NEURAL NETWORKS

1.0.0 Introduction

The recent rise of interest in neural networks has its roots in the recognition that the brain performs computations in a different manner than do conventional digital computers. Computers are extremely fast and precise at executing sequences of instructions that have been formulated for them. A human information processing system is composed of neurons switching at speeds about a million times slower than computer gates. Yet, humans are more efficient than computers at computationally complex tasks such as speech understanding. Moreover, not only humans, but also even animals, can process visual information better than the fastest computers.

Artificial neural systems, or neural networks (NN), are physical cellular systems, which can acquire, store, and utilize experiential knowledge. The knowledge is in the form of stable states or mappings embedded in networks that can be recalled in response to the presentation cues. Neural network processing typically involves dealing with large-scale problems in terms of dimensionality, amount of data handled, and the volume of simulation or neural hardware processing. This large-scale approach is both essential and typical for real-life applications. By keeping view of all these, the research community has made an effort in designing and implementing the various neural network models for different applications. Now let us formally define the basic idea of neural network:

Definition: *A neural network is a computing system made up of a number of simple, highly interconnected nodes or processing elements, which process information by its dynamic state response to external inputs.*

1.1.0 Humans and Computers

Human beings are more intelligent than computers. Computers could only do logical things well. But in case of solving cross word puzzles, vision problem, controlling an arm to pick it up or something similar, that requires exceptionally complex techniques. Like these problems, human beings do better than computers.

Computers are designed to carry out one instruction after another, extremely rapid, whereas our brains work with many slower units. Whereas a computer can typically carry out a few million operations every second, the units in the brain respond about ten per second. However, they work on many different things at once, which computer can’t do.

The computer is a high-speed, serial machine and is used as such, compared to the slow, highly parallel nature of the brain. Counting is an essentially serial activity, as is adding, with the thing done one after another, and so the computer can beat the brain any time. For vision, or speech recognition, the problem is a highly parallel one, with many different and conflicting inputs, triggering many different and conflicting ideas and memories, and it is only the combining of all these different factors that allow us to perform such feats, but then, our brains are able to operate in parallel easily and so we leave the computers far behind.

The conclusion that we can reach from all of this is that the problems that we are trying to solve are immensely parallel ones.

1.2.0 History of artificial neural networks

- The field of neural networks is not new. The first formal definition of a synthetic neuron model based on the highly simplified considerations of the biological model proposed by McCulloch and Pitts in 1943. The McCulloch-Pitts (MP) neuron model resembles what is known as a binary logic device.
- The next major development, after the MP neuron model was proposed, occurred in 1949, when D.O. Hebb proposed a learning mechanism for the brain that become the starting point for artificial neural networks (ANN) learning (training) algorithms. He postulated that as the brain learns, it changes its connectivity patterns.
- The idea of learning mechanism was first incorporated in ANN by E. Rosenblatt 1958.
- By introducing the least mean squares (LMS) learning algorithm, Widrow and Hoff developed in 1960 a model of a neuron that learned quickly and accurately. This model was called ADALINE for ADAptive LInear NEuron. The applications of ADALINE and its extension to MADALINE (for Many ADALINES) include pattern recognition, weather forecasting, and adaptive controls. The monograph on learning machines by Nils Nilsson (1965) summarized the developments of that time.
- In 1969, research in the field of ANN suffered a serious setback. Minsky and Papert published a book on perceptrons in which they proved that single layer neural networks have limitations in their abilities to process data, and are capable of any mapping that is linearly separable. They pointed out, carefully applying mathematical techniques, that are logical Exclusive-OR (XOR) function could not be realized by perceptrons.
- Further, Minsky and Papert argued that research into multi-layer neural networks would be unproductive. Due to this pessimistic view of Minsky and Papert, the field of ANN entered into an almost total eclipse for nearly two decades. Fortunately, Minsky and Papert's judgment has been disapproved; multi-layer perceptron networks can solve all nonlinear separable problems.
- Nevertheless, a few dedicated researchers such as Kohonen, Grossberg, Anderson and Hopfield continued their efforts.
- The study of learning in networks of threshold elements and of the mathematical theory of neural networks was pursued by Sun - Ichi – Amari (1972, 1977). Also Kunihiko Fukushima developed a class of neural network architectures known as neocognitrons in 1980.
- There have been many impressive demonstrations of ANN capabilities: a network has been trained to convert text to phonetic representations, which were then converted to speech by other means (Sejnowsky and Rosenberg 1987); other network can recognize handwritten characters (Burr 1987); and a neural network based image-compression system has been devised (Cottrell, Munro, and Zipser 1987). These all use the backpropagation network, perhaps the most successful of the current algorithms. Backpropagation, invented independently in three separate research efforts (Werbos 1974, Parker 1982, and Rumelhart, Hinton and Williams 1986) provides a systematic means for training multi-layer networks, thereby overcoming limitations presented by Minsky.

1.3.0 Characteristics of ANN

Artificial neural networks are biologically inspired; that is, they are composed of elements that perform in a manner that is analogous to the most elementary functions of the biological neuron. The important characteristics of artificial neural networks are learning from experience; generalize from previous examples to new ones, and abstract essential characteristics from inputs containing irrelevant data.

1.3.1 Learning

ANN can modify their behavior in response to their environment. Shown a set of inputs (perhaps with desired outputs), they self-adjust to produce consistent responses. A wide variety of training algorithms has been discussed in later units.

1.3.2 Generalization

Once trained, a network's response can be to a degree, insensitive to minor variations in its input. This ability to see through noise and distortion to the pattern that lies within is vital to pattern recognition in a real-world environment. It is important to note that the ANN generalizes automatically as a result of its structure, not by using human intelligence embedded in the form of adhoc computer programs.

1.3.3 Abstraction

Some ANN's are capable of abstracting the essence of a set of inputs.

1.3.4 Applicability

ANN's are not a panacea. They are clearly unsuited to such tasks as calculating the payroll. They are preferred for a large class of pattern-recognition tasks that conventional computers do poorly, if at all.

1.4.0 Applications

Neural networks are preferred when the task is related to large-amount data processing. The following are the potential applications of neural networks:

- Classification
- Prediction
- Data Association
- Data Conceptualization
- Data Filtering
- Optimization

In addition to the above fields, neural networks can apply to the fields of Medicine, Commercial and Engineering, etc.

Fundamentals of Artificial Neural Networks

Objective: An objective of the unit-2 is, the reader should be able to define biological neuron, artificial neuron, comparison between biological and artificial neurons and McCulloch-Pitts model.

2.0.0 Introduction

Artificial neural networks have been developed in a wide variety of forms. To understand them it is require to get familiarize with basics of neural networks and terminology. In this unit we define biological and artificial neurons and also we discuss the early model of artificial neuron.

2.1.0 The Biological Prototype

ANN's are biologically inspired; that is viewing at the organization of the brain considering network configurations and algorithms. The human nerve system, built of

cells called neurons is of staggering complexity. It contains approximately ten thousand million (10^{11}) basic neurons. Each of these neurons is connected to about ten thousand (10^4) others. The connection of each neuron with other neurons forms a densely network called a neural network. These massive interconnections provide an exceptionally large computing power and memory. The neuron accepts many inputs, which are all added up in some fashion. If enough active inputs are received at once, then the neuron will be activated at once, then the neuron will be activated and "fire"; if not, then the neuron will remain in its inactive, quit state. The schematic diagram of biological neuron is shown in Fig.2.1. From a systems theory, the neuron considered to be as a multiple-input-single-output (MISO) system as shown in Fig.2.2.

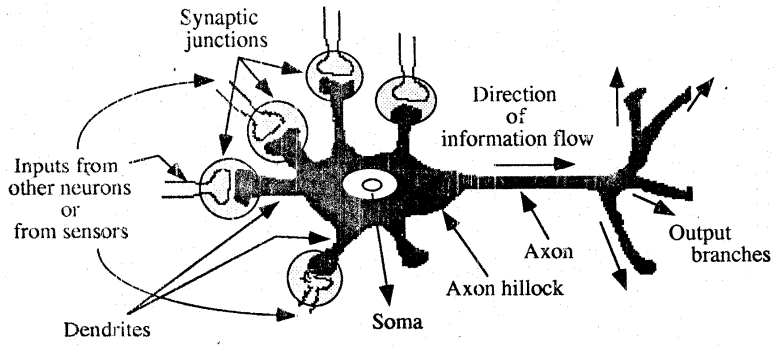


Fig. 2.1. A Schematic view of the biological neuron

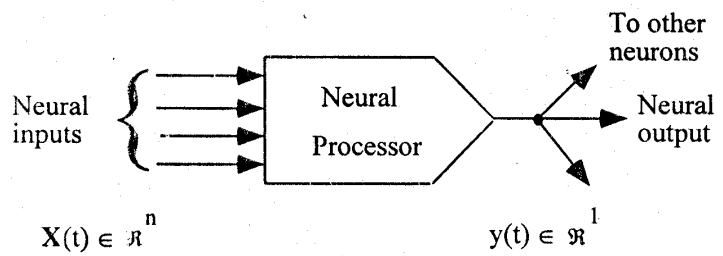


Fig.2.2 Model representation of a biological neuron with multiple inputs

The soma is the body of the neuron. Attached to the soma there are long irregularly shaped filaments, called dendrites. These nerve processes are often less than a micron in diameter, and have complex branching shapes. The dendrites act as the connections through which all the inputs to the neuron arrive. These cells are able to perform more complex functions than simple addition on the inputs they receive, but considering simple summation is a reasonable approximation.

Another type of nerve process attached to the soma is called an axon. This is electrically active, unlike the dendrite, and serves as the output channel of the neuron. Axons always appear on output cell, but are often absent from interconnections, which have both inputs and outputs on dendrites. The axon is a non-linear threshold device, producing a voltage pulse, called an action potential, that last about 1 millisecond (10^{-3} sec) when the resting potential within the soma rises above a certain critical threshold.

The axon terminates in a specialized contact called a synapse that couples the axon with the dendrite of another cell. There is no direct linkage across the junction; rather, it is temporally chemical one. The synapse releases chemicals called neurotransmitters when its potential is raised sufficiently by the action potential. It may take the arrival of more than one action potential before the synapse is triggered. The neurotransmitters that are released by the synapse diffuse across the gap, any chemically activate gates on the dendrites, which, when open, allow charged ions to flow. It is this flow of ions that alters the dendrite potential, and provides a voltage pulse on the dendrite, which is then conducted along into the next neuron body. Each dendrite may have many synapses acting on it, allowing massive interconnectivity to be achieved.

2.2.0 **Artificial Neuron**

The artificial neuron is developed to mimic the first-order characteristics of the biological neuron. In similar to the biological neuron, the artificial neuron receives many inputs representing the output of other neurons. Each input is multiplied by a corresponding weight, analogous to the synaptic strength. All of these weighted inputs are then summed and passed through an activation function to determine the neuron input. This artificial neuron model is shown in Fig.2.3.

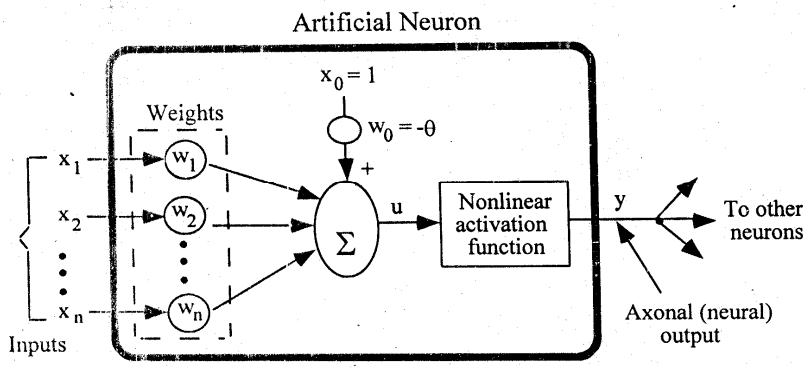


Fig. 2.3 An artificial neuron

The mathematical model of the artificial neuron may written as

$$u(t) = w_1x_1 + w_2x_2 + w_3x_3 + \ldots\ldots\ldots + w_nx_n$$

$$= \sum_{i=1}^{n} w_i x_i - \theta = \sum_{i=0}^{n} w_i x_i \qquad (2.1)$$

Assuming $w_0 = -\theta$ and $x_0 = 1$

$$y(t) = f[u(t)] \qquad (2.2)$$

where f[.] is a nonlinear function called as the activation function, the input-output function or the transfer function. In equation (2.1) and Fig.2.3, $[x_0, x_1, \ldots, x_n]$ represent the inputs, $[w_0, w_1, \ldots, w_n]$ represents the corresponding synaptic weights. In vector form, we can represent the neural inputs and the synaptic weights as

$X = [x_0, x_1, \ldots, x_n]^T$, and $W = [w_0, w_1, \ldots, w_n]$

Equations (2.1) and (2.2) can be represented in vector form as:

$$U = WX \qquad (2.3)$$

$$Y = f[U] \qquad (2.4)$$

The activation function f[.] is chosen as a nonlinear function to emulate the nonlinear behavior of conduction current mechanism in biological neuron. The behavior of the artificial neuron depends both on the weights and the activation function.

Sigmoidal functions are the commonly used activation functions in multi-layer static neural networks. Other types of activation functions are discussed in later units.

2.0.0 McCulloch-Pitts Model

The McCulloch-Pitts model of the neuron is shown in Fig. 2.4(a). The inputs x_i, for i= 1,2, . . .,n are 0 or 1, depending on the absence or presence of the input impulse at instant k. The neuron's output signal is denoted as **o.** The firing rule for this model is defined as follows

$$\mathbf{o}^{k+1} = \begin{cases} 1 & if \quad \sum_{i=1}^{n} w_i x_i^k \geq T \\ 0 & if \quad \sum_{i=1}^{n} w_i x_i^k < T \end{cases}$$

where super script k = 0,1,2,. . . . , denotes the discrete – time instant, and w_i is the multiplicative weight connecting the i'th input with the neuron's membrane. Note that w_i = +1 for excitatory synapses, w_i = -1 for inhibitory synapses for this model, and T is the neuron's threshold value, which needs to be exceeded by the weighted sum of the signals for the neuron to fire.

This model can perform the basic operations NOT, OR and AND, provided its weights and thresholds are approximately selected. Any multivariable combinational function can be implemented using either the NOT and OR, or alternatively the NOT and AND, Boolean operations. Examples of three-input NOR and NAND gates using the McCulloch-Pitts neuron model are shown in (Fig.2.4 (b) and Fig 2.4(c)).

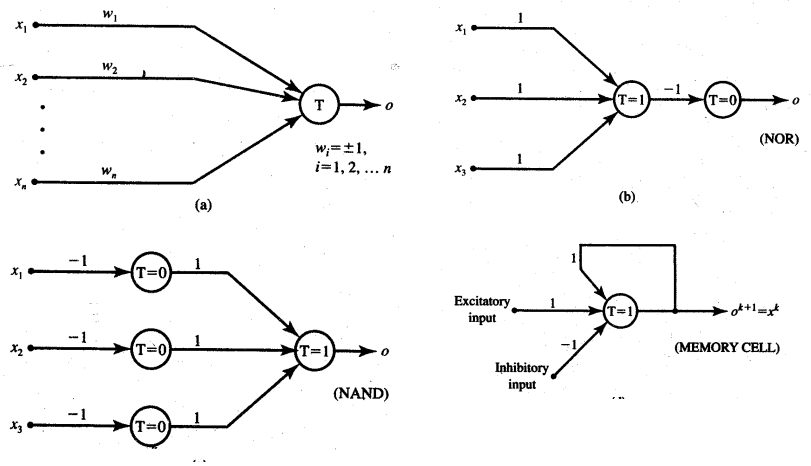


Fig.2.4 McCulloch-Pitts

2.1.0 **Keyword definitions**

Action potential: The pulse of electrical potential generated across the membrane of a neuron (or an axon) following the application of a stimulus greater than threshold value.

Axon: The output fiber of a neuron, which carries the information in the form of action potentials to other neurons in the network.

Dendrite: The input line of the neuron that carries a temporal summation of action potentials to the soma.

Excitatory neuron: A neuron that transmits an action potential that has excitatory (positive) influence on the recipient nerve cells.
Inhibitory neuron: A neuron that transmits an action potential that has inhibitory (negative) influence on the recipient nerve cells.
Lateral inhibition: The local spatial interaction where the neural activity generated by one neuron is suppressed by the activity of its neighbors.
Latency: The time between the application of the stimulus and the peak of the resulting action potential output.
Refractory period: The minimum time required for the axon to generate two consecutive action potentials.
Neural state: A neuron is active if it's firing a sequence of action potentials.
Neuron: The basic nerve cell for processing biological information.
Soma: The body of a neuron, which provides aggregation, thresholding and nonlinear activation to dendrite inputs.
Synapse: The junction point between the axon (of a pre-synaptic neuron) and the dendrite (of a post-synaptic neuron). This acts as a memory (storage) to the past-accumulated experience (knowledge).

Activation Functions

Objective: An objective of the unit-3 is, the reader should be able to define activation function, types and their selection.

3.0.0 **Introduction**

In order to understand this unit, the reader is suggested to have glance at unit 2. The functioning of artificial neuron mainly depends upon the selection of its activation function.

3.1.0 **Operations of Artificial Neuron**

The schematic diagram of artificial neuron is shown in Fig.3.1. The artificial neuron mainly performs two operations, one is the summing of weighted net input and the second is passing the net input through an activation function. The activation function also called nonlinear function and some time transfer function of artificial neuron.

The net input of j^{th} neuron may be written as

$$NET_j = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n - \theta_j \quad (3.1)$$

where θ_j is the threshold of j^{th} neuron,

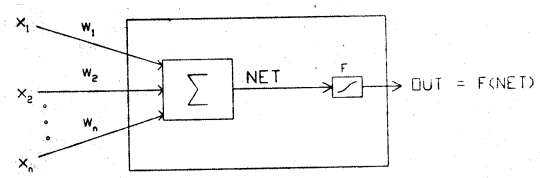


Fig. 3.1 Artificial neuron.

$X = [x_1\ x_2\ \ldots\ x_n]$ in the input vector and $W = [w_1\ w_2\ \ldots\ w_n]$ is the synaptic weight vector. The NET_j signal is processed by an activation function F to produce the neuron's output signal.

$$OUT_j = F(NET_j) \quad (3.2)$$

What functional form for F(.) should be selected? Can it be – square root, log, e^x, x^3 and so on. Mathematicians and computer scientists, however, have found that, the sigmoid (S-shaped) function is more useful. In addition to this sigmoid function, there are number of other function are using in artificial neural networks. They are discussed in the next section.

3.2.0 **Types activation functions**

The behavior of the artificial neuron depends both on the synaptic weights and the activation function. Sigmoid functions are the commonly used activation functions in multi-layered feed forward neural networks. Neurons with sigmoid functions bear a greater resemblance to the biological neurons than with other activation functions. The other feature of sigmoid function is that it is differentiable, and gives a continuous values output. Some of the popular activation functions are described below along with their other characteristics.

1. **Sigmoid function (Unipolar sigmoid).** The characteristics of this function is shown in Fig.3.2 and its mathematical description is

$$y(x) = f(x) = \frac{1}{1+e^{-x}} \qquad (3.1)$$

and its range of signal is 0<y<1.

The derivative of the above function is written as

$$y'(x) = f'(x) = f(x)\,(1-f(x)) \qquad (3.2)$$

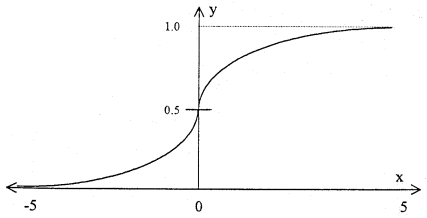


Fig.3.2 A sigmoid (S-shaped) function

Moreover, sigmoid functions are continuous and monatonic, and remain finite even as x approaches to ±∞. Because they are monatonic, they also provide for more efficient network training.

Hyperbolic tangent (bipolar sigmoid) function. The characteristics of this function is shown in Fig.3.3 and its mathematical description is y(x)=f(x)=

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \qquad (3.3)$$

range of signal is -1<y<1 and its derivative can be obtained as

$$y'(x) = f'(x) = 1-[f(x)]^2 \qquad (3.4)$$

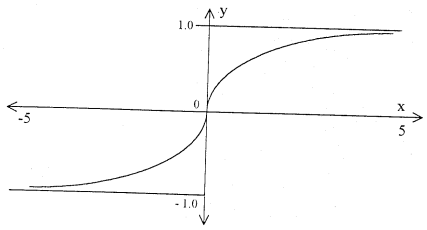


Fig.3.3 A hyperbolic tangent (bipolar sigmoid) function

2. **Radial basis function**: The Gaussian function is the most commonly used " radially symmetric" function, the characteristics of this function is shown in Fig.3.4 and its mathematical description is

$$y(x) = f(x) = \exp\left(\frac{-x^2}{2}\right) \qquad (3.5)$$

Range of signal is 0<y<1 and its derivative can be obtained as

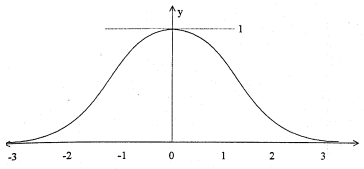


Fig.3.4. A Gaussian function

$$y'(x) = f'(x) = -x\ \exp\left(\frac{-x^2}{2}\right) \quad (3.6)$$

The function has maximum response, f(x) =1, when the input is x=0, and the response decreases to f(x)=0 as the input approaches x=±∞.

3. **Hard Limiter**: The hard limiter function is the mostly used in classification of patterns, the characteristics of this function is shown in Fig.3.5 and its mathematical description is

$$f(u(t)) = \textbf{sign}\ (u(t)) = \begin{cases} +1, & u(t) > 0 \\ -1 & u(t) < 0 \end{cases} \quad (3.7)$$

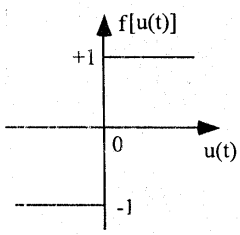


Fig. 3.5 Hard Limiter

This function is not differentiable. Therefore it cannot be used for continuation type of applications.

4. **Piecewise linear**: The piecewise linear function characteristics is shown in Fig.3.6 and its mathematical description is

$$f(u(t)) = \begin{cases} +1 & \text{if } gu > 1 \\ gu & \text{if } |gu| < 1 \\ -1 & \text{if } gu < -1 \end{cases} \quad (3.8)$$

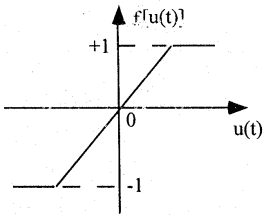


Fig. 3.6 Piecewise linear

5. **Unipolar multimodal**: The unipolar multimodal function characteristics is shown in Fig.3.7 and its mathematical description is

$$f(u(t)) = \frac{1}{2}\left[1 + \frac{1}{M}\sum_{m=1}^{M} \tanh\left(g^m\left(u(t) - w_0^m\right)\right)\right] \quad (3.9)$$

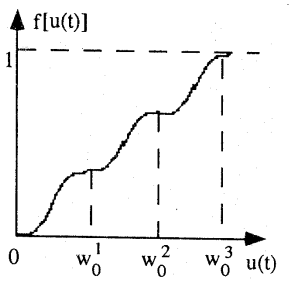


Fig. 3.7 Unipolar multimodal

6. **Linear**: The Linear function characteristics is shown in Fig.3.8 and its mathematical description is

$$f(u(t)) = gu(t) \qquad (3.10)$$

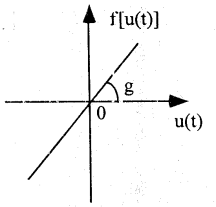


Fig. 3.8 Linear function

It is differentiable and is mostly used for output nodes of the networks.

3.3.0 **Selection of activation function**

The selection of an activation function is depends upon the application to which the neural network used and also the level (in which layer) neuron. The activation functions that are mainly used are the sigmoid (unipolar sigmoidal), the hyperbolic tangent (bipolar sigmoid), radial basis function, hard limiter and linear functions. The sigmoid and hyperbolic tangent functions perform well for the prediction and the process-forecasting types of problems. However, they do not perform as well for classification networks. Instead, the radial basis function proves more effective for those networks, and highly recommended function for any problems involving fault diagnosis and feature categorization. The hard limiter suits well for classification problems. The linear function may be used at output layer in feed forward networks.

Classification of Artificial Neural Networks

Objective: An objective of the unit-4 is, the reader should be able to classify the artificial neural networks, to define their suitable applications, and types are connections, to describe single layer networks and multi-layer networks.

4.0.0 Introduction

The development of artificial neuron based on the understanding of the biological neural structure and learning mechanisms for required applications. This can be summarized as (a) development of neural models based on the understanding of biological neurons, (b) models of synaptic connections and structures, such as network topology and (c) the learning rules. Researchers are explored different neural network architectures and used for various applications. Therefore the classification of artificial neural networks (ANN) can be done based on structures and based on type of data.

A single neuron can perform a simple pattern classification, but the power of neural computation comes from neuron connecting networks. The basic definition of artificial neural networks as physical cellular networks that are able to acquire, store and utilize experimental knowledge has been related to the network's capabilities and performance. The simplest network is a group of neurons are arranged in a layer. This configuration is known as single layer neural networks. There are two types of single layer networks namely, feed-forward and feedback networks. The single linear neural (that is activation function is linear) network will have very limited capabilities in solving nonlinear problems, such as classification etc., because their decision boundaries are linear. This can be made little more complex by selecting nonlinear neuron (that is activation function is nonlinear) in single layer neural network. The nonlinear classifiers will have complex shaped decision boundaries, which can solve complex problems. Even nonlinear neuron single layer networks will have limitations in classifying more close

nonlinear classifications and fine control problems. In recent studies shows that the nonlinear neural networks in multi-layer structures can simulate more complicated systems, achieve smooth control, complex classifications and have capabilities beyond those of single layer networks. In this unit we discuss first classifications, single layer neural networks and multi-layer neural networks. The structure of a neural network refers to how its neurons are interconnected.

4.1.0 Classification of ANN

The artificial neural networks broadly categorized into static and dynamic networks, and single –layer and multi – layer networks. The detail classification of ANN is shown in Fig. 4.1.

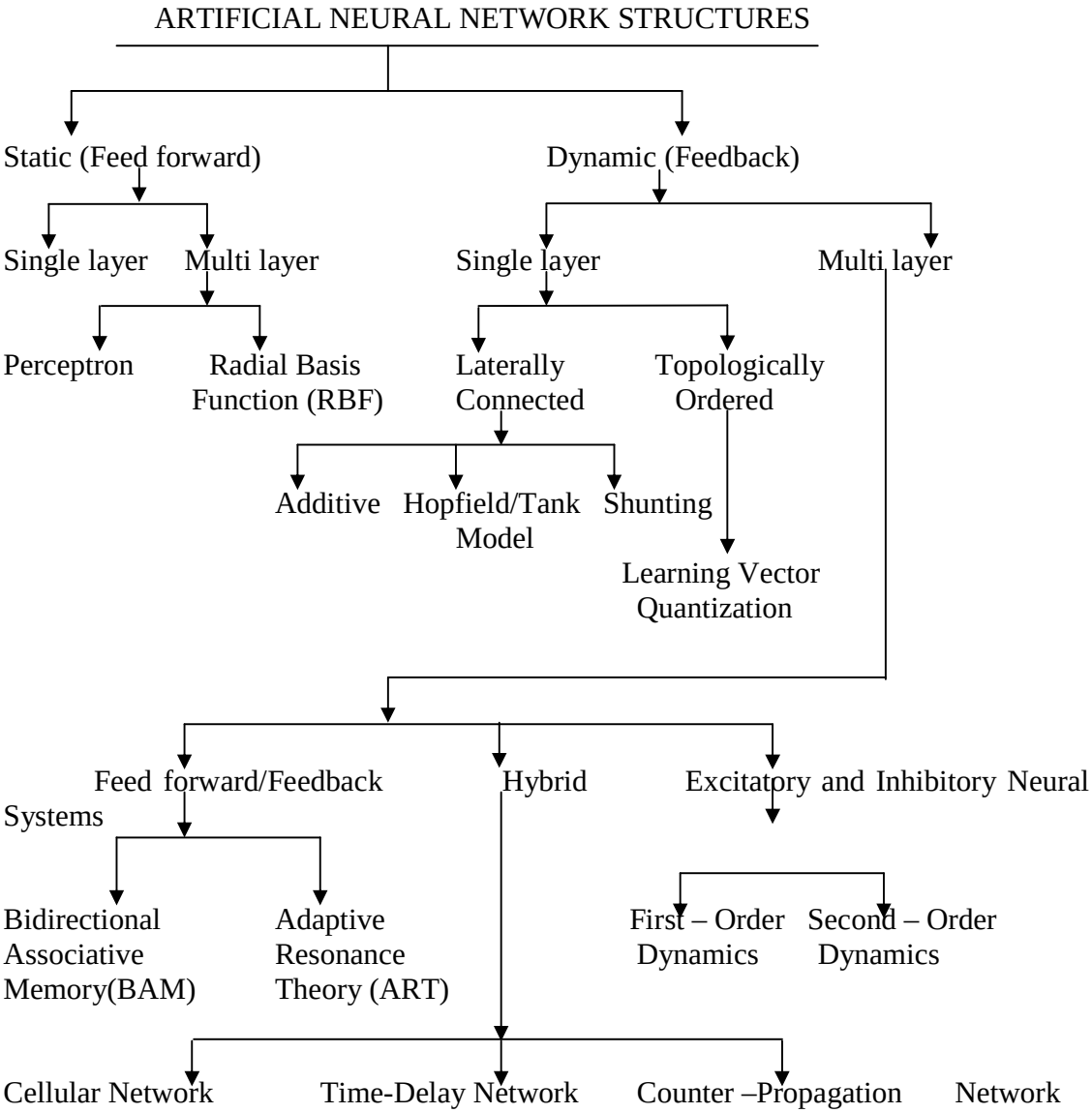


Fig 4.1. The taxonomy of neural structures

4.2.0 Applications

Having different types artificial neural networks, these networks can be used to broad classes of applications, such as (i) Pattern Recognition and Classification, (ii) Image Processing and Vision, (iii) System Identification and Control, and (iv) Signal Processing. The details of suitability of networks as follows:

(i) **Pattern Recognition and Classification**: Almost all networks can be used to solve these types of problems.

