output vector. Then the actual output vector is compared with the desired (target) output vector. If there exists a difference between the two output vectors then an error signal is generated by the network. This error signal is used for adjustment of weights until the actual output matches the desired (target) output. In this type of training, a supervisor or teacher is required for error minimization. Hence, the network trained by this method is said to be using supervised training methodology. In supervised learning, it is assumed that the correct "target" output values are known for each input pattern.

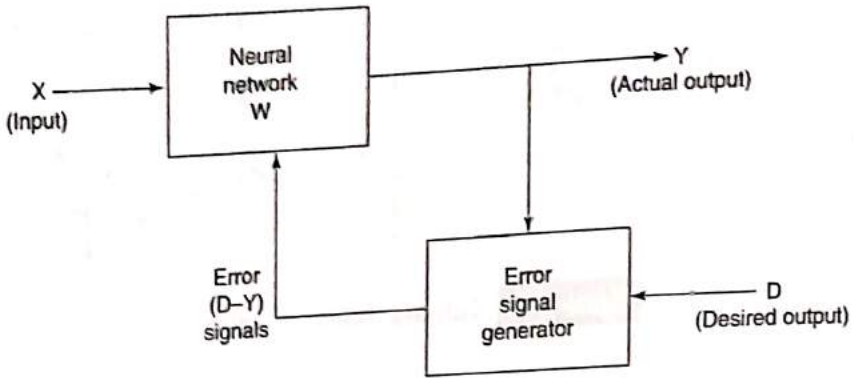


Figure 2-12 Supervised learning.

2.3.2.2 *Unsupervised Learning*

The learning here is performed without the help of a teacher. Consider the learning process of a tadpole, it learns by itself, that is, a child fish learns to swim by itself, it is not taught by its mother. Thus, its learning process is independent and is not supervised by a teacher. In ANNs following unsupervised learning, the input vectors of similar type are grouped without the use of training data to specify how a member of each group looks or to which group a number belongs. In the training process, the network receives the input patterns and organizes these patterns to form clusters. When a new input pattern is applied, the neural network gives an output response indicating the class to which the input pattern belongs. If for an input, a pattern class cannot be found then a new class is generated. The block diagram of unsupervised learning is shown in Figure 2-13.

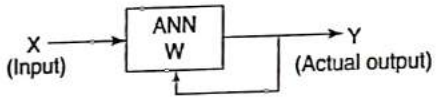


Figure 2-13 Unsupervised learning.

From Figure 2-13 it is clear that there is no feedback from the environment to inform what the outputs should be or whether the outputs are correct. In this case, the network must itself discover patterns, regularities, features or categories from the input data and relations for the input data over the output. While discovering all these features, the network undergoes change in its parameters. This process is called *self-organizing* in which exact clusters will be formed by discovering similarities and dissimilarities among the objects.

2.3.2.3 *Reinforcement Learning*

This learning process is similar to supervised learning. In the case of supervised learning, the correct target output values are known for each input pattern. But, in some cases, less information might be available. For example, the network might be told that its actual output is only "50% correct" or so. Thus, here only critic information is available, not the exact information. The learning based on this critic information is called *reinforcement learning* and the feedback sent is called *reinforcement signal.*

The block diagram of reinforcement learning is shown in Figure 2-14. The reinforcement learning is a form of supervised learning because the network receives some feedback from its environment. However, the feedback obtained here is only evaluative and not instructive. The external reinforcement signals are processed in the critic signal generator, and the obtained critic signals are sent to the ANN for adjustment of weights properly so as to get better critic feedback in future. The reinforcement learning is also called learning with a critic as opposed to learning with a teacher, which indicates supervised learning.

So, now you've a fair understanding of the three generalized learning rules used in the training process of ANNs.

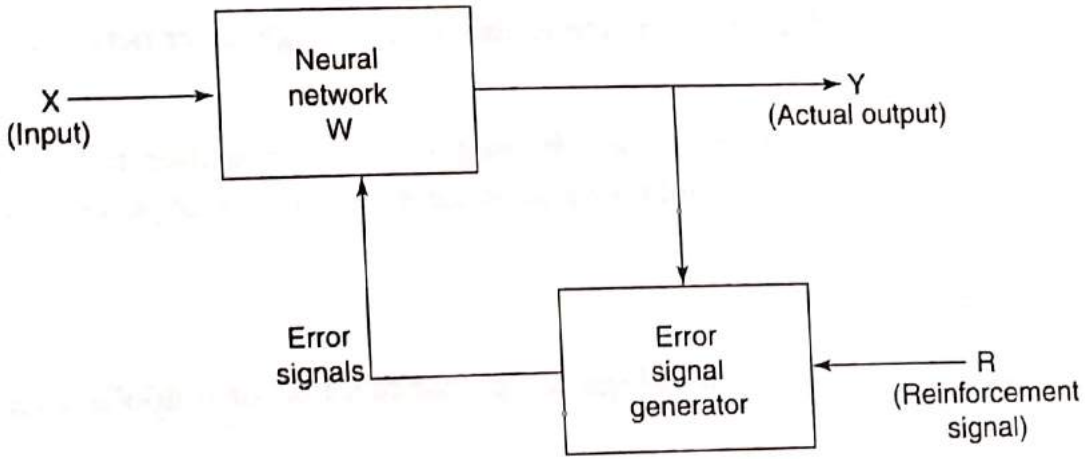


Figure 2-14 Reinforcement learning.

2.5 McCULLOCH–PITTS NEURON

2.5.1 Theory

The McCulloch–Pitts neuron was the earliest neural network discovered in 1943. It is usually called as *M–P neuron*. The M–P neurons are connected by directed weighted paths. It should be noted that the activation of a M–P neuron is binary, that is, at any time step the neuron may fire or may not fire. The weights associated with the communication links may be excitatory (weight is positive) or inhibitory (weight is negative). All the excitatory connected weights entering into a particular neuron will have same weights. The threshold plays a major role in M–P neuron. There is a fixed threshold for each neuron, and if the net input to the neuron is greater than the threshold then the neuron fires. Also, it should be noted that any nonzero inhibitory input would prevent the neuron from firing. The M–P neurons are most widely used in the case of logic functions.

2.5.2 Architecture

A simple M–P neuron is shown in Figure 2-18. As already discussed, the M–P neuron has both excitatory and inhibitory connections. It is excitatory with weight ($w > 0$) or inhibitory with weight $-p(p < 0)$. In Figure 2-18, inputs from x_1 to x_n possess excitatory weighted connections and inputs from x_{n+1} to x_{n+m} possess inhibitory weighted interconnections. Since the firing of the output neuron is based upon the threshold, the activation function here is defined as

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

For inhibition to be absolute, the threshold with the activation function should satisfy the following condition:

$$\theta > nw - p$$

The output will fire if it receives say "k" or more excitatory inputs but no inhibitory inputs, where

$$kw \geq \theta > (k-1)w$$

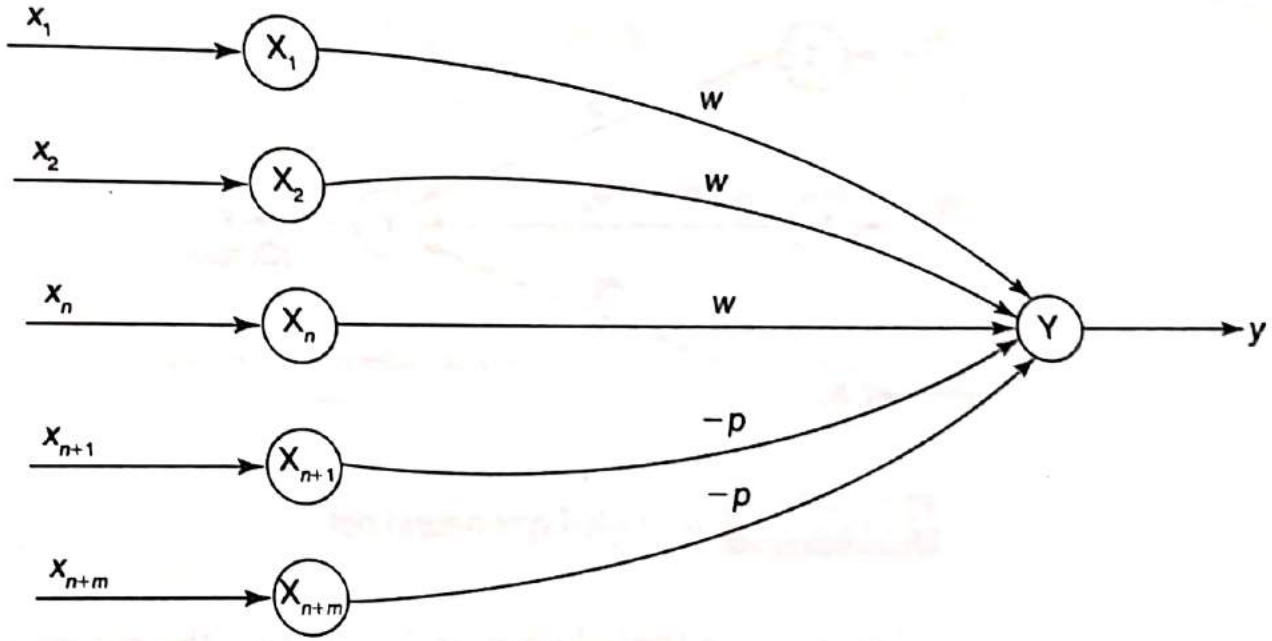


Figure 2-18 McCulloch–Pitts neuron model.

The M–P neuron has no particular training algorithm. An analysis has to be performed to determine the values of the weights and the threshold. Here the weights of the neuron are set along with the threshold to make the neuron perform a simple logic function. The M–P neurons are used as building blocks on which we can model any function or phenomenon, which can be represented as a logic function.

2.7.3 Training Algorithm

The training algorithm of Hebb network is given below:

Step 0: First initialize the weights. Basically in this network they may be set to zero, i.e., $w_i = 0$ for $i = 1$ to "n" where n may be the total number of input neurons.

Step 1: Steps 2–4 have to be performed for each input training vector and target output pair, $s:t$.

Step 2: Input units activations are set. Generally, the activation function of input layer is identity function: $x_i = s_i$ for $i = 1$ to n.

Step 3: Output units activations are set: $y = t$.

Step 4: Weight adjustments and bias adjustments are performed:

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

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

The above five steps complete the algorithmic process. In Step 4, the weight updation formula can also be given in vector form as

$$w(\text{new}) = w(\text{old}) + xy$$

Here the change in weight can be expressed as

$$\Delta w = xy$$

As a result,

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

The Hebb rule can be used for pattern association, pattern categorization, pattern classification and over a range of other areas.

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.

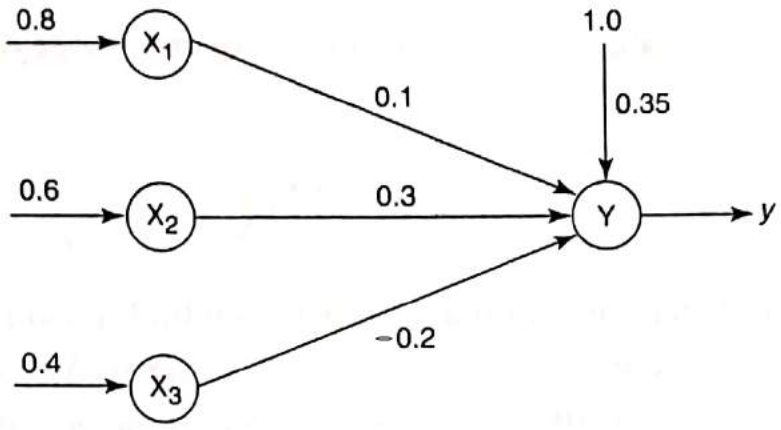


Figure 3 Neural net.

Solution: The given network has three input neurons with bias and one output neuron. These form a single-layer network. The inputs are given as $[x_1, x_2, x_3] = [0.8, 0.6, 0.4]$ and the weights are $[w_1, w_2, w_3] = [0.1, 0.3, -0.2]$ with bias $b = 0.35$ (its input is always 1).

The net input to the output neuron is

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

[$n = 3$, because only 3 input neurons are given]

$$= b + x_1 w_1 + x_2 w_2 + x_3 w_3$$

$$= 0.35 + 0.8 \times 0.1 + 0.6 \times 0.3 + 0.4 \times (-0.2)$$

$$= 0.35 + 0.08 + 0.18 - 0.08 = 0.53$$

(i) For binary sigmoidal activation function,

$$y = f(y_{in}) = \frac{1}{1 + e^{-y_{in}}} = \frac{1}{1 + e^{-0.53}} = 0.625$$

(ii) For bipolar sigmoidal activation function,

$$y = f(y_{in}) = \frac{2}{1 + e^{-y_{in}}} - 1 = \frac{2}{1 + e^{-0.53}} - 1 = 0.259$$

$$y_{in} = b + x_1 w_1 + x_2 w_2$$
$$= 0.45 + 0.2 \times 0.3 + 0.6 \times 0.7$$
$$= 0.45 + 0.06 + 0.42 = 0.93$$

Therefore $y_{in} = 0.93$ is the net input.

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.

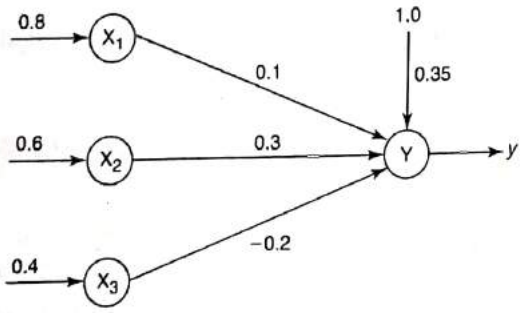


Figure 3 Neural net.

Solution: The given network has three input neurons with bias and one output neuron. These form a single-layer network. The inputs are given as $[x_1, x_2, x_3] = [0.8, 0.6, 0.4]$ and the weights are $[w_1, w_2, w_3] = [0.1, 0.3, -0.2]$ with bias $b = 0.35$ (its input is always 1).

The net input to the output neuron is

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

[$n = 3$, because only 3 input neurons are given]

$$= b + x_1 w_1 + x_2 w_2 + x_3 w_3$$
$$= 0.35 + 0.8 \times 0.1 + 0.6 \times 0.3 + 0.4 \times (-0.2)$$
$$= 0.35 + 0.08 + 0.18 - 0.08 = 0.53$$

(i) For binary sigmoidal activation function,

$$y = f(y_{in}) = \frac{1}{1 + e^{-y_{in}}} = \frac{1}{1 + e^{-0.53}} = 0.625$$

(ii) For bipolar sigmoidal activation function,

$$y = f(y_{in}) = \frac{2}{1 + e^{-y_{in}}} - 1 = \frac{2}{1 + e^{-0.53}} - 1 = 0.259$$

4. Implement AND function using McCulloch–Pitts neuron (take binary data).

Solution: Consider the truth table for AND function (Table 1).

TABLE 1

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

In McCulloch–Pitts neuron, only analysis is being performed. Hence, assume the weights be $w_1 = 1$ and $w_2 = 1$. The network architecture is shown in Figure 4. With these assumed weights, the net input is calculated for four inputs: For inputs

$(1, 1)$, $y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 1 \times 1 = 2$

$(1, 0)$, $y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 0 \times 1 = 1$

$(0, 1)$, $y_{in} = x_1 w_1 + x_2 w_2 = 0 \times 1 + 1 \times 1 = 1$

$(0, 0)$, $y_{in} = x_1 w_1 + x_2 w_2 = 0 \times 1 + 0 \times 1 = 0$

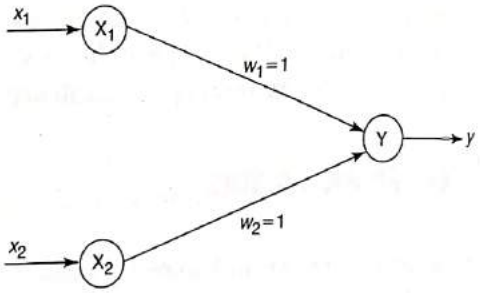


Figure 4 Neural net.

For an AND function, the output is high if both the inputs are high. For this condition, the net input is calculated as 2. Hence, based on this net input, the threshold is set, i.e. if the threshold value is greater than or equal to 2 then the neuron fires, else it does not fire. So the threshold value is set equal to $2(\theta = 2)$. This can also be obtained by

$$\theta \geq nw - p$$

Here, $n = 2$, $w = 1$ (excitatory weights) and $p = 0$ (no inhibitory weights). Substituting these values in the above-mentioned equation we get

$$\theta \geq 2 \times 1 - 0 \Rightarrow \theta \geq 2$$

Thus, the output of neuron Y can be written as

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq 2 \\ 0 & \text{if } y_{in} < 2 \end{cases}$$

where "2" represents the threshold value.

2.1. Hebbian Learning Rule

The **Hebbian rule** was the first learning rule. In 1949 Donald Hebb developed it as learning algorithm of the unsupervised neural network. We can use it to identify how to improve the weights of nodes of a network.

The **Hebb learning rule** assumes that – If two neighbor neurons activated and deactivated at the same time. Then the weight connecting these neurons should increase. For neurons operating in the opposite phase, the weight between them should decrease. If there is no signal correlation, the weight should not change.

When inputs of both the nodes are either positive or negative, then a strong positive weight exists between the nodes. If the input of a node is positive and negative for other, a strong negative weight exists between the nodes. At the start, values of all weights are set to zero. This learning rule can be usedo for both soft- and hard-activation functions. Since desired responses of neurons are not used in the learning procedure, this is the unsupervised learning rule. The absolute values of the weights are usually proportional to the learning time, which is undesired.

The Hebbian learning rule describes the formula as follows:

$$W_{ij} = x_i * x_j$$

Mathematical Formula of Hebb Learning Rule in Artificial Neural Network

2.3. Delta Learning Rule

Developed by Widrow and Hoff, the delta rule, is one of the most common learning rules. It depends on supervised learning.

This rule states that the modification in sympatric weight of a node is equal to the multiplication of error and the input.

In Mathematical form the delta rule is as follows:

$$\Delta w = \eta (t - y) x_i$$

Mathematical Formula of Delta Learning Rule in Artificial Neural Network

For a given input vector, compare the output vector is the correct answer. If the difference is zero, no learning takes place; otherwise, adjusts its weights

to reduce this difference. The change in weight from ui to uj is: dwij = r* ai * ej.
where r is the learning rate, ai represents the activation of ui and ej is the difference between the expected output and the actual output of uj. If the set of input patterns form an independent set then learn arbitrary associations using the delta rule.
It has seen that for networks with linear activation functions and with no hidden units. The error squared vs. the weight graph is a paraboloid in n-space. Since the proportionality constant is negative, the graph of such a function is concave upward and has the least value. The vertex of this paraboloid represents the point where it reduces the error. The weight vector corresponding to this point is then the ideal weight vector.
We can use the delta learning rule with both single output unit and several output units.

While applying the delta rule assume that the error can be directly measured.
The aim of applying the delta rule is to reduce the difference between the actual and expected output that is the error.

3.2.2 Perceptron Learning Rule

In case of the perceptron learning rule, the learning signal is the difference between the desired and actual response of a neuron. The perceptron learning rule is explained as follows:

Consider a finite "n" number of input training vectors, with their associated target (desired) 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.

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

The weight updation in case of perceptron learning is as shown.

If $y \neq t$, then

$$w(\text{new}) = w(\text{old}) + \alpha t x \quad (\alpha - \text{learning rate})$$

else, we have

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

The weights can be initialized at any values in this method. The perceptron rule convergence theorem states that "If there is a weight vector W, such that $f(x(n)W) = t(n)$, for all n, then for any starting vector w_1, the perceptron learning rule will converge to a weight vector that gives the correct response for all training patterns, and this learning takes place within a finite number of steps provided that the solution exists."

3.2.3 Architecture

In the original perceptron network, the output obtained from the associator unit is a binary vector, and hence that output can be taken as input signal to the response unit, and classification can be performed. Here only the weights between the associator unit and the output unit can be adjusted, and the weights between the sensory and associator units are fixed. As a result, the discussion of the network is limited to a single portion. Thus, the associator unit behaves like the input unit. A simple perceptron network architecture is shown in Figure 3-2.

In Figure 3-2, there are n input neurons, 1 output neuron and a bias. The input-layer and output-layer neurons are connected through a directed communication link, which is associated with weights. The goal of the perceptron net is to classify the input pattern as a member or not a member to a particular class.

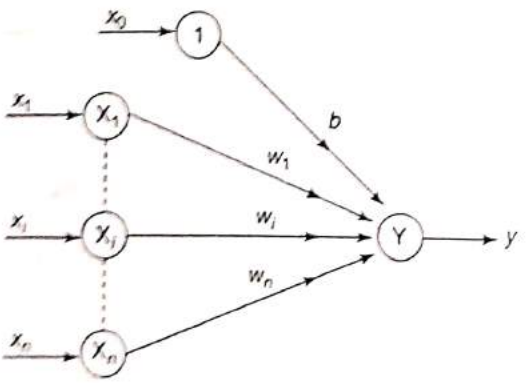


Figure 3-2 Single classification perceptron network.

As depicted in the flowchart, first the basic initialization required for the training process is performed. 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.

3.2.5 Perceptron Training Algorithm for Single Output Classes

The perceptron algorithm can be used for either binary or bipolar input vectors, having bipolar targets, threshold being fixed and variable bias. The algorithm discussed in this section is not particularly sensitive to the initial values of the weights or the value of the learning rate. In the algorithm discussed below, 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. The algorithm of a perceptron network is as follows:

Step 0: Initialize the weights and the bias (for easy calculation they can be set to zero). Also initialize the learning rate $\alpha(0< \alpha \leq 1)$. For simplicity α 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.

The algorithm discussed above is not sensitive to the initial values of the weights or the value of the learning rate.

3.2.6 Perceptron Training Algorithm for Multiple Output Classes

For multiple output classes, the perceptron training algorithm is as follows:

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:

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

Step 5: Make adjustment 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.

It can be noticed that after training, the net classifies each of the training vectors. The above algorithm is suited for t architecture shown in Figure 3–4.

3.2.7 Perceptron Network Testing Algorithm

It is best to test the network performance once the training process is complete. For efficient performance of the netwo it should be trained with more data. The testing algorithm (application procedure) is as follows:

3.3 ADAPTIVE LINEAR NEURON (ADALINE)

3.3.1 Theory

The units with linear activation function are called linear units. A network with a single linear unit is called an *Adaline* (adaptive linear neuron). That is, in an Adaline, the input–output relationship is linear. Adaline uses bipolar activation for its input signals and its target output. The weights between the input and the output are adjustable. The bias in Adaline acts like an adjustable weight, whose connection is from a unit with activations being always 1. Adaline is a net which has only one output unit. The Adaline network may be trained using delta rule. The delta rule may also be called as *least mean square* (LMS) rule or Widrow-Hoff rule. This learning rule is found to minimize the mean-squared error between the activation and the target value.

3.3.2 Delta Rule for Single Output Unit

The Widrow-Hoff rule is very similar to perceptron learning rule. However, their origins are different. The perceptron learning rule originates from the Hebbian assumption while the delta rule is derived from the gradient-descent method (it can be generalized to more than one layer). Also, the perceptron learning rule stops after a finite number of learning steps, but the gradient-descent approach continues forever, converging only asymptotically to the solution. The delta rule updates the weights between the connections so as to minimize the difference between the net input to the output unit and the target value. The major aim is to minimize the error over all training patterns. This is done by reducing the error for each pattern, one at a time.

The delta rule for adjusting the weight of ith pattern ($i = 1$ to n) is

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

where Δw_i is the weight change; α the learning rate; x the vector of activation of input unit; y_{in} the net input to output unit, i.e., $Y = \sum_{i=1}^{n} x_i w_i$; t the target output. The delta rule in case of several output units for adjusting the weight from ith input unit to the jth output unit (for each pattern) is

$$\Delta w_{ij} = \alpha(t_j - y_{inj})x_i$$

3.3.3 Architecture

As already stated, Adaline is a single-unit neuron, which receives input from several units and also from one unit called bias. An Adaline model is shown in Figure 3-5. The basic Adaline model consists of trainable weights. Inputs are either of the two values (+1 or −1) and the weights have signs (positive or negative). Initially, random weights are assigned. The net input calculated is applied to a quantizer transfer function (possibly activation function) that restores the output to +1 or −1. The Adaline model compares the actual output with the target output and on the basis of the training algorithm, the weights are adjusted.