(ii) **Image Processing and Vision**: The following networks are used for the applications in this area: Static single layer networks, Dynamic single layer networks, BAM, ART, Counter – propagation networks, First – Order dynamic networks.

(iii) **System Identification and Control**: The following networks are used for the applications in this area: Static multi layer networks, Dynamic multi layer networks of types time-delay and Second-Order dynamic networks.

(iv) **Signal Processing:** The following networks are used for the applications in this area: Static multi layer networks of type RBF, Dynamic multi layer networks of types Cellular and Second-Order dynamic networks.

4.3.0 Single Layer Artificial Neural Networks

The simplest network is a group of neuron arranged in a layer. This configuration is known as single layer neural networks. There are two types of single layer networks namely, feed-forward and feedback networks.

4.3.1 **Feed forward single layer neural network**

Consider m numbers of neurons are arranged in a layer structure and each neuron receiving n inputs as shown in Fig.4.2.

Output and input vectors are respectively

$$O = [o_1 \; o_2 \; \ldots \; o_m]^T \qquad (4.1)$$

$$X = [x_1 \; x_2 \; \ldots \; x_n]^T$$

Weight w_{ji} connects the j^{th} neuron with the i^{th} input. Then the activation value for j^{th} neuron as

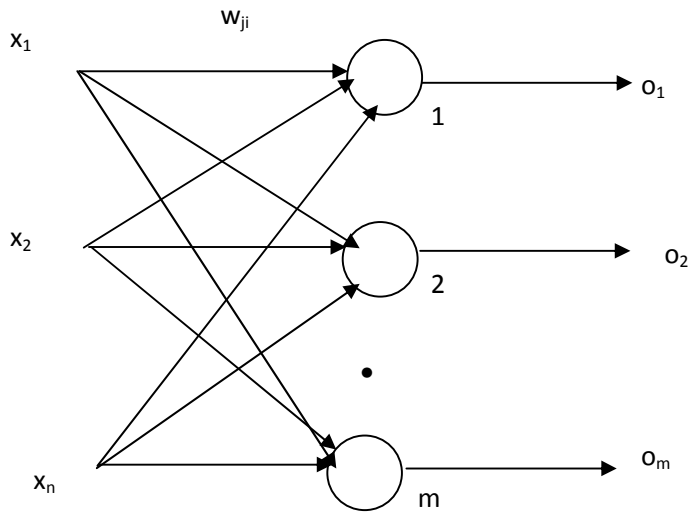
$$net_j = \sum_{i=1}^{n} w_{ji} x_i \qquad \text{for } j = 1,2, \ldots m \qquad (4.2)$$

The following nonlinear transformation involving the activation function $f(net_j)$, for j=1,2,. . .m, completes the processing of X. The transformation will be done by each of the m neurons in the network.

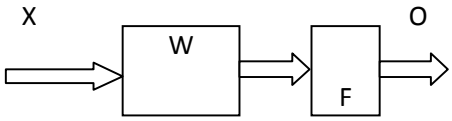
$$o_j = f(W_j^t X), \qquad \text{for } j = 1, 2, \ldots m \qquad (4.3)$$

where weight vector w_j contains weights leading toward the j^{th} output node and is defined as follows

$$W_j = [w_{j1} \; w_{j2} \; \ldots \; w_{jn}] \qquad (4.4)$$



(a) Single layer neural network



(b) Block diagram of single layer network

Fig. 4.2 Feed forward single layer neural network

Introducing the nonlinear matrix operator F, the mapping of input space X to output space O implemented by the network can be written as

$$O = F(W X) \tag{4.5a}$$

Where W is the weight matrix and also known as connection matrix and is represented as

$$W = \begin{bmatrix} w_{11} & w_{12} & . & . & w_{1n} \\ w_{21} & w_{22} & . & . & w_{2n} \\ . & . & . & . & . \\ . & . & . & . & . \\ w_{m1} & w_{m2} & . & . & w_{mn} \end{bmatrix} \tag{4.5b}$$

The weight matrix will be initialized and it should be finalized through appropriate training method.

$$F(.) = \begin{bmatrix} f(.) & 0 & . & . & 0 \\ 0 & f(.) & . & . & 0 \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & 0 & . & . & f(.) \end{bmatrix} \qquad (4.5c)$$

The nonlinear activation function *f*(.) on the diagonal of the matrix operator F(.) operates component-wise on the activation values net of each neuron. Each activation value is, in turn, a scalar product of an input with the respective weight vector, X is called input vector and O is called output vector. The mapping of an input to an output is shown as in (4.5) is of the feed-forward and instantaneous type, since it involves no delay between the input and the output . Therefore the relation (4.5a) may be written in terms of time t as

$$O(t) = F(W X(t)) \qquad (4.6)$$

This type of networks can be connected in cascade to create a multiplayer network. Though there is no feedback in the feedforward network while mapping from input X(t) to output O(t), the output values are compared with the "teachers" information, which provides the desired output values. The error signal is used for adapting the network weights. The details about will be discussed in later units.

Example: To illustrate the computation of output O(t), of the single layer feed forward network consider an input vector X(t) and a network weight matrix W (say initialized weights), given below. Consider the neurons uses the hard limiter as its activation function.

$$X = [-1 \;\; 1 \;\; -1]^T \qquad W = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & -1 & -3 \end{bmatrix}$$

The out vector may be obtained from the (4.5) as

$$O = F(W X) = [\, \text{sgn}(-1-1) \;\; \text{sgn}(+1+2) \;\; \text{sgn}(1) \;\; \text{sgn}(-1+3) \,]$$

$$= [\, -1 \;\; 1 \;\; 1 \;\; 1]$$

The output vector of the above single layer feedforward network is = [-1 1 1 1].

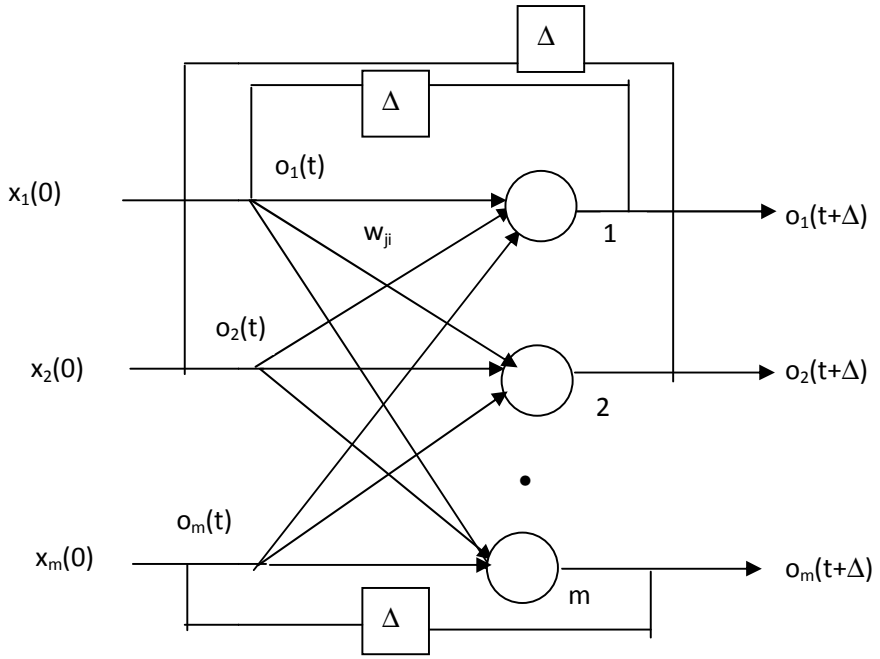
4.3.2 Feedback single layer neural network

A feedback network can be formed from feed forward network by connecting the neurons outputs to their inputs, as shown in Fig.4.3. The idea of closing the feedback loop is to enable control of output o_i through outputs o_j , for j=1,2, . . . ,m. Such control is meaningful if the present output, say o(t), controls the output at the following instant, o(t+Δ). The time Δ elapsed between t and t+Δ is introduced by the delay elements in the feedback loop as shown in Fig.4.3. Here the time delay Δ is similar meaning as that of the

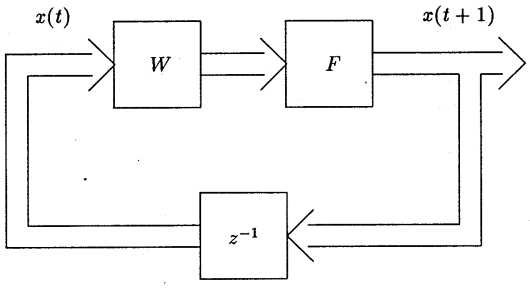
refractory period of an elementary biological neuron model. The mapping of O(t) into O(t+Δ) can now be written as

$$O(t+\Delta) = F(W\,O(t)) \qquad (4.7)$$

Note that the input X(t) is only needed to initialize this network so that O(0) = X(0). The input is then removed and the system remains autonomous for t>0.



(a) Feedback Single layer neural network



(b) Block diagram of Feed back single layer Network

Fig. 4.3 Feedback single layer neural network

There are two main categories of single-layer feedback networks and they are *discrete – time* network and *continuous-time* network.

Discrete – time network: If we consider time as a discrete variable and decided to know the network performance at discrete time instants Δ, 2Δ, 3Δ , 4Δ, . . . , the system is called discrete-time system. For notational convenience, the time step in discrete-time networks is equated to unity, and the time instances are indexed by positive integers, i.e., Δ=1. The choice of indices as natural numbers is convenient, since we initialize the study of the

system at t = 0 and are interested in its response thereafter. The relation (4.7) may be written as

$$O^{k+1} = F(W O^{k}) \ , \quad \text{for } k = 1,2,3, \ldots \qquad (4.8a)$$

Where k is the instant number. The network in Fig.4.3 is called recurrent since its response at the (k+1)th instant depends on the entire history of the network starting at k=0. From the relation (4.8a) a series of nested solution as follows:

$$O^{1} = F[W X^{0}]$$

$$O^{2} = F[W F[W X^{0}]]$$

$$O^{k+1} = F[W F[\ldots F[W X^{0}] \ldots]] \ldots\ldots\ldots\ldots \qquad (4.8b)$$

Recurrent networks typically operate with a discrete representation of data; they employ neurons with a hard-limiting activation function. A system with discrete-time inputs and a discrete data representation is called an *automaton*. Thus, recurrent neural networks of this class can be considered as automatons.

The relation (4.8) describes system *state* O^{k} at instants k =1,2,3,4.. and they yield the sequence of *state transitions*. The network begins the state transitions once it is initialized at instants 0 with X^{0} , and it goes through state transitions O^{k} , for k=1,2,3, . . , until it possibly finds an equilibrium state. This equilibrium state often called an attractor.

A vector X^{e} is said to be an equilibrium state of the network if

$$F[W X^{e}] = X^{e}$$

Determination of the final weight matrix will be discussed in later units of this course.

Continuous- time network: The feedback concept can also be implemented with any infinitesimal delay between output and input introduced in the feedback loop. The assumption of such a small delay between input and output is that the output vector can be considered to be a continuous-time function. As a result, the entire network operates in continuous time. The continuous time network can be obtained by replacing delay elements in Fig.4.3 with suitable continuous time lag.

A simple electric network consisting of resistance and capacitance may be considered as one of the simple form of continuous-time feedback network, as shown in Fig.4.4. In fact, electric networks are very often used to model the computation performed by the neural networks.

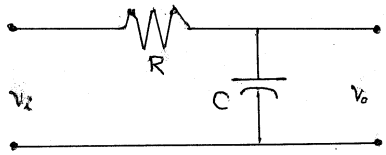


Fig. 4.4 R-C Network

4.4 Types of connections

There are three different options are available for connecting nodes to one another, as shown in Fig. 4.5 They are:

Intralayer connection: The output from a node fed into other nodes in the same layer.

Interlayer connection: The output from a node in one layer fed into nodes in other layer.

Recurrent connection: The outputs from a node fed into itself.

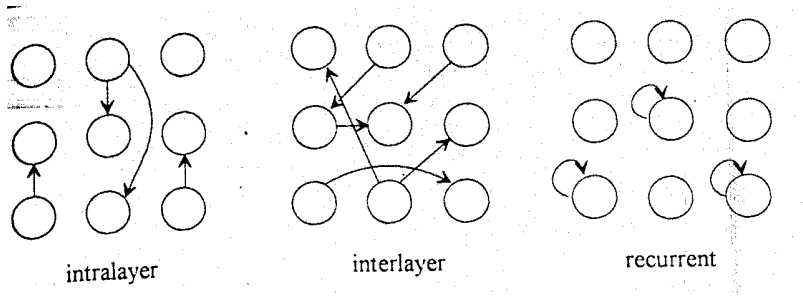


Fig. 4.5 Different connection option of Neural Networks

In general, when building a neural network its structure will be specified. In engineering applications, suitable and mostly preferred structure is the interlayer connection type topology. Within the interlayer connections, there are two more options: (i) feed forward connections and (ii) feedback connections, as shown in Fig. 4.6.

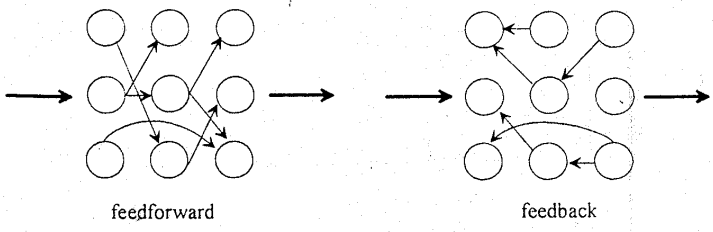


Fig. 4.6 Feed forward and feed back connections of Neural Networks

4.4.0 **Multi Layer Artificial Neural Networks**

Cascading a group of single layers networks can form a multi-layer neural network. There are two types of multi layer networks namely, feed-forward and feedback networks.

4.4.1 **Feed forward multi layer neural networks**

Cascading a group of single layers networks can form the feed forward neural network. This type networks also known as feed forward multi layer neural network. In which, the output of one layer provides an input to the subsequent layer. The input layer gets input from outside; the output of input layer is connected to the first hidden layer as input. The output layer receives its input from the last hidden layer. The multi layer neural network provides no increase in computational power over single layer neural networks unless there is a nonlinear activation function between layers. Therefore, due to nonlinear activation function of each neuron in hidden layers, the multi layer neural networks able to solve many of the complex problems, such as, nonlinear function approximation, learning generalization, nonlinear classification etc.

A multi layer neural network consists of input layer, output layer and hidden layers. The number of nodes in input layer depends on the number of inputs and the number of nodes in the output layer depends upon the number of outputs. The designer selects the number of hidden layers and neurons in respective layers. According to the Kolmogorov's theorem single hidden layer is sufficient to map any complicated input – output mapping.

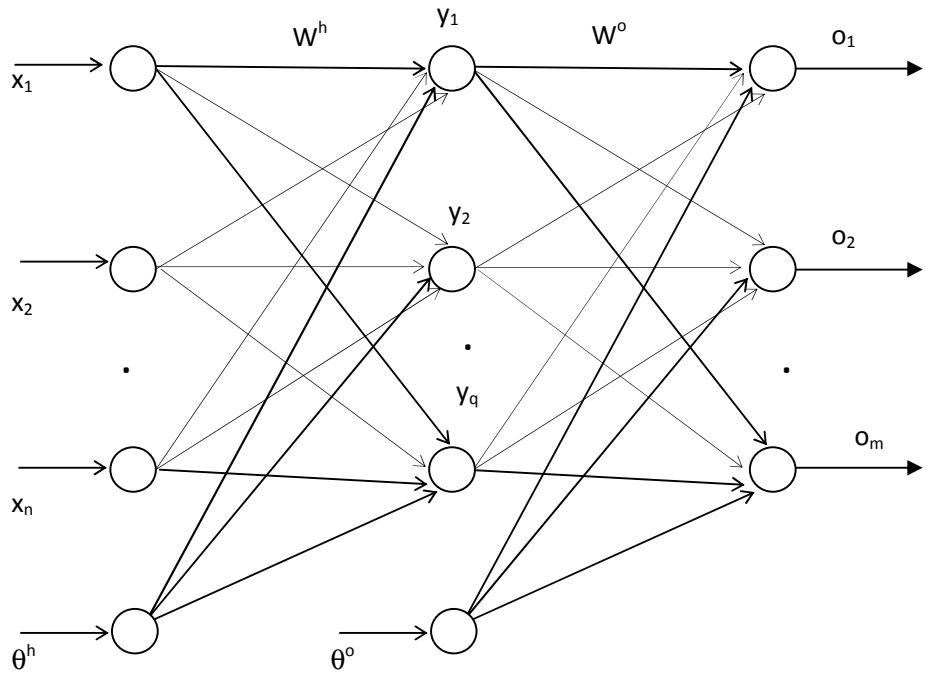
Kolmogorov's mapping neural network existence theorem is stated below:

Theorem: *Given any continuous function* $f:[0,1]^n \rightarrow R^m$, $f(x)=y$, *f can be implemented exactly by a three- layer feed forward neural network having n fanout processing elements in the first (**x**-input) layer, (2n +1) processing elements in the middle layer, and m processing elements in the top (**y**-output) layer.*

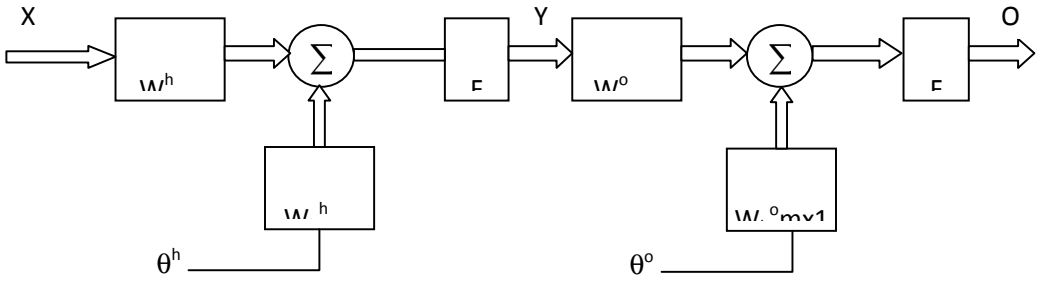
4.4.1.1 A typical three layer feed forward ANN

A typical configuration of a three-layer network as shown in Fig 4.7: The purpose of this section is to show how signal is flowing from input to output of the network. The network consists of **n** number of neurons in input layer, **q** number of neurons in hidden layer, and **m** number of neurons in out put layer. In which $\mathbf{X} = [x_1\ x_2\ \ldots\ x_n]$ is input vector of n-dimensional, $\mathbf{Y} = [y_1, y_2\ \ldots\ y_q]$ is the hidden layer output vector of q-dimensional , $\mathbf{O} = [o_1\ o_2\ \ldots\ o_m]$ is the output vector of neural network of m-dimensional. w^h_{ij}, denotes the synaptic weight between j^{th} neuron of input layer to i^{th} neuron of hidden layer, w^o_{ki}, denotes the synaptic weight between k^{th} neuron of hidden layer to i^{th} neuron of output layer, θ^h is the threshold for hidden layer neurons, θ^o is the threshold for output layer neurons and the threshold values may be varied through its connecting weights. The block diagram of three layer network as shown in (b) and (c).

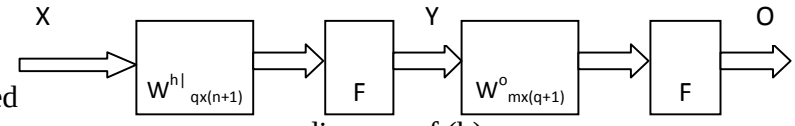
In general the neurons in input layer are dummy and they just pass on the input information to the next layer through the next layer weights. In Fig. 4.7(b), $W^h_{q\times n}$ is the synaptic weight matrix between input layer and hidden layer, $W^o_{m\times q}$ is the synaptic weight matrix between hidden layer and output layer, $Wb^h_{q\times 1}$ is the bias weight vector for hidden layer neurons, $Wb^o_{m\times 1}$ is the bias weight vector for output layer neurons. $W^h_{q\times n}$ and $Wb^h_{q\times 1}$ can have single representation with appropriate dimension as $W^h_{q\times(n+1)}$ and similarly $W^o_{m\times q}$ and $Wb^o_{m\times 1}$ can have single representation with appropriate dimension as $W^o_{m\times(q+1)}$. The simplified block diagram as shown in Fig.4.7(c).



(a) A three layer feed forward neural network



(b) A detailed Block diagram of (a)



(c) Simplified block diagram of (b)

Fig. 4.7 Configurations of three layer neural network

F denotes is the activation functions vector. The different layers may have different activation functions. Even with in a layer the different neurons may have different activation functions and forms an activation functions vector. Of course, a single activation function may be used by all neurons in a layer and without loss of generality it can be considered as a vector. For more details on activation functions may be referred to unit-3.

Forward signal flow: In general the signal flow in the feed forward multi layer networks is in two directions: forward and backward. The forward signal flow indicates the

activation dynamics of each neuron and backward flow indicates synaptic weight dynamics of the neural network. The synaptic weight dynamics is nothing but the training of neural network and this will be discussed in later units of this course. Before the training, the network will be initialized by known weights or with small random weights. As we know that, the neuron has two operations, one is finding the net input to the neuron and the second is the passing the net input through an activation function. The processing of signal flow in the network of Fig 4.7 as follows:

The net input of i^{th} neuron in the hidden layer may be obtained as

$$net_i^h = \sum_{j=0}^{n} w_{ij}^h x_j \qquad (4.9)$$

where $x_0=\theta^h$ and $w_{i0}{}^h$ bias weight element of hidden units. The output of i^{th} neuron y_i in hidden layer may be obtained as

$$y_i = f_i(net_i^h) \qquad (4.10)$$

where $f_i(.)$ is a activation function, for example $f(a)=\frac{1}{1+\bar{e}^a}$ and other types of functions are discussed in Unit 3. The net input of k^{th} neuron in the output layer may be obtained as

$$net_k^o = \sum_{i=0}^{q} w_{ki}^o y_i \qquad (4.11)$$

where $y_0=\theta^o$ and $w_{i0}{}^o$ bias weight element of output units. The output of k^{th} neuron o_k in output layer may be obtained as

$$o_k = f_k(net_k^o) \qquad (4.12)$$

o_k is the actual output of k^{th} neuron in the output layer. In training stage this value will be compared with that of desired output if the learning mechanism is supervised.

The most popular networks of this type are multi layer perceptron, Radial basis networks, Hamming network, Kohonen networks etc.

Example: Consider a typical neural network shown in Fig. 4.8, find the outputs of the network for given set of inputs in Table 1. Consider the activation function as a unipolar sigmoid function.

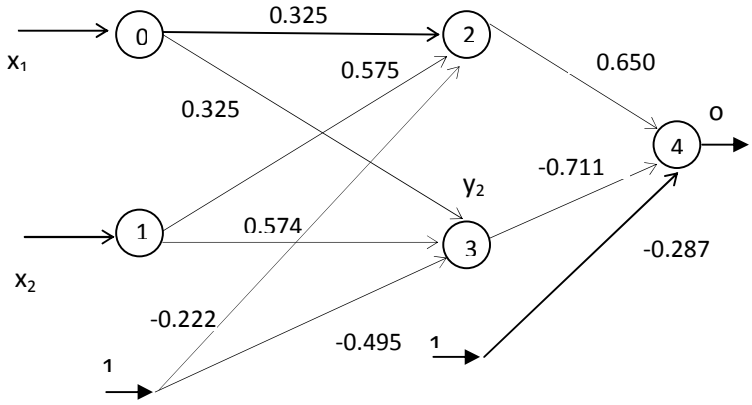


Fig. 4.8 A typical three layer network

Table 1: Inputs

S.No.	x_1	X_2
1	0	0
2	0	1
3	1	0
4	1	1

From the given network the parameters are as follows: n=2, q = 2, and m=1 and the weights matrices are:

$$W^h_{2\times2} = \begin{bmatrix} 0.325 & 0.575 \\ 0.325 & 0.574 \end{bmatrix} \text{ and } Wb^h_{2\times1} = \begin{bmatrix} -0.222 \\ -0.495 \end{bmatrix}$$

$$W^o_{1\times2} = \begin{bmatrix} 0.650 & -0.711 \end{bmatrix} \text{ and } Wb^o_{1\times1} = \begin{bmatrix} -0.287 \end{bmatrix}$$

Let the bias elements are $\theta^h = 1.0$ and $\theta^o=1.0$. Let us calculate for input X = [0 1].

$$Net^h = W^h_{2\times2} \times X^T + Wb^h_{2\times1} \times \theta^h = \begin{bmatrix} 0.325 & 0.575 \\ 0.325 & 0.574 \end{bmatrix} \times \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix} + \begin{bmatrix} -0.222 \\ -0.495 \end{bmatrix} \times \begin{bmatrix} 1.0 \\ \end{bmatrix} = \begin{bmatrix} 0.353 \\ 0.0790 \end{bmatrix}$$

$$Y = \begin{bmatrix} \frac{1.0}{1.0+e^{-0.353}} \\ \frac{1.0}{1.0+e^{-0.0790}} \end{bmatrix} = \begin{bmatrix} 0.5873 \\ 0.5197 \end{bmatrix}$$

$$Net^o = W^o_{1\times2} \times Y^T + Wb^o_{1\times1} \times \theta^o = \begin{bmatrix} 0.650 & -0.711 \end{bmatrix} \times \begin{bmatrix} 0.5873 \\ 0.5197 \end{bmatrix} + \begin{bmatrix} -0.287 \end{bmatrix} \times \begin{bmatrix} 1.0 \end{bmatrix} = -0.27476$$

Output **o** = $\frac{1.0}{1.0+e^{-0.2058}} = 0.43173.$

Similarly we can obtain the outputs for other inputs.

4.4.2 Feedback multi layer ANNs

Single layer feedback networks are discussed in previous section 4.3, in which output of neurons is fed back as inputs. In multi layer feed back networks can be formulated by different combination of single layer networks. One type of these is two single layer networks connected in feed back configuration with a unit time delay and it is shown in Fig. 4.9. The time delays are known as dynamical element. The networks with time delays known as dynamical systems.

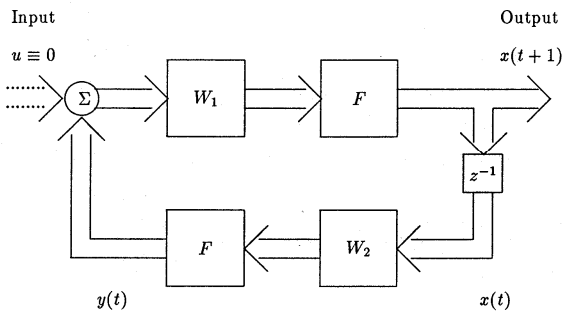


Fig.4.9 Two layer recurrent network

In Fig 4.9, the network consists of a weight matrix W_1 and nonlinear function F (.) in the forward path and a similar network with a weight matrix W_2 and nonlinear function F (.) in the feed back path. The mathematical relation may be described as

$$X(t+1) = F \{ W_1 F(W_2 X(t))\}$$

Where F(.) is the nonlinear function operator and are discussed in unit 3. The weights of the network have to adjust so that the given set of vectors is realized as fixed points of the dynamical transformation realized. The procedure for adjusting the weights will be discussed in later units of this course.

The most complex network among the ensemble is the one shown in Fig. 4.10. The system consists of three networks with weights W_1, W_2 and W_3 connected in series and feedback with a time delay in the feedback path. Alternately, a time delay can also be included in the forward path so that both the feed forward and feedback signals are dynamic variables. Such a network represents a complex nonlinear dynamical transformation from input to output.

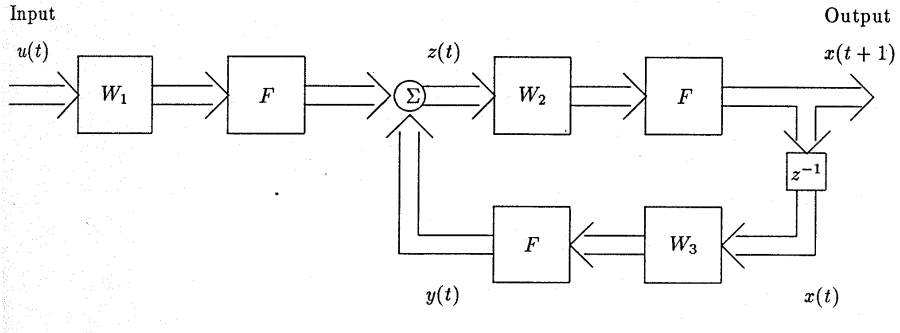


Fig. 4.10 Multi layer networks in dynamical systems

The well-known dynamical networks are Bi-directional associate memory networks, Adaptive Resonance Theory, Cellular networks, Time-delay networks, and Counter-Propagation networks.

Neural Dynamics

5.0 Introduction

An artificial neural network consists of several processing units and is governed by set of rules. These rules are specified in terms of activation and synaptic dynamics equations, which characterizes the behavior of a network. The activation state of a network may be described by the set of activation values of the neurons in the network at any given instant of time. A point in the activation state space specifies the state of the network. The trajectory of the activation states, leading to a solution state, refers the dynamics of the network. The trajectory depends upon the activation dynamics built into the network. The activation dynamics is prescribed by a set of equations, which can be used to determine the activation state of the network at the next instant.

For a given information, the weights of the network are adjusted to enable the network learn the knowledge in the given information. The set of weight values in a network at any given instant defines the weight state, which can be viewed as a point in the weight space. The trajectory of the weight states in the weight space is determined by the synaptic dynamics of the network.

A network is led to one of its steady activation states by the activation dynamics and the input. Since the steady activation state depends on the input and is referred as short-term memory. On the other hand, the steady weight state of a network is determined by the synaptic dynamics for a given set of training patterns, and is does not change thereafter. Hence this steady weight state is referred to as long-term memory.

5.1 Activation Dynamic models

An activation dynamics is described by the first derivative of the activation value of a neuron [1]. Consider a network of size N neurons fully connected. For the i^{th} neuron as shown in Fig 5.1, its activation is expressed as

$$\dot{x}_i = \frac{dx_i}{dt} = h(.) \qquad (5.1)$$

where h(.) is a function of the activation state and synaptic weights of the network.