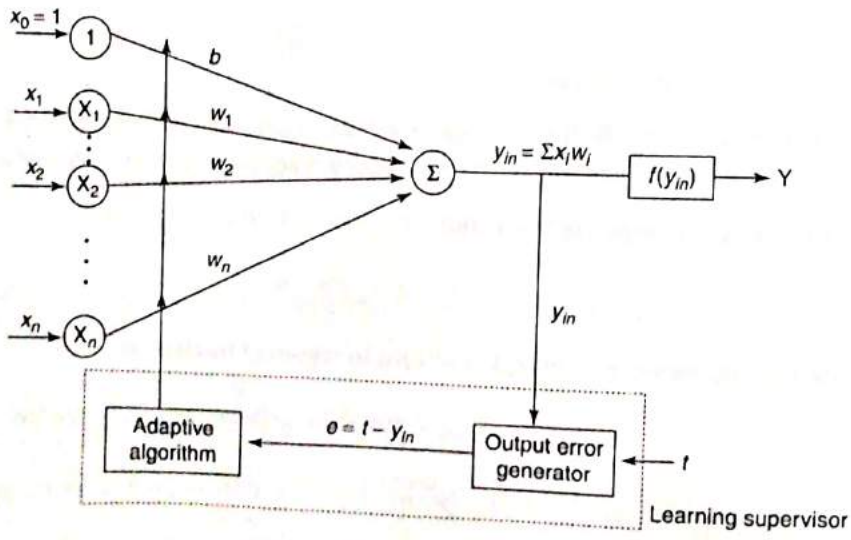


Figure 3-5 Adaline model.

3.3.5 Training Algorithm

The Adaline network training algorithm is as follows:

Step 0: Weights and bias are set to some random values but not zero. Set the learning rate parameter α.

Step 1: Perform Steps 2–6 when stopping condition is false.

Step 2: Perform Steps 3–5 for each bipolar training pair $s:t$.

Step 3: Set activations for input units $i = 1$ to n.

$$x_i = s_i$$

Step 4: Calculate the net input to the output unit.

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

Step 5: Update the weights and bias for $i = 1$ to n:

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

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

Step 6: If the highest weight change that occurred during training is smaller than a specified tolerance then stop the training process, else continue. This is the test for stopping condition of a network.

The range of learning rate can be between 0.1 and 1.0.

3.3.6 Testing Algorithm

It is essential to perform the testing of a network that has been trained. When training is completed, the Adaline can be used to classify input patterns. A step function is used to test the performance of the network. The testing procedure for the Adaline network is as follows:

Step 0: Initialize the weights. (The weights are obtained from the training algorithm.)

Step 1: Perform Steps 2–4 for each bipolar input vector x.

Step 2: Set the activations of the input units to x.

Step 3: Calculate the net input to the output unit:

$$y_{in} = b + \sum x_i w_i$$

Step 4: Apply the activation function over the net input calculated:

$$y = \begin{cases} 1 & \text{if } y_{in} \geq \theta \\ -1 & \text{if } y_{in} < \theta \end{cases}$$

3.4 MULTIPLE ADAPTIVE LINEAR NEURONS

3.4.1 Theory

The multiple adaptive linear neurons (Madaline) model consists of many Adalines in parallel with a single output unit whose value is based on certain selection rules. It may use majority vote rule. On using this rule, the output would have as answer either true or false. On the other hand, if AND rule is used, the output is true if and only if both the inputs are true, and so on. The weights that are connected from the Adaline layer to the Madaline layer are fixed, positive and possess equal values. The weights between the input layer and the Adaline layer are adjusted during the training process. The Adaline and Madaline layer neurons have a bias of excitation "1" connected to them. The training process for a Madaline system is similar to that of an Adaline.

3.4.2 Architecture

A simple Madaline architecture is shown in Figure 3-7, which consists of "n" units of input layer, "m" units of Adaline layer and "1" unit of the Madaline layer. Each neuron in the Adaline and Madaline layers has a bias of excitation 1. The Adaline layer is present between the input layer and the Madaline (output) layer; hence, the Adaline layer can be considered a hidden layer. The use of the hidden layer gives the net computational capability which is not found in single-layer nets, but this complicates the training process to some extent.

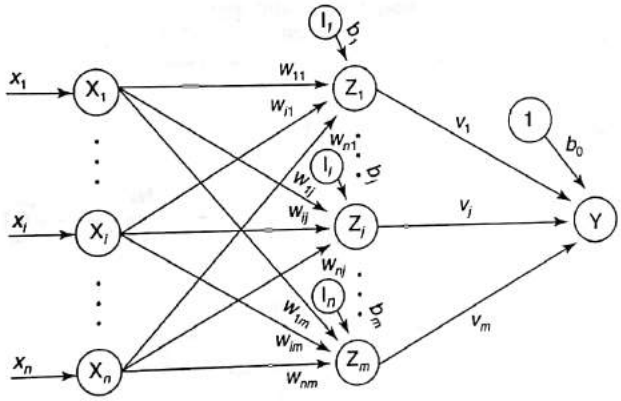


Figure 3-7 Architecture of Madaline layer.

The Adaline and Madaline models can be applied effectively in communication systems of adaptive equalizers and adaptive noise cancellation and other cancellation circuits.

3.4.3 Flowchart of Training Process

The flowchart of the training process of the Madaline network is shown in Figure 3-8. In case of training, the weights between the input layer and the hidden layer are adjusted, and the weights between the hidden layer and the output layer are fixed. The time taken for the training process in the Madaline network is very high compared to that of the Adaline network.

3.4.4 Training Algorithm

In this training algorithm, only the weights between the hidden layer and the input layer are adjusted, and the weights for the output units are fixed. The weights $v_1, v_2, \ldots, v_m$ and the bias b_0 that enter into output unit Y are determined so that the response of unit Y is 1. Thus, the weights entering Y unit may be taken as

$$v_1 = v_2 = \cdots = v_m = \frac{1}{2}$$

Step 0: Initialize the weights. The weights entering the output unit are set as above. Set initial small random values for Adaline weights. Also set initial learning rate α.

Step 1: When stopping condition is false, perform Steps 2–3.

Step 2: For each bipolar training pair *s:t*, perform Steps 3–7.

Step 3: Activate input layer units. For $i = 1$ to n,

$$x_i = s_i$$

Step 4: Calculate net input to each hidden Adaline unit:

$$z_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}, \quad j = 1 \text{ to } m$$

Step 5: Calculate output of each hidden unit:

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

Step 6: Find the output of the net:

$$y_{in} = b_0 + \sum_{j=1}^{m} z_j v_j$$

$$y = f(y_{in})$$

Step 7: Calculate the error and update the weights.

1. If $t = y$, no weight updation is required.
2. If $t \neq y$ and $t = +1$, update weights on z_j, where net input is closest to 0 (zero):

$$b_j(\text{new}) = b_j(\text{old}) + \alpha(1 - z_{inj})$$

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha(1 - z_{inj})x_i$$

3. If $t \neq y$ and $t = -1$, update weights on units z_k whose net input is positive:

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z_{ink})x_i$$

$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z_{ink})$$

Step 8: Test for the stopping condition. (If there is no weight change or weight reaches a satisfactory level, or if a specified maximum number of iterations of weight updation have been performed then stop, or else continue).

Madalines can be formed with the weights on the output unit set to perform some logic functions. If there are only two hidden units present, or if there are more than two hidden units, then the "majority vote rule" function may be used.

3.5 BACK-PROPAGATION NETWORK

3.5.1 Theory

The back-propagation learning algorithm is one of the most important developments in neural networks (Bryson and Ho, 1969; Werbos, 1974; Lecun, 1985; Parker, 1985; Rumelhart, 1986). This network has re-awakened the scientific and

3.5 BACK-PROPAGATION NETWORK

3.5.1 Theory

The back-propagation learning algorithm is one of the most important developments in neural networks (Bryson and Ho, 1969; Werbos, 1974; Lecun, 1985; Parker, 1985; Rumelhart, 1986). This network has re-awakened the scientific and

engineering community to the modeling and processing of numerous quantitative phenomena using neural networks. This learning algorithm is applied to multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions. The networks associated with back-propagation learning algorithm are also called back-propagation networks (BPNs). 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 basic concept for this weight update algorithm is simply the gradient-descent method as used in the case of simple perceptron networks with differentiable units. This is a method where the error is propagated back to the hidden unit. 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 the weights are calculated during the learning period of the network. The general difficulty with the multilayer perceptrons is calculating the weights of the hidden layers in an efficient way that would result in a very small or zero output error. 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. It should be noted that 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, and 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.

3.5.2 Architecture

A back-propagation neural network is a multilayer, feed-forward neural network consisting of an input layer, a hidden layer and an output layer. The neurons present in the hidden and output layers have biases, which are the connections from the units whose activation is always 1. The bias terms also acts as weights. Figure 3-9 shows the architecture of a BPN, depicting only the direction of information flow for the feed-forward phase. During the back-propagation phase of learning, signals are sent in the reverse direction.

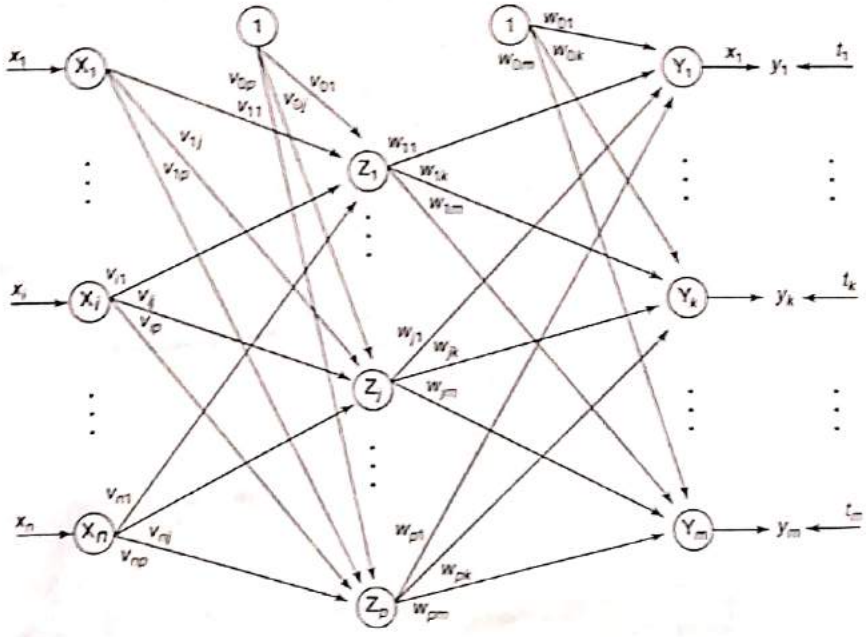


Figure 3-9 Architecture of a back-propagation network.

3.5.4 Training Algorithm

The error back-propagation learning algorithm can be outlined in the following 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 I):

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 $z_j (j = 1 \text{ to } p)$ 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 functions 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 II):

Step 6: Each output unit $y_k (k = 1 \text{ to } m)$ 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 as in Section 2.3.3. 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 δ_k to the hidden layer backwards.

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

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

The term δ_{inj} gets multiplied with the derivative of $f(z_{inj})$ to calculate the error term:

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

The derivative $f'(z_{inj})$ can be calculated as discussed in Section 2.3.3 depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated δ_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 III):

Step 8: Each output unit $(y_k, k = 1 \text{ to } m)$ 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 $(z_j, j = 1 \text{ to } p)$ 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

1. Implement AND function using perceptron networks for bipolar inputs and targets.

Solution: Table 1 shows the truth table for AND function with bipolar inputs and targets:

TABLE 1

x_1	x_2	t
1	1	1
1	−1	−1
−1	1	−1
−1	−1	−1

The perceptron network, which uses perceptron learning rule, is used to train the AND function. The network architecture is as shown in Figure 1. The input patterns are presented to the network one by one. When all the four input patterns are presented, then one epoch is said to be completed. The initial weights and threshold are set to zero, i.e., $w_1 = w_2 = b = 0$ and $\theta = 0$. The learning rate α is set equal to 1.