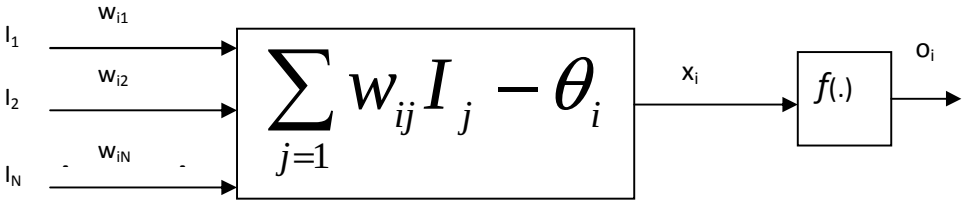


Fig. 5.1

Schematic representation of i^{th} neuron in a Network of size N

The activation value is generally associated with the cell membrane potential. The output function f(.) determines the output signal generated at the axon hillock due to a given membrane potential. The input values to the processing units are coming from internal sources especially through sensory inputs and from other neurons. Both of these types of inputs may have excitatory components, which tend to increase the activation of the node, or inhibitory components which tend to decrease the activation value of the node.

Activation dynamics can be formed in additive models and in more general shunting or multiplicative models.

5.1.1 Additive Activation models

The activation value x_i of the i^{th} neuron can be interpreted as the cell membrane potential and is a function of time, i.e., $x_i = x_i(t)$. The activation models are described by the first derivative of the activation value of a neuron. For the simplest case of a passive decay situation:

$$\dot{x}_i(t) = -A_i x_i(t) \tag{5.2}$$

where $A_i(>0)$ is a constant for the membrane and can be interpreted as the passive decay rate. The solution of the above equation is given by

$$x_i(t) = x_i(0)e^{-A_i t} \tag{5.3}$$

In electrical circuit analogy, A_i can be interpreted as membrane conductance, which is inverse of the membrane resistance (R_i). The initial value of x_i is $x_i(0)$. The steady state value of x_i is given by $x_i(\infty)=0$, which is also called the resting potential.
The passive decay time constant is altered by the membrane capacitance C_i which can also be viewed as a time scaling parameter. With C_i, the passive decay model is given by

$$C_i \dot{x}_i(t) = -A_i x_i(t) \tag{5.4}$$

and the solution is given by

$$x_i(t) = x_i(0)e^{-\frac{A_i}{C_i}t} \tag{5.5}$$

without loss of generality, we can assume $C_i=1$ in the discussion.
If we assume a nonzero resting potential, then the activation model can be expressed by adding a constant P_i to the passive decay term as

$$\dot{x}_i(t) = -A_i x_i(t) + P_i \tag{5.6}$$

whose solution is given by $x_i(t) = x_i(0)e^{-A_i t} + \frac{P_i}{A_i}(1 - e^{-A_i t})$ (5.7)

The steady state activation value is given by $x_i(\infty) = \frac{P_i}{A_i}$, the resting potential.

Assuming the resting potential to be zero ($P_i=0$), if there is a constant external excitatory input I_i, then additive activation model is given by

$$\dot{x}_i(t) = -A_i x_i(t) + B_i I_i \tag{5.8}$$

where $B_i(>0)$ is the weightage given to I_i. The solution of this equation is given by

$$x_i(t) = x_i(0)e^{-A_i t} + \frac{B_i I_i}{A_i}(1 - e^{-A_i t}) \tag{5.9}$$

The steady state value is given by $x_i(\infty) = \frac{B_i I_i}{A_i}$, which shows that activation value directly depends on the external input, and thus it is unbounded. In addition to the external input, if there is an input from the outputs of the other units, then the model becomes an additive auto-associative model, and the configuration this model can be visualized as in Fig 5.2. The n inputs are applied to a current summing junction and it acts as a summing node for the input currents. The total input current to nonlinear element (activation element) may be written as

$$\sum_{j=1}^{n} w_{ij}\, f_j(x_j(t)) + B_i I_i \qquad (5.10)$$

where I_i is the current source representing an external bias and $B_i(>0)$ is the weightage given to I_i. Let $x_i(t)$ denote the induced local field at the input of the activation function f(.). The total current flowing away from node is written as

$$\frac{x_i(t)}{R_i} + C_i \frac{d}{dt} x_i(t) \qquad (5.11)$$

where the first element is due to leakage resistance R_i and the second term is due to leakage capacitance C_i. By applying Kirchhoff current law to the node of the nonlinearity, we get

$$C_i \frac{d}{dt} x_i(t) + \frac{x_i(t)}{R_i} = \sum_{j=1}^{n} w_{ij}\, f_j(x_j(t)) + B_i I_i \qquad (5.12)$$

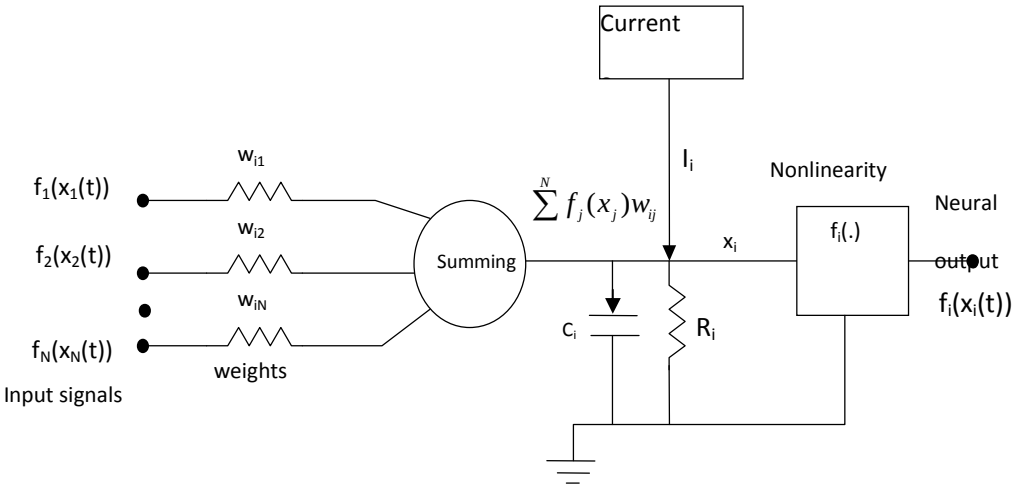


Fig. 5.2 Additive Dynamical Model of a Neuron ([1])

The output of neuron i may be written as $o_i = f(x_i(t))$. A commonly used activation function is the logistic function

$$f(x_i(t)) = \frac{1}{1+\exp(-x_i(t))}, \qquad i = 1,2,\ldots n. \qquad (5.13)$$

with C_i=1 and $A_i = \frac{1}{R_i}$, the equation (5.12) may be expressed as

$$\dot{x}_i(t) = -A_i x_i(t) + \sum_{j=1}^{N} f_j(x_j(t)) w_{ij} + B_i I_i \qquad (5.14)$$

For inhibitory feedback connections or for inhibitory external input, the equations will be similar to the above except for the signs of the second and third terms in the above equation. The classical neural circuit described by Parkel is a special case of the additive auto-associative model ([3]) and is given by

$$C_i \frac{d}{dt} x_i(t) = -\frac{x_i(t)}{R_i} + \sum_{j=1}^{N} \frac{x_j - x_i}{R_{ij}} + B_i I_i = -\frac{x_i(t)}{R_i'} + \sum_{j=1}^{N} \frac{x_j}{R_{ij}} + B_i I_i \qquad (5.15)$$

where R_{ij} is the resistance between the neurons i and j, and $\frac{1}{R_i'} = \frac{1}{R_i} + \sum_{j=1}^{N} \frac{1}{R_{ij}}$ Perkels model assumes a linear output function f(x)=x, thus resulting in a signal which is unbounded. The connection weights are symmetric, ie., $w_{ij}=w_{ji}$, then the resulting model is called Hopfield model.

A network consisting of two layers of processing units, where each unit in one layer (say layer 1) is connected to every unit in the other layer (say layer 2) and vice versa, is called a hetero-associative network. The additive activation model for a hetero-associative network is given by

$$\dot{x}_i(t) = -A_i x_i(t) + \sum_{j=1}^{M} f_j(y_j(t)) w_{ij} + I_i \text{ , } i = 1,2,..,N \qquad (5.16)$$

$$\dot{y}_j(t) = -A_j y_j(t) + \sum_{i=1}^{N} f_i(x_i(t)) w_{ji} + J_j \text{ , } j = 1,2,..,M \qquad (5.17)$$

where I_i and J_j are the net external inputs to the units i and j respectively. A_i and $f_i(.)$ could be different for each unit and for each layer.

5.1.2 Shunting Activation Models

Grossberg has proposed a shunting model to restrict the range of the activation values to a specified operating range irrespective of the dynamic range of the external inputs. Consider the saturation model, where the activation value is bounded to an upper limit. For an excitatory external input I_i, the shunting activation model is given by

$$\dot{x}_i(t) = -A_i x_i(t) + [B_i - x_i(t)] I_i \qquad (5.18)$$

The steady state activation value is obtained by setting $\dot{x}_i(t) = 0$ and then solve for x_i. The result is

$$x_i(\infty) = \frac{B_i I_i}{A_i + I_i} = \frac{B_i}{1 + A_i / I_i} \qquad (5.19)$$

As the input $I_i \to \infty$, then $x_i(\infty) \to B_i$. That is, the steady state value of x_i saturates at B_i. In other words, if the initial value $x_i(0) \le B_i$, then $x_i(t) \le B_i$ for all t.

5.2.0 Synaptic Dynamic models

Synaptic dynamics is attributed to learning in a biological neural network. The synaptic weights are adjusted to learn the pattern information in the input samples. The adjustment of the synaptic weights is represented by a set of learning equations, which describe the synaptic dynamics of the network. The learning equation describing a synaptic dynamics model is given as an expression for the first derivative (first difference) of the synaptic weight w_{ij} connecting the unit j to the unit i. The set of equations for all the weights in the network determine the trajectory of the weight states in the weight space from a given initial weight state. The first order difference equation may be written as

$$w_{ij}(l+1) = w_{ij}(l) + \phi(f_i(x_i(t)), f_j(x_j(t))) \qquad (5.20)$$

The second term in the above equation will vary depending up on the learning rule. In the next unit we will discuss popular learning rules.

Training Methods of Artificial Neural Networks

6.0.0 Introduction

The dynamics of neuron consists of two parts. One is the dynamics of the activation state and the second one is the dynamics of the synaptic weights. The Short Term Memory (STM) in neural networks is modeled by the activation state of the network and the Long Term Memory is encoded the information in the synaptic weights due to learning. The main property of artificial neural network is that, the ability of the learning from its environment and history. The network learns about its environment and history through its interactive process of adjustment applied to its synaptic weights and bias levels. Generally, the network becomes more knowledgeable about its environment and history, after completion each iteration of learning process. It is important to distinguish between representation and learning. Representation refers to the ability of a perceptron (or other network) to simulate a specified function. Learning requires the existence of a systematic procedure for adjusting the network weights to produce that function. Here we will discuss most of popular learning rules.

6.1.0 Definition of learning

There are too many activities associated with the notion of learning and we define learning in the context of neural networks [1] as

"*Learning is a process by which the free parameters of neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter change takes place*"

Based on the above definition the learning process of ANN can be divided into the following sequence of steps:

1. The ANN is stimulated by an environment.

2. The ANN undergoes changes in its free parameters as a result of the above stimulation.
3. The ANN responds in a new way to the environment because of the changes that have occurred in its internal structure.

6.1.1 Types of Training Methods

A set of defined rules for the solution of a learning problem is called algorithm. There are different approaches to train an ANN. Most of the methods fall into one of two classes namely supervised learning and unsupervised learning.

Supervised learning: An external signal known as teacher controls learning and incorporates information.

Supervised training requires the pairing of each input vector with a target vector representing the desired output; together these are called a training pair. Usually a network is trained over a number of such training pairs. An input vector is applied, the output of the network is calculated and compared to the corresponding target vector and the difference (error) is fed back through the network and weights are changed according to an algorithm that tends to minimize the error. The vectors of the training set are applied sequentially, and errors are calculated and weights adjusted for each vector, until the error for the entire training set is at the acceptably low value.

Unsupervised learning: No external signal (teacher) is used in the learning process. The neural network relies upon both internal and local information.

Unsupervised training is a far more plausible model of training in the biological system. Developed by Kohonen (1984) and many others, it requires no target vector for the outputs, and hence, no comparisons to predetermined ideal responses. The training set consists solely of input vectors. The training algorithm modifies network weights to produce output vectors that consistent; i.e., both application of one of the training vectors and application of a vector that is sufficiently similar to it will produce the same patterns of outputs.

- The training process, therefore, extracts the statistical properties of the training set and group's similar vector into classes.
- Applying a vector from a given class as a input will produce a specific output vector, but there is no way to determine prior to training which specific output pattern will be produced by a given input vector class. Hence, the outputs of such a network must generally be transformed into a comprehensible form subsequent to the training process.

6.1.2 Types of basic learning mechanisms

In general the training of any artificial neural network has to use one of the following basic learning mechanisms.

The basic learning mechanisms of neural networks are:

i. Error-correction learning,
ii. Memory-based learning,
iii. Hebbian learning,
iv. Competitive learning and
v. Boltzmann learning.

In this unit we will discuss the above learning rules.

6.2.0 Error-correction Learning

A network is trained so that application of a set of inputs produces the desired (or at least consistent) set of outputs. Each such input (or output) set is referred to as a vector. Training is accomplished by sequentially applying input vectors, while adjusting network weights according to a predetermined procedure. During training the network weight gradually converge to values such that each input vector produces the desired output vector.

Let us consider n is time index, then the error at n^{th} instant can be obtain as a function of actual output of network (i.e. output of the output layer) denoted by $o_k(n)$ for k^{th} neuron and desired response or target output of network denoted by $d_k(n)$ for k^{th} node of a output layer. The error signal denoted by $e_k(n)$ is defined as

$$e_k(n) = d_k(n) - o_k(n) \qquad (6.1)$$

The error signal $e_k(n)$ is used to adjust the synaptic weights of neuron k. The corrective adjustment are designed to make the output signal $o_k(n)$ come closer to the desired response $d_k(n)$ in a step-by-step procedure. The schematic diagram of the error-correction mechanism is shown in Fig. 6.1.

The cost function or performance index E(n) is defined in terms of error signal $e_k(n)$ as

$$E(n) = \frac{1}{2} e_k^{\,2}(n) \qquad (6.2)$$

This kind of learning is rooted in optimum filtering. Based on this concept many learning rules are derived in order to update the weights of network. The popular rules based this concept are

1. Perceptron learning rule.
2. Delta learning rule.
3. Widrow Hoff learning rule.

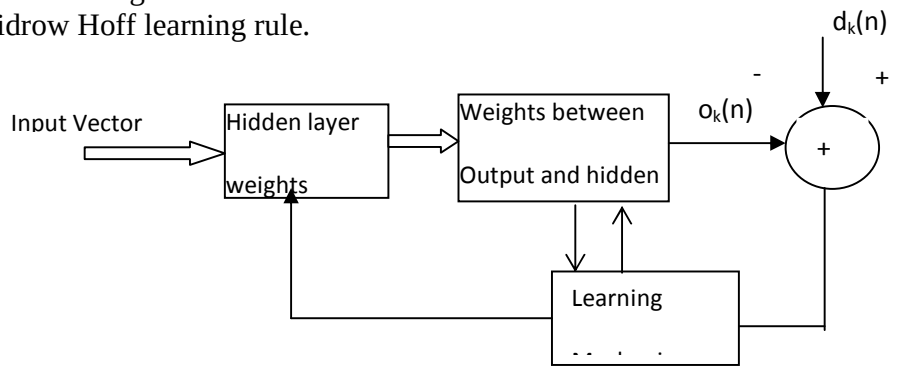


Fig. 6.1 A schematic representation of error-correction mechanism

6.2.1 Perceptron learning rule

The learning signal, know as error signal defined as the difference between the desired and actual neurons response [Rosenblatt 1958 – Zurada].

$$e_i = d_i - o_i \qquad (6.3)$$

where $o_i = \text{sgn}(W_i X)$ and d_i is the desired response as shown in Fig. 6.2.

Weight adjustments in this method, ΔW_i and ΔW_{ij} are defined as

$$\Delta W_i^T = c[d_i - \text{sgn}(W_i X)]X \qquad (6.4a)$$

$$\Delta w_{ij} = c[d_i - \text{sgn}(W_i X)]x_j \qquad \text{for } j = 1, 2, \ldots\ldots n \qquad (6.4b)$$

This rule is applicable only for binary neuron response and the relationship (6.4) express the rule for the bipolar binary case. Under this rule, weights are adjusted if and only if o_i is incorrect.

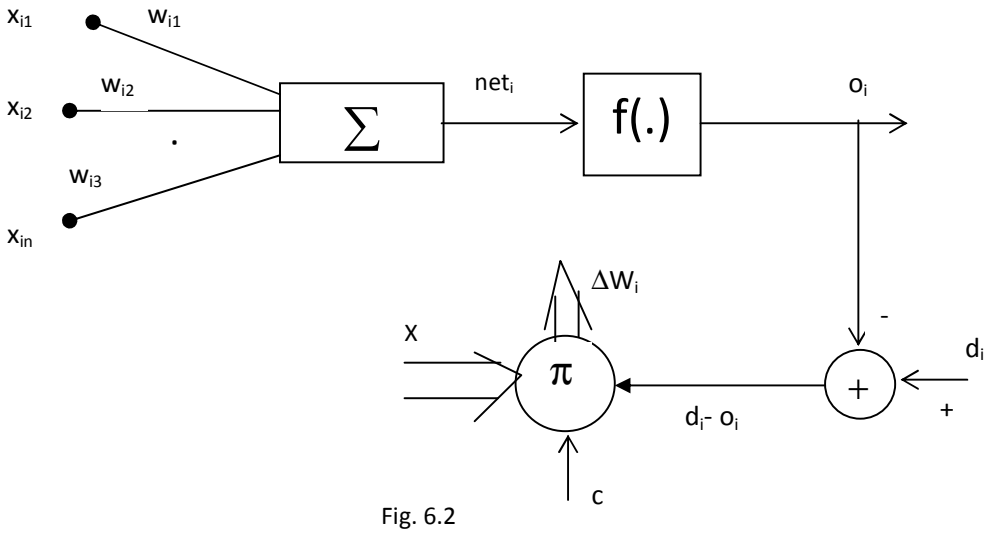


Fig. 6.2

Since the desired response is either 1 or –1, the weight adjustment (6.4a) reduces to

$$\Delta W_i^T = \pm 2cX \qquad (6.5)$$

ΔW_i is positive when $d_i = 1$ and sgn(WX) = -1 and is negative when $d_i = -1$ and sgn(WX) = 1. The weight adjustment is zero when there is an agreement between desired response and actual response.

Example: Consider the following set of input vectors X_i, i=1, 2, 3, and the initial weights vector W^1

$$X_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}, \quad X_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} \text{ and } W^1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$$

The learning constant is assumed to be c=0.1. The desired responses for X_1, X_2, X_3 are d_1=-1, d_2=-1 and d_3=1 respectively. Find the weight vectors for one cycle of training for three input vectors.

Step 1: Apply the input X_1, desired output is d_1:

$$net_1 = W^1 X_1 = [1 \ -1 \ \ 0 \ \ 0.5] \times \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = 2.5, o_1 = f(net_1) = sgn(2.5) = 1 \neq d.$$ Therefore,

correction in this step is necessary and the change in weight vector is

$$\therefore \Delta W^{1T} = c(d_1 - 1)X_1 = 0.1 \times [-1 \ \ -1] \times \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ 0.2 \end{bmatrix}$$

The updated weight vector W^2as $$W^{2T} = W^{1T} + \Delta W^{1T} = \begin{bmatrix} 1 \\ -1 \\ 0.0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$$

Step 2: Apply an input X_2 and the corresponding desired output is d_2. The present weight vector is W^2. The net input to the activation function net_2.

$$net_2 = W^2 X_2 = [0.8 \ \ -0.6 \ \ \ 0 \ \ \ 0.7] \times \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix} = -1.6, \quad o_2 = f(net_2) = sgn(-1/6) = -1 = d_2 .$$

Therefore the correction is not required. $\therefore W^3 = W^2$

Step 3: Input is X_3, desired output is d_3, and present weight vector is W^3.

$$net_3 = W^3 X_3 = [0.8 \ \ -0.6 \ \ \ 0.0 \ \ \ 0.7] \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = -2.1$$

$o_3 = f(net_3) = sgn(-2.1) = -1 \neq d_3$. Therefore correction is necessary in this step. The modified weight vector can be written as

$$\therefore\ W^{4T} = W^{3T} + c(d_3 - o_3)X_3 = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} + 0.2 \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.1 \\ 0.5 \end{bmatrix}$$

This terminates the sequence of learning steps unless the training set is recycled.

6.2.2 Delta learning rule

McClelland and Rumelhart introduced the delta rule in 1986 for training of neural networks. This learning rule is applicable only for continuous activation function. This rule can be derived from the condition of least square error between d_i and o_i. The squared error may be defined as

$$E \cong \frac{1}{2}(d_i - o_i)^2 = \frac{1}{2}[d_i - f(W_i X)]^2 \qquad (6.6)$$

The error gradient vector with respect to W_i may be written as $\nabla E = -(d_i - o_i) f'(W_i X) X$ (6.7)

The elements of gradient vector are , $\frac{\partial E}{\partial w_{ij}} = -(d_i - o_i) f'(W_i X) x_j$ *for* $j = 1, 2, \ldots, n.$ (6.8)

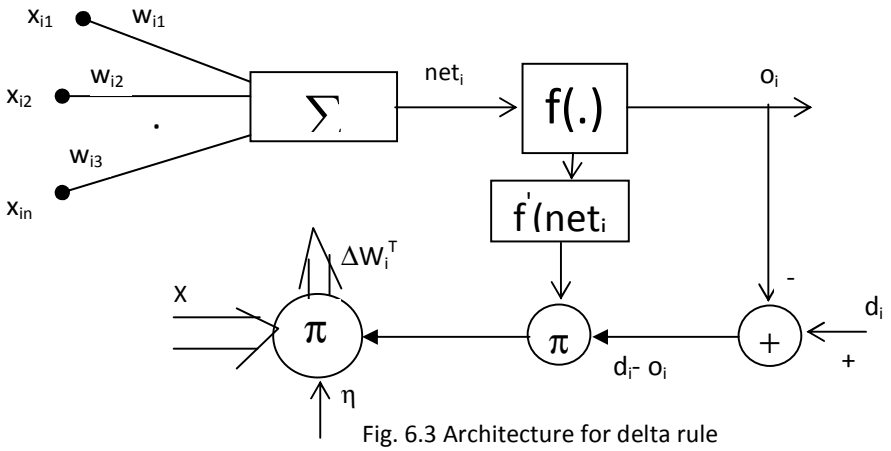


Fig. 6.3 Architecture for delta rule

In order to minimize the cost function E, it requires the weight changes to be in the negative gradient direction. Therefore

$$\Delta W_i = -\eta \nabla F \qquad (6.9)$$

where η is a positive constant.

From equations (6.7) and (6.9) we can obtain

$$\Delta W_i^T = \eta (d_i - o_i) f'(W_i X) X = \eta (d_i - o_i) f'(net_i) X \qquad (6.10)$$

Since $net_i = W_i X$ and the single weight adjustment becomes

$$\Delta w_{ij} = \eta (d_i - o_i) f'(net_i) x_j \qquad \text{for } i = 1, 2, \ldots\ldots n. \qquad (6.11)$$

For this rule, the weights can be initialized to any value. The diagrammatic representation of delta rule is shown in Fig. 6.3.

Example: This example illustrates the delta-learning rule of the Fig. 6.3. Consider the input vectors and the initial weight vectors are

$$X_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}, X_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}, \quad X_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} \text{ and } W^{1T} = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} \text{ respectively.}$$

The desired responses are $d_1 = -1$, $d_2 = -1$ and $d_3 = 1$. Assume the activation function to be bipolar sigmoid function and $\eta = 0.1$.

$$f(\text{net}) = \frac{2}{1+e^{-\text{net}}} - 1, \quad f'(\text{net}) = \frac{1}{2}\left[1 - f(\text{net})^2\right] = \frac{1}{2}\left(1 - o^2\right)$$

Step 1: Apply the input vector X_1,

$$\text{net}_1 = W^1 X_1 = \begin{bmatrix} 1 & -1 & 0 & 0.5 \end{bmatrix} \times \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = 2.5, \quad o_1 = f(\text{net}_1) = \frac{2}{1+e^{-2.5}} - 1 = 0.848$$

$$f'(\text{net}_1) = \frac{1}{2}\left[1 - (o_1)^2\right] = 0.140$$

$$\Delta W^{1T} = \eta (d_1 - o_1) f'(\text{net}_1) X_1 = 0.1(-1 - 0.848) \times 0.140 \times \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} -0.026 \\ 0.052 \\ 0 \\ 0.026 \end{bmatrix}$$

$$W^{2T} = W^{1T} + \Delta W^{1T} = \begin{bmatrix} 0.974 \\ -0.948 \\ 0 \\ 0.526 \end{bmatrix}$$

Step 2 : Apply the input vector X_2 and consider the present weight vector is W^2

$$\text{net}_2 = W^2 X_2 = -1.948 \text{ and} \qquad o_2 = f(\text{net}_2) = -0.750$$

$$f'(\text{net}_2) = \frac{1}{2}\left[1 - (o_2)^2\right] = 0.218$$

$$\Delta W^{2T} = 0.1(-1 + 0.75) \times 0.218 \times \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ -0.008 \\ 0.002 \\ 0.005 \end{bmatrix}$$

$$W^{3T} = W^{2T} + \Delta W^{2T} = \begin{bmatrix} 0.974 \\ -0.948 \\ 0 \\ 0.526 \end{bmatrix} + \begin{bmatrix} 0 \\ -0.008 \\ 0.002 \\ 0.005 \end{bmatrix} = \begin{bmatrix} 0.974 \\ -0.956 \\ 0.002 \\ 0.531 \end{bmatrix}$$

Step 3: Apply the input vector X_3 and consider the weight vector W^3

$$net_3 = W^3 X_3 = -2.46 \text{ and } o_3 = f(net_3) = -0.842, \qquad f'(net_3) = \frac{1}{2}\left[1 - (o_3)^2\right] = 0.145$$

$$\Delta W^{3T} = 0.1 \times (1 + 0.842) \times 0.145 \times \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} -0.026 \\ 0.026 \\ 0.013 \\ -0.026 \end{bmatrix}$$

$$W^{4T} = W^{3T} + \Delta W^{3T} = \begin{bmatrix} 0.974 \\ -0.956 \\ 0.002 \\ 0.531 \end{bmatrix} + \begin{bmatrix} -0.027 \\ 0.027 \\ 0.014 \\ -0.027 \end{bmatrix} = \begin{bmatrix} 0.947 \\ -0.929 \\ 0.016 \\ 0.504 \end{bmatrix}$$

This process will be repeated and the corrections will be performed in each step till its error $(d_i - o_i)$ should be within tolerable limits.

6.2.3 **Widrow-Hoff learning rule**

This learning rule was developed by Bernard Widrow and Marcian Hoff (1960, 1962). This rule is independent of the activation function of neurons. It minimizes the squared error between the desired output d_i and the neuron's activation value $net_i = W_i X$. Therefore, the error may be defined as

$$e_i = d_i - o_i = d_i - W_i X \qquad (6.12)$$

Using this learning rule, the change in weight vector

$$\Delta W_i = \eta (d_i - W_i X) X \qquad (6.13)$$

This single weight adjustment may be written as

$$\Delta w_{ij} = \eta (d_i - W_i X) x_j \qquad \text{for } j = 1, 2, \ldots\ldots\ldots, n. \qquad (6.14)$$

This rule is a special case of the delta learning rule, i.e. f(net) = net and f'(net) = 1. This rule also called the LMS (Least Mean Square) learning rule.

6.3.0 Memory Based Learning

In memory – based learning [1] , most of the experiences are explicitly stored in a large memory of correctly classified input-output exemplar patterns. $\{(X_i, d_i)\}_{i=1}^{N}$, where X_i denotes an input vector and d_i denotes the corresponding desired response. When classification of test vector X_{test} is required, the algorithm responds by retrieving and analyzing the training data in a "local neighborhood" of X_{test} memory-based learning algorithms. All memory-based learning algorithms involves the following:

- Criterion used for defining the local neighborhood of the test vector X_{test}
- Learning rule applied to the training examples in the neighborhood of X_{test}.

The algorithms differ from each other in the way in which these two ingredients defined.

A simple type of memory-based learning known as the nearest neighbor rule, the local neighborhood is defined as the training example that lies in the immediate neighborhood of the test vector X_{test}.

For example, $X'_N \in \{X_1, X_2, \ldots . X_N\}$ is said to be the nearest neighbor of X_{test} of $\min d(X_i, X_{test}) = d(X'_N, X_{test})$ where $d(X_i, X_{test})$ is the Euclidean distance between the vectors X_i and X_{test}.

The class associated with minimum distance, that is vector X'_N is reported as the classification of X_{test}.

A variant of the nearest neighbor classifier is the k-nearest neighbor classifier, which proceeds as follows

- Identify the k classified pattern that lie nearest to the test vector X_{test} for some integer k.
- Assign X_{test} to the class (hypothesis) that is most frequently represented in the k nearest to X_{test}.

Thus the k nearest neighbor classifies acts like an averaging device. In particular, it discriminates against a single outlier, as shown in Fig. 6.4 for k = 3. An outlier is an observation that is improbably large for a nominal model of interest.

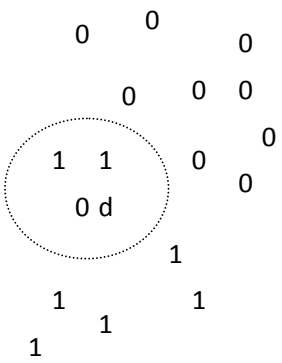


Fig. 6.4 Class 1

Fig. 6.4 The area lying inside the dashed circle includes two points pertaining to class 1 and an outlier from class 0. The point d *corresponds to the test vector* X_{test}*. With k=3, the k-nearest*

= 3, the k-nearest neighbor classifier assigns Class 1 to point d even though it lies closest to the outlier.

6.5.0 Hebbian Learning

It is the oldest and famous of all learning rules. It is name in honor of neuro-psychologist Hebb (1949). The basic theory comes from a book by Hebb (1949), *The Organization of behavior*. When an axon of cell A is near enough to excite a cell B and repeatedly or persistently take part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased. Hebb proposed this change as a basis of associative learning, which would result in an enduring modification on activity pattern of a spatially distributed "assembly of nerve cells". This statement can be rephrase as a two-part rule:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased.
2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

The synapse defined by following the above rules is called Hebbian synapse. According to Haykin [1], Hebbian synapse defined as synapse that uses a time-dependent, highly local, and strongly interactive mechanism to increase synaptic efficiency as a function of the correlation between the presynaptic and postsynaptic activities.

The key properties that characterizes a Hebbian synapse are:

1. *Time-dependant mechanism*: It refers that; the modifications in a Hebbian synapse depend on the exact time of occurrence of the presynaptic and postsynaptic signals.
2. *Local mechanism*: A synapse is the transmission site where information-bearing signals are in spatiotemporal contiguity.
3. *Interactive mechanism*: The change in a Hebbian synapse depends on signals on both sides of the synapse. That is, a Hebbian learning depends on a "true interaction" between presynaptic and postsynaptic signals in the sense that we can not make a predictions from either one of these two activities by itself.
4. *Conjunctional or correlational mechanism*: According to one of Hebbian interpretation, the co-occurrence of presynaptic and postsynaptic signals is sufficient to produce the synaptic modification. It is the reason to refer the Hebbian synapse as a conjunctional synapse. For another interpretation of hebbian learning, one may think of the interactive mechanism characterizing a Hebbian synapse an statistical terms. In particular, the correlation over time between pre-synaptic and postsynaptic signals is viewed as being responsible for a synaptic change. Accordingly, a Hebbian synapse is also referred to as a correlation synapse.

6.5.1 Mathematic relations of Hebbian learning

Consider a synaptic weight w_{kj} of neuron k with presynaptic and postsynaptic signals by x_j and o_k respectively. The modification applied to the synaptic weight w_{kj} at time step n is expressed in the general form

$$\Delta w_{kj}(n) = f\left(o_k(n), x_j(n)\right)$$

where $f\left(o_k(n), x_j(n)\right)$ is a function of both postsynaptic and presynaptic signals.

The simple from of Hebbian learning is described by

$$\Delta w_{kj}(n) = \eta\, o_k(n)\, x_j(n) = \eta\, f(W_i X)\, x_j(n) \qquad \text{for } j = 1, 2, \ldots, n \qquad (6.16)$$

where η is a learning rate and is a positive constant.

Covariance hypothesis [1] : In this hypothesis, the presynaptic and postsynaptic signals in (6.16) are replaced by the deviation of presynaptic and postsynaptic signals from their respective average values over a certain time interval. Let $\bar{x}$ and $\bar{o}$ denote the time – averaged values of the presynaptic signal x_j and postsynaptic signal o_k, respectively. According to the covariance hypothesis, the adjustment applied to the synaptic weight w_{kg} is defined by

$$\Delta w_{kj} = \eta\,(x_j - \bar{x})\,(o_k - \bar{o}) \qquad (6.17)$$

where η is the learning rate.

The Hebbian learning rules represents a purely feed forward, unsupervised learning. The rule state that: if the cross product of output and input or correlation term $o_k\, x_j$ is positive, this results in an increase of weight w_{kj}, otherwise the weight decreases.

Example: This example illustrates the Hebbian learning rule with binary and continuous of activation functions. Consider the initial weigh vector and set of three input vectors

$$W^{IT} = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix},\ X_1 = \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix},\ X_2 = \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix} \text{ and } X_3 = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix}$$

The network needs to be trained using the above data and the learning rate to be $\eta = 1$.

Assume the network shown in Fig. 6.5

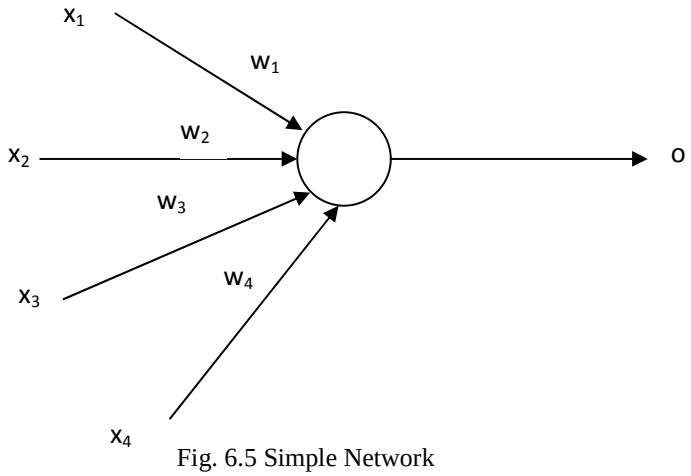


Fig. 6.5 Simple Network

Input Initial Weight

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \qquad W^{IT} = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$$

Case 1 : Assume that bipolar binary neurons are used and takes f (net) = sgn(net).

Step 1 : Apply the input X_1 to network

$$net_1 = W^1 X_1 = \begin{bmatrix} 1 & -1 & 0 & 0.5 \end{bmatrix} \times \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = 3.0, \text{ and} \qquad o_1 = f(net_1) = f(3) = 1$$

$$W^{2T} = W^{1T} + sgn(net_1) X_1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 1.5 \\ 0.5 \end{bmatrix}$$

Step 2: Apply the input X_2

$$net_2 = W^2 X_2 = \begin{bmatrix} 2 & -3 & 1.5 & 0.5 \end{bmatrix} \times \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix} = -0.25 \text{ and } o_2 = f(net_2) = f(-0.25) = -1$$

$$W^{3T} = W^{2T} + sgn(net_2) X_2 = \begin{bmatrix} 2 \\ -3 \\ 1.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix} = \begin{bmatrix} 1 \\ -2.5 \\ 3.5 \\ 2.0 \end{bmatrix}$$

Step 3: For input X_3

$$net_3 = W^3 X_3 = \begin{bmatrix} 1 & -2.5 & 3.5 & 2.0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix} = -3.0, \text{ and } o_3 = f(net_3) = f(-3) = -1$$

$$W^{4T} = W^{3T} + sgn(net_3) X_3 = \begin{bmatrix} 1 \\ -2.5 \\ 3.5 \\ 2.0 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 1 \\ -3.5 \\ 4.5 \\ 0.5 \end{bmatrix}$$

It can be seen that learning with discrete activation function and $\eta = 1$ results in adding or subtracting the entire pattern vectors to and from the weight vector respectively.

Case 2: Assume that, the activation function is continuous bipolar function f(net) and also assume learning rate $\eta = 1$.

Step 1 : For input X_1, the net input is, $\text{net}_1 = 3$, $o_1 = \frac{2}{1+e^{-3}} - 1.0 = 0.905$

$$\Delta W^{1T} = \eta o_1 X_1 = 0.905 \times \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.905 \\ -1.81 \\ 1.3575 \\ 0 \end{bmatrix}$$

$$W^{2T} = W^{1T} + \Delta W^{1T} = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 0.905 \\ -1.81 \\ 1.3575 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.905 \\ -2.81 \\ 1.357 \\ 0.5 \end{bmatrix}$$

Step 2: For input X_2

$$\text{net}_2 = -0.155,\ o_2 = \frac{2}{1+e^{0.155}} = -0.077 \text{ and } \Delta W^{2T} = \eta o_2 X_2 = -0.077 \times \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix} = \begin{bmatrix} -0.077 \\ 0.038 \\ 0.154 \\ 0.116 \end{bmatrix}$$

$$W^{3T} = W^{2T} + \Delta W^{2T} = \begin{bmatrix} 1.828 \\ -2.771 \\ 1.512 \\ 0.616 \end{bmatrix}$$

Step 3 : For input X_3

$$net_3 = W^3 X_3 = -3.359,\ o_3 = \frac{2}{1+e^{3..359}} - 1 = -0.932$$

$$\Delta W^{3T} = \eta o_3 X_3 = \begin{bmatrix} 0 \\ -0.932 \\ 0.932 \\ -1.398 \end{bmatrix} \text{ and } W^{4T} = W^{3T} + \Delta W^{3T} = \begin{bmatrix} 1.828 \\ -3.703 \\ 2.444 \\ -0.782 \end{bmatrix}$$

Comparison of learning using discrete and continuous activation functions indicates that the weight adjustments are tapered for continuous f(net) but are generally in the same direction.

6.6.0 Competitive Learning

In competitive learning, the output of neurons of a neural network competes among themselves to become active (fired). Whereas in a neural network based on Hebbian learning, several output neurons may be active simultaneously, in competitive learning only a single output neuron is active at any one time.

The basic elements in the competitive learning rule (Rumelhart and Zipser, 1985) are:

- A set of neurons that are all the same except for some randomly distributed synaptic weights, which therefore respond differently to given set of input patterns.
- A limit imposed on the "strength" of each neuron.
- A mechanism that permits two neurons to compete for the right to respond to a given subset of outputs, such that only one output neuron or only one neuron per group is active (i.e. "on") at a time. The neuron that wins the competition is called a "Winner-takes-all neuron".

In the simplest form of competitive learning the neural network has a single layer of output neurons each of which is fully connected to the input nodes. The network may include of feedback connections among the neurons as indicated in Fig.6.6. In the network architecture described herein the feedback connections perform lateral inhibition, with each neuron tending to exhibit the neuron to which it is laterally connected. The feed forward and synaptic connections in the above network are all excitatory.

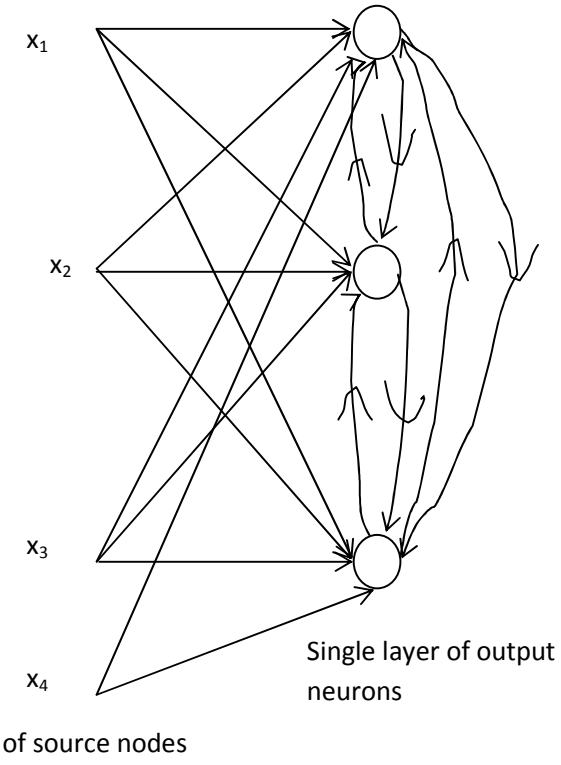


Figure 6.6. Architectural graph of a simple competitive learning network with feedforward (excitory) connections from the source nodes to the neurons, and lateral (inhibirory) connections among the neurons, the lateral connection are signified by open arrows[1].

6.6.1 Mathematical Relations of competitive learning

For a neuron k to be winning neuron, its induced local field v_k for a specified input pattern X must be the largest among all the neurons in the network. The output

signal o_k of winning neuron k is set equal to one and the output signals of all the neurons that lose the competition are set equal to zero. This can be written as

$$o_k = \begin{cases} 1 & \text{if } v_k > v_j \quad \text{for all j, } j \neq k \\ 0 & \text{otherwise} \end{cases} \qquad (6.18)$$

where the induced local field v_k represents the combined action of all the forward and feedback inputs to the neuron k. Let w_{kj} denote the synaptic weight connecting input node j to neuron k. Suppose that each neuron is allotted a fixed amount of synaptic weight (i.e.; all synaptic weights are positive), which is distributed among its input nodes, i.e.

$$\sum_j w_{kj} = 1 \qquad \text{for all k} \qquad (6.19)$$

A neuron then learns by shifting synaptic weights from its inactive to active input nodes. If a neuron does not respond to a particular input pattern, no learning takes places in that neuron. If a particular neuron wins the competition, each input nodes of that neuron relinquish some proportion of its synaptic weight, and the weight relinquished is then distributed equally among the active input nodes.
According this rule, the change Δw_{kj} applied to synaptic weight w_{kj} is defined by

$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}) & \text{if neuron k wins the competition.} \\ 0 & \text{if neuron k loses the competition.} \end{cases} \qquad (6.20)$$

where η is the learning rate parameter. This rule has the overall effect of moving the synaptic weight vector w_k of winning neuron k toward the input pattern X. The example of this mechanism is "winner-take-all learning rule" and discussed in the following section.

6.6.2 Winner-Take-All Learning rule

This is an example of competitive learning and is used for unsupervised training network. The Winner-Take all learning is used for learning statistical properties of inputs. The learning is based on the premise that one of the neurons in the layer, say the m^{th}, has the maximum response due to input X. This neuron is declared the winner. As a result of this winning even, the weight vector W_m.

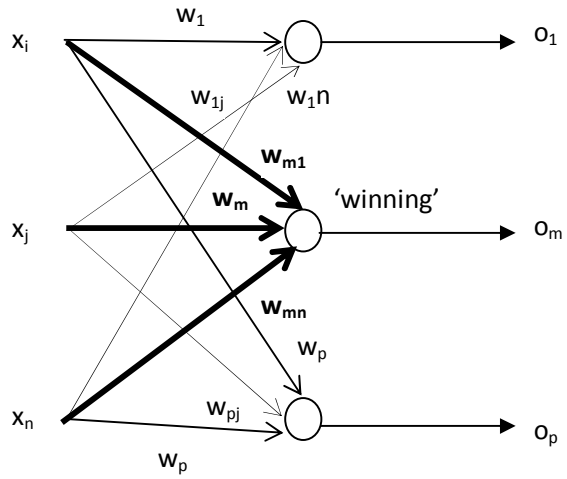
$W_m = [W_{m1}, W_{m2}, \ldots W_{mn}]$

Containing weights highlighted in the figure is the one adjusted in the given unsupervised learning step. Its changed is computed as follows $\Delta W_m^T = \eta\left(X - W_m^T\right)$ (6.21a)
or, the individual weight adjustment becomes, $\Delta w_{mj} = \eta(x_j - w_{mj}), \quad \text{for } j = 1, 2, \ldots n$ (6.21b)

where η is a small learning constant, typically decreasing as learning progress. The winner selection is based on the criterion of maximum activation among all p neurons participating in a competition. Its diagrammatic representation has shown in Fig.6.7.

$$W_m X = \max (W_i \; X) \qquad i = 1, 2, \ldots p \qquad (6.22)$$

The rule then reduces to incrementing W_m by a fraction of $(X - W_m^T)$.



(adjusted weights are highlighted)

Fig. 6.7 Competitive unsupervised "Winner-Take-All"

6.7.0 Boltzmann Learning

The Boltzman learning rule, named on honor of Ludwig Boltzman, is a statistical learning algorithm derived from ideas rooted in statistical machines. In a Boltzman machine the neurons forms a recurrent structure and they operate in a binary manner. That is, neuron 'on' state denoted by '+1' or in an "off" state denoted by –1.

The machine is characterized by an energy function, E the value of which is determined by the particular states occupied by the individual neurons of the machine, as given by

$$E = -\frac{1}{2}\sum_{j}\sum_{\substack{k \\ j \neq k}} w_{kj}\ x_k\ x_j \qquad (6.23)$$

where x_j is the state of neuron j, and w_{kj} is the synaptic weight connecting neuron j to neuron k. The fact that $j \neq k$ means simply none of the neurons in the machine has self-feedback. The machine operates by choosing a neuron at random for example, neuron k at some step of learning process, then flipping the state of neuron k from state x_k to state $-x_k$ at some temperate T with probability.

$$P(x_k \rightarrow -x_k) = \frac{1}{1+\exp(-\Delta E_k/T)} \qquad (6.24)$$

where ΔE_k is the energy change (i.e. the change in the energy function of the machine) resulting from such a flip.

Note that T is not a physical temperature, but rather a pseudo temperature. If this rule is applied repeatedly, the machine will reach thermal equilibrium.

The neurons of a Boltzman machine partition into two functional groups : visible and hidden. The visible neurons provide an interface between the network and the environment in which it operates, whereas the hidden neurons always operate freely.

There are two modes of operation to be considered.

- Clamped condition in which the visible neurons are all clamped into specific states determined by environment.
- Free running condition in which all the neurons (visible and hidden) are allowed to operate freely.

Let P_{kj}^{+} denote the correlation between the states of neurons j and k, with the network in its clamped condition. Let P_{kj}^{-} denote the correlation between the states of neurons j and k with the network in its free-running condition.

Both correlations are averaged over all possible states of the machine when it is in thermal equilibrium. Then, according to the Boltzmann learning rule, the change Δw_{kj} applied to the synaptic weight w_{kj} from the neuron j to neuron k defined by

$$\Delta w_{kj} = \eta \left(P_{kj}^{+} - P_{kj}^{-} \right), \qquad j \neq k \qquad (6.25)$$

where η is the learning rate parameter. Both P_{kj}^{+} and P_{kj}^{-} range in value from –1 to +1.

6.8.0 Outstar Learning rule

The outstar learning rule [2] is designed to produce a desired response **d** of the layer of P neurons in Fig. 6.8 (Grosberg 1974, 1982). It is used to provide learning of repetitive and characteristic properties of input / output relations. It is concerned with supervised learning; however it is supposed to allow the network to extract statistical properties of the input and output signals. The weight adjustment may be written as

$$\Delta W_j = \beta (d - W_j) \qquad (6.26a)$$

or the individual weight adjustments are

$$\Delta w_{mj} = \beta (d_m - w_{mj}), \qquad \text{for } m = 1, 2, \ldots, P \qquad (6.26b)$$

the weight vector in equation (6.26a) is defined as $W_j = \left[w_{1j}, w_{2j}, \ldots . w_{pj} \right]^T$ and β is a small positive constant.

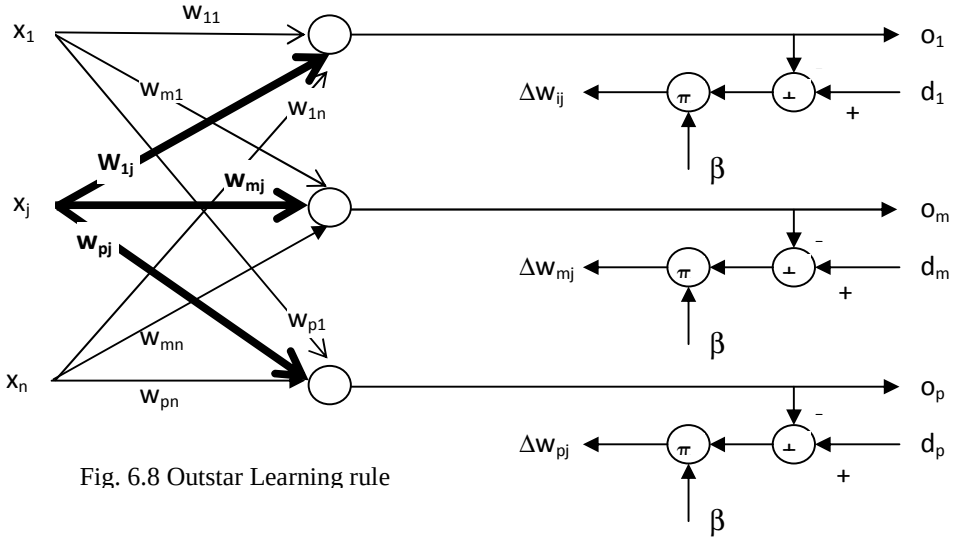


Fig. 6.8 Outstar Learning rule

UNCONSTRAINT OPTIMIZATION TECHNIQUES

7.0.0 **Introduction**

In this unit we describe the learning of single neuron is a process of adaptive filtering. As we learnt that, artificial neural network technique is one offers optimum solutions to many real world problems. That is, training of the ANN involves some of the optimization tools. We now discuss the relevant unconstraint optimization techniques, which will be useful in training of artificial neural networks.

7.1.0 **Adaptive Filtering Problem**

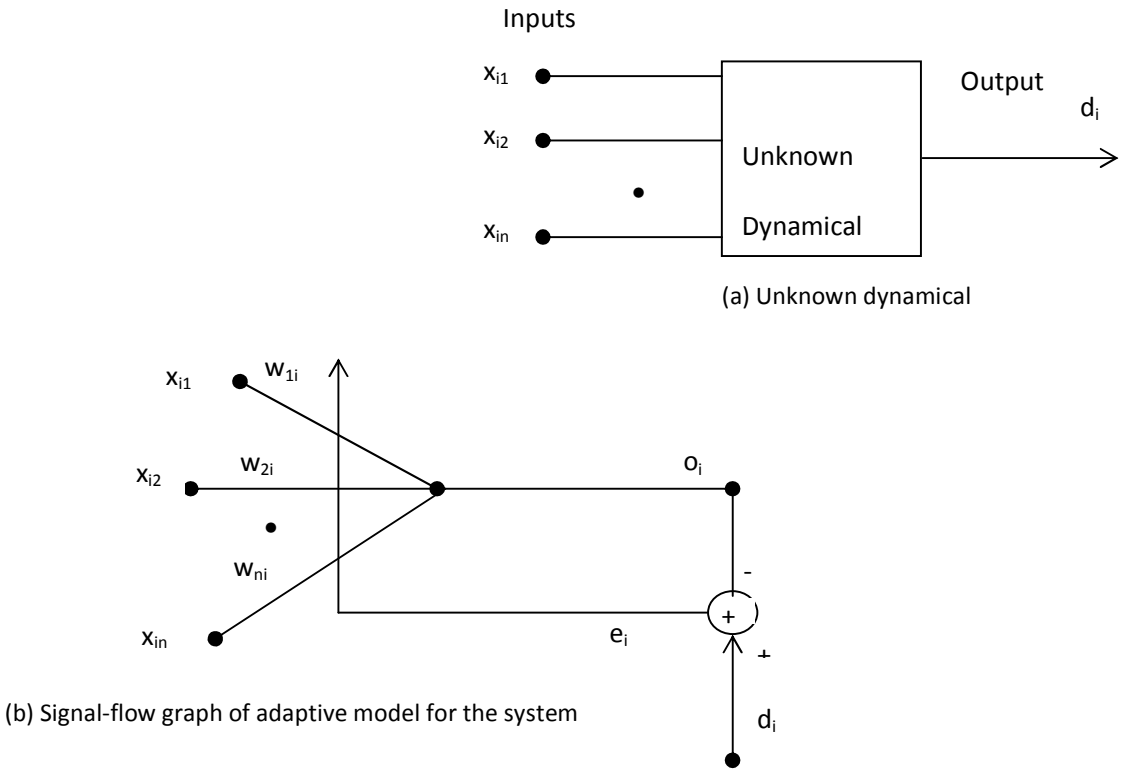
Consider a dynamical system, its mathematical description is unknown, as shown in Fig.(7.1a). But its input-output data generated by the system at discrete instants of time with uniform rate are available. When an n-dimensional stimulus X_i is the i^{th} input applied across n input nodes of the system, the system produces a scalar output d_i, where i = 1, 2, . . . , m. Thus the behavior of the system is described by the data set.

$$\Im : \{X_i, d_i, \quad i = 1, 2, \ldots m\} \tag{7.1}$$

where $X_i = [x_{i1}, x_{i2}, \ldots . x_{in}]^T$

The samples comprising $\Im$ are identically distributed according to an unknown probability law.

Now, the problem is how to design a multiple input-single output model of the unknown dynamical system by building it around a single linear neuron. The neural net model operates under the influence of an algorithm that controls necessary adjustment to the synaptic weights of the neuron.



(a) Unknown dynamical

(b) Signal-flow graph of adaptive model for the system

Fig.7.1. Schematic representation of Adaptive filtering process

In order to implement the above mechanism, the following steps should be followed:

- The algorithm starts from arbitrary synaptic weights.
- Adjustments to the synaptic weights are made on a continuous basis (i.e. time is incorporated into the constitution of algorithm).
- Computations of adjustments to the synaptic weights are completed inside a time interval that is one sampling period long.

The neural model described is referred to as an adaptive filter. Fig. (7.1b) shows a signal-flow graph of the adaptive filter. Its operation consists of two continuous processes.

1. *Filtering process*, which involves the computation of two signals. An output denoted by o_i is produced in response to input vector X_i. An error signal for i^{th} input denoted by e_i is obtained by comparing the output o_i to corresponding output d_i produced by the unknown system. In effect d_i acts as a desired response or target signal.
2. *Adaptive process*, which involves the automatic adjustment of the synaptic weights of the neuron in accordance with the error signal e_i.

Thus, the combination of these two processes working together constitutes feedback loop acting around the neuron. Since the neuron is liner, the output o_i is exactly the same as the induced local field v_i that i.e,

$$o_i = v_i = \sum_{k=1}^{n} w_{ik} \; x_{ik} \qquad (7.2)$$

where $w_{i1}, w_{i2}, \ldots . w_{in}$ are the n synaptic weights of the neuron, measured as time i.
Also the output may be written as an inner product of the vectors X_i and W_i as

$$o_i = X_i^T W_i \qquad (7.3)$$

where $W_i = [w_{i1}, w_{i2}, \ldots w_{in}]^T$ and the error signal e_i may be defined as

$$e_i = d_i - o_i \qquad (7.4)$$

The way in which the error e_i as used to control the adjustments to the neurons synaptic weights is determined by the cost function used to derive the adaptive filtering algorithm of interest. This is nothing but problem of optimization.

7.2.0 Unconstrained optimization techniques

Consider a cost function E(W) that is a continuously differentiable function of some unknown weight (parameter) vector W. It is a measure of how to choose the weight vector W of an adaptive filtering algorithm so that it behaves in an optimum manner. It is desire to find an optimal solution W* that satisfies the condition

$$E(W^*) \leq E(W) \qquad (7.5)$$

The unconstrained optimization problem may be states as

Minimize the cost function E(W) with respect to the weigh vector W. (7.6)

The necessary condition for optimality is

$$\nabla E(W^*) = 0 \qquad (7.7)$$

where ∇ is the gradient operator. $\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots\ldots, \frac{\partial}{\partial w_n} \right]^T$ (7.8)

$\nabla E(W)$ is the gradient vector of the cost function and

∇ written as $\nabla E(W) = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots\ldots, \frac{\partial E}{\partial w_n} \right]^T$ (7.9)

A class of unconstrained optimization algorithms suited for the design of adaptive filter is based on the idea of local iterative descent [1].

Starting with an initial guess denoted by W(0), generate a sequence of weight vector W(1), W(2), ….. , such that the cost function E(W) is reduced at each iteration of the algorithm, as shown by

$$E\left[(W(k+1)\right] < E\left[(W(k))\right] \quad (7.10)$$

Where W(k) is the old value of the weight vector and W(k+1) is its updated value.

The algorithm will eventually converge onto the optimal solution W*.

7.3.0 Method of Steepest Descent

In this method, the successive adjustments will be made to the weight vector are in the direction of steepest descent, i.e. in a direction opposite to the gradient vector $\nabla E(W)$.

$$\text{Let } g = \nabla E(W). \quad (7.11)$$

The steepest descent algorithm may be described as

$$W(k+1) = W(k) - \eta\, g(k) \quad (7.12)$$

where η is a positive constant called the learning-rate parameter and g(k) is the gradient vector at the point W(k). The adjustment term may be written as

$$\Delta W(k) = W(k+1) - W(k) = -\eta\, g(k) \quad (7.13)$$

Let us show that the steepest descent algorithm satisfies the optimization rule. Consider the first order Taylor series approximation around W(k) to approximate W(k+1) as

$$E\left[W(k+1)\right] = E\left[W(k)\right] + g^{T}(k)\,\Delta W(k)$$

Substituting (7.13) in this approximate relation yields

$$E\left[W(k+1)\right] = E\left[W(k)\right] - \eta\, g^{T}(k)\, g(k) = E\left[W(k)\right] - \eta \left\| g(k) \right\|^{2}$$

which shows that, for a positive learning-rate parameter η the cost function is decreased as the algorithm progresses from one iteration to the next. Also note that, these results true for small enough learning rates.

The steepest –descent converges to the optimal solution W* slowly. The learning rate η has influence on its convergence. When η is small, the transient response of the algorithm is over damped. When η is large, the transient response of the algorithm is under damped. When η exceeds a certain critical value, the algorithm becomes unstable.

7.4.0 Newton's Method

This method minimizes the quadratic approximation of the cost function E(W) around the current point W(k). A second-order Taylor series expansion of the cost function around the point W(k) may be written as

$$\Delta E\left[W(k)\right] = E\left[W(k+1)\right] - E\left[W(k)\right] \cong g^{T}(k)\,\Delta W(k) + \frac{1}{2}\Delta W^{T}(k)\, H(k)\, \Delta W(k) \quad (7.14)$$

g(k) is the n×1 gradient vector of the cost function E(W) evaluated at the gradient W(k). The matrix H(k) is the n×n Hassian matrix of E(W) also evaluated at W(k). The Hessian of E(W) is defined by

$$H(k) = \nabla^2 E(W) = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \bullet\bullet & \frac{\partial^2 E}{\partial w_1 \partial w_n} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \bullet\bullet & \frac{\partial^2 E}{\partial w_2 \partial w_n} \\ \bullet & \bullet & \bullet\bullet & \bullet \\ \frac{\partial^2 E}{\partial w_n \partial w_1} & \frac{\partial^2 E}{\partial w_n \partial w_2} & \bullet\bullet & \frac{\partial^2 E}{\partial w_n^2} \end{bmatrix} \quad (7.15)$$

Equation (7.15) requires the cost function E(W) to be differentiated twice with respect to W. Differentiating equation (7.14) with respect to ΔW, the change $\Delta E(W)$ is minimized when $g(k) + H(k)\,\Delta W(k) = 0$.

Solving the above equation yields

$$\Delta W(k) = -H^{-1}(k)\, g(k)$$

$$\therefore\ W(k+1) = W(k) + \Delta W(k) = W(k) - H^{-1}(k)\, g(k)$$

$$W(k+1) = W(k) - H^{-1}(k)\, g(k) \quad (7.16)$$

This method work only, if Hessian H(k) has to be positive definite matrix for all k. Unfortunately, in general there is no guarantee that H(k) is positive definite at every iteration of the algorithm.