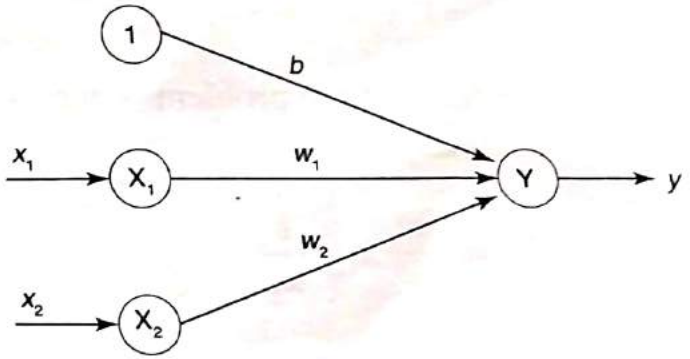


Figure 1 Perceptron network for AND function.

For the first input pattern, $x_1 = 1, x_2 = 1$ and $t = 1$, with weights and bias, $w_1 = 0$, $w_2 = 0$ and $b = 0$:

- Calculate the net input

$$y_{in} = b + x_1 w + x_2 w$$
$$= 0 + 1 \times 0 + 1 \times 0 = 0$$

- The output y is computed by applying activations over the net input calculated:

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

Here we have taken $\theta = 0$. Hence, when, $y_{in} = 0, y = 0$.

- Check whether $t = y$. Here, $t = 1$ and $y = 0$, so $t \neq y$, hence weight updation takes place:

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

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

Here, the change in weights are

$$\Delta w_1 = \alpha t x_1;$$
$$\Delta w_2 = \alpha t x_2;$$
$$\Delta b = \alpha t$$

The weights $w_1 = 1$, $w_2 = 1$, $b = 1$ are the final weights after first input pattern is presented. The same process is repeated for all the input patterns. The process can be stopped when all the targets become equal to the calculated output or when a separating line is obtained using the final weights for separating the positive responses from negative responses. Table 2 shows the training of perceptron network until its target and calculated output converge for all the patterns.

TABLE 2

Input			Target	Net input	Calculated output	Weight changes			Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
EPOCH-1											
1	1	1	1	0	0	1	1	1	1	1	1
1	−1	1	−1	1	1	−1	1	−1	0	2	0
−1	1	1	−1	2	1	+1	−1	−1	1	1	−1
−1	−1	1	−1	−3	−1	0	0	0	1	1	−1
EPOCH-2											
1	1	1	1	1	1	0	0	0	1	1	−1
1	−1	1	−1	−1	−1	0	0	0	1	1	−1
−1	1	1	−1	−1	−1	0	0	0	1	1	−1
−1	−1	1	−1	−3	−1	0	0	0	1	1	−1

The final weights and bias after second epoch are

$$w_1 = 1, w_2 = 1, b = -1$$

Since the threshold for the problem is zero, the equation of the separating line is

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$

Here

$$w_1 x_1 + w_2 x_2 + b > \theta$$
$$w_1 x_1 + w_2 x_2 + b > 0$$

Thus, using the final weights we obtain

$$x_2 = -\frac{1}{1} x_1 - \frac{(-1)}{1}$$
$$x_2 = -x_1 + 1$$

It can be easily found that the above straight line separates the positive response and negative response region, as shown in Figure 2.

The same methodology can be applied for implementing other logic functions such as OR, ANDNOT, NAND, etc. If there exists a threshold value $\theta \neq 0$, then two separating lines have to be obtained, i.e., one to separate positive response from zero and the other for separating zero from the negative response.

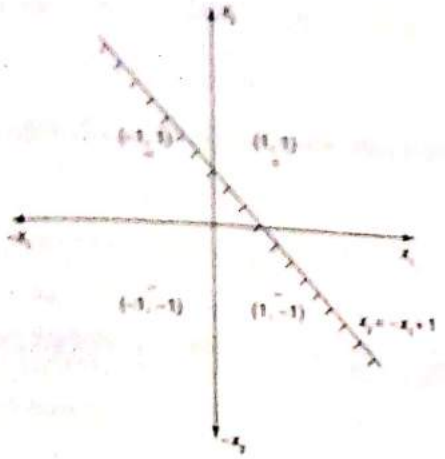


Figure 2 Decision boundary for AND function in perceptron training ($\theta = 0$).

2. Implement OR function with binary inputs and bipolar targets using perceptron training algorithm upto 3 epochs.

Solution: The truth table for OR function with binary inputs and bipolar targets is shown in Table 3.

TABLE 3

x_1	x_2	t
1	1	1
1	0	1
0	1	1
0	0	−1

The perceptron network, which uses perceptron learning rule, is used to train the OR function. The network architecture is shown in Figure 3. The initial values of the weights and bias are taken as zero, i.e.,

$$w_1 = w_2 = b = 0$$

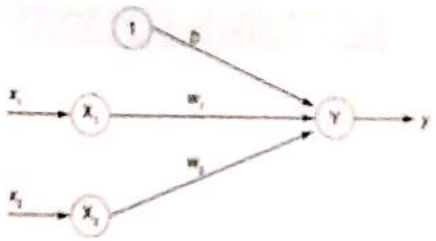


Figure 3 Perceptron network for OR function.

Also the learning rate is 1 and threshold is 0.2. So, the activation function becomes

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0.2 \\ 0 & \text{if } -0.2 \le y_{in} \le 0.2 \end{cases}$$

The network is trained as per the perceptron training algorithm and the steps are as in problem 1 (given for first pattern). Table 4 gives the network training for 3 epochs.

The final weights at the end of third epoch are

$$w_1 = 2,\ w_2 = 1,\ b = -1$$

Further epochs have to be done for the convergence of the network.

TABLE 4

Input			Target	Net input	Calculated output	Weight changes			Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
EPOCH-1											
1	1	1	1	0	0	1	1	1	1	1	1
1	0	1	1	2	1	0	0	0	1	1	1
0	1	1	1	2	1	0	0	0	1	1	0
0	0	1	−1	1	1	0	0	−1	1	1	0
EPOCH-2											
1	1	1	1	2	1	0	0	0	1	1	0
1	0	1	1	1	1	0	0	0	1	1	0
0	1	1	1	1	1	0	0	0	1	1	0
0	0	1	−1	0	0	0	0	0	1	1	−1
EPOCH-3											
1	1	1	1	1	1	0	0	0	1	1	1
1	0	1	1	0	0	1	0	1	2	1	0
0	1	1	1	1	1	0	0	0	2	1	0
0	0	1	−1	0	0	0	0	−1	2	1	−1

TABLE 14—CONTINUED

Inputs			Target	Net input		Weight changes			Weights			Error
x_1	x_2	1	t	y_{in}	$(t - y_{in})$	Δw_1	Δw_2	Δb	w_1 (0.2	w_2 0.2	b 0.2)	$(t - y_{in})^2$
EPOCH-2									0.28	−0.43	−0.46	0.95
1	1	1	−1	−0.02	−0.98	−0.195	−0.195	−0.195	0.43	−0.58	−0.31	0.57
1	−1	1	1	0.25	0.76	0.15	−0.15	0.15	0.37	−0.51	−0.25	0.106
−1	1	1	−1	−1.33	0.33	−0.065	0.065	0.065	0.55	−0.38	0.43	0.8
−1	−1	1	−1	−0.11	−0.90	0.18	0.18	−0.18				

TABLE 15

Epoch	Total mean square error
Epoch 1	5.71
Epoch 2	2.43

Hence from Table 15, it is clearly understood that the mean square error decreases as training progresses. Also, it can be noted that at the end of the sixth epoch, the error becomes approximately equal to 1. The network architecture for ANDNOT function using Adaline network is shown in Figure 9.

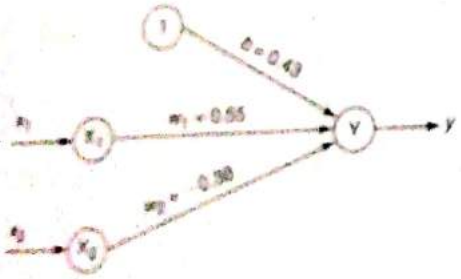


Figure 9 Network architecture for ANDNOT function using Adaline network.

8. Using Madaline network, implement XOR function with bipolar inputs and targets. Assume the required parameters for training of the network.

Solution: The training pattern for XOR function is given in Table 16.

TABLE 16

x_1	x_2	1	t
1	1	1	−1
1	−1	1	1
−1	1	1	1
−1	−1	1	−1

The Madaline Rule I (MRI) algorithm in which the weights between the hidden layer and output layer remain fixed is used for training the network. Initializing the weights to small random values, the network architecture is as shown in Figure 10, with initial weights. From Figure 10, the initial weights and bias are $[w_{11}\ w_{21}\ b_1] = [0.05\ 0.2\ 0.3]$, $[w_{12}\ w_{22}\ b_2] = [0.1\ 0.2\ 0.15]$ and $[v_1\ v_2\ b_3] = [0.5\ 0.5\ 0.5]$. For first input sample, $x_1 = 1$, $x_2 = 1$, target $t = -1$, and learning rate α equal to 0.5:

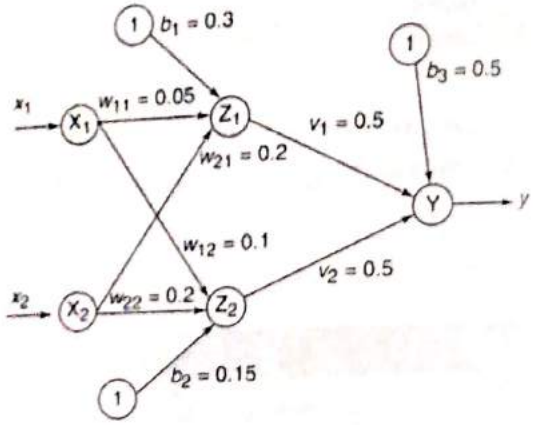


Figure 10 Network architecture of Madaline for XOR functions (initial weights given).

- Calculate net input to the hidden units:

$$z_{in1} = b_1 + x_1 w_{11} + x_2 w_{21}$$
$$= 0.3 + 1 \times 0.05 + 1 \times 0.2 = 0.55$$
$$z_{in2} = b_2 + x_1 w_{12} + x_2 w_{22}$$
$$= 0.15 + 1 \times 0.1 + 1 \times 0.2 = 0.45$$

- Calculate the output z_1, z_2 by applying the activations over the net input computed. The activation function is given by

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

Hence,

$$z_1 = f(z_{in1}) = f(0.55) = 1$$
$$z_2 = f(z_{in2}) = f(0.45) = 1$$

- After computing the output of the hidden units, then find the net input entering into the output unit:

$$y_{in} = b_3 + z_1 v_1 + z_2 v_2$$
$$= 0.5 + 1 \times 0.5 + 1 \times 0.5 = 1.5$$

- Apply the activation function over the net input y_{in} to calculate the output y:

$$y = f(y_{in}) = f(1.5) = 1$$

Since $t \neq y$, weight updation has to be performed. Also since $t = -1$, the weights are updated on z_1 and z_2 that have positive net input. Since here both net inputs z_{in1} and z_{in2} are positive, updating the weights and bias on both hidden units, we obtain

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha(t - z_{inj})x_i$$
$$b_j(\text{new}) = b_j(\text{old}) + \alpha(t - z_{inj})$$

This implies:

$$w_{11}(\text{new}) = w_{11}(\text{old}) + \alpha(t - z_{in1})x_1$$
$$= 0.05 + 0.5(-1 - 0.55) \times 1 = -0.725$$
$$w_{12}(\text{new}) = w_{12}(\text{old}) + \alpha(t - z_{in2})x_1$$
$$= 0.1 + 0.5(-1 - 0.45) \times 1 = -0.625$$
$$b_1(\text{new}) = b_1(\text{old}) + \alpha(t - z_{in1})$$
$$= 0.3 + 0.5(-1 - 0.55) = -0.475$$
$$w_{21}(\text{new}) = w_{21}(\text{old}) + \alpha(t - z_{in1})x_2$$
$$= 0.2 + 0.5(-1 - 0.55) \times 1 = -0.575$$
$$w_{22}(\text{new}) = w_{22}(\text{old}) + \alpha(t - z_{in2})x_2$$
$$= 0.2 + 0.5(-1 - 0.45) \times 1 = -0.525$$
$$b_2(\text{new}) = b_2(\text{old}) + \alpha(t - z_{in2})$$
$$= 0.15 + 0.5(-1 - 0.45) = -0.575$$

All the weights and bias between the input layer and hidden layer are adjusted. This completes the training for the first epoch. The same process is repeated until the weight converges. It is found that the weight converges at the end of 3 epochs. Table 17 shows the training performance of Madaline network for XOR function.

TABLE 17

Inputs			Target												
x_1	x_2	1	(t)	z_{in1}	z_{in2}	z_1	z_2	y_{in}	Y	w_{11}	w_{21}	b_1	w_{12}	w_{22}	b_2
EPOCH-1															
1	1	1	−1	0.55	0.45	1	1	1.5	1	−0.725	−0.58	−0.475	−0.625	−0.525	−0.575
1	−1	1	1	−0.625	−0.675	−1	−1	−0.5	−1	0.0875	−1.39	0.34	−0.625	−0.525	−0.575
−1	1	1	1	−1.1375	−0.475	−1	−1	−0.5	−1	0.0875	−1.39	0.34	−1.3625	0.2125	0.1625
−1	−1	1	−1	1.6375	1.3125	1	1	1.5	1	1.4065	−0.069	−0.98	−0.207	1.369	−0.994
EPOCH-2															
1	1	1	−1	0.3565	−0.168	1	1	1.5	1	0.7285	−0.75	−1.66	−0.791	−0.207	−1.58
1	−1	1	1	−0.1845	−3.154	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−0.791	0.785	−1.58
−1	1	1	1	−3.728	−0.002	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−1.29	0.785	−1.08
−1	−1	1	−1	−1.0495	−1.071	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−1.29	1.29	−1.08
EPOCH-3															
1	1	1	−1	−1.0865	−1.083	−1	−1	−0.5	−1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
1	−1	1	1	1.5915	−3.655	1	−1	0.5	1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
−1	1	1	1	−3.728	1.501	−1	1	0.5	1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
−1	−1	1	−1	−1.0495	−1.701	−1	−1	−0.5	−1	1.32	−1.34	−1.07	−1.29	1.29	−1.08

The network architecture for Madaline network with final weights for XOR function is shown in Figure 11.

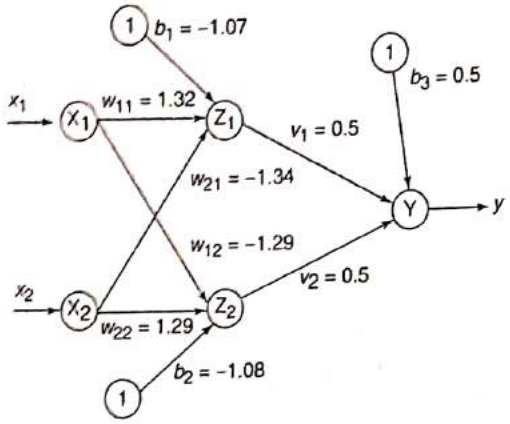


Figure 11 Madaline network for XOR function (final weights given).

9. Using back-propagation network, find the new weights for the net shown in Figure 12. It is presented with the input pattern [0, 1] and the target output is 1. Use a learning rate $\alpha = 0.25$ and binary sigmoidal activation function.

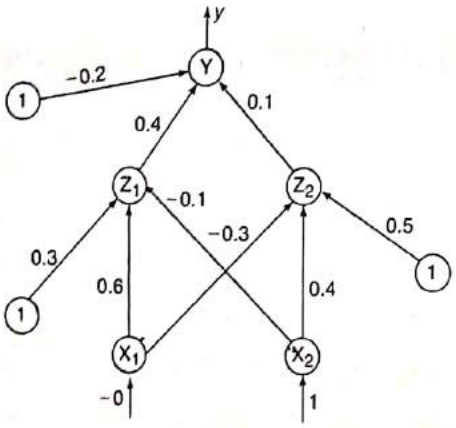


Figure 12 Network.

Solution: The new weights are calculated based on the training algorithm in Section 3.5.4. The initial weights are $[v_{11}\ v_{21}\ v_{01}] = [0.6\ -0.1\ 0.3]$,

$[v_{12}\ v_{22}\ v_{02}] = [-0.3\ 0.4\ 0.5]$ and $[w_1\ w_2\ w_0] = [0.4\ 0.1\ -0.2]$, and the learning rate is $\alpha = 0.25$. Activation function used is binary sigmoidal activation function and is given by

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

Given the output sample $[x_1, x_2] = [0, 1]$ and ta $t = 1$,

- Calculate the net input: For z_1 layer

$$z_{in1} = v_{01} + x_1 v_{11} + x_2 v_{21}$$
$$= 0.3 + 0 \times 0.6 + 1 \times -0.1 = 0.2$$

For z_2 layer

$$z_{in2} = v_{02} + x_1 v_{12} + x_2 v_{22}$$
$$= 0.5 + 0 \times -0.3 + 1 \times 0.4 = 0.9$$

Applying activation to calculate the output, we obta

$$z_1 = f(z_{in1}) = \frac{1}{1+e^{-z_{in1}}} = \frac{1}{1+e^{-0.2}} = 0.5498$$

$$z_2 = f(z_{in2}) = \frac{1}{1+e^{-z_{in2}}} = \frac{1}{1+e^{-0.9}} = 0.7109$$

- Calculate the net input entering the output lay For y layer

$$y_{in} = w_0 + z_1 w_1 + z_2 w_2$$
$$= -0.2 + 0.5498 \times 0.4 + 0.7109 \times 0.1$$
$$= 0.09101$$

Applying activations to calculate the output, we obta

$$y = f(y_{in}) = \frac{1}{1+e^{-y_{in}}} = \frac{1}{1+e^{-0.09101}} = 0.5227$$

- Compute the error portion δ_k :

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

Now

$$f'(y_{in}) = f(y_{in})[1 - f(y_{in})] = 0.5227[1 - 0.5227]$$

$$f'(y_{in}) = 0.2495$$

This implies

$$\delta_1 = (1 - 0.5227)(0.2495) = 0.1191$$

Find the changes in weights between hidden an output layer:

$$\Delta w_1 = \alpha \delta_1 z_1 = 0.25 \times 0.1191 \times 0.5498$$
$$= 0.0164$$

$$\Delta w_2 = \alpha \delta_1 z_2 = 0.25 \times 0.1191 \times 0.7109$$
$$= 0.02117$$
$$\Delta w_0 = \alpha \delta_1 = 0.25 \times 0.1191 = 0.02978$$

- Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

$$\delta_j = \delta_{inj} f'(z_{inj})$$
$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$
$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$
$$\Rightarrow \delta_{in1} = \delta_1 w_{11} = 0.1191 \times 0.4 = 0.04764$$
$$\Rightarrow \delta_{in2} = \delta_1 w_{21} = 0.1191 \times 0.1 = 0.01191$$

Error, $\delta_1 = \delta_{in1} f'(z_{in1})$

$$f'(z_{in1}) = f(z_{in1})[1 - f(z_{in1})]$$
$$= 0.5498[1 - 0.5498] = 0.2475$$
$$\delta_1 = \delta_{in1} f'(z_{in1})$$
$$= 0.04764 \times 0.2475 = 0.0118$$

Error, $\delta_2 = \delta_{in2} f'(z_{in2})$

$$f'(z_{in2}) = f(z_{in2})[1 - f(z_{in2})]$$
$$= 0.7109[1 - 0.7109] = 0.2055$$
$$\delta_2 = \delta_{in2} f'(z_{in2})$$
$$= 0.01191 \times 0.2055 = 0.00245$$

Now find the changes in weights between input and hidden layer:

$$\Delta v_{11} = \alpha \delta_1 x_1 = 0.25 \times 0.0118 \times 0 = 0$$
$$\Delta v_{21} = \alpha \delta_1 x_2 = 0.25 \times 0.0118 \times 1 = 0.00295$$
$$\Delta v_{01} = \alpha \delta_1 = 0.25 \times 0.0118 = 0.00295$$
$$\Delta v_{12} = \alpha \delta_2 x_1 = 0.25 \times 0.00245 \times 0 = 0$$
$$\Delta v_{22} = \alpha \delta_2 x_2 = 0.25 \times 0.00245 \times 1 = 0.0006125$$
$$\Delta v_{02} = \alpha \delta_2 = 0.25 \times 0.00245 = 0.0006125$$

- Compute the final weights of the network:

$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11} = 0.6 + 0 = 0.6$$
$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12} = -0.3 + 0 = -0.3$$
$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21}$$
$$= -0.1 + 0.00295 = -0.09705$$
$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22}$$
$$= 0.4 + 0.0006125 = 0.4006125$$
$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.4 + 0.0164$$
$$= 0.4164$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1 + 0.02117$$
$$= 0.12117$$
$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.00295$$
$$= 0.30295$$
$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02}$$
$$= 0.5 + 0.0006125 = 0.5006125$$
$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0 = -0.2 + 0.02978$$
$$= -0.17022$$

Thus, the final weights have been computed for the network shown in Figure 12.

10. Find the new weights, using back-propagation network for the network shown in Figure 13. The network is presented with the input pattern [– 1, 1] and the target output is + 1. Use a learning rate of $\alpha = 0.25$ and bipolar sigmoidal activation function.

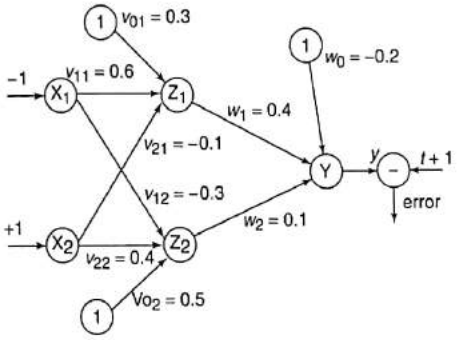


Figure 13 Network.

Solution: The initial weights are $[v_{11} v_{21} v_{01}] = [0.6 \ -0.1 \ 0.3]$, $[v_{12} \ v_{22} \ v_{02}] = [-0.3 \ 0.4 \ 0.5]$ and $[[w_1 \ w_2 \ w_0] = [0.4 \ 0.1 \ -0.2]$, and the learning rate is $\alpha = 0.25$.