7.5.0 Least-Mean-Square Algorithm

The least-mean-square (LMS) algorithm is based on the use of instantaneous values of the cost function and it may be written as

$$E[(W(k)] = \frac{1}{2} e^2(k) \quad (7.17)$$

Where e(k) is the error signal measured at time k. Differentiating $E[W(k)]$ with respect to W(k) as

$$\frac{\partial E[W(k)]}{\partial W(k)} = e(k) \frac{\partial E(k)}{\partial W(k)} \quad (7.18)$$

We have the relation for error is

$$e(k) = d(k) - X^T(k)\, W(k) \quad (7.19)$$

Hence $\frac{\partial e(k)}{\partial W(k)} = -X(k)$ and $\frac{\partial E[W(k)]}{\partial W(k)} = -X(k)\, e(k)$ (7.20)

Using this later relation (7.20) as an *estimator* for the gradient vector, we may write

$$\hat{g}(k) = -X(k)\, e(k) \quad (7.21)$$

Using the above gradient vector of (7.21) let us find out the update weight vector by using the method of steepest descent, therefore, we can write the LMS algorithm as follows:

$$\Delta W(k) = -\eta\, \hat{g}(k)$$

$$\hat{W}(k+1) = \hat{W}(k) + \Delta W(k) = \hat{W}(k) - \eta\, \hat{g}(k) = \hat{W}(k) + \eta\, X(k)\, e(k) \quad (7.22)$$

where η is the learning-rate parameter. The feedback loop around the weight vector $\hat{W}(k)$ in the LMS algorithm behaves like a low-pass filter.

The average time constant of this low pass filter is inversely proportional to the learning rate. The behavior of this filter much depends on the learning rate. If the learning rate small, then the adaptive process will be slow. The LMS algorithm is some times referred to as a 'Stochastic gradient algorithm'.

7.5.1 The summary of the LMS algorithm

Training sample : Input vector = X(k)

Desired response = d(k)

Learning rate = η

Initial guess weight vector $\hat{W}(k) = 0$, Computation for k = 1, 2, ...

$$e(k) = d(k) - \hat{W}^{T}(k)\, X(k) \text{ and } \hat{W}(k+1) = \hat{W}(k) + \eta\, X(k)\, e(k).$$

7.5.2 Diagrammatic representation of the LMS algorithm

The LMS algorithm can be represented as a Block diagram. Consider the equation (7.22) and substitute the expression for error of equation (7.19) in (7.22), the result may be written as

$$\hat{W}(k+1) = \hat{W}(k) + \eta\, X(k)\left[d(k) - X^{T}(k)\,\hat{W}(k)\right] = \left[I - \eta\, X(k)\, X^{T}(k)\right]\hat{W}(k) + \eta\, X(k)\, d(k) \qquad (7.23)$$

where I is the identity matrix. By using LMS algorithm, we can consider the following related.

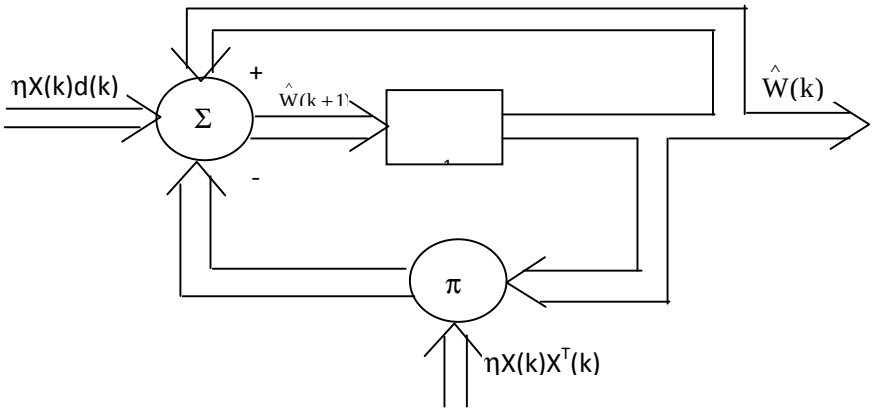


Fig. 7.2 Diagrammatic representation of the LMS algorithm

$$\hat{W}(k) = Z^{-1}\left[\hat{W}(k+1)\right] \qquad (7.24)$$

where Z^{-1} is the unit-delay operator. By using equations (7.23) and (7.24) we may represent the LMS algorithm as block algorithm as shown in Fig. 7.2.

PERCEPTRONS

8.0.0 **Introduction**

We know that perceptron is one of the early models of artificial neuron. It was proposed by Rosenblatt in 1958. It is a single layer neural network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The perceptron is a program that learns *concepts*, i.e. it can learn to respond with *True* (1) or *False* (0) for inputs we present to it, by repeatedly "studying" examples presented to it. The training technique used is called *the perceptron learning rule*. The perceptron generated great interest due to its ability to *generalize* from its training vectors and work with randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. In this also we give the perceptron convergence theorem.

8.1.0 Perceptron Model

In the 1960, perceptrons created a great deal of interest and optimism. Rosenblatt (1962) proved a remarkable theorem about perceptron learning. Widrow (Widrow 1961, 1963, Widrow and Angell 1962, Widrow and Hoff 1960) made a number of convincing demonstrations of perceptron like systems. Perceptron learning is of the supervised type. A perceptron is trained by presenting a set of patterns to its input, one at a time, and adjusting the weights until the desired output occurs for each of them.

The schematic diagram of perceptron is shown inn Fig. 8.1. Its synaptic weights are denoted by $w_1, w_2, \ldots w_n$. The inputs applied to the perceptron are denoted by $x_1, x_2, \ldots x_n$. The externally applied bias is denoted by b.

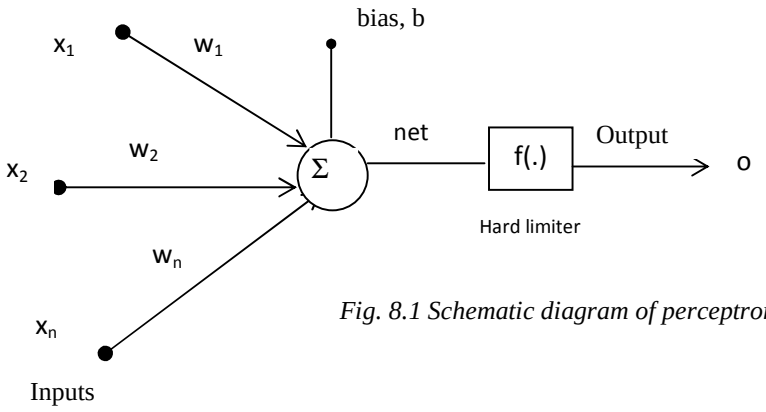


Fig. 8.1 Schematic diagram of perceptron

The net input to the activation of the neuron is written as

$$net = \sum_{i=1}^{n} w_i\, x_i + b \qquad (8.1)$$

The output of perceptron is written as $o = f(net)$ (8.2)

where f(.) is the activation function of perceptron. Depending upon the type of activation function, the perceptron may be classified into two types

i) Discrete perceptron, in which the activation function is ***hard limiter*** or ***sgn()*** function
ii) Continuous perceptron, in which the activation function is sigmoid function, which is differentiable. The input-output relation may be rearranged by considering $w_0=b$ and fixed bias $x_0 = 1.0$. Then

$$net = \sum_{i=0}^{n} w_i x_i = WX \qquad (8.3)$$

where $W = [w_0, w_1, w_2, \ldots . w_n]$ and $X = [x_0, x_1, x_2, \ldots x_n]^T$.

The learning rule for perceptron has been discussed in unit 7. Specifically the learning of these two models is discussed in the following sections.

8.2.0 Single Layer Discrete Perceptron Networks

For discrete perceptron the activation function should be *hard limiter* or *sgn()* function. The popular application of discrete perceptron is a pattern classification. To develop insight into the behavior of a pattern classifier, it is necessary to plot a map of the decision regions in n-dimensional space, spanned by the n input variables. The two decision regions separated by a hyper plane defined by

$$\sum_{i=0}^{n} w_i x_i = 0 \qquad (8.4)$$

This is illustrated in Fig. 8.2 for two input variables x_1 and x_2, for which the decision boundary takes the form of a straight line.

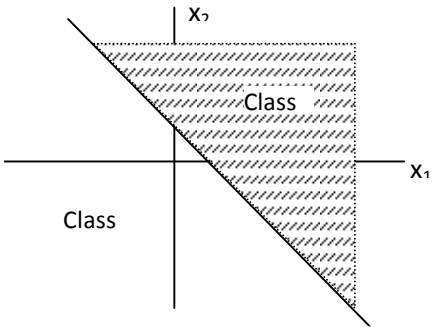


Fig. 8.2 Illustration of the hyper plane (in this example, a

For the perceptron to function properly, the two classed C_1 and C_2 must be linearly separable. This in turn, means that the patterns to be classified must be sufficiently separated from each other to ensure that the decision surface consists of a hyper plane. This is illustrated in Fig. 8.3.

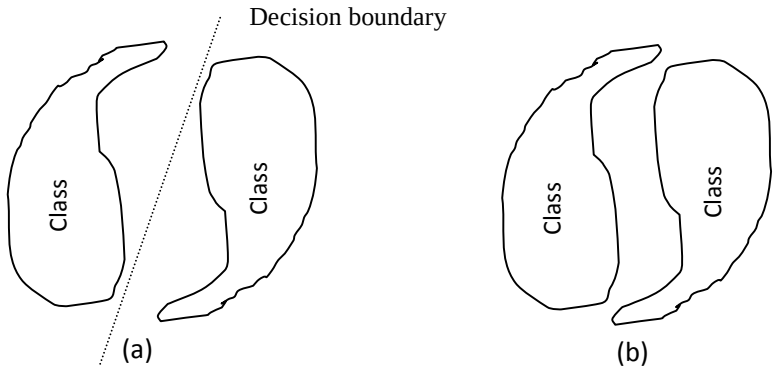


Fig. 8.3 (a) A pair of linearly separable

In Fig. 8.3(a), the two classes C_1 and C_2 are sufficiently separated from each other to draw a hyper plane (in this it is a straight line) as the decision boundary. If however, the two classes C_1 and C_2 are allowed to move too close to each other, as in Fig. 8.3 (b), they become nonlinearly separable, a situation that is beyond the computing capability of the perceptron.

Suppose then that the input variables of the perceptron originate from two linearly separable classes. Let æ$_1$ be the subset of training vectors $X_1(1)$, $X_1(2)$, ... , that belongs to class C_1 and æ$_2$ be the subset of train vectors $X_2(1)$, $X_2(2)$, ... , that belong to class C_2. The union of æ$_1$ and æ$_2$ is the complete training set æ. Given the sets of vectors æ$_1$ and æ$_2$ to train the classifier, the training process involves the adjustment of the W in such a way that the two classes C_1 and C_2 are linearly separable. That is, there exists a weight vector W such that we may write,

$$\left.\begin{array}{l} WX > 0 \quad \text{for every input vector X belonging to class } C_1 \\ WX \leq 0 \quad \text{for every input vector X belonging to class } C_2 \end{array}\right\} \qquad (8.5)$$

In the second condition, it is arbitrarily chosen to say that the input vector X belongs to class C_2 if WX = 0.

The algorithm for updating the weights may be formulated as follows:

1. If the k^{th} member of the training set, X_k is correctly classified by the weight vector W(k) computed at the k^{th} iteration of the algorithm, no correction is made to the weight vector of perceptron in accordance with the rule.

$$W^{k+1} = W^k \quad \text{if } W^k X_k > 0 \text{ and } X_k \text{ belongs to class } C_1 \qquad (8.6)$$

$$W^{k+1} = W^k \quad \text{if } W^k X_k \leq 0 \text{ and } X_k \text{ belongs to class } C_2 \qquad (8.7)$$

2. Otherwise, the weight vector of the perceptron is updated in accordance with the rule.

$W^{(k+1)T} = W^{kT} - \eta X_k$ if $W^k X_k > 0$ and X_k belongs to class C_2 (8.8a)

$W^{(k+1)T} = W^{kT} + \eta X_k$ if $W^k X_k \leq 0$ and X_k belongs to class C_1 (8.8b)

where the learning rule parameter η controls the adjustment applied to the weight vector. Equations (8.8a) and (8.8b) may be written general expression as

$$W^{(k+1)} = W^{kT} + \frac{\eta}{2}(d_k - o_k)X_k \quad (8.9)$$

8.2.1 Summary of the discrete perceptron training algorithm

Given are P training pairs of patterns

$\{X_1, d_1, X_2, d_2, \ldots . Xp, dp\}$, where X_i is $(n\times1)$, d_i is (1×1), $i = 1, 2, \ldots P$. Define $w_0=b$ is bias and $X_0 = 1.0$, then the size of augmented input vector is X_i $((n+1)\times)$.

In the following, k denotes the training step and p denotes the step counter with the training cycle.

Step 1: $\eta > 0$ is chosen and define E_{max}.

Step 2: Initialize the weights at small random values, $W = [w_{ij}]$, augmented size is $(n+1)\times1$ and initialize counters and error function as:

$k \leftarrow 1, \quad p \leftarrow 1 \quad E \leftarrow 0.$

Step 3: The training cycle begins. Apply input and compute the output:

$X \leftarrow X_p, \quad d \leftarrow d_p, \quad o \leftarrow sgn(WX)$

Step 4: Update the weights: $W^T \leftarrow W^T + \eta (d - o) X$

Step 5: Compute the cycle error: $E \leftarrow \frac{1}{2}(d-o)^2 + E$

Step 6: If $p < P$, the $p \leftarrow p+1$, $k \leftarrow k+1$ and go to step 3, otherwise go to step 7.

Step 7: The training cycle is completed. For $E < E_{max}$ terminates the training session with output weights and k. If $E > E_{max}$, then $E \leftarrow 0$, $p \leftarrow 1$ and enter the new training cycle by going to step 3.

In general, a continuous perceptron element with sigmoidal activation function will be used to facilitate the training of multi layer feed forward networks used for classification and recognition.

8.3.0 Single-Layer Continuous Perceptron networks

In this, the concept of an error function in multidimensional weight space has been introduced. Also the hard limiter (*sgn(.)*) with weights will be replaced by the continuous perceptron. By introduction of this continuous activation function, there are two advantages (i) finer control over the training procedure and (ii) differential characteristics of the activation function, which is used for computation of the error gradient.

The gradient or steepest descent is used in updating weights starting from any arbitrary weight vector W, the gradient $\nabla E(W)$ of the current error function is computed. The next value of W as obtained by moving in the direction of the negative gradient along the multidimensional error surface. Therefore the relation of modification of weight vector may be written as

$$W^{(k+1)T} = W^{kT} - \eta \nabla E(W^k) \qquad (8.10)$$

where η is the learning constant and is the positive constant and the superscript k denotes the step number. Let us define the error function between the desired output d_k and actual output o_k as

$$E_k = \frac{1}{2}(d_k - o_k)^2 \qquad (8.11a)$$

or

$$E_k = \frac{1}{2}\left[d_k - f\left(W^k X\right)\right]^2 \qquad (8.11b)$$

where the coefficient ½ in from of the error expression is only for convenience in simplifying the expression of the gradient value and it does not effect the location of the error function minimization. The error minimization algorithm (8.10) requires computation of the gradient of the error function (8.11) and it may be written as

$$\nabla E(W^k) = \frac{1}{2}\nabla\left[d - f(net_k)\right]^2 \qquad (8.12)$$

The n+1 dimensional gradient vector is defined as

$$\nabla E(W^k) = \begin{bmatrix} \frac{\partial E}{\partial w_0} \\ \frac{\partial E}{\partial w_1} \\ . \\ . \\ \frac{\partial E}{\partial w_n} \\ \end{bmatrix} \qquad (8.13)$$

Using (8.12), we obtain the gradient vector as

$$\nabla E(W^k) = -(d_k - o_k) f'(net_k) \begin{bmatrix} \frac{\partial(net_k)}{\partial w_0} \\ \frac{\partial(net_k)}{\partial w_1} \\ . \\ . \\ \frac{\partial(net_k)}{\partial w_n} \end{bmatrix} \quad (8.14)$$

Since $net_k = W^k X$, we have

$$\frac{\partial(net_k)}{\partial w_i} = x_i, \qquad \text{for } i = 0, 1, \ldots n. \quad (8.15)$$

(x_0=1 for bias element) and equation (8.15) can be written as

$$\nabla E(W^k) = -(d_k - o_k) f'(net_k) X \quad (8.16a)$$

or

$$\frac{\partial E}{\partial w_i} = -(d_k - o_k) f'(net_k) x_i \qquad \text{for } i = 0, 1, \ldots n \quad (8.16b)$$

$$\therefore \Delta w_i^k = -\eta \nabla E(W^k) = \eta (d_k - o_k) f'(net_k) x_i \quad (8.17)$$

Equation (8.17) is the training rule for the continuous perceptron. Now the requirement is how to calculate $f'(net)$ in terms of continuous perceptron output. Consider the bipolar activation function f(net) of the form

$$f(net) = \frac{2}{1 + \exp(-net)} - 1 \quad (8.18)$$

Differentiating the equation (8.18) with respect to net: $f'(net) = \frac{2 \times \exp(-net)}{[1 + \exp(-net)]^2}$ (8.19)

The following identity can be used in finding the derivative of the function.

$$\frac{2 \times \exp(-net)}{[1 + \exp(-net)]^2} = \frac{1}{2}(1 - o^2) \quad (8.20)$$

The relation (8.20) may be verified as follows:

$$\frac{1}{2}(1 - o^2) = \frac{1}{2}\left[1 - \left(\frac{1 - \exp(-net)}{1 + \exp(-net)}\right)^2\right] \quad (8.21)$$

The right side of (8.21) can be rearranged as

$$\frac{1}{2}\left[1-\left(\frac{1-\exp(-net)}{1+\exp(-net)}\right)^2\right]=\frac{2\exp(-net)}{[1+\exp(-net)]^2} \qquad (8.22)$$

This is same as that of (8.20) and now the derivative may be written as

$$\therefore\ f'(net_k) = \frac{1}{2}(1-o_k^{\,2}) \qquad (8.23)$$

The gradient (8.16a) can be written as $\nabla E(W^k) = -\frac{1}{2}(d_k - o_k)(1 - o_k^{\,2})X$ (8.24)

and the complete delta training for the bipolar continuous activation function results from (8.24) as

$$W^{(k+1)T} = W^{kT} + \frac{1}{2}\eta(d_k - o_k)(1 - o_k^{\,2})X_k \qquad (8.25)$$

where k denotes the reinstated number of the training step.

The weight adjustment rule (8.25) corrects the weights in the same direction as the discrete perceptron learning rule as in equation (8.8). The main difference between these two is the presence of the moderating factor $(1-o_k^{\,2})$. This scaling factor is always positive and smaller than 1. Another main difference between the discrete and continuous perceptron training is that the discrete perceptron training algorithm always leads to a solution for linearly separable problems. In contrast to this property, the negative gradient-based training does not guarantee solutions for linearly separable patterns.

8.3.1 Summary of the Single Continuous Perceptron Training Algorithm

Given are P training pairs

$\{X_1, d_1, X_2, d_2, \ldots\ldots X_p, d_p\}$, where X_i is $((n+1)\times 1)$, d_i is (1×1), for i = 1, 2, . . . P

$$X_i = \begin{bmatrix} x_{i0} \\ x_{i1} \\ . \\ . \\ x_{in} \end{bmatrix}, \text{ where } x_{i0} = 1.0 \text{ (bias element)}$$

Let k is the training step and p is the step counter within the training cycle.

Step 1: $\eta > 0$ and $E_{max} > 0$ chosen.

Step 2: Weights are initialized at W at small random values, $W = [w_{ij}]$ is $(n+1)\times 1$. Counter and error function are initialized.

$k \leftarrow 1, \qquad p \leftarrow 1 \qquad E \leftarrow 0.$

Step 3: The training cycle begins. Input is presented and output is computed.

$$X \leftarrow X_p, \qquad d \leftarrow d_p, \qquad o \leftarrow f(WX)$$

Step 4: Weights are updated: $W^T \leftarrow W^T + \frac{1}{2}\eta (d - o)(1 - o^2)X$

Step 5: Cycle error is computed: $E \leftarrow \frac{1}{2}(d - o)^2 + E$

Step 6: If p < P, the p ← p+1, k ← k+1 and go to step 3, otherwise go to step 7.

Step 7: The training cycle is completed. For $E < E_{max}$ terminated the training session with output weights, k and E. If $E \geq E_{max}$, then $E \leftarrow 0$, $p \leftarrow 1$ and enter the new training cycle by going to step 3.

8.4.0 Perceptron Convergence Theorem

This theorem states that the perceptron learning law converges to a final set of weight values in a finite number of steps, if the classes are linear separable. The proof of this theorem is as follows:

Let X and W are the augmented input and weight vectors respectively. Assume that there exits a solution W^* for the classification problem, we have to show that W^* can be approached in a finite number of steps, starting from some initial weight values. We know that the solution W^* satisfies the following inequality as per the equation (8.5):

$W^*X > \alpha > 0$, for each $X \in C_1$ (8.26)

where $\alpha = \min_{X \in C_1}(W^{*T} X)$

The weight vector is updated if $W^{kT}X \leq 0$, for $X \in C_1$. That is,

$W^{k+1} = W^k + \eta X(k)$, for $X(k) = X \in C_1$ (8.27)

where X(k) is used to denote the input vector at step k. If we start with W(0)=0, where 0 is an all zero column vector, then

$$W^k = c\sum_{i=0}^{k-1} X(i) \tag{8.28}$$

Multiplying both sides of equation (8.28) by W^{*T}, we get

$$W^{*T}W^k = c\sum_{i=0}^{k-1} W^{*T}X(i) > ck\alpha \tag{8.29}$$

since $W^{*T}X(k) > \alpha$ according to equation (8.26). Using the Cauchy-Schwartz inequality

$$\left\|W^{*T}\right\|^2 . \left\|W^k\right\|^2 \geq [W^{*T}W^k]^2 \tag{8.30}$$

We get from equation (8.29)

$$\|W^k\|^2 > \frac{c^2 k^2 \alpha^2}{\|W^{*T}\|^2} \tag{8.31}$$

We also have from equation (8.27)

$$\begin{aligned} \|W^{k+1}\|^2 &= (W^k + cX(k))^T (W^k + cX(k)) \\ &= \|W^k\|^2 + c^2 \|X(k)\|^2 + 2cW^{kT} X(k) \\ &\leq \|W^k\|^2 + c^2 \|X(k)\|^2 \end{aligned} \tag{8.32}$$

since for learning $W^{kT} X(k) \leq 0$ when $X(k) \in C_1$. Therefore, starting from $W^0=0$, we get from equation (8.32)

$$\|W^k\|^2 = c^2 \sum_{i=0}^{k-1} \|X(i)\|^2 < c^2 k\beta \tag{8.33}$$

where $\beta = \max_{X(i) \in C_1} \|X(i)\|^2$. Combining equations (8.31) and (8.33), we obtain the optimum value of k by solving $\frac{k^2 \alpha^2}{\|W^{*T}\|^2} = \beta k$ (8.34) or

$$k = \frac{\beta}{\alpha^2} \|W^{*T}\|^2 = \frac{\beta}{\alpha^2} \|W^*\|^2 \tag{8.35}$$

Since β is positive, equation (8.35) shows that the optimum weight value can be approached in a finite number of steps using the perceptron learning law.

8.5.0 Problems and Limitations of the perceptron training algorithms

It may be difficult to determine if the caveat regarding linear separability is satisfied for the particular training set at hand. Further more, in many real world situations the inputs are often time varying and may be separable at one time and not at another. Also, these is no statement in the proof of the perceptron learning algorithm that indicates how many steps will be required to train the network. It is small consolation; to know that training will only take a finite number of steps if the time it takes is measured in geological units.

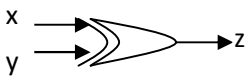
Further more, there is no proof that perceptron training algorithm is faster than simply trying all possible adjustment of the weights; in some cases this brute force approach may be superior.

8.5.1 Limitations of perceptrons

There are limitations to he capabilities of perceptrons however. They will learn the solution, if there is a solution to be found. First, the output values of a perceptron can take on only one of two values (True or False). Second, perceptrons can only classify *linearly separable* sets of vectors. If a straight line or plane can be drawn to separate the input vectors into their correct categories, the input vectors are linearly separable and the

perceptron will find the solution. If the vectors are not linearly separable learning will never reach a point where all vectors are classified properly. The most famous example of the perceptron's inability to solve problems with linearly non-separable vectors is the boolean exclusive-OR problem.

Consider the case of the exclusive-or (XOR) problem. The XOR logic function has two inputs and one output, how below.



(a) Exclusive –OR gate

x	y	Z
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 8.4 (b) Truth Table

It produces an output only if either one or the other of the inputs is on, but not if both are **off** or both are **on.** It is shown in above table. We can consider this has a problem that we want the perceptron to learn to solve; output a 1 of the x is **on** and y is **off** or y is **on** and x is **off**, otherwise output a '0'. It appears to be a simple enough problem.

We can draw it in pattern space as shown in Fig. (8.5). The x-axis represents the value of x, the y-axis represents the value of y. The shaded circles represent the inputs that produce an output of 1, whilst the un-shaded circles show the inputs that produce an output of 0. Considering the shaded circles and un-shaded circles as separate classes, we find that, we cannot draw a straight line to separate the two classes. Such patterns are known as linearly inseparable since no straight line can divide them up successfully. Since we cannot divide them with a single straight line, the perceptron will not be able to find any such line either, and so cannot solve such a problem. In fact, a single-layer perceptron cannot solve any problem that is linearly inseparable.

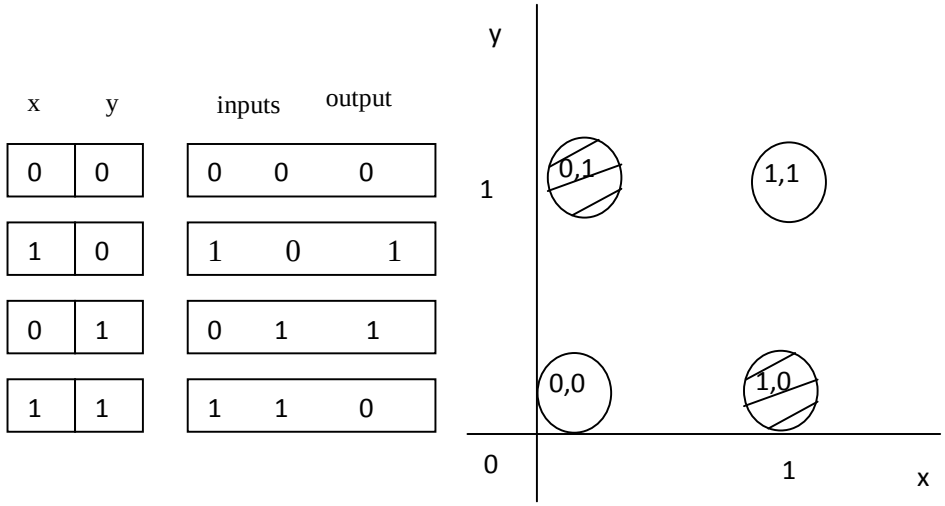


Fig. 8.5 The XOR problem in pattern space

Feed Forward Multi Layer Artificial Neural Networks (FFANN)

9.0.0 Introduction

A multiplayer neural network is a feed forward neural network with at least one hidden layer, as shown in Fig.9.1. It can deal with nonlinear classification problems because it forms more complex decision regions (rather than just hyper planes). Each node in the first layer can create a hyper plane. Each node in the second layer can combine hyper plane to create convex decision regions. Each mode in the third layer can combine convex regions to form concave regions. The different structures of feed forward artificial neural networks are discussed in unit-4. If reader is not familiar with the structure of multi layer neural network, suggested to go through the unit 4. In this we will discuss the training of multi layer neural networks.

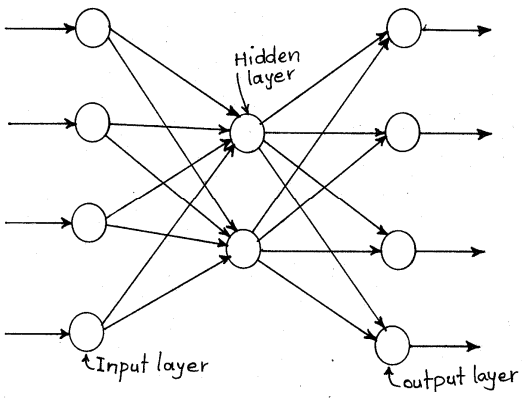


Fig.9.1 Multi-layer Perceptron

The learning rule used for multi-layer neural network is the "generalized delta rule" or the back propagation rule popularly known as backpropagation (BP) learning algorithm. In this we will discuss the derivation of backpropagation, applications and its limitations.

The feed forward multi layer ANN's can be constructed by cascading the single layer networks. The connections are fully interconnected in the forward direction. Thus, there are no feedback connections and no connections that bypass one layer to go directly to a layer. The multi layer network consists of input layer nodes, one or more hidden layers nodes and output layer nodes. The training mechanism based on error–correction-learning rule. By using this rule, the error terms are defined as function of error function at output layer. These error terms are used to adjust the synaptic weights between output layer and its preceding layer. The error terms for hidden layers are obtained as function of error terms and weights of its succeeding layers. The process of determining error terms in multi layer is nothing but, propagating error from output layer to its preceding layers until it reaches to input layer. Because of this process it is known as back propagation (BP) algorithm.

The backpropagation algorithm consists of two passes through the different layers of network: a forward pass and a backward pass. In the forward pass, an input vector is applied to the input nodes of the network and its effect propagates through the network layer by layer. Finally, a set of outputs is produced as response of the networks. In the forward pass the synaptic weights remain fixed. In the backward pass, first the error terms are determined at output layer and these terms are used in the process of determining the error terms in its preceding layers. Then the synaptic weights are modified in accordance with an error – correction-training rule.

Backpropagation Algorithm

In this section, we derive the algorithm of error-backpropagation for multi layer neural networks based generalized delta-learning rule (GDR). Consider a typical three-layer structure of multi layer neural network as shown in Fig 9.2 for derivation of this algorithm.Let us consider a set of P vector pairs, (X_1, Y_1), (X_2, Y_2), . . . (X_P, Y_P), which are examples of a functional mapping $Y = \phi(X)$; $x \in R^n$, $y \in R^m$. X_i, i=1, 2, 3, . . .P are the input vectors and Y_i, i=1, 2, 3, . . P are the corresponding desires output vectors. Objective is to train the network so that it will learn an approximation $O = Y = \phi(X)$. Remember that learning in a neural network means finding an appropriate set of final weights. In the process of training of the neural network, the signal flows in two

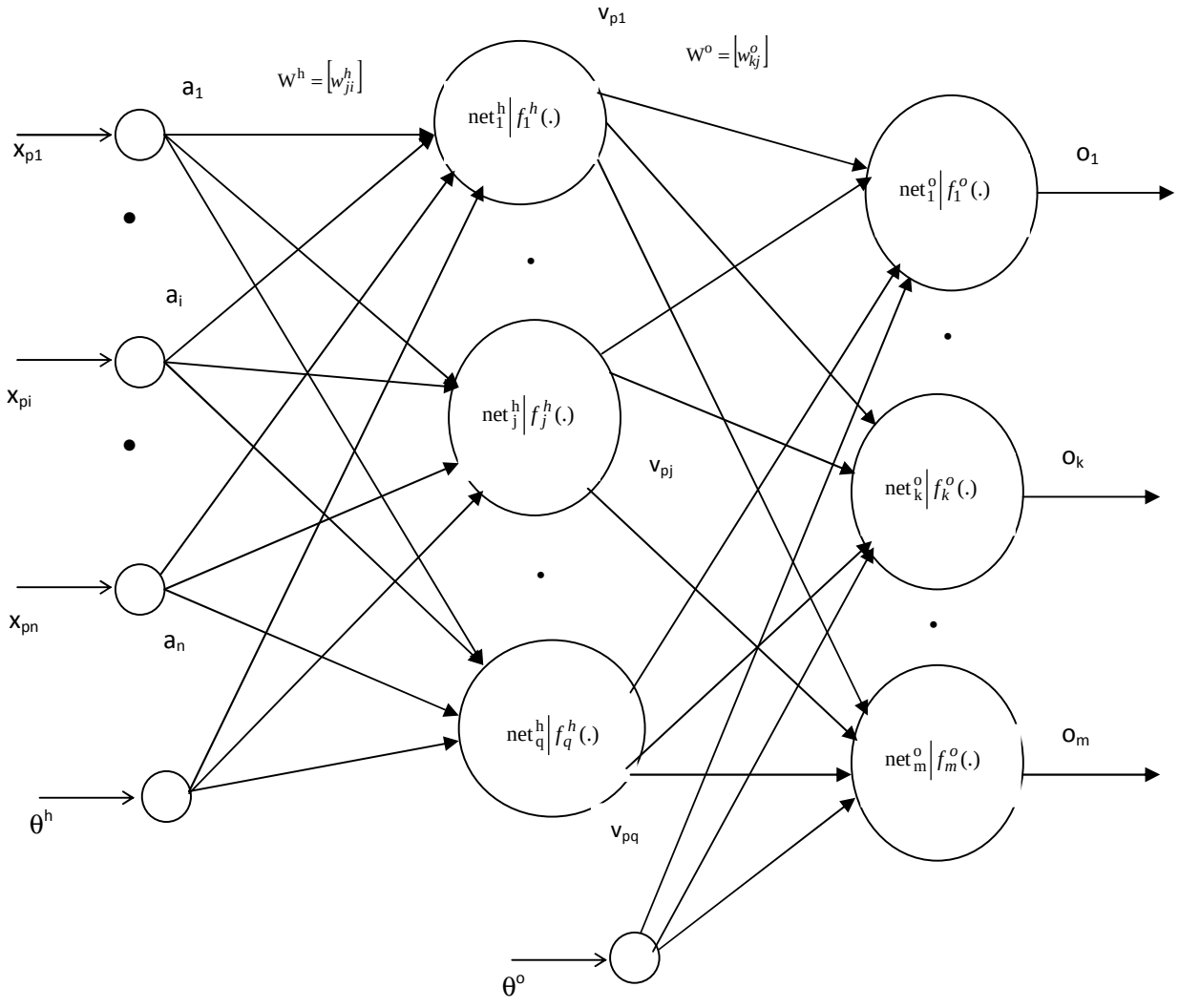


Fig. 9.2 A Three layer Feed Forward Neural Network architecture with the bias weights, θ_j^h and θ_k^θ and bias units are optional.

directions and they are forward pass and backward pass. In the forward pass, the outputs of the network are computed for the presented input vector and in backward pass the error terms are computed for input – output pair with existing weights. By using these error terms, the weights are updated and it completes one iteration process of training.

9.0.1 **Forward Pass**

An input vector, $X_p = (x_{p1}, x_{p2}, \ldots x_{PN})^T$, is applied to the input layer of the network, Fig. 9.2. The input units distribute the values to the hidden layer units. The net input to the j^{th} hidden unit is

$$\text{net}_{pj}^{h} = \sum_{i=1}^{n} w_{ji}^{h} x_{pi} + \theta_j^h \qquad (9.1)$$

where w_{ji}^h is the weight on the connection from the i^{th} input unit to j^{th} unit in hidden layer and θ_j^h is the bias term for j^{th} unit in hidden layer. Then the output of this node is

$$v_{pj} = f_j^h\left(\text{net}_{pj}^{h}\right) \qquad (9.2)$$

Similarly, the relations for output layer nodes are

$$\text{net}_{pk}^{O} = \sum_{j=1}^{q} w_{kj}^{O} v_{pj} + \theta_k^O \qquad (9.3)$$

$$o_{pk} = f_k^O\left(\text{net}_{pk}^{O}\right) \qquad (9.4)$$

where the "o" superscript refer to quantities on the output layer.

9.0.2 **Backward pass**

In this error terms are computed at different layers of the network. First error terms are computed for output layer, last hidden layer, so on and finally for the first hidden layer. That is, the computation process takes place in backward direction and is know as backward pass. In this we have considered only one hidden layer, therefore it consists only two levels of error terms, that is, at output layer and at hidden layer.

9.1.0 **Updating the Weight Elements of Network**

In the previous section we have discussed the signal processing in both directions. In this we derive the relation used for updating the weight elements of the multi-layer feed-forward neural network.

9.2.1 **Update of output layer weights**

Let us define the error at a single output unit to be $e_{pk} = y_{pk} - o_{pk}$, where the subscript "p" refers to the p^{th} training vector, and "k" refers to the k^{th} output node. y_{pk} is the desired output value and o_{pk} is the actual output from the k^{th} node. The error function is minimized by the method steepest descent method. The Generalized Delta Rule (GDR) is the sum of the squares of the error for all output nodes is obtained as

$$E_P = \frac{1}{2}\sum_{k=1}^{m} e_{pk}^2 \tag{9.5}$$

The factor of $\frac{1}{2}$ in equation (9.5) is only for mathematical convenience in calculating derivatives. To determine the direction in which to change the weights, we calculate the negative of the gradient of E_P, ∇E_P with respect to the weights, w_{kj}. Then, we can adjust the values of the weights such that the total error is reduced.

From equation (9.5) and the definition of e_{pk}, the energy function can be written as

$$E_P = \frac{1}{2}\sum_{k} \left(y_{Pk} - o_{Pk}\right)^2 \tag{9.6}$$

and
$$\frac{\partial E_P}{\partial w_{kj}^O} = -\left(y_{pk} - o_{pk}\right)\frac{\partial f_k^{\,0}(.)}{\partial\left(net_{pk}^0\right)}\frac{\partial\left(net_{pk}^0\right)}{\partial w_{kj}^0} \tag{9.7}$$

where, we have used equation (9.4) for obtaining the output value o_{pk}, and the chain rule for partial derivations. For the moment, we shall not try to evaluate the derivative of $f_k^{\,0}$, but instead will write it simply as $f^{o'}\left(net_{pk}^{o}\right)$.

$$\therefore \frac{\partial\left(net_{pk}^{o}\right)}{\partial w_{kj}^{o}} = \left(\frac{\partial}{\partial w_{kj}^{0}}\sum_{j=1}^{q} w_{kj}^{o} v_{pj} + \theta_k^o\right) = v_{pj} \tag{9.8}$$

Combining equations (9.7) and (9.8), the negative gradient can be written as

$$-\frac{\partial E_p}{\partial w_{kj}^{o}} = \left(y_{Pk} - o_{Pk}\right) f_k^{o^1}\left(net_{pk}^{o}\right) v_{pj} \tag{9.9}$$

By using steepest descent method, the change in weight is proportional to the negative gradient of error function. Therefore, the change in weights on the output layer can be written as

$$\Delta_P w_{kj}^0 = \eta\left(y_{pk} - o_{pk}\right) f_k^{\,0^1}\left(net_{pk}^0\right) v_{pj} \tag{9.10}$$

Now the new weights of the output layer may be written as

$$w_{kj}^0\left(t+1\right) = w_{kj}^0\left(t\right) + \Delta_p w_{kj}^0\left(t\right) \tag{9.11}$$

The factor η is called the learning rate parameter. Here the requirement is that the function $f_k^{\,0}(.)$ must be differentiable. There are three forms of the output function that are of interest here;

1. $f_k^{\,0}\left(net_{pk}^0\right) = net_{pk}^0$

2. $f_k^0\left(net_{pk}^0\right) = \left(1 + e^{-net_{pk}^0}\right)^{-1}$

3. $f_k^0\left(net_{pk}^0\right) = \tanh\left(net_{pk}^0\right)$

The first function defines the for linear output nodes, the second function is called a unipolar sigmoid and the third function is known as bipolar sigmoid function. The choice of output function depends on how you choose to represent the output data.

In first case, $f_k^{0^1} = 1$;

In the second case, we have $f_k^{0^1} = f_k^0\left(1 - f_k^0\right) = o_{pk}\left(1 - o_{pk}\right)$

In the third case, $f_k^{0^1} = \left(1 - (f_k^0)^2\right) = (1 + f_k^0)\left(1 - f_k^0\right) = (1 + o_{pk})\left(1 - o_{pk}\right)$

Depending upon the activation function selected, the weight update equation may be written as

For case 1: $w_{kj}^0(t+1) = w_{kj}^0(t) + \eta\left(y_{pk} - o_{pk}\right)v_{pj}$ for the linear output nodes, (9.12)

For case 2: $w_{kj}^0(t+1) = w_{kj}^0(t) + \eta\left(y_{pk} - o_{pk}\right)o_{pk}\left(1 - o_{pk}\right)v_{pj}$

for the unipolar sigmoidal output (9.13)

For case 3: $w_{kj}^0(t+1) = w_{kj}^0(t) + \eta\left(y_{pk} - o_{pk}\right)(1 + o_{pk})\left(1 - o_{pk}\right)v_{pj}$

for the bipolar sigmoidal output (9.14)

Let us summarize the weight update equations by defining a quantity

$$\delta_{pk}^0 = \left(y_{pk} - o_{pk}\right)f_k^{0^1}\left(\text{net}_{\text{pk}}^0\right) = e_{pk}\, f_k^{0^1}\left(\text{net}_{\text{pk}}^0\right) \quad (9.15)$$

We can write the weight update equation as

$$w_{kj}^0(t+1) = w_{kj}^0(t) + \eta\delta_{pk}^0 v_{pj} \quad (9.16)$$

where δ_{pk}^0, is known as error terms on the output layer regardless of the activation function at output layer, $f_k^o(.)$.

We wish to make a comment regarding the relationship between the gradient descent method described here and the least squares technique. If we were trying to make the generalized delta rule entirely analogous to a least-squares method, we would not actually change any of the weight values until all of the training patterns had been presented to the network once. We would simply accumulate the changes as each pattern was processed, sum them, and make update to the weights. We would then repeat the process until the error was acceptably low. The error that this process minimizes as

$$E = \sum_{p=1}^{P} E_p \qquad (9.17)$$

where P is the number of patterns in training set. In practice, we have found little advantage to this strict adherence to analogy with the least squares method. Moreover, you must store a large amount of information to use this method. We recommend that you perform weight updates as each training pattern is processed.

9.2.2 **Update of Hidden-Layer weights**

Here, we would like to repeat for the hidden layer the same type of calculation as we did for the output layer. A problem arises when we try to determine a measure of the error of the outputs of the hidden-layer nodes. We know what the actual output is, but we have no way of knowing in advance what the correct output should be for these nodes. Intuitively, the total error, E_P, must some how be related to the output values on the hidden layer.

$$E_p = \frac{1}{2}\sum_k (y_{Pk} - o_{Pk})^2 = \frac{1}{2}\sum_k \left(y_{Pk} - f_k^0\left(net_{Pk}^0\right)\right)^2 = \frac{1}{2}\sum_k \left[y_{Pk} - f_k^0\left(\sum_j w_{kj}^0 v_{pj} + \theta_k^o\right)\right]^2 \qquad (9.18)$$

We know that v_{pj} depends on the weights on the hidden layer. We can exploit this fact to calculate the gradient of E_P with respect to the hidden layer weights.

$$\frac{\partial E_p}{\partial w_{ji}^h} = \frac{1}{2}\sum_K \frac{\partial}{\partial w_{ji}^h}\left(y_{Pk} - o_{Pk}\right)^2$$

$$= -\sum_k \left(y_{Pk} - o_{Pk}\right)\frac{\partial o_{Pk}}{\partial\left(net_{Pk}^0\right)}\frac{\partial\left(net_{Pk}^O\right)}{\partial v_{pj}}\cdot\frac{\partial v_{pj}}{\partial\left(net_{Pj}^h\right)}\frac{\partial\left(net_{Pk}^h\right)}{\partial w_{ji}^h}$$

$$= -\sum_k \left(y_{Pk} - o_{Pk}\right)\frac{\partial o_{Pk}}{\partial\left(net_{Pk}^0\right)}\frac{\partial\left(net_{Pk}^0\right)}{\partial v_{pj}}\cdot\frac{\partial v_{pj}}{\partial\left(net_{pj}^h\right)}\frac{\partial\left(net_{Pj}^h\right)}{\partial w_{ji}^h} \qquad (9.19)$$

The partial derivative factors in the above equation (9.19) can be calculated from previous relations. Now the gradient of E_p with respect to w_{ji}^h may be written as

$$\frac{\partial E_P}{\partial w_{ji}^h} = -\sum_k \left(y_{pk} - o_{pk}\right) f_k^{0'}\left(net_{pk}^0\right) w_{kj}^0 f_j^{h'}\left(net_{pj}^h\right)\ x_{pi} \qquad (9.20)$$

As we know that, the adjustments in weights are proportional to the negative gradient of error function. Now, the updating equations for the hidden-layer weights can be written as

$$\Delta_p w_{ji}^h = \eta f_j^{h'}\left(net_{pj}^h\right) x_{pi} \sum_k \left(y_{pk} - o_{pk}\right) f_k^{0'}\left(net_{pk}^0\right) w_{kj}^0 \qquad (9.21)$$

where η is the again the learning rate. Equation (9.21) may be written in terms of error terms of output layer, δ_{pk}^{0},

$$\Delta_p w_{ji}^{h} = \eta f_j^{h^1}\left(net_{pj}^{h}\right) x_{pi} \sum_k \delta_{pk}^{o} w_{kj}^{0} \tag{9.22}$$

Notice that, every weight update on the hidden layer depends on all the error terms, δ_{pk}^{0}, on the output layer. This result is where the notion of back propagation arises. The known errors on the output layer are propagated back to the hidden layer to determine the appropriate weight changes on that layer.

The hidden-layer error terms may be defined as the function of error terms on the output layer and the weights connected to the output layer are

$$\delta_{pj}^{h} = f_j^{h^1}\left(net_{pj}^{h}\right) \sum_k \delta_{pk}^{0} w_{kj}^{0} \tag{9.23}$$

we can use the weight update equations to become analogous to those for the output layer:

$$w_{ji}^{h}(t+1) = w_{ji}^{h}(t) + \eta \delta_{pj}^{h} x_i \tag{9.24}$$

9.3 Summary of backpropagation algorithm

1. Construct the network to match with the given number of inputs and outputs to the nodes at input and output layers respectively. Initialize the weights with small random values.
2. Apply the input vector, $X_p = (x_{p1}, x_{p2} \ldots\ldots x_{pn})^T$ to the input layer units.
3. Calculate the net input values for the hidden layer units: $\text{net}_{\text{pj}}^{\text{h}} = \sum_{i=1}^{n} w_{ji}^{h} x_{pi} + \theta_j^{h}$
4. Calculate the outputs of the nodes in the hidden-layer: $v_{pj} = f_j^{h}\left(\text{net}_{\text{pj}}^{\text{h}}\right)$
5. Calculate the net input values for the output layer units: $\text{net}_{\text{pk}}^{\text{o}} = \sum_{j=1}^{q} w_{kj}^{o} i_{pj} + \theta_k^{o}$
6. Calculate the outputs of the nodes in the output layer: $o_{pk} = f_k^{0}\left(\text{net}_{\text{pk}}^{0}\right)$
7. Calculate the error terms for the output units: $\delta_{pk}^{0} = \left(y_{pk} - o_{pk}\right) f_k^{0^1}\left(\text{net}_{\text{pk}}^{0}\right)$
8. Calculate the error terms for the hidden units: $\delta_{pj}^{h} = f_j^{h'}\left(\text{net}_{\text{pj}}^{\text{h}}\right) \sum_k \delta_{pk}^{0} w_{kj}^{0}$

Notice that the error terms on the hidden units are calculated before the connection weights to the output layer units have been updated.

9. Update weights on the output layer: $w_{kj}^{0}(t+1) = w_{kj}^{0}(t) + \eta \delta_{pk}^{0} v_{pj}$
10. Update weights on the hidden layer: $w_{ji}^{h}(t+1) = w_i^{h}(t) + \eta \delta_{pj}^{h} x_i$

The order of the weight updates on an individual layer is not important. Be sure to calculate the error function: $E_p = \frac{1}{2} \sum_{k=1}^{m} e_{Pk}^{2}$

Since this quantity of the measure of how well the network is learning when the error is acceptably small for each of the training vector pairs, training can be disconnected.

9.4 Learning difficulties and improvements

Occasionally the network settles into a stable solution that does not provide the correct output. In these cases, the energy function is in a local minimum. This means that in every direction in which the network could move in the energy landscape, the energy is higher than at the current position. It may be that there is only a slight "lip" to cross before reaching an actual deeper minimum, but the network has no way of knowing this, since learning is accomplished by following the energy function down in the steepest direction, until it reaches the bottom of well, at which point there is no direction to move in order to reduce the energy. There are alternative approaches to minimizing these occurrences.

9.4.1 Momentum term

The weight changes can be given some "momentum" by introducing an extra term into the weight adaptation equation that will produce a large change in the weight if the changes are currently large, and will decrease as the changes become less. This means that the network is less likely to get stuck in local minima early on, since the momentum term will push the changes over they local increases in the energy function, following the overall downward trend. Momentum is of great assistance in speeding up convergence along shallow gradients, allowing the path the network takes towards the solution to pickup speed in the down hill direction. The energy landscape may consist of long gradually sloping ravines, which finish at minima. Convergence along these ravines is slow, since the direction that has to be followed has only a slight gradient, and usually the algorithm oscillate across the ravine valley as it meander towards a solution.

This momentum term can be written as (e.g, for hidden layer weights) follows:

$$w_{ji}^{h}(t+1) = w_{ji}^{h}(t) + \eta \delta_{pj}^{h} x_{pi} + \alpha\left[w_{ji}^{h}(t) - w_{ji}^{h}(t-1)\right] \qquad (9.25)$$

where α is the momentum factor $0<\alpha<1$.

9.4.2 Lowering the gain term

If the rate at which the weights are altered is progressively decreased, then the gradient descent algorithm is able to achieve a better solution. If the gain term η is made large to begin with, large steps are taken across the weight and energy space towards the solution. As the gain is decreased, the network weights settle into a minimum energy configuration without overshooting the stable position, as the gradient descent takes smaller downhill steps. This approach enables the network to bypass local minimum at first, then hopefully locate, and settle in, some deeper minima without oscillating wildly. However, the reduction in the gain term will mean that the network will take longer to coverage.

9.4.3 Addition of internal nodes

Local minima can be considered to occur when two or more disjoint classes are categorized as the same. This amounts to a poor central representation within the hidden

units, and so adding more units to this layer will allow a better recoding of the inputs and lesser the occurrence of these minima.

9.4.4 Addition of noise

If random noise is added, this perturbs the gradient descent algorithm from the time of steepest descent, and often this noise is enough to know the system out of a local minimum. This approach has the advantage that it takes very little extra computational time, and so is not noticeably slower than the direct gradient descent algorithm.

9.5 Simulation of backpropagation algorithm

Example ([5]): To illustrate the basic concepts of backpropagation learning, let us consider a three – layer feed forward neural network (structure same as in Fig 9.2) for fault diagnosis of process data from a chemical reactor. Table I lists the input and output vectors for the network. The desired output, d_k , from the neural network is Boolean: 0 indicates that no operational fault exists, and 1 indicates that a fault does exists. The actual output from the neural network is a numeric value between 0 and 1, and can be viewed almost as the " probability" that a given input will produce the operational fault (0 = the fault definitely will not occur; 1 = the fault definitely will occur).

Table I: Input and output for fault-diagnosis network

Input vector	Output vector
I_1: reactor inlet temperature, ^oF	o_1: low conversion
I_2: reactor inlet pressure, psia	o_2: low catalyst selectivity
I_3: feed flow rate, lb/min	o_3: catalyst sintering

Given the above inputs and outputs, we will now illustrate how to use backpropagation to train the network to recognize the following specific conditions:

Input: $I_1 = 300/1000\ ^oF = 0.3$ *Desired Output*: $d_1 = 1$ (low conversion)

$I_2 = 100/1000$ psia $= 0.1$ $d_2 = 0$ (no problem)

$I_3 = 200/1000$ lb/min $= 0.2$ $d_3 = 0$ (no problem)

Note that we have divided all the input values by 1000. In neural network training, we recommend normalizing the input and output values to a finite range, such as [0,1] or [-1,1]. Also consider the nodes in input layer also does the processing (i.e., they are not dummy nodes).

Step 1: Randomly assign values between -1 and 1 to weights w_{ji}^h and w_{kj}^o. Likewise, assign internal threshold values (θ_i^i, θ_j^h and θ_k^o) for every node, also between -1 and +1. For the delta rule, the internal threshold values must be assigned as follows: all input-layer thresholds must equal zero, i.e. $\theta_i^i = 0$ (superscription i indicates input and subscription i indicates input node number); all hidden and output-layer thresholds must equal one, i.e. $\theta_j^h = \theta_k^o = 1$. Remember that i = 1, 2,. . . , n, where n is the number of nodes in input layer ; j = 1, 2, . . . , q, where q is the number of nodes in hidden layer ; and k =

1, 2, . . . , m, where m is the number of nodes in output layer. The assigned values are as follows:

$$W^{hT} = [w^h_{ji}] = \begin{bmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & -0.5 \\ 0.5 & -0.5 & 0.5 \end{bmatrix} \quad \text{and} \quad W^h = [w^h_{ij}] = \begin{bmatrix} -1.0 & 1.0 & 0.5 \\ 0.5 & 0.0 & -0.5 \\ 0.5 & -0.5 & 0.5 \end{bmatrix}$$

$$W^{oT} = [w^o_{jk}] = \begin{bmatrix} -1.0 & -0.5 & 0.5 \\ 1.0 & 0.0 & 0.5 \\ 0.5 & -0.5 & 0.5 \end{bmatrix} \text{ and } W^o = [w^o_{kj}] = \begin{bmatrix} -1.0 & 1.0 & 0.5 \\ -0.5 & 0.0 & -0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix}, \ [\theta] = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$$

This θ-matrix serves as a bias elements for neurons in the network.

Step 2: Introduce the input I_i into the neural network, and calculate the output from the first layer according to the equations:

$$x_i = I_i + \theta^i_i = I_i + 0 = I_i \qquad (9.26)$$

$$a_i = f(x_i) = \frac{1}{1+e^{-x_i}} \qquad (9.27)$$

Once again, f(.) here is an unipolar sigmoid function. We introduce the input vector into the neural network and calculate the outputs from layer 1:

$$x_1 = I_1 + \theta^i_1 = 0.3 + 0 = 0.3$$

$$x_2 = I_2 + \theta^i_2 = 0.1 + 0 = 0.1$$

$$x_3 = I_3 + \theta^i_3 = 0.2 + 0 = 0.2$$

Substituting these values into the sigmoid function, we get:

$$a_1 = f(x_1) = \frac{1}{1+e^{-x_1}} = 0.57444$$

$$a_2 = f(x_2) = \frac{1}{1+e^{-x_2}} = 0.52498$$

$$a_3 = f(x_3) = \frac{1}{1+e^{-x_3}} = 0.54983$$

Step 3: Knowing the output from the first layer, calculate outputs from the second layer, using the equation: $v_j = f\left(\sum_{i=1}^{n} (w^h_{ji}\, a_i) + \theta^h_j\right)$ (9.28)

where f(.) is an unipolar sigmoid function. Note that $\theta_i^h = 1$ in this algorithm, and we are gain adding it to the weighted inputs. When used in this mode, θ_j^h acts as a bias function rather that an internal threshold. We calculate the output from each node in layer two:

$$v_1 = f\left(w_{11}^h a_1 + w_{12}^h a_2 + w_{13}^h a_3 + \theta_1^h\right) = f(1.22546) = 0.77302$$

$$v_2 = f\left(w_{21}^h a_1 + w_{22}^h a_2 + w_{23}^h a_3 + \theta_2^h\right) = f(1.012305) = 0.733471$$

$$v_3 = f\left(w_{31}^h a_1 + w_{32}^h a_2 + w_{33}^h a_3 + \theta_1^i\right) = f(1.29965) = 0.78578$$

Step 4: Knowing the output from the second layer, calculate the result from the output layer, according to the equation:

$$o_k = f\left(\sum_{j=1}^{q}\left(w_{kj}^o v_j\right) + \theta_k^o\right) \qquad (9.29)$$

where f(.) is the unipolar sigmoid function. Note again, $\theta_k^o = 1.0$, and θ_k^o acts as a bias function added to the weighted input. We calculate the output from each node in layer three:

$$o_1 = f\left(w_{11}^o v_1 + w_{12}^o v_2 + w_{13}^o v_3 + \theta_1^o\right) = f(1.353341) = 0.794775$$

$$o_2 = f\left(w_{21}^o v_1 + w_{22}^o v_2 + w_{23}^o v_3 + \theta_2^o\right) = f(0.22060) = 0.55493$$

$$o_3 = f\left(w_{31}^o v_1 + w_{32}^o v_2 + w_{33}^o v_3 + \theta_3^o\right) = f(2.146136) = 0.89531$$

Step 5: Continue steps 1-4 for P number of training patterns presented to the input layer. Calculated the mean-squares error, E, according to the following equation:

$$E = \sum_{p=1}^{p}\sum_{k=1}^{m}\left(d_{pk} - o_{pk}\right)^2 \qquad (9.30)$$

where P is the number of training patterns presented to the input layer, m is the number of nodes on the output layer, d_{pk} is the desired output value from the k^{th} node in the p^{th} training pattern, and o_{pk} is the actual output value from the k^{th} node in the p^{th} training pattern. We are training the network with just one input pattern (P = 1). With desired output values $d_1 = 1$, $d_2 = 0$, and $d_3 = 0$, our total mean squares error is:

$$E = \sum_{k=1}^{3}\left(d_k - o_k\right)^2 = (d_1 - o_1)^2 + (d_2 - o_2)^2 + (d_3 - o_3)^2 = 1.15168$$

Step 6: Knowing the p^{th} pattern, calculate δ^o_{pk}, the gradient-descent term (error-term) for the k^{th} node in the output layer (layer 3) for training pattern p. Use the following equation:

$$\delta^o_{pk} = (d_{pk} - o_{pk})\,\frac{\partial f(net_k^o)}{\partial(net_k^o)} \qquad (9.31)$$

where f(.) is the sigmoid function: $f(\text{net}) = \frac{1}{1+e^{-\text{net}}}$

and its partial derivative is : $\frac{\partial f(.)}{\partial(net_k^o)} = \frac{e^{-net_k^o}}{\left(1+e^{-net_k^o}\right)^2}$

Note that net_k^o here is the sum of the weighted inputs to the k[th] node on the output layer plus the bias function (i.e., for the p[th] training session): $net_{pk}^o = \sum_j w_{kj}^o v_{pj} + \theta_k^o$ (9.32)

where for training pattern p, v_{pj} is the output value of the j[th] node in the hidden layer and θ_k^o is the threshold value on the output layer. Now, we calculate δ_{pk}^o, the gradient-descent term (error terms) for output layer (layer 3).