Activation function used is binary sigmoidal activation function and is given by

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

Given the input sample $[x_1, x_2] = [-1, 1]$ and target $t = 1$:

- Calculate the net input: For z_1 layer

$$z_{in1} = v_{01} + x_1 v_{11} + x_2 v_{21}$$
$$= 0.3 + (-1) \times 0.6 + 1 \times -0.1 = -0.4$$

For z_2 layer

$$z_{in2} = v_{02} + x_1 v_{12} + x_2 v_{22}$$
$$= 0.5 + (-1) \times -0.3 + 1 \times 0.4 = 1.2$$

Applying activation to calculate the output, we obtain

$$z_1 = f(z_{in1}) = \frac{1 - e^{-z_{in1}}}{1 + e^{-z_{in1}}} = \frac{1 - e^{0.4}}{1 + e^{0.4}} = -0.1974$$

$$z_2 = f(z_{in2}) = \frac{1 - e^{-z_{in1}}}{1 + e^{-z_{in2}}} = \frac{1 - e^{-1.2}}{1 + e^{-1.2}} = 0.537$$

- Calculate the net input entering the output layer. For y layer

$$y_{in} = w_0 + z_1 w_1 + z_2 w_2$$
$$= -0.2 + (-0.1974) \times 0.4 + 0.537 \times 0.1$$
$$= -0.22526$$

Applying activations to calculate the output, we obtain

$$y = f(y_{in}) = \frac{1 - e^{-y_{in}}}{1 + e^{-y_{in}}} = \frac{1 - e^{0.22526}}{1 + e^{0.22526}} = -0.1122$$

- Compute the error portion δ_k:

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

Now

$$f'(y_{in}) = 0.5[1 + f(y_{in})][1 - f(y_{in})]$$
$$= 0.5\,[1 - 0.1122]\,[1 + 0.1122] = 0.4937$$

This implies

$$\delta_1 = (1 + 0.1122)(0.4937) = 0.5491$$

Find the changes in weights between hidden and output layer:

$$\Delta w_1 = \alpha \delta_1 z_1 = 0.25 \times 0.5491 \times -0.1974$$
$$= -0.0271$$

$$\Delta w_2 = \alpha \delta_1 z_2 = 0.25 \times 0.5491 \times 0.537 = 0.0737$$
$$\Delta w_0 = \alpha \delta_1 = 0.25 \times 0.5491 = 0.1373$$

- Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

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

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

$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$

$$\Rightarrow \delta_{in1} = \delta_1 w_{11} = 0.5491 \times 0.4 = 0.21964$$
$$\Rightarrow \delta_{in2} = \delta_2 w_{21} = 0.5491 \times 0.1 = 0.05491$$

Error, $\delta_1 = \delta_{in1} f'(z_{in1})$
$$= 0.21964 \times 0.5 \times (1 + 0.1974)(1 - 0.1974)$$
$$= 0.1056$$

Error, $\delta_2 = \delta_{in2} f'(z_{in2})$
$$= 0.05491 \times 0.5 \times (1 - 0.537)(1 + 0.537)$$
$$= 0.0195$$

Now find the changes in weights between input and hidden layer:

$$\Delta v_{11} = \alpha \delta_1 x_1 = 0.25 \times 0.1056 \times -0.0264$$
$$\Delta v_{21} = \alpha \delta_1 x_2 = 0.25 \times 0.1056 \times 1 = 0.0264$$
$$\Delta v_{01} = \alpha \delta_1 = 0.25 \times 0.1056 = 0.0264$$
$$\Delta v_{12} = \alpha \delta_2 x_1 = 0.25 \times 0.0195 \times -1 = 0.0049$$
$$\Delta v_{22} = \alpha \delta_2 x_2 = 0.25 \times 0.0195 \times 1 = 0.0049$$
$$\Delta v_{02} = \alpha \delta_2 = 0.25 \times 0.0195 = 0.0049$$

- Compute the final weights of the network:

$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11} = 0.6 - 0.0264 = 0.5736$$
$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12} = -0.3 - 0.0049 = -0.3049$$
$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21} = -0.1 + 0.0264 = -0.0736$$
$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22} = 0.4 + 0.0049 = 0.4049$$
$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.4 - 0.0271 = 0.3729$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1 + 0.0737 = 0.1737$$
$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.0264 = 0.3264$$
$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02} = 0.5 + 0.0049 = 0.5049$$
$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0 = -0.2 + 0.1373 = -0.0627$$

Thus, the final weight has been computed for the network shown in Figure 13.

CHAPTER 4

Associative Memory Networks

LEARNING OBJECTIVES

- Gives details on associative memories.
- Discusses the training algorithm used for pattern association networks – Hebb rule and outer products rule.
- The architecture, flowchart for training process, training algorithm and testing algorithm of autoassociative, heteroassociative and bidirectional associative memory are discussed in detail.
- Variants of BAM – continuous BAM and discrete BAM are included.
- Hopfield network with its electrical model is described with training algorithm.
- Analysis of energy function was performed for BAM, discrete and continuous Hopfield networks.
- An overview is given on the iterative autoassociative network – linear autoassociator memory brain-in-the-box network and autoassociator with threshold unit.
- Also temporal associative memory is discussed in brief.

4.1 INTRODUCTION

An associative memory network can store a set of patterns as memories. When an associative memory is being presented with a key pattern, it responds by producing one of the stored patterns, which closely resembles or relates to the key pattern. Thus, the recall is through association of the key pattern, with the help of information memorized. These types of memories are also called as *content-addressable memories* (CAM) in contrast to that of traditional *address-addressable memories* in digital computers where stored pattern (in bytes) is recalled by its address. It is also a matrix memory as in RAM/ROM. The CAM can also be viewed as associating data to address, i.e., for every data in the memory there is a corresponding unique address. Also, it can be viewed as data correlator. Here input data is correlated with that of the stored data in the CAM. It should be noted that the stored patterns must be unique, i.e., different patterns in each location. If the same pattern exists in more than one location in the CAM, then, even though the correlation is correct, the address is noted to be ambiguous. The basic structure of CAM is given in Figure 4-1.

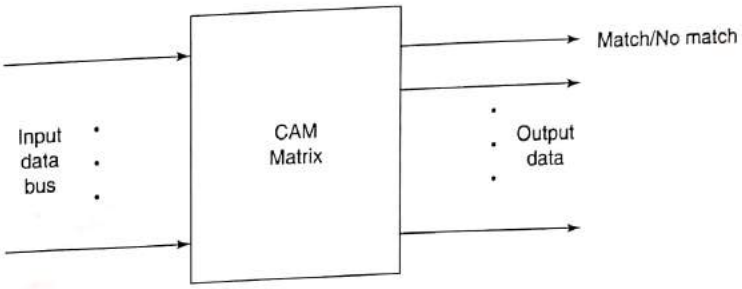


Figure 4-1 CAM architecture.

Associative memory makes a parallel search within a stored data file. The concept behind this search is to output any one or all stored items which match the given search argument and to retrieve the stored data either completely or partially.

Two types of associative memories can be differentiated. They are *autoassociative memory* and *heteroassociative memory*. Both these nets are single-layer nets in which the weights are determined in a manner that the net stores a set of pattern associations. Each of this association is an input–output vector pair, say, *s*:*t*. If each of the output vectors is same as the input vectors with which it is associated, then the net is a said to be autoassociative memory net. On the other hand, if the output vectors are different from the input vectors then the net is said to be heteroassociative memory net.

If there exist vectors, say, $x = (x_1, x_2, \ldots, x_n)^T$ and $x' = (x_1', x_2', \ldots, x_n')^T$, then the hamming distance (HD) is defined as the number of mismatched components of x and x' vectors, i.e.,

$$\text{HD}(x, x') = \begin{cases} \sum_{i=1}^{n} |x_i - x_i'| & \text{if } x_i, x_i' \in [0, 1] \\ \frac{1}{2}\sum_{i=1}^{n} |x_i - x_i'| & \text{if } x_i, x_i' \in [-1, 1] \end{cases}$$

The architecture of an associative net may be either feed-forward or iterative (recurrent). As is already known, in a feed-forward net the information flows from the input units to the output units; on the other hand, in a recurrent neural net, there are connections among the units to form a closed-loop structure. In the forthcoming sections, we will discuss the training algorithms used for pattern association and various types of association nets in detail.

4.2 TRAINING ALGORITHMS FOR PATTERN ASSOCIATION

There are two algorithms developed for training of pattern association nets. These are discussed below.

4.2.1 Hebb Rule

The Hebb rule is widely used for finding the weights of an associative memory neural net. The training vector pairs here are denoted as *s*:*t*. The flowchart for the training algorithm of pattern association is as shown in Figure 4-2. The weights are updated until there is no weight change. The algorithmic steps followed are given below:

Step 0: Set all the initial weights to zero, i.e.,

$$w_{ij} = 0 \quad (i = 1 \text{ to } n, j = 1 \text{ to } m)$$

Step 1: For each training target input output vector pairs *s*:*t*, perform Steps 2–4.

Step 2: Activate the input layer units to current training input,

$$x_i = s_i \quad (\text{for } i = 1 \text{ to } n)$$

Step 3: Activate the output layer units to current target output,

$$y_j = t_j \quad (\text{for } j = 1 \text{ to } m)$$

Step 4: Start the weight adjustment

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + x_i y_j \quad (\text{for } i = 1 \text{ to } n, \ j = 1 \text{ to } m)$$

4.2.2 Outer Products Rule

Outer products rule is an alternative method for finding weights of an associative net. This is depicted as follows:

$$\text{Input} \Rightarrow s = (s_1, \ldots, s_i, \ldots, s_n)$$

$$\text{Output} \Rightarrow t = (t_1, \ldots, t_j, \ldots, t_m)$$

The outer product of the two vectors is the product of the matrices $S = s^T$ and $T = t$, i.e., between $[n \times 1]$ matrix and $[1 \times m]$ matrix. The transpose is to be taken for the input matrix given.

The matrix multiplication is done as follows:

$$ST = s^T t$$

$$= \begin{bmatrix} s_1 \\ \vdots \\ s_i \\ \vdots \\ s_n \end{bmatrix}_{n\times 1} [t_1 \ldots t_j \ldots t_m]_{1\times m}$$

$$W = \begin{bmatrix} s_1t_1 & \cdots & s_1t_j & \cdots & s_1t_m \\ \vdots & & & & \vdots \\ s_it_1 & \cdots & s_it_j & \cdots & s_it_m \\ \vdots & & & & \vdots \\ s_nt_1 & \cdots & s_nt_j & \cdots & s_nt_m \end{bmatrix}_{n\times m}$$

This weight matrix is same as the weight matrix obtained by Hebb rule to store the pattern association *s:t*.

For storing a set of associations, $s(p):t(p)$, $p=1$ to P, wherein,

$$s(p) = (s_1(p),\ldots,s_i(p),\ldots,s_n(p))$$
$$t(p) = (t_1(p),\ldots,t_j(p),\ldots,t_m(p))$$

the weight matrix $W = \{w_{ij}\}$ can be given as

$$w_{ij} = \sum_{p=1}^{P} s_i^T(p)t_j(p)$$

This can also be rewritten as

$$W = \sum_{p=1}^{P} s^T(p)t(p)$$

for finding the weights of the net using Hebbian learning. Similar to the Hebb rule, even the delta rule discussed in Chapter 2 can be used for storing weights of pattern association nets.

4.3 AUTOASSOCIATIVE MEMORY NETWORK

4.3.1 Theory

In the case of an autoassociative neural net, the training input and the target output vectors are the same. The determination of weights of the association net is called storing of vectors. This type of memory net needs suppression of the output noise at the memory output. The vectors that have been stored can be retrieved from distorted (noisy) input if the input is sufficiently similar to it. The net's performance is based on its ability to reproduce a stored pattern from a noisy input. It should be noted, that in the case of autoassociative net, the weights on the diagonal can be set to zero. This can be called as auto associative net with no self-connection. The main reason behind setting the weights to zero is that it improves the net's ability to generalize or increase the biological plausibility of the net. This may be more suited for iterative nets and when delta rule is being used.

4.3.2 Architecture

The architecture of an autoassociative neural net is shown in Figure 4-3. It shows that for an autoassociative net, the training input and target output vectors are the same. The input layer consists of n input units and the output layer also consists of n output units. The input and output layers are connected through weighted interconnections. The input and output vectors are perfectly correlated with each other component by component.

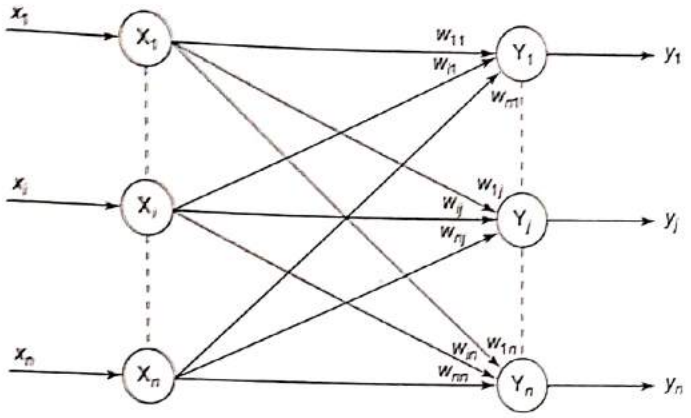


Figure 4-3 Architecture of autoassociative net.

4.3.3 Flowchart for Training Process

The flowchart here is the same as discussed in Section 4.2.1, but it may be noted that the number of input units and output units are the same. The flowchart is shown in Figure 4-4.

4.3.4 Training Algorithm

The training algorithm discussed here is similar to that discussed in Section 4.2.1 but there are same numbers of output units as that of the input units.

Step 0: Initialize all the weights to zero,

$$w_{ij} = 0 \quad (i = 1 \text{ to } n,\ j = 1 \text{ to } n)$$

Step 1: For each of the vector that has to be stored perform Steps 2–4.

Step 2: Activate each of the input unit,

$$x_i = s_i \ (i = 1 \text{ to } n)$$

Step 3: Activate each of the output unit,

$$y_j = s_j \ (j = 1 \text{ to } n)$$

Step 4: Adjust the weights,

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

The weights can also be determined by the formula

$$W = \sum_{p=1}^{P} s^{T}(p) s(p)$$

Figure 4-4 Flowchart for training of autoassociative net.

4.3.5 Testing Algorithm

An autoassociative memory neural network can be used to determine whether the given input vector is a "known" vector or an "unknown" vector. The net is said to recognize a "known" vector if the net produces a pattern of activation on the output units which is same as one of the vectors stored in it. The testing procedure of an autoassociative neural net is as follows:

Step 0: Set the weights obtained for Hebb's rule or outer products.

Step 1: For each of the testing input vector presented perform Steps 2–4.

Step 2: Set the activations of the input units equal to that of input vector.

Step 3: Calculate the net input to each output unit $j = 1$ to n:

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

Step 4: Calculate the output by applying the activation over the net input:

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

This type of network can be used in speech processing, image processing, pattern classification, etc.

This type of network can be used in speech processing, image processing, pattern classification, etc.

4.4 HETEROASSOCIATIVE MEMORY NETWORK

4.4.1 Theory

In case of a heteroassociative neural net, the training input and the target output vectors are different. The weights are determined in a way that the net can store a set of pattern associations. The association here is a pair of training input target output vector pairs $(s(p), t(p))$, with $p = 1,\ldots,P$. Each vector $s(p)$ has n components and each vector $t(p)$ has m components. The determination of weights is done either by using Hebb rule or delta rule. The net finds an appropriate output vector, which corresponds to an input vector x, that may be either one of the stored patterns or a new pattern.

4.4.2 Architecture

The architecture of a heteroassociative net is shown in Figure 4-5. From the figure, it can be noticed that for a heteroassociative net, the training input and target output vectors are different. The input layer consists of n number of input units and the output layer consists of m number of output units. There exist weighted interconnections between the input and output layers. The input and output layer units are not correlated with each other. The flowchart of the training process and the training algorithm are the same as discussed in Section 4.2.1.

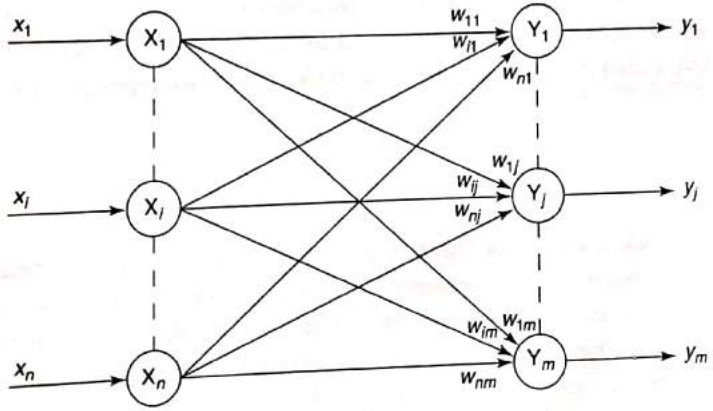


Figure 4-5 Architecture of heteroassociative net.

4.4.3 Testing Algorithm

The testing algorithm used for testing the heteroassociative net with either noisy input or with known input is as fo

Step 0: Initialize the weights from the training algorithm.

Step 1: Perform Steps 2–4 for each input vector presented.

Step 2: Set the activation for input layer units equal to that of the current input vector given, x_i.

Step 3: Calculate the net input to the output units:

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij} \quad (j = 1 \text{ to } m)$$

Step 4: Determine the activations of the output units over the calculated net input:

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > 0 \\ 0 & \text{if } y_{inj} = 0 \\ -1 & \text{if } y_{inj} < 0 \end{cases}$$

Thus, the output vector y obtained gives the pattern associated with the input vector x.

Note: Heteroassociative memory is not an iterative memory network. If the responses of the net are binary, then the activation function to be used is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} \geq 0 \\ 0 & \text{if } y_{inj} < 0 \end{cases}$$

4.10 SOLVED PROBLEMS

1. Train a heteroassociative memory network using Hebb rule to store input row vector $s=(s_1,s_2,s_3,s_4)$ to the output row vector $t=(t_1,t_2)$. The vector pairs are given in Table 1.

TABLE 1

Input targets	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	0	1	0	1	0
2nd	1	0	0	1	1	0
3rd	1	1	0	0	0	1
4th	0	0	1	1	0	1

Solution: The network for the given problem is as shown in Figure 1. The training algorithm based on Hebb rule is used to determine the weights.

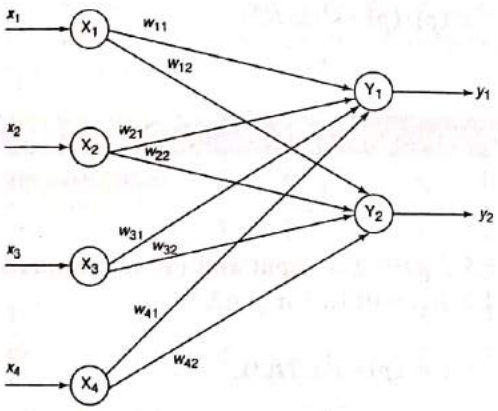


Figure 1 Neural net.

For 1st input vector:

Step 0: Initialize the weights, the initial weights are taken as zero.

Step 1: For first pair (1, 0, 1, 0):(1, 0)

Step 2: Set the activations of input units:

$$x_1=1,\quad x_2=0,\quad x_3=1,\quad x_4=0$$

Step 3: Set the activations of output unit:

$$y_1=1,\quad y_2=0$$

Step 4: Update the weights,

$$w_{ij}(\text{new})=w_{ij}(\text{old})+x_i y_j$$
$$w_{11}(\text{new})=w_{11}(\text{old})+x_1y_1=0+1\times1=1$$
$$w_{21}(\text{new})=w_{21}(\text{old})+x_2y_1=0+0\times1=0$$
$$w_{31}(\text{new})=w_{31}(\text{old})+x_3y_1=0+1\times1=1$$
$$w_{41}(\text{new})=w_{41}(\text{old})+x_4y_1=0+0\times1=0$$
$$w_{12}(\text{new})=w_{12}(\text{old})+x_1y_2=0+1\times0=0$$
$$w_{22}(\text{new})=w_{22}(\text{old})+x_2y_2=0+0\times0=0$$
$$w_{32}(\text{new})=w_{32}(\text{old})+x_3y_2=0+1\times0=0$$
$$w_{42}(\text{new})=w_{42}(\text{old})+x_4y_2=0+0\times0=0$$

For 2nd input vector:

The input–output vector pair is (1, 0, 0, 1):(1, 0)

$$x_1=1,\quad x_2=0,\quad x_3=0,\quad x_4=1,$$
$$y_1=1,\quad y_2=0$$

The final weights obtained for the input vector pair is used as initial weight here:

$$w_{11}(\text{new})=w_{11}(\text{old})+x_1y_1=1+1\times1=2$$
$$w_{41}(\text{new})=w_{41}(\text{old})+x_4y_1=0+1\times1=1$$

Since $x_2=x_3=y_2=0$, the other weights remains the same.

The final weights after second input vector is presented are

$$w_{11}=2, w_{21}=0, w_{31}=1, w_{41}=1$$
$$w_{12}=0, w_{22}=0, w_{32}=0, w_{42}=0$$

For 3rd input vector:

The input–output vector pair is (1, 1, 0, 0):(0, 1)

$$x_1=1,\ x_2=1, x_3=0,\ x_4=0,\ y_1=0,\ y_2=1$$

Training, using Hebb rule, evolves the final weights as follows:

Since $y_1=0$, the weights of y_1 are going to the same. Computing the weights of y_2 unit, we obtain

$$w_{12}(\text{new})=w_{12}(\text{old})+x_1y_2=0+1\times1=1$$

$$w_{12}(\text{new}) = w_{12}(\text{old}) + x_1 y_2 = 0 + 1 \times 1 = 1$$
$$w_{22}(\text{new}) = w_{22}(\text{old}) + x_2 y_2 = 0 + 0 \times 1 = 0$$
$$w_{42}(\text{new}) = w_{42}(\text{old}) + x_4 y_2 = 0 + 0 \times 1 = 0$$

The final weights after presenting third input vector are

$$w_{11} = 2,\ w_{21} = 0,\ w_{31} = 1,\ w_{41} = 1$$
$$w_{12} = 1,\ w_{22} = 1,\ w_{32} = 0,\ w_{42} = 0$$

For 4th input vector:

The input–output vector pair is (0, 0, 1, 1):(0, 1)

$$x_1 = 0,\ x_2 = 0,\ x_3 = 1,\ x_4 = 1,\ y_1 = 0,\ y_2 = 1$$

The weights are given by

$$w_{32}(\text{new}) = w_{32}(\text{old}) + x_3 y_2 = 0 + 1 \times 1 = 1$$
$$w_{42}(\text{new}) = w_{42}(\text{old}) + x_4 y_2 = 0 + 1 \times 1 = 1$$

Since, $x_1 = x_2 = y_1 = 0$, the other weights remains the same. The final weights after presenting the fourth input vector are

$$w_{11} = 2,\ w_{21} = 0,\ w_{31} = 1,\ w_{41} = 1$$
$$w_{12} = 1,\ w_{22} = 1,\ w_{32} = 1,\ w_{42} = 1$$

Thus, the weight matrix in matrix form is

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 0 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

2. Train the heteroassociative memory network using outer products rule to store input row vectors $s = (s_1, s_2, s_3, s_4)$ to the output row vectors $t = (t_1, t_2)$. Use the vector pairs as given in Table 2.

TABLE 2

Input and targets	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	0	1	0	1	0
2nd	1	0	0	1	1	0
3rd	1	1	0	0	0	1
4th	0	0	1	1	0	1

Solution: Use outer products rule to determine the weight matrix:

$$W = \sum_{p=1}^{P} s^{\mathrm{T}}(p)\, t(p)$$

For 1st pair: The input and output vectors are $s = (1\,0\,1\,0)$, $t = (1\,0)$. For $p = 1$,

$$s^{\mathrm{T}}(p)\, t(p) = s^{\mathrm{T}}(1)\, t(1)$$
$$= \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}_{4\times 1} [1 \quad 0]_{1\times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}_{4\times 2}$$

For 2nd pair: The input and output vectors are $s = (1\,0\,0\,1)$, $t = (1\,0)$. For $p = 2$,

$$s^{\mathrm{T}}(p)\, t(p) = s^{\mathrm{T}}(2)\, t(2)$$
$$= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}_{4\times 1} [1 \quad 0]_{1\times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}_{4\times 2}$$

For 3rd pair: The input and output vectors are $s = (1\,1\,0\,0)$, $t = (0\,1)$. For $p = 3$,

$$s^{\mathrm{T}}(p)\, t(p) = s^{\mathrm{T}}(3)\, t(3)$$
$$= \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}_{4\times 1} [0 \quad 1]_{1\times 2} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}_{4\times 2}$$

For 4th pair: The input and output vectors are $s = (0\,0\,1\,1)$, $t = (0\,1)$. For $p = 4$,

$$s^{\mathrm{T}}(p)t(p) = s^{\mathrm{T}}(4)t(4)$$
$$= \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}_{4\times 1} [0 \quad 1]_{1\times 2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}_{4\times 2}$$

The final weight matrix is the summation of all the individual weight matrices obtained for each pair.

$$W = \sum_{p=1}^{4} s^T(p)t(p)$$

$$= s^T(1)t(1) + s^T(2)t(2) + s^T(3)t(3) + s^T(4)t(4)$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

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

3. Train a heteroassociative memory network to store the input vectors $s = (s_1, s_2, s_3, s_4)$ to the output vectors $t = (t_1, t_2)$. The vector pairs are given in Table 3. Also test the performance of the network using its training input as testing input.

TABLE 3

Input and targets	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	0	0	0	0	1
2nd	1	1	0	0	0	1
3rd	0	0	0	1	1	0
4th	0	0	1	1	1	0

Solution: The network architecture for the given input–target vector pair is shown in Figure 2. Training the network means the determination of weights of the network. Here outer products rule is used to determine the weight.