$\delta_{p1}^o = (d_1-o_1)o_1(1.0-o_1) = (1-0.794675)\times 0.794675\times (1.0-0.794675) = 0.033502$

$\delta_{p2}^o = (d_2-o_2)o_2(1.0-o_2) = (0.0-0.55493)\times 0.55493\times (1.0-0.55493) = -0.1370581$

$\delta_{p3}^o = (d_3-o_3)o_3(1.0-o_3) = (0.0-0.89531)\times 0.89531 \times (1.0-0.89531) = -0.08392$

Step 7: Again knowing the p[th] pattern, calculate δ_{pj}^h, the gradient-descent term (error term for hidden layer) for the j[th] node on the hidden layer (layer 2). Use the equation:

$$\delta_{pj}^h = \left(\sum_k \delta_{pk}^o w_{kj}^o\right)\frac{\partial f(.)}{\partial(net_j^h)} \qquad (9.33)$$

where the subscript k denotes a node in the output layer, recall the net input value net_{pj}^h is defined by: $\text{net}_{pj}^h = \sum_{i=1}^{n} w_{ji}^h x_{pi} + \theta_j^h$

and the partial derivative of the sigmoid function, again, is:

$$\frac{\partial f(.)}{\partial(net_{pj}^h)} = \frac{e^{-\text{net}_{pj}^h}}{\left(1+e^{-\text{net}_{pj}^h}\right)^2} = f(\text{net}_{pj}^h)(1-f(\text{net}_{pj}^h)) = v_{pj}(1-v_{pj}) \qquad (9.34)$$

We calculate δ_{pj}^h, the gradient-descent term (error terms) for layer 2. Then, the δ_j^h's are:

$$\delta_1^h = (\delta_1^o w_{11}^o + \delta_2^o w_{21}^o + \delta_3^o w_{31}^o)\frac{\partial f(.)}{\partial(net_1^h)} = (\delta_1^o w_{11}^o + \delta_2^o w_{21}^o + \delta_3^o w_{31}^o)v_1(1-v_1)$$

$=\{0.033502\times(-1.0)+(-0.1370581)\times(-0.5)+(-0.08392)\times 0.5\}\times(0.77302)\times(1.0-0.77302)=-0.001216.$

$$\delta_2^h = (\delta_1^o w_{12}^o + \delta_2^o w_{22}^o + \delta_3^o w_{32}^o)\frac{\partial f(.)}{\partial(net_2^h)} = (\delta_1^o w_{12}^o + \delta_2^o w_{22}^o + \delta_3^o w_{32}^o)v_2(1-v_2)$$

$=\{0.033502\times 1.0+(-0.1370581)\times(0.0)+(-0.08392)\times 0.5\}\times(0.733471)\times(1.0-0.733471)= -0.0016534.$

$$\delta_3^h = (\delta_1^o\, w_{13}^o + \delta_2^o\, w_{23}^o + \delta_3^o\, w_{33}^o)\frac{\partial f(.)}{\partial(net_3^h)} = (\delta_1^o\, w_{13}^o + \delta_2^o\, w_{23}^o + \delta_3^o\, w_{33}^o)v_3(1\text{-}v_3)$$

$$=\{0.033502\times(0.5)+(-0.1370581)\times(-0.5)+(-0.08392)\times 0.5\}\times(0.78578)\times(1.0-0.78578)=0.007292.$$

Step 8: Knowing δ_{pj}^h for the hidden layer and δ_{pk}^o for the output layer, calculate the weight changes using the equations:

$$\Delta_p w_{ji}^h(l) = \eta\, \delta_{pj}^h a_{pi} + \alpha\, \Delta_p w_{ji}^h(l-1) \qquad (9.35)$$

$$\Delta_p w_{kj}^o(l) = \eta\, \delta_{kj}^o v_{pj} + \alpha\, \Delta_p w_{kj}^o(l-1) \qquad (9.36)$$

where η is the learning rate, and α is the momentum coefficient and l is the iteration number. The momentum is simply an added weight used to speed up the training rate. The momentum coefficient, α, is usually restricted such that $0 < \alpha < 1$. Thus, the momentum terms, $\alpha\, \Delta_p w_{ji}^h(l-1)$ and $\alpha\, \Delta_p w_{kj}^o(l-1)$, are fractional values of the weight changes from the previous training iteration.

We calculate the changes in weights, $\Delta_p w_{ji}^h(l)$ and $\Delta_p w_{kj}^o(l)$. We arbitrarily set the learning rate, η, to 0.9, and the momentum coefficient, α, to 0.7. Thus, for $\Delta_p w_{ji}^h(l)$:

$$\Delta_p w_{11}^h(0) = \eta\, \delta_1^h a_1 + \alpha\, \Delta_p w_{11}^h(0-1) = (0.9)(-0.001216)(0.57444) + (0.7)(0) = -0.00628667$$

$$\Delta_p w_{12}^h(0) = \eta\, \delta_1^h a_2 + \alpha\, \Delta_p w_{12}^h(0-1) = (0.9)(-0.001216)(0.52498) + (0.7)(0) = -0.00057453$$

$$\Delta_p w_{13}^h(0) = \eta\, \delta_1^h a_3 + \alpha\, \Delta_p w_{13}^h(0-1) = (0.9)(-0.001216)(0.54983) + (0.7)(0) = -0.00060173$$

For the first iteration, $\Delta_p w_{ji}^h(l-1)$ (from the "previous step") is zero, since no previous step exists. We continue this procedure for all $\Delta_p w_{ji}^h(l)$'s, and end up with:

$$\left[\Delta_p w_{ji}^h(0)\right] = \begin{bmatrix} -6.28667\times10^{-4} & -5.7453\times10^{-4} & -6.0173\times10^{-4} \\ -8.548\times10^{-4} & -7.812\times10^{-4} & -8.1818\times10^{-4} \\ 3.76993\times10^{-3} & 3.44533\times10^{-3} & 3.60843\times10^{-3} \end{bmatrix}$$

We also calculate the weight change $\Delta_p w_{kj}^o(l)$ using the same learning rate ($\eta = 0.9$) and momentum coefficient ($\alpha = 0.7$):

$$\Delta_p w_{11}^o(0) = \eta\, \delta_1^o v_1 + \alpha\, \Delta_p w_{11}^o(0-1) = (0.9)(0.033502)(0.77302) + (0.7)(0) = 0.0233078$$

$$\Delta_p w_{12}^o(0) = \eta\, \delta_1^o v_2 + \alpha\, \Delta_p w_{12}^o(0-1) = (0.9)(0.033502)(0.733471) + (0.7)(0) = 0.0221254$$

$\Delta_p w^{o}_{13}(0) = \eta \delta^{o}_{1} v_3 + \alpha \Delta_p w^{o}_{13}(0-1)$ = (0.9)(0.033502)(0.78578) + (0.7) (0) = 0.0236926

Again, on this first time step, $\Delta_p w^{o}_{kj}(l-1)$ = 0, since no previous time step exists. We continue this procedure for all $\Delta_p w^{o}_{kj}(l)$'s to yield:

$$\left[\Delta_p w^{o}_{kj}(0)\right] = \begin{bmatrix} 0.0233078 & 0.0221154 & 0.0236926 \\ -0.0953537 & -0.0904753 & -0.0969277 \\ -0.0583846 & -0.0553975 & -0.059348 \end{bmatrix}$$

Step 9: Knowing the weight changes, update the weights according to the equations:

$$w^{o}_{kj}(l+1) = w^{o}_{kj}(l) + \Delta_p w^{o}_{kj}(l) \qquad (9.37)$$

$$w^{h}_{ji}(l+1) = w^{h}_{ji}(l) + \Delta_p w^{h}_{ji}(l) \qquad (9.38)$$

After first iteration the update weights are as follows: $W^{o}(1) = W^{o}(0) + \Delta_p W^{o}(0)$

$$= \begin{bmatrix} -1.0 & 1.0 & 0.5 \\ -0.5 & 0.0 & -0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} + \begin{bmatrix} 0.0233078 & 0.0221154 & 0.0236926 \\ -0.0953537 & -0.0904753 & -0.0969277 \\ -0.0583846 & -0.0553975 & -0.059348 \end{bmatrix} = \begin{bmatrix} -0.9767 & 1.0221 & 0.5237 \\ -0.5954 & -0.0905 & -0.5969 \\ 0.4416 & 0.4446 & 0.4407 \end{bmatrix}$$

$$W^{h}(1) = W^{h}(0) + \Delta_p W^{h}(0)$$

$$= \begin{bmatrix} -1.0 & 1.0 & 0.5 \\ 0.5 & 0.0 & -0.5 \\ 0.5 & -0.5 & 0.5 \end{bmatrix} + \begin{bmatrix} -6.28667\times10^{-4} & -5.7453\times10^{-4} & -6.0173\times10^{-4} \\ -8.548\times10^{-4} & -7.812\times10^{-4} & -8.1818\times10^{-4} \\ 3.76993\times10^{-3} & 3.44533\times10^{-3} & 3.60843\times10^{-3} \end{bmatrix} = \begin{bmatrix} -1.0006 & 0.9994 & 0.4994 \\ 0.4991 & -0.0008 & -0.5008 \\ 0.5038 & -0.4966 & 0.5036 \end{bmatrix}$$

It completes one iteration of weight updation. Now let us find the error function value with weight matrices. The input vector $a = \begin{bmatrix} 0.57444 \\ 0.52498 \\ 0.54983 \end{bmatrix}$

The net input to the hidden layer units can be computed as

$$Net^{h} = W^{h} * a + \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} -1.0006 & 0.9994 & 0.4994 \\ 0.4991 & -0.0008 & -0.5008 \\ 0.5038 & -0.4966 & 0.5036 \end{bmatrix} \times \begin{bmatrix} 0.57444 \\ 0.52498 \\ 0.54983 \end{bmatrix} + \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1.2245 \\ 1.0109 \\ 1.3056 \end{bmatrix}$$

The output of hidden layer units as: $V=F(Net^{h}) = \begin{bmatrix} f(1.2245) \\ f(1.0109) \\ f(1.3056) \end{bmatrix} = \begin{bmatrix} \frac{1}{1+\exp(-1.2245)} \\ \frac{1}{1+\exp(-1.0109)} \\ \frac{1}{1+\exp(-1.3056)} \end{bmatrix} = \begin{bmatrix} 0.7729 \\ 0.7331 \\ 0.7868 \end{bmatrix}$

The net inputs to the output layer units can be computed as $Net^{o} = W^{o} * V + \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} =$

$$\begin{bmatrix} -0.9767 & 1.0221 & 0.5237 \\ -0.5954 & -0.0905 & -0.5969 \\ 0.4416 & 0.4446 & 0.4407 \end{bmatrix} \times \begin{bmatrix} 0.7729 \\ 0.7331 \\ 0.7868 \end{bmatrix} + \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1.4065 \\ 0.0038 \\ 2.0140 \end{bmatrix}$$

The output of hidden layer units as:

$$O = F(Net^{o}) = \begin{bmatrix} f(1.4065) \\ f(0.0038) \\ f(2.0140) \end{bmatrix} = \begin{bmatrix} \frac{1}{1+\exp(-1.4065)} \\ \frac{1}{1+\exp(-0.0038)} \\ \frac{1}{1+\exp(-2.0140)} \end{bmatrix} = \begin{bmatrix} 0.8032 \\ 0.50095 \\ 0.8823 \end{bmatrix}$$

With desired output values $d_1 = 1$, $d_2 = 0$, and $d_3 = 0$, our total mean squares error after first iteration is:

$$E = \sum_{k=1}^{3} (d_k - o_k)^2 = (1 - 0.8032)^2 + (0.0 - 0.50095)^2 + (0.0 - 8823)^2 = 1.0681$$

Repeat steps 2-9 for all training patterns until the squared error is zero or sufficiently low.

The delta-rule algorithm repeated for much iteration to achieve less than a 1% error on variable d_1 (i.e. $\in_1 < 0.01$). Note that in the above example, we used only one data point (input pattern). However, the delta-rule algorithm can accommodate multiple input patterns, and the procedure outlined here may be applied to a neural network with P number of input patterns. Backpropagation, as noted, is a form of error-correction learning, and hence attempts to properly map given inputs with desired outputs by minimizing an error function. Typically, we use the sum-of-squares error. The component of the output-error vector from the k^{th} node on the output layer, $\in_k$, is defined as: $\in_k = d_k - c_k$, where d_k is the desired output value and c_k is the calculated value. The total mean-squares error function, *E*, is:

$$E = \sum_{k} \in_k^2 = \sum_{k} (d_k - o_k)^2$$

Associative Memories

10.0.0 **Introduction**

In nature there are many problems, which are unknown to us, or in other words, we need time to gain knowledge of them through our learning capability. How to gain knowledge through learning?, how to organize?, how to store and recall knowledge, is a topic that has been studied for decades by mankind, and referred to as memory. How does our memory works? As for all memory there should be a storage medium. For a type of storage medium, there must be a certain storage mechanism and memory recall functions. In some sense, for the purposes of recognition, reasoning, and behavior

control, our brain should be able to form an inside model so that the environment can be perceived, interpreted, and the knowledge of the environment can be stored. Neural networks provide us with a brand new storage mechanism although the storage medium is still magnetic material or semiconductor. Our brain inspires the new storage mechanism. It is recognized, that brain stores information by modification of synaptic junctions between neurons. Such synaptic junction modifications are distributed. We cannot identify which modification corresponds to which piece of information.

Artificial memory has been in the form of *localized memory* before the beginning of neural network research. Think about the way we make files of documents, or the method in which we write our diary. In the diary all messages are indexed by date. We put those messages of certain events where the date of the event is indicated. When we look for a message, we should first find the date, which is acting as the address here.

Nowadays, information storage inside a computer is still in the form of localized memory. Localized memory is simple and easy to implement. It does not matter whether we use semiconductor memory or magnetic memory, information recording and retrieval are all done through addresses. Addresses are formed in terms of a minimum unit, such as byte for semiconductor memory and the sector for magnetic storage. When searching for a particular data item, we should first find its correct address. This is exactly the same as the situation when you are wandering among bookshelves in a library looking for a book, you cannot get it until you know where it is.

The problem arises when we really use such a simple mechanism for information storage. All the information we are dealing with is ultimately for our brain information processing. The information storage mechanism therefore should match the way our brain works. Nevertheless, there is no any evidence that in our brain the information storage is localized. We recall information by its content. Memory that works this way is referred to as content-addressable memory. One may say that our libraries and database systems work in a content-addressable memory manner too, because every time we seek a particular book or data item, we use key words, which are an abstract of the content. Indeed, key words are used in libraries and databases, but these key words should be first converted into an absolute address through indexes and then the book or data item can be physically retrieved. It is virtually content-addressable, and actually localized.

Our memories function in what is called an *associative* or *content-addressable* fashion. That is, a memory does not exist in some isolated fashion, located in a particular set of neurons. All memories are in some sense strings of memories - you remember someone in a variety of ways - by the color of their hair or eyes, the shape of their nose, their height, the sound of their voice, or perhaps by the smell of a favorite perfume. Thus memories are stored in *association* with one another. These different sensory units lie in completely separate parts of the brain, so it is clear that the memory of the person must be distributed throughout the brain in some fashion. Indeed, PET scans reveal that during memory recall there is a pattern of brain activity in many widely different parts of the brain.

Notice also that it is possible to access the full memory (all aspects of the person's description for example) by initially remembering just one or two of these characteristic features. We access the memory by its contents not by where it is stored in the neural

pathways of the brain. This is very powerful; given even a poor photograph of that person we are quite good at reconstructing the persons face quite accurately. This is very different from a traditional computer where specific facts are located in specific places in computer memory. If only partial information is available about this location, the fact or memory cannot be recalled at all.

10.1.0 Paradigms of Associate Memory

The common paradigm of memory here may be described as follows. There is some underlying collection of data which is ordered and interrelated in some way and which is stored in memory. The data may be thought of, therefore, as forming a stored pattern. In the recollection examples below, it is the cluster of memories associated with the celebrity or the phase in childhood. In the case of character recognition, it is the parts of the letter (pixels) whose arrangement is determined by an archetypical version of the letter. When part of the pattern of data is presented in the form of a sensory cue, the rest of the pattern (memory) is recalled or *associated* with it. Notice that it often doesn't matter which part of the pattern is used as the cue, the whole pattern is always restored.

Conventional computers (von Neumann machines) can perform this function in a very limited way. The typical software for this is usually referred to as a database. Here, the `sensory cue' is called the *key*, which is to be searched on. For example, the library catalogue is a database, which stores the authors, titles, class marks and data of publication of books and journals. We may search on any one of these discrete items for a catalogue entry by typing the complete item after selecting the correct option from a menu.

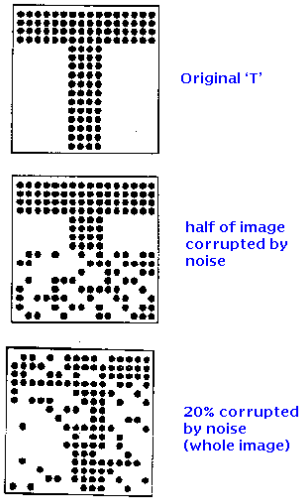


Fig. 10.1 slide of `T's

Suppose now we have only the fragment `ion, Mar' from the full title `Vision, Marr D.'. There is no way that the database can use this fragment of information to even start searching. We don't know if it pertains to the author or the title, and even if we did, we might get titles or authors that start with `ion'. The kind of input to the database has to be very specific and complete.

10.2.0 A physical analogy with memory

The networks that are used to perform associative recall are specific examples of a wider class of physical systems, which may be thought of as doing the same thing. This allows the net operation to be viewed as a dynamics of a physical system and its behaviour to be described in terms of the network's `energy'. Consider a bowl in which a ball bearing is allowed to roll freely as shown 10.2 and this is more easily drawn using a 2D cross section as in 10.3.

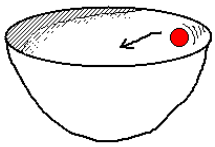


Fig. 10.2 Bowl and ball bearing in 3D

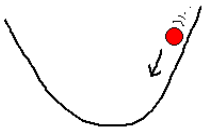


Fig. 10.3. 2d X-section of bowl

Suppose we let the ball go from a point somewhere up the side of the bowl with, possibly, a push to one side as well. The ball will roll back and forth and around the bowl until it comes to rest at the bottom. The physical description of what has happened may be couched in terms of the energy of the system. The ball initially has some *potential* energy. That is work was done to push it up the side of the bowl to get it there and it now has the potential to gain speed and acquire energy. When the ball is released, the potential energy is released and the ball rolls around the bowl (it gains *kinetic* energy). Eventually the ball comes to rest where its energy (potential and kinetic) is zero. (The kinetic energy gets converted to heat via friction with the bowl and the air). The main point is that the ball comes to rest in the same place every time and this place is determined by the energy minimum of the system (ball + bowl). The resting state is said to be *stable* because the system remains there after it has been reached

There is another way of thinking of this process, which ties in with our ideas about memory. We suppose that the ball comes to rest in the same place each time because it `remembers' where the bottom of the bowl is. We may push the analogy further by giving the ball a coordinate description. Thus, its position or *state* at any time is given by the three coordinates (*x, y, z*) or the position vector **x**. The location of the bottom of the bowl, $\mathbf{x}_0$ represents the pattern, which is stored. By writing the ball's vector as the sum of $\mathbf{x}_0$ and a displacement $\Delta\mathbf{x}$ thus, $\mathbf{x} = \mathbf{x}_0 + \Delta\mathbf{x}$, we may think of the ball's initial position as representing the partial knowledge or cue for recall, since it approximates to the memory $\mathbf{x}_0$. If we now use a corrugated surface instead of a single depression (the bowl) we may store many `memories'.

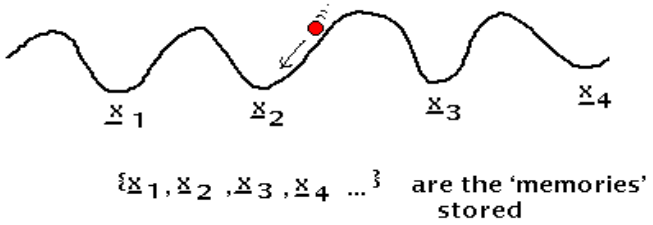


Fig. 10.4 X-section through corrugated surface

If now the ball is started somewhere on this surface, it will eventually come to rest at the local depression which is closest to its initial starting point. That is it evokes the stored pattern, which is closest to its initial partial pattern or cue. Once again, this is an energy minimum for the system. There are therefore two complementary ways of looking at what is happening. One is to say that the system falls into an energy minimum; the other is that it stores a set of patterns and recalls that which is closest to its initial state. If

we are to build a network, which behaves like this, we must include the following key elements

1. It is completely described by a *state vector* $\mathbf{V} = (v_1, v_2, \ldots, v_n)$.
2. There are a set of *stable states* $V_1, V_2, \ldots, V_n$. These will correspond to the stored patterns and, in, the corrugated surface example, were the bottoms of the depressions in the surface.
3. The system evolves in time from any arbitrary starting state **v** to one of the stable states, and this may be described as the system decreasing its energy *E*. This corresponds to the process of memory recall.

10.3.0 Associative Memory

Our memories function in what is called an *associative*. Content-addressable memory is also referred to as *associative memory,* or *distributed memory.* It is distributed because a piece of information is not stored in a particular memory location; rather its storage happens to be changes of synaptic junctions. Such synaptic junction changes are distributed. A stored piece of information appears only when it is being recalled. Distributed memory tolerates partial damage of the storage units. The stored information can be still recalled even though part of the storage units is damaged. The effect of partial damage of the storage only causes quality degradation of the information recall.

The capability of content-addressable memory is achieved through association. There are two types of association: auto-association and hetero-association. Auto-association means that a piece of information is associated with itself in the process of its storage. In the retrieval phase, one can recall the information through a part or degraded version of this piece of information. With hetero-association, a piece of information is associated with another piece of information. These two pieces of information can have the relationship such as having similar or opposite meaning/appearance, being closely related, etc. After two pieces of information are stored in the manner of hetero-association, one piece of information can recall another. Associative memory has raised the mechanism of memory from simple mechanical level to semantic level. It provides efficient tools for information storage and retrieval. It is expected that with associative memory some human brain functions can be artificially simulated.

10.3.1 Basic Model

Neural networks for associative memory have two distinct modes of operation when they are used to store data items. If each data item is distinct and it is necessary to keep them separate, then it is called memory for particular events. On the other hand, the neural network forms equivalence classes in the *stor*age by forming "concepts" or "categories", the memory becomes a memory for concepts. In the "memory for events" operation mode, the storage is severely limited to a fraction of the network dimension because each data item must not be degraded. In the mode of "memory of concepts", the storage capacity for data items is not so restricted. What matters is correct learning and classification of new data items. However, "memory of events" can be considered as an *extreme* of "memory of concepts", where each category has exactly one data item.'

10.3.2 Pattern mathematics

The term *pattern* is to represent a piece of knowledge. In a general sense, a pattern can be numerical (a vector in a n-dimensional space), symbolic (a syntactic sentence), or semantic (a semantic network). Pattern vectors do have semantic

interpretations when used in applications. For example, in remote sensing applications, the spectral signature of a ground object is a vector in a space defined by the sensor array. A tuple of a relational database can also be considered a vector although the field of the relation can be either numerical or string.

Obviously, associative memory and pattern recognition are mutually overlapping research fields. The storage phase of associative memory corresponds to the learning (or training) in pattern recognition, while the recall phase of associative memory coincides with performing actual recognition. The difference between these two fields is that in pattern recognition more emphasis is placed on the pattern theory and recognition methodology, while in neural network associative memory we are looking for neural network models and algorithms which can actually perform associative memory functions. However, there are a lot of concepts, mathematical methods can be shared by these two related fields. For example, in pattern recognition, feature selection is a feature vector preprocessing process to improve the efficiency of the recognition. A typical feature selection method is principal component analysis. It maps the feature vectors to an orthogonal space with the coordinates pointing in the main directions of the feature vectors. The same problem happens to associative memory: The memory works perfectly to store a set of uncorrelated patterns. There will be interference between patterns if the patterns to be stored are correlated. Such interference reduces the capacity of the memory as well. To reduce the interference and to expand the storage capacity methods similar to principal component analysis are developed. Now we will start with some general pattern mathematics.

Patterns are represented as n-dimensional vectors $\boldsymbol{a} = (a_1, ..., a_n)$. They are usually points in an n-dimensional vector space. It is very important to consider a pattern as a point in an n-dimensional vector space, hence the mathematical methods from linear algebra can be used to establish measures of patterns and to define operations on patterns.

A basic operation between two patterns is the inner product. The inner product of pattern $\boldsymbol{a}$ and $\boldsymbol{b}$ is written as $\boldsymbol{a} \cdot \boldsymbol{b}$ and defined as

$$\boldsymbol{a.b} = a_1 b_1 + a_2 b_2 + ... + a_n b_n \qquad (10.1)$$

The normalized correlation between $\boldsymbol{a}$ and $\boldsymbol{b}$ is the inner product divided by their norm: $\frac{a.b}{\|a\|\|b\|}$. It is the cosine of the angle between these two vectors. If the inner product of two patterns is zero they are said to be *uncorrelated* or *orthogonal*, otherwise they are correlated.

For a given set of patterns $\boldsymbol{a}^1, \boldsymbol{a}^2, ..., \boldsymbol{a}^m$, if there exits a set of scalar parameters $k_1, k_2, ..., k_m \in F$ which are not zero simultaneously, such that

$$k_1 a^1 + k_2 a^2 + ... + k_m a^m = 0 \qquad (10.2)$$

then we can say this set of patterns are *linearly dependent* in *F*. They are *linearly independent* otherwise. Obviously, a set of patterns are linearly dependent if there is at least one pattern which can be represented as a linear combination of other patterns.

10.3.3 General concepts of associative memory

A general block diagram of associative memory is depicted in Fig. 10.5. An input pattern $\boldsymbol{a} = \{a_1, ..., a_N\}$ is input into associative memory which is characterized by $\boldsymbol{M}$. The output of the memory is pattern $\boldsymbol{b} = \{b_1, ..., b_p\}$. The memory system can be written as $\boldsymbol{b} = f(a, \boldsymbol{M})$.

Assume that for a set of input patterns $\boldsymbol{A} = \{\boldsymbol{a^1}, \ldots, \boldsymbol{a^k}\}$, and with associative memory we are able to get a set of output patterns $\boldsymbol{B} = \{\boldsymbol{b^1}, \ldots, \boldsymbol{b^k}\}$with a one-to-one correspondence

$\boldsymbol{a^1 \rightarrow b^1}$

.

.

$\boldsymbol{a^k \rightarrow b^k}$

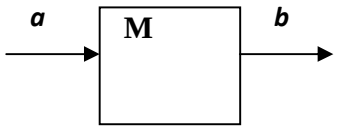


Fig.10.5 Block diagram of associative memory

We then call this memory system hetero-associative memory. Auto-associative memory can be considered a special case of hetero-associative memory. Temporal association is another extension of hetero-association where a time series can be associated with its initial state, when its initial state is presented the whole time series is recalled.

10.3.4 Associative Matrix

Matrix **M** in associative memory is defined as an N×p positive matrix. Its elements can be regarded as representing the synaptic junction. There are N input neurons $\{u_1^i, \ldots, u_N^i\}$and p output neurons $\{u_1^o, \ldots, u_p^o\}$. These neurons are acting *as* short-term memory. For convenience, we use $\{a_1, \ldots, a_N\}$and $\{b_1, \ldots, b_p\}$to denote the activations of these input and output neurons, and w_{ij} to denote the synaptic junction from a_j to b_i. Synaptic junctions are long-term memory. A positive synaptic junction represents an excitatory link while a negative synaptic junction represents an inhibitory link. Usually associative memory is any mapping *from* input pattern space into output space *T:* $R^N \rightarrow R^p$. *T* can be nonlinear. The mapping represented by **M**: $R^N \rightarrow R^p$ is linear associative memory. When $\{a_1, ..., a_N\}$ and $\{b_l, ..., b_p\}$ are different, **M** is hetero-association memory. It stores pattern pair (a_i, b_i). When the input patterns and output patterns are the same, **M** is auto-association memory. It stores the input pattern a_i.

10.3.5 Association rules: For neural networks, which act as associative memory, associative matrix **M** appears as the weights of the links between units and usually denoted by **W**. Patterns are stored via learning. The Hebbian rule is often used to update weights. After learning the weight matrix will be:

$$w_{ij} = \frac{1}{N}\sum_{l=1}^{m} a_i^l a_j^l \qquad (10.3)$$

where a_i^l denote the i-th component of the l-th pattern.

The Hopfield network is the example of associative memory and the bi-directional associative memory is the example of heteroassociative memory.

10.4.0 Associative- Memory Definitions

The concept of an associative memory is a fairly intuitive one. Associative memory appears to be one of the primary functions of the brain. We easily associate the face of a friend which that friends name, or a name with telephone number.

Many devices exhibit associative memory characteristics; For example the memory bank in a computer is a type of associative memory; it associates address with data. A simple associative memory is called linear associator.

10.4.1 Hamming Distance

In general, Hamming space can be defined by the expression

$$H^n = \left\{x = (x_1, x_2, ...x_n)^T \in R_1^n, x_i \in (\pm 1)\right\} \qquad (10.4)$$

In words, n-dimensional Hamming space is the set of n-dimensional vectors, with each component an element of the real number R, subject to the condition that each component is restricted to the values ±1. This space has 2^n points, all equidistant from the origin of Euclidian space. Many neural network models use the concept of the distance between two vectors. There are, however, many different measures of distance. Hamming distance is the one of the methods available. Here we are trying to find the relationship between Hamming distance and Euclidean distance.

Let $X = (x_1, x_2, \ldots .x_n)^T$ and $Y = (y_1, y_2, \ldots .y_n)^T$ be two vectors in n-dimensional Euclidean space, subject to the restriction that $x_i, y_i \in \{\pm 1\}$, so that X and Y are also vectors in n-dimensional Hamming space. The Euclidean distance between the two vector end point is $d = \sqrt{(x_1 - x_2)^2 + (x_2 - y_2)^2 + ...(x_n - y_n)^2}$ (10.5)

since $x_i, y_i \in \{\pm 1\}$, then $(x_i, - y_i)^2 \in \{0, 4\}$; $(x_i - y_i)^2 = \begin{cases} 0 & x_i = y_i \\ 4 & x_i \neq y_i \end{cases}$

Thus, the Euclidean distance can be written as $d = \sqrt{4(\#\text{ mismatched components of X and Y})}$

we define the Hamming distance as: h = # mismatched components of X and Y or the number of bits that are different between X and Y.

The Hamming distance is related to the Euclidean distance by the equation

$$d = 2\sqrt{h} \quad \text{or} \quad h = \frac{d^2}{4} \qquad (10.6)$$

Example: Determine the Euclidean distance between $(1, 1, 1, 1, 1)^T$ and $(-1, -1, 1, -1, 1)^T$. Use this result to determine the Hamming distance.

$$d = \sqrt{4+4+0+4+0} = \sqrt{12}\ , \qquad h = \frac{d^2}{4} = \frac{12}{4} = 3$$

10.4.2 The Linear Associator

Suppose we have L pairs vectors $\{(x_1, y_1), (x_2, y_2), \ldots . (x_L, y_L)\}$, with $x_i \in R^n$ and $y_i \in R^n$. We call these vectors exemplars, because we will use them as examples of correct associations we can distinguish three types of associative memories.

1. Associative Memory: Assumes $y_i = x_i$ and implements a mapping, ϕ, of X to X such that $\phi(x_i) = x_i$, and if some arbitrary x is closer x_i than to any other x_j, j = 1, . . .L, then $\phi(x) = x_i$.