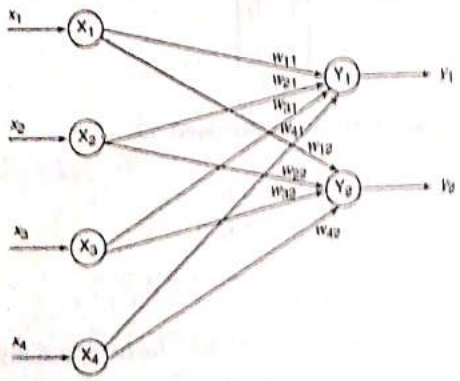


Figure 2 Network architecture.

The weight matrix W using outer products rule is given by

$$W = \sum_{p=1}^{4} s^T(p)t(p)$$

For $p = 1$ to 4,

$$W = \sum_{p=1}^{4} s^T(p)t(p)$$

$$= s^T(1)t(1) + s^T(2)t(2) + s^T(3)t(3) + s^T(4)t(4)$$

$$= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

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

This is the final weight of the matrix.

Testing the Network

Method 1

The testing algorithm for a heteroassociative memory network is used to test the performance of the net. The weight obtained from training algorithm is the initial weight in testing algorithm.

For 1st testing input

Step 0: Initialize the weights:

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 2 & 0 \end{bmatrix}$$

Step 1: Performs Steps 2–4 for each testing input–output vector.

Step 2: Set the activations, $x = [1 \quad 0 \quad 0 \quad 0]$.

Step 3: Compute the net input, $n = 4$, $m = 2$. For $i = 1$ to 4 and $j = 1$ to 2:

$$y_{in_j} = \sum_{i=1}^{4} x_i w_{ij}$$

$$y_{in_1} = \sum_{i=1}^{4} x_i w_{i1}$$

$$= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41}$$
$$= 1 \times 0 + 0 \times 0 + 0 \times 1 + 0 \times 2 = 0$$

$$y_{in_2} = \sum_{i=1}^{4} x_i w_{i2}$$

$$= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42}$$
$$= 1 \times 2 + 0 \times 1 + 0 \times 0 + 0 \times 0 = 2$$

Step 4: Applying activation over the net input to calculate the output.

$$y_1 = f(y_{in_1}) = f(0) = 0$$
$$y_2 = f(y_{in_2}) = f(2) = 1$$

The output is [0, 1] which is correct response for first input pattern.

For 2nd testing input

Set the activation $x = [1\ 1\ 0\ 0]$. Computing the net input, we obtain

$$y_{in1} = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41}$$
$$= 0 + 0 + 0 + 0 = 0$$
$$y_{in2} = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42}$$
$$= 2 + 1 + 0 + 0 = 3$$

Compute the output by applying activations over net input,

$$y_1 = f(y_{in1}) = f(0) = 0$$
$$y_2 = f(y_{in2}) = f(3) = 1$$

The output is [0, 1] which is correct response for second input pattern.

For 3rd testing input

Set the activation $x = [0\ 0\ 0\ 1]$. Computing net input, we obtain

$$y_{in1} = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41}$$
$$= 0 + 0 + 0 + 2 = 2$$
$$y_{in2} = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42}$$
$$= 0 + 0 + 0 + 0 = 0$$

Calculate output of the network,

$$y_1 = f(y_{in1}) = f(2) = 1$$
$$y_2 = f(y_{in2}) = f(0) = 0$$

The output is [1 0] which is correct response for third testing input pattern.

For 4th testing input

Set the activation $x = [0\ 0\ 1\ 1]$. Calculating the net input, we obtain

$$y_{in1} = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41}$$
$$= 0 + 0 + 1 + 2 = 3$$
$$y_{in2} = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42}$$
$$= 0 + 0 + 0 + 0 = 0$$

Calculate the output of the network,

$$y_1 = f(y_{in1}) = f(3) = 1$$
$$y_2 = f(y_{in2}) = f(0) = 0$$

The output is [1 0] which is correct response for fourth testing input pattern.

Method II

Since net input is the dot product of the input row vector with the column of weight matrix, hence a method using matrix multiplication can be used to test performance of network. The initial weights for the network are

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

The binary activations are used, i.e.,

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

For 1st testing input

Set the activation $x = [1\ 0\ 0\ 0]$. The net input is given by $y_{in} = xW$ (in vector form):

$$[y_{in1}\ y_{in2}] = [1\ \ 0\ \ 0\ \ 0]_{1\times 4}\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}_{4\times 2}$$

$$= [0+0+0+0 \quad 2+0+0+0]$$

$$= [0\ \ 2]$$

Applying activations over the net input, we get

$$[y_1\ y_2] = [0\ \ 1]$$

The correct response is obtained for first testing input pattern.

For 2nd testing input

Set the activation $x = [1\ 1\ 0\ 0]$. The net input is obtained by

$$[y_{in1}\ y_{in2}] = [1\ \ 1\ \ 0\ \ 0]\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$

$$= [0+0+0+0 \quad 2+1+0+0]$$

$$= [0\ \ 3]$$

Apply activations over the net input to obtain output, we get

$$[y_{in1}\ y_{in2}] = [0\ \ 1]$$

The correct response is obtained for second testing input.

For 3rd testing input

Set the activation $x = [0\ 0\ 0\ 1]$. The net input is obtained by

$$[y_{in1}\ y_{in2}] = [0\ \ 0\ \ 0\ \ 1]\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$

$$= [0+0+0+2 \quad 0+0+0+0]$$

$$= [2\ \ 0]$$

Applying activations to calculate the output, we get

$$[y_1\ \ y_2] = [1\ \ 0]$$

Thus, correct response is obtained for third testing input.

For 4th testing input

Set the activation $x = [0\ 0\ 1\ 1]$. The net input is calculated as

$$[y_{in1}\ y_{in2}] = [0\ \ 0\ \ 1\ \ 1]\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$

$$= [0+0+1+2 \quad 0+0+0+0]$$

$$= [3\ \ 0]$$

The output is obtained by applying activations over the net input:

$$[y_1\ \ y_2] = [1\ \ 0]$$

The correct response is obtained for fourth testing input. Thus, training and testing of a hetero associative network is done here.

4. For Problem 3, test a heteroassociative network with a similar test vector and unsimilar test vector.

Solution: The heteroassociative network has to be tested with similar and unsimilar test vector.

With similar test vector: From Problem 3, the second input vector is $x = [1\ 1\ 0\ 0]$ with target $y = [0\ 1]$. To test the network with a similar vector, making a change in one component of the input vector, we get

$$x = [0\ 1\ 0\ 0]$$

The weight matrix is

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

The net input is calculated for the similar vector,

$$[y_{in1}\ y_{in2}] = [0\ \ 1\ \ 0\ \ 0]\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$

$$= [0+0+0+0 \quad 0+1+0+0]$$

$$= [0\ \ 1]$$

The output is obtained by applying activations over the net input

$$[y_1 \quad y_2] = [0 \quad 1]$$

The correct response same as the target is found, hence the vector similar to the input vector is recognized by the network.

With unsimilar input vector: The second input vector is $x = [1\ 1\ 0\ 0]$ with target $y = [0 \quad 1]$. To test the network with unsimilar vectors by making a change in two components of the input vector, we get

$$x = [0 \quad 1 \quad 1 \quad 0]$$

The weight matrix is

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

The net input is calculated for unsimilar vector,

$$[y_{in1}\ y_{in2}] = [0 \quad 1 \quad 1 \quad 0] \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 2 & 0 \end{bmatrix}$$

$$= [0+0+1+0 \quad 0+1+0+0]$$

$$= [1 \quad 1]$$

The output is obtained by applying activations over the net input

$$[y_1 \quad y_2] = [1 \quad 1]$$

The correct response is not obtained when the vector unsimilar to the input network is presented to the network.

5. Train a heteroassociative network to store the input vectors $s = (s_1\ s_2\ s_3\ s_4)$ to the output vector $t = (t_1\ t_2)$. The training input-target output vector pairs are in binary form. Obtain the weight vector in bipolar form. The binary vector pairs are as given in Table 4.

TABLE 4

	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	0	0	0	0	1
2nd	1	1	0	0	0	1
3rd	0	0	0	1	1	0
4th	0	0	1	1	1	0

Solution: In this case, the hybrid representation of the network is adopted to find the weight matrix in bipolar form. The weight matrix can be formed using

$$w_{11} = (2\times1-1)(2\times0-1)+(2\times1-1)(2\times0-1)$$
$$+(2\times0-1)(2\times1-1)+(2\times0-1)(2\times1-1)$$

$$w_{12} = (2\times1-1)(2\times1-1)+(2\times1-1)(2\times1-1)$$
$$+(2\times0-1)(2\times0-1)+(2\times0-1)(2\times0-1)$$
$$= 1+1+1+1 = 4$$

$$w_{21} = -1\times-1+1\times-1+-1\times1+-1\times1$$
$$= 1-1-1-1 = -2$$

$$w_{22} = -1\times1+1\times1+-1\times-1+-1\times-1$$
$$= -1+1+1+1 = 2$$

$$w_{31} = -1\times-1+-1\times-1+-1\times1+1\times1$$
$$= 1+1-1+1 = 2$$

$$w_{32} = -1\times1+-1\times1+-1\times-1+1\times-1$$
$$= -1-1+1-1 = -2$$

$$w_{41} = -1\times-1+-1\times-1+1\times1+1\times1$$
$$= 1+1+1+1 = 4$$

$$w_{42} = -1\times1+-1\times1+1\times-1+1\times-1$$
$$= -1-1-1-1 = -4$$

The weight matrix W is given by

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} = \begin{bmatrix} -4 & 4 \\ -2 & 2 \\ 2 & -2 \\ 4 & -4 \end{bmatrix}$$

6. Train a heteroassociative network to store the given bipolar input vectors $s = (s_1\ s_2\ s_3\ s_4)$ to the output vector $t = (t_1\ t_2)$. The bipolar vector pairs are as given in Table 5.

TABLE 5

	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	−1	−1	−1	−1	1
2nd	1	1	−1	−1	−1	1
3rd	−1	−1	−1	1	1	−1
4th	−1	−1	1	1	1	−1

Solution: To store a bipolar vector pair, the weight matrix is

$$w_{ij} = \sum_{p=1}^{P} s_i(p) t_j(p)$$

If the outer products rule is used, then

$$W = \sum_{p} s^T(p) t(p)$$

For 1st pair

$$s = [1 \quad -1 \quad -1 \quad -1], \quad t = [-1 \quad 1]$$

$$s^T(1)t(1) = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} [-1 \quad 1] = \begin{bmatrix} -1 & 1 \\ 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

For 2nd pair

$$s = [1 \quad 1 \quad -1 \quad -1], \quad t = [-1 \quad 1]$$

$$s^T(2)t(2) = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} [-1 \quad 1] = \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

For 3rd pair

$$s = [-1 \quad -1 \quad -1 \quad 1], \quad t = [1 \quad -1]$$

$$s^T(3)t(3) = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix} [1 \quad -1] = \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix}$$

For 4th pair

$$s = [-1 \quad -1 \quad 1 \quad 1], \quad t = [1 \quad -1]$$

$$s^T(4)t(4) = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} [1 \quad -1] = \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

The final weight matrix is

$$W = \sum_{p=1}^{4} s^T(p)t(p) = \begin{bmatrix} -1 & 1 \\ 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$+ \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$= \begin{bmatrix} -4 & 4 \\ -2 & 2 \\ 2 & -2 \\ 4 & -4 \end{bmatrix}$$

7. For Problem 6, test the performance of the network with missing and mistaken data in the test vector.

Solution: *With missing data*

Let the test vector be $x = [0 \quad 1 \quad 0 \quad -1]$ with changes made in two components of second input vector $[1 \quad 1 \quad -1 \quad -1]$. Computing the net input, we get

$$[y_{in1} \quad y_{in2}] = [0 \quad 1 \quad 0 \quad -1] \begin{bmatrix} -4 & 4 \\ -2 & 2 \\ 2 & -2 \\ 4 & -4 \end{bmatrix}$$

$$= [0 - 2 + 0 - 4 \quad 0 + 2 + 0 + 4]$$

$$= [-6 \quad 6]$$

Applying activations to compute the output, we get

$$[y_1 \quad y_2] = [-1 \quad 1]$$