2. Heteroassociative memory: Implements a mapping, ϕ, of X to Y such that $\phi(x_i) = y_i$, and if an arbitrary x is closer to x_i than to any other x_j, j = 1, . . .L, then $\phi(x) = y_i$. In this and the following definitions closer means with respect to Hamming distance.

3. Interpolative Associative Memory: Implements a mapping, ϕ, of X to Y such that $\phi(x_i) = y_i$, but if the input vectors differs from one of the exemplars by the vector d, such that $x = x_i + d$, then the output of the memory also differs from one of the exemplars by some vector e; $\phi(x) = \phi(x_i+d) = y_i + e$.

10.5.0 Summary

In this unit we have discussed the basic concepts of associative *memory*. Two fundamental operations of *memory* are storage and retrieval, or in the other words, write and read. Two fundamental objects that the memory deals with are addresses and contents. Associative *memory* is content-addressable. There is no address at all. Storing an item in the network involves the network weight modification rule, and retrieval of an item from the network is accomplished by network evolution.

Content-addressable memory requires that the contents to be stored should be distinctive; Otherwise, the stored items may interfere with each other. When we use the Hebbs rule to store patterns, there is no interference between stored items, provided they are orthogonal. In most cases, patterns to be stored are not orthogonal, this limits the storage capacity of the network to a small portion of the network dimension. To increase the network storage capacity and to reduce the interference between stored items the projection method is often invoked. Content-addressable *memory* leads to categorization. Instead of storing an exact copy, patterns presented to the network are categorized into certain categories. When presenting a sample pattern to the network, the stored template, which is most similar to the sample, is retrieved. This is a kind of "best match" problem.

BIDIRECTIONAL ASSOCIATIVE MEMORIES

11.0.0 Introduction

The bi-directional associative memory (BAM) consists of two layers of processing elements that are fully interconnected between the layers. It is an hetero-associative, that is, it accepts an input vector on one set of neurons and produces a related, but different, output vector on another set. It has the characteristic of a content addressable memory rather than needing the address of a piece of information you can ask for the location of the information that you want. Applications for content-addressable memories include paging systems (i.e. get me the physical address of the virtual address. The virtual address is the content and it is what we know. The physical address is where the virtual address resides.).

Since a BAM is bi-directional, it can take input in either layer, however this implementation currently only supports loading inputs into the input layer. Although a BAM can be binary or bipolar, this version also only has the weight formula for bipolar and the transfer function (activation function) for bipolar (with bias is fixed at 0). The BAM is capable of generalization, producing correct outputs despite corrupted inputs. Also adaptive versions can abstract, extracting the ideal from a set of noisy examples.

11.1.0 BAM Architecture

As in other neural networks, in the BAM architecture there are weights associated with the connections between processing elements. The general network architecture of BAM is shown as in Fig. 11.1. In this we have considered as n units on the X-layer and m units on the Y layer. For convenience, we shall define the X vector, $x \in R^n$ as input vector and the Y-vector, $y \in R^m$ as its output vector. All connections between units are bi-directional, with weights at each end. Information passes back and forth from one layer to the other, through these connections. In this network, the weights can be determined in advance, if all of the training vectors, can be identified. This is one main difference from other networks

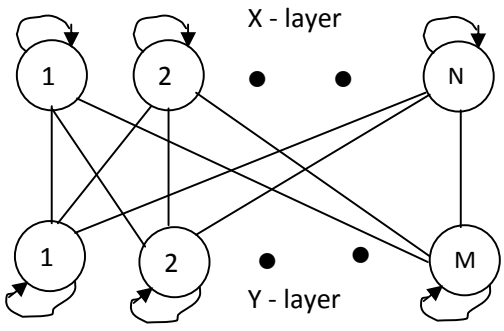


Fig. 11.1 The BAM here has n units in X layer, and m units on the Y layer.

This equation on gives the weights on the connections from the X layer to the Y layer, say W. For example, the values W_{23} as the weight on the connection from the third

unit on the X layer to the second unit on the Y layer. To construct the weight matrix for from Y layer to X layer units, we simply take the transpose of the weight matrix W^T as shown in Fig.11.2.

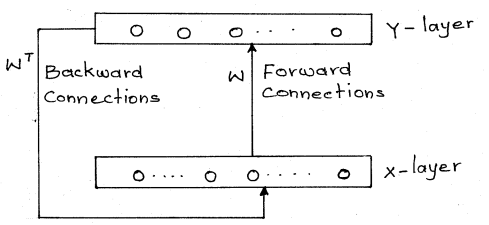


Fig. 11.2 Schematic representation of weight matrices of BAM

We can make the BAM into an auto associative memory by constructing the weight matrix as square and symmetric. In this network, the dimensionality of vectors X and Y are same.

11.2.0 Mathematical modeling of BAM

The basic processing done by each unit of the BAM is similar to that done by the general processing element. We have discussed the activation dynamics of general processing node in unit-5. Now, the additive activation dynamics of BAM can be written as

$$\dot{x}_i(t) = -A_i x_i(t) + \sum_{j=1}^{M} f_j(y_j(t)) w_{ij} + \mathrm{I}_i \,, \mathrm{i} = 1,2,..,\mathrm{N} \qquad (11.1)$$

$$\dot{y}_j(t) = -A_j y_j(t) + \sum_{i=1}^{N} f_i(x_i(t)) w_{ji} + \mathrm{J}_j \,, \mathrm{j} = 1,2,..,\mathrm{M} \qquad (11.2)$$

where I_i and J_j are the net external inputs to the units i and j respectively. A_i and $f_i(.)$ could be different for each unit and for each layer. Once the neurons reaches to equilibrium state, where $\dot{x}_i(t)$=0 and $\dot{y}_j(t)$=0, the output of the network is computed. We need to consider one layer as its input layer and the other as its output layer. Now, let us consider X-layer as input layer and Y-layer as its output layer, that is, the signal flow is from X-layer to Y-layer. The output of the network may computed as follows:

1. Compute the net input values of nodes on Y-layer: $\mathrm{Net}^y = WX$ (11.3)

where Net^y is the vector of net-input values on the Y layer. For the individual units, it is written as $\mathrm{net}_j^y = \sum_{i=1}^{N} w_{ji} x_i$ (11.4)

2. Compute the outputs (new values of Y-layer units) of the output layer nodes:

$$\mathrm{y}_j = f_j\left(\sum_{i=1}^{N} w_{ji} x_i\right) \qquad (11.5)$$

where $f_j(.)$ is a nonlinear activation function of the neuron j on Y-layer. For a typical type of activation function, its output relations may be written as:

$$y_j(l+1) = \begin{cases} +1 & \text{net}_j^y > 0 \\ y_j(l) & \text{net}_j^y = 0 \\ -1 & \text{net}_j^y < 0 \end{cases} \qquad (11.6)$$

where *l* is the step number and j =1, 2, . . .M.

Now, let us consider Y-layer as input layer and X-layer as its output layer, that is, the signal flow is from Y-layer to X-layer. The output of the network may computed as follows:

3. Compute the net input values of nodes on X-layer: $\text{Net}^x = WY$ (11.7)

where Net^x is the vector of net-input values on the X - layer. For individual units it is written as $\text{net}_i^x = \sum_{j=1}^{M} y_j w_{ij}$ (11.8)

4. Compute the outputs (new values of X-layer units) of the output layer nodes:

$$x_i = f_i(\sum_{j=1}^{M} w_{ij} y_j) \qquad (11.9)$$

where $f_i(.)$ is a nonlinear activation function of the neuron i on X-layer. For a typical type of activation function, its output relations may be written as:

$$x_i(l+1) = \begin{cases} +1 & \text{net}_i^x > 0 \\ x_i(l) & \text{net}_i^x = 0 \\ -1 & \text{net}_i^x < 0 \end{cases} \qquad (11.10)$$

where *l* is the step number and i=1, 2, . . . N. The quantities N and M are the dimensions of the X and Y layers respectively.

11.3.0 Training of BAM

The training of BAM network is the process of storing information. It is one step process for given set information. Given L vector pairs that constitute the set of examples that would like to store, we can construct the weight matrix as:

$$W = Y_1 X_1^T + Y_2 X_2^T + \ldots\ldots + Y_L X_L^T \qquad (11.11)$$

We can make the BAM into an auto associative memory by constructing the weight matrix as square and symmetric. That is,

$$W = X_1 X_1^T + X_2 X_2^T + \ldots\ldots + X_L X_L^T \qquad (11.12)$$

11.4.0 Recall Algorithm of BAM

Once weight matrix has been constructed, the BAM can be used to recall information, when presented with some key information. If the desired information is

only partially known in advance or is noisy, the BAM may be able to complete the information.
To recall information using the BAM, we perform the following steps:

1. Apply an initial vector pair, (X_0, Y_0) to the processing elements of the BAM.
2. Propagate the information from X layer to the Y layer, and update the values on the Y-layer units.
3. Propagate the updated Y information back to the X layer and update the units there.
4. Repeat steps 2 and 3 until there is no further change in the units on each layer.

This algorithm is what gives the BAM its bidirectional nature. The terms inputs and output refer to different quantities, depending on the current direction of the propagation. For example, in going from Y to X, the 'Y' vector is considered as the input to the network, and the X vector is the output. The opposite is true when propagating from X to Y. If all goes well, the final, stable state will recall one of the exemplars used to construct the weight matrix.

If we try to put too much information in a given BAM, a phenomenon known as cross talk occurs between exemplar patterns. Cross talk occurs when exemplar patterns are too close to each other. The interaction between these patterns can result in the creation of spurious stable states. In that wise, the BAM could stabilize on meaningless vectors. The spurious stable states corresponds to minima that appear between the minima that correspond to the exemplars.

Example: For the following given pairs vectors construct weight matrix for appropriate structure of BAM.

$X_1 = (1, -1, -1, 1, -1, 1, 1, -1, -1, 1)^T$, $Y_1 = (1, -1, -1, -1, -1, 1)^T$

$X_2 = (1, 1, 1 -1, -1, -1, 1, 1, -1, -1)^T$, $Y_2 = (1, 1, 1, 1, -1, -1)^T$

$$W = Y_1 X_1^T + Y_2 X_2^T$$

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

$$= \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{matrix} \\ \begin{bmatrix} 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 \end{bmatrix} \end{array} + \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 \\ -1 & -1 & -1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & 1 & 1 \end{bmatrix} \end{array}$$

$$W = \begin{bmatrix} 2 & 0 & 0 & 0 & -2 & 0 & 2 & 0 & -2 & 0 \\ 0 & 2 & 2 & -2 & 0 & -2 & 0 & 2 & 0 & -2 \\ 0 & 2 & 2 & -2 & 0 & -2 & 0 & 2 & 0 & -2 \\ 0 & 2 & 2 & -2 & 0 & -2 & 0 & 2 & 0 & -2 \\ -2 & 0 & 0 & 0 & 2 & 0 & -2 & 0 & 2 & 0 \\ 0 & -2 & -2 & 2 & 0 & 2 & 0 & -2 & 0 & 2 \end{bmatrix}$$ X to Y and $$W^T = \begin{bmatrix} 2 & 0 & 0 & 0 & -2 & 0 \\ 0 & 2 & 2 & 2 & 0 & -2 \\ 0 & 2 & 2 & 2 & 0 & -2 \\ 0 & -2 & -2 & -2 & 0 & 2 \\ -2 & 0 & 0 & 0 & 2 & 0 \\ 0 & -2 & -2 & -2 & 0 & 2 \\ 2 & 0 & 0 & 0 & -2 & 0 \\ 0 & 2 & 2 & 2 & 0 & -2 \\ -2 & 0 & 0 & 0 & 2 & 0 \\ 0 & -2 & -2 & -2 & 0 & 2 \end{bmatrix}$$ Y to X

For our first trial, we choose an X vector with a Hamming distance of 1 from X_1;

$$X_0 = (-1, -1, -1, 1, -1, 1, 1, -1, -1, 1)^t$$

This situation could represent noise on the input vector. The starting Y_0 vector is one of the training vector Y_2; $Y_0 = (1, 1, 1, -1, -1)^T$. (Note that in a realistic problem you may not have prior knowledge of the output vector. Use a random bipolar vector if necessary).

We will propagate first from X to Y

$$\text{Net}^y = WX = \begin{bmatrix} 2 & 0 & 0 & 0 & -2 & 0 & 2 & 0 & -2 & 0 \\ 0 & 2 & 2 & -2 & 0 & -2 & 0 & 2 & 0 & -2 \\ 0 & 2 & 2 & -2 & 0 & -2 & 0 & 2 & 0 & -2 \\ 0 & 2 & 2 & -2 & 0 & -2 & 0 & 2 & 0 & -2 \\ -2 & 0 & 0 & 0 & 2 & 0 & -2 & 0 & 2 & 0 \\ 0 & -2 & -2 & 2 & 0 & 2 & 0 & -2 & 0 & 2 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \\ -1 \\ 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

$$\therefore \text{Net}^y = (4, -12, -12, -12, -4, 12)^t$$

The new Y vector $Y_{new} = (1 -1 -1 -1 -1 1)^t$ and which is also one of the training vector .

Propagating back to X layer:

$$\text{Net}^x = W^T Y = \begin{bmatrix} 2 & 0 & 0 & 0 & -2 & 0 \\ 0 & 2 & 2 & 2 & 0 & -2 \\ 0 & 2 & 2 & 2 & 0 & -2 \\ 0 & -2 & -2 & -2 & 0 & 2 \\ -2 & 0 & 0 & 0 & 2 & 0 \\ 0 & -2 & -2 & -2 & 0 & 2 \\ 2 & 0 & 0 & 0 & -2 & 0 \\ 0 & 2 & 2 & 2 & 0 & -2 \\ -2 & 0 & 0 & 0 & 2 & 0 \\ 0 & -2 & -2 & -2 & 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = [4 -8 -8 \quad 8 -4 \quad 8 \quad 4 -8 -4 \quad 8]^T$$

$$X_{new} = [1 -1 -1 \; 1 -1 \; 1 \; 1 -1 -1 \; 1]^t$$

Further passes result in no change, so we are finished. The BAM successfully recalled the first training set.

11.5.0 BAM Energy Function

In supervised learning (multiplayer perceptron), during the training process the weights form a dynamical system. That is, the weights change as a function of time, and those changes can be represented as a set of coupled differential equations.

For BAM, a slightly different situation occurs. The weights are calculated on advance, and are not part of a dynamical system. On the other hand, an unknown pattern presented to the BAM may require several passes before the network stabilizes on a final result. In this situation, the X and Y vectors change as a function of time, and they form a dynamical system.

In the theory of dynamical systems, a theorem can be proved concerning the existence of stable states that uses the concept of a function called a Lyapunov function or Energy function. If a bounded function of the state variables of a dynamical system can be found, such that all state changes results in a decrease in the value of the function, then the system has a stable solution. This function is called a *Lyapunov* function or energy function. In the case of the BAM, such a function exists we shall call it the BAM energy function. It has the form

$$E(x, y) = -Y^T WX \qquad (11.13)$$

or in terms of components

$$E = -\sum_{i=1}^{M}\sum_{j=1}^{N} y_i w_{ij} x_j \qquad (11.14)$$

In order to say that, the BAM is stable, it has to satisfy the following theorem on BAM energy function:

The theorem has three parts:

1. Any change in X or Y during BAM processing results in a decease in E.
2. E is bounded below by $E_{min} = - \sum_{i,,j} |w_{ij}|$
3. When E changes, it must change by a finite amount

Items 1 and 2 prove that E is a Lyapunov function, and that the dynamical system has a stable state. In particular, item 2 show that E can decrease only to a certain value; it can't continue down to negative infinity, so that eventually the vectors X and Y vectors must stop changing. Item 3 prevents the possibility that changes in E might be infinitesimally small; resulting in an infinite amount of time spent before the minimum E is reached.

The initial state of the BAM is determined by the choice of the staring vectors, (X and Y). As the BAM processes the data, X and Y change, resulting in movement of the energy over the landscape, which is guaranteed by BAM energy theorem to be downward.

Initially, the changes in the calculated values of E (X, Y) are large. As the X and Y vectors reach their stable state, the value of E changes by smaller amounts, and eventually stops changing when the minimum point is reached. This situation corresponds to a physical system such as a ball rolling down a hill into a valley, but with enough friction that, by the time the ball reaches the bottom, it has no more energy and therefore it stops. Thus, the BAM resembles a dissipative dynamic system in which the E function corresponds as the energy of the physical system.

11.0.1 **Proof of BAM energy theorem**

In this we prove the first part of the theorem and the part 2 & 3 are left to the readers to prove. Consider the equation (11.14) as $E = -\sum_{i=1}^{M}\sum_{j=1}^{N} y_i w_{ij} x_j$

According to the theorem, any change in X or Y must result in a decrease in the value of E. For simplicity, let us consider a change in a single component of Y, specifically y_k. Now, we can write the above equation showing the term with y_k explicitly:

$$E = -\sum_{j=1}^{N} y_k w_{kj} x_j - \sum_{\substack{i=1 \\ i\neq k}}^{M}\sum_{j=1}^{N} y_i w_{ij} x_j \qquad (11.15)$$

Now, consider the change $y_k \rightarrow y_k^{new}$. The new energy value is

$$E^{new} = -\sum_{j=1}^{N} y_k^{new} w_{kj} x_j - \sum_{\substack{i=1 \\ i\neq k}}^{M}\sum_{j=1}^{N} y_i w_{ij} x_j \qquad (11.16)$$

The change in energy can be written as $\Delta E = (E^{new} - E) = (y_k - y_k^{new})\sum_{j=1}^{N} w_{kj} x_j$

(11.17)

From equation (11.6), the value of y_k can be determined as

$$y_k^{new} = \begin{cases} +1 & \sum_{j=1}^{N} w_{kj} x_j > 0 \\ y_k & \sum_{j=1}^{N} w_{kj} x_j = 0 \\ -1 & \sum_{j=1}^{N} w_{kj} x_j < 0 \end{cases}$$

The above equation gives scope to consider two possible changes of y_k:

First, suppose y_k = +1 and it changes to –1; in this case, $(y_k - y_k^{new})$>0. But, in according to the procedure for calculating y_k^{new}, this transition can occur only if $\sum_{j=1}^{N} w_{kj} x_j$ <0. Therefore, the value of ΔE is the product of one factor that is greater than zero and one that is less than zero. The result is that ΔE <0.

The second possibility is that, y_k = -1 and it changes to +1; in this case, $(y_k - y_k^{new})$ <0, but this transition can occurs only if $\sum_{j=1}^{N} w_{kj} x_j$ >0. Again, the value of ΔE is the product of one factor less than zero and one greater than zero. In both cases where y_k changes, ΔE decreases. Note that, for the case where y_k does not change, both factors in the equation for ΔE are zero, so the energy does not change unless on of the vectors changes.

12.0.0 Introduction

We have seen in unit-10 about the memories and their functions. The *Hopfield network* is a simple and one of the oldest artificial network, which can store certain memories or patterns in a manner rather similar to the brain - the full pattern can be recovered if the network is presented with only partial information. Furthermore there is a degree of stability in the system - if just a few of the connections between nodes (neurons) are severed, the recalled memory is not too badly corrupted - the network can respond with a "best guess". Of course, a similar phenomenon is observed with the brain - during an average lifetime many neurons will die but we do not suffer a catastrophic loss of individual memories - our brains are quite robust in this respect (by the time we die we may have lost 20 percent of our original neurons).

The nodes in the network are vast simplifications of real neurons - they can only exist in one of two possible "*states*" - *firing* or *not firing*. Every node is connected to every other node with some strength. At any instant of time a node will change its state (i.e start or stop firing) depending on the inputs it receives from the other nodes.

If we start the system *off* with any general pattern of firing and non-firing nodes then this pattern will in general change with time. To see this, think of starting the network with just one firing node. This will send a signal to all the other nodes via its connections so that a short time later some of these other nodes will fire. These new firing nodes will then excite others after a further short time interval and a whole cascade of different firing patterns will occur. One might imagine that the firing pattern of the network would change in a complicated perhaps random way with time.

The crucial property of the Hopfield network, which renders it useful for simulating memory recall, is the following:

- we are *guaranteed* that the pattern will settle down after a long enough time to some fixed pattern.
- Certain nodes will be always "*on*" and others "*off*".
- Furthermore, it is possible to arrange that these *stable firing patterns* of the network correspond to the desired memories we wish to store. The reason for this is somewhat technical but we can proceed by analogy.

Imagine a ball rolling on some bumpy surface. We imagine the position of the ball at any instant to represent the activity of the nodes in the network. Memories will be represented by special patterns of node activity corresponding to wells in the surface. Thus, if the ball is let go, it will execute some complicated motion but we are certain that eventually it will end up in one of the wells of the surface. We can think of the height of the surface as representing the energy of the ball. We know that the ball will seek to minimize its energy by seeking out the lowest spots on the surface.

Furthermore, the well it ends up in well usually be the one it started off closest to. In the language of memory recall, if we start the network off with a pattern of firing which approximates one of the "stable firing patterns" (memories) it will "under its own steam" end up in the nearby well in the energy surface thereby recalling the original perfect memory.

The smart thing about the Hopfield network is that there exists a rather simple way of setting up the connections between nodes in such a way that any desired set of patterns can be made "stable firing patterns". Thus any set of memories can be burned into the network at the beginning. Then if we kick the network off with any old set of node activity we are *guaranteed* that a "memory" will be recalled. Not too surprisingly, the memory that is recalled is the one, which is "closest" to the starting pattern. In other words, we can give the network a corrupted image or memory and the network will "all by itself" try to reconstruct the perfect image. Of course, if the input image is sufficiently poor, it may recall the incorrect memory - the network can become "confused" - just like the human brain. We know that when we try to remember someone's telephone number we will sometimes produce the wrong one! Notice also that the network is reasonably robust - if we change a few connection strengths just a little the recalled images are "roughly right". We don't lose any of the images completely.

12.1.0 **Architecture of Hopfield Network**

Among all the auto-associative networks, the Hopfield network (Hopfield and tank) is the most widely known today. The Hopfield net consists of a number of nodes, each connected to every other node. We might ask the question - *how many memories can be stored in such a network, how big (how many nodes) does the network need to have to be to store a given number of stable firing patterns?*

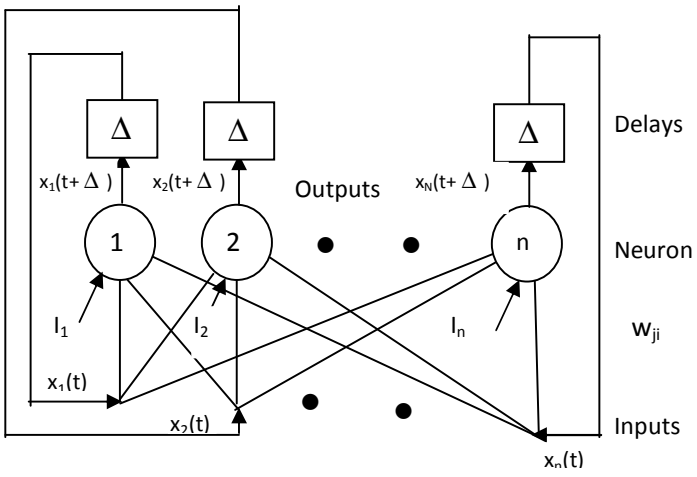


Fig. 12.1 Architecture of Hopfield network

For the Hopfield network the answer is known (for other networks it is generally not!). That is the number of nodes equal to the number of elements in an input vector. It is a fully connected network with a set of neurons and a corresponding set of delays, forming multiple-loop feedback system as shown in Fig 12.1. However, recent studies reveals that, by using this model new models are derived by adding some nodes to the existing n neurons.

The number of feedback loops is equal to the number of neurons. The output of each neuron is fed back via delay element; this may be unit delay (in case discrete type of network) or fraction delay (in case continuous time networks), to each other neurons in the network. It is also a symmetrically weighted network, since the weights on the link from one node to another are the same in the both directions.

Because recurrent networks have feedback paths from their outputs back to their inputs, the response of such network is dynamic, that is, after applying a new input, the output calculated and fed back to modify the input. The output is then recalculated, and the process is repeated again and again. For a stable network, successive iterations produce smaller and smaller output changes until eventually the outputs become constant. For many networks, the process never ends, and such networks are said to be unstable.

Inputs to the network are applied to all nodes at once, and consist of a set of starting values, +1, or –1. The network is then left alone, and it proceeds to cycle through a succession of states, until it converges on a stable solution, happens when the values of the nodes no longer alter. The output of the network is taken to the value of all the nodes, when the network has reached a stable steady state. This state is known as equilibrium state. There are two types of Hopfield networks namely continuous-time Hopfield networks and discrete-time Hopfield networks.

12.2.0 Continuous-time Hopfield Networks

To study the dynamics of the continuous-time Hopfield networks, we use the neuron additive activation dynamics of the neuron discussed in unit 5. It involves differential operators and is known continuous-time single layer feedback networks. It is also called gradient type networks. Consider $v_i(t)$ be the field (activation) of the i^{th} neuron and its output can be written as, $x_i(t) = f_i(v_i(t))$. Then the additive activation dynamics of the i^{th} neuron can be written in the following form:

$$C_j \frac{d}{dt} v_j(t) = \frac{-v_j(t)}{R_j} + \sum_{i=1}^{n} w_{ji} f_i(v_i(t)) + I_j \qquad i = 1, 2, \ldots n \qquad (12.1)$$

In order to proceed further in constructing the network, we make the following assumptions

1. The weight matrix is symmetric, i.e.

$$w_{ji} = w_{ij} \qquad \text{for all i and j} \qquad (12.2)$$

2. Each neuron has a nonlinear activation of its own, hence the use $f_i(\cdot)$ in equation (12.1)

3. If the inverse of the nonlinear activation function exists, then we may write

$$v = f_i^{-1}(x) \qquad (12.3)$$

Let the sigmoid function $f_i(v)$ be defined by the hyperbolic tangent function

$$x = f_i(v) = \tanh\left(\frac{a_i v}{2}\right) = \frac{1 - \exp(-a_i v)}{1 + \exp(a_i v)} \qquad (12.4)$$

which has a slope of $(a_i/2)$ at the origin as shown by $\frac{a_i}{2} = \left.\frac{df_i(.)}{dv}\right|_{v=0}$ (12.5a)

where a_i is the gain of neuron i, and the inverse output-input relation of (12.3) may thus be rewritten in the form $v = f_i^{-1}(x) = \frac{-1}{a_i}\log\left(\frac{1-x}{1+x}\right)$ (12.5b)

The standard form of the inverse output-input relation for a neuron of unit gain is defined by $f^{-1}(x) = -\log\left(\frac{1-x}{1+x}\right)$ (12.5c)

We may rewrite equation (12.5b) in terms of this as $f_i^{-1}(x) = \frac{1}{a_i} f^{-1}(x)$ (12.5d)

(a) (b)

Fig. 12.2 (a) the standard sigmoid nonlinearity (b) Inverse of (a), $v=g(x)=f^{-1}(x)$.

Let us rewrite the dynamic equation as

$$C_j \frac{d}{dt} v_j(t) = \frac{-v_j(t)}{R_j} + \sum_{i=1}^{n} w_{ji}\, x_i(t) + I_j \qquad (12.5e)$$

$$x_i(t) = f_i\left(v_i(t)\right) \qquad (12.5f)$$

The above equations can be expressed in matrix vector form as

$$C\frac{dV(t)}{dt} = WX(t) - GV(t) + I \qquad (12.6a)$$

$$X(t) = F\left(V(t)\right) \qquad (12.6b)$$

The final equations of the entire model network consist of the state equation (12.6a) and the output equation (12.6b) written in matrix form. Let us study the stability of the system described by the ordinary nonlinear differential equations (12.6). It is evaluated using a generalized computational energy function E[X(t)] and is discussed in the next section.

12.2.1 Stability of continuous-time Hopfield Networks

In order to evaluate the stability of the network first we need to construct energy function E(X(t)). Let us assume that, the net input to a neuron is the potential, which is formulated by the currents from the other nodes and the external input such as from current source and also the connection strengths equivalent to the conductance. Then we can define the energy function in terms of local potential, currents from other nodes and the current from the external source. With the help of dynamic equations (12.1), the energy function of the Hopfield network is defined by

$$E(X(t)) = \frac{-1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ji}\, x_i\, x_j + \sum_{j=1}^{n}\frac{1}{R_j}\int_0^{x_j} f_j^{-1}(z)\, dz \; - \; \sum_{j=1}^{n} i_j\, x_j \qquad (12.7)$$

$$= \frac{-1}{2} X^T(t)\, WX(t) - X^T(t)\, I + \sum_{j=1}^{n}\frac{1}{R_j}\int_0^{x_j} f_j^{-1}(z)dz$$

The above computational energy function E(X(t)) is defined in n-dimensional input space x^n (where the designed neural system specifications are usually known). The corresponding energy function in the state space v^n of an asymptotically stable, the function would be the system's Liapunov function. In order to say that, the network is stable, and then it should satisfy the Liapunov stability conditions. Let us prove that, the energy function of Hopfield network is stable.

The time derivation of E(X(t)) can easily be obtained using the chain rule as

$$\frac{dE(X(t))}{dt} = \sum_{i=1}^{n}\frac{\partial E(X(t))}{\partial x_i(t)}\,\dot{x}_i(t) \qquad (12.8a)$$

where $\dot{x}_i = \frac{d}{dt}x_i(t)$

This may be written as $\frac{d}{dt}E(X(t)) = \nabla E^T(X(t))\dot{X}(t)$ (12.8b)

where $\nabla E(X(t))$ denotes the gradient vector ,

$$\nabla E(X(t)) = \begin{bmatrix} \frac{\partial E(X(t))}{\partial x_1} \\ \frac{\partial E(X(t)}{\partial x_2} \\ . \\ . \\ \frac{E(X(t))}{\partial x_n} \end{bmatrix}$$

Differentiating E[X(t)] with respect to t, we get

$$\frac{dE}{dt} = -\sum_{j=1}^{n}\left(\sum_{i=1}^{n} w_{ji}\, x_i - \frac{v_j}{R_j} + I_j\right)\frac{dx_j}{dt} \qquad (12.9)$$

But we have from equation (12.5e)

$$\sum_{i=1}^{n} w_{ji}\, x_i - \frac{v_j}{R_j} + I_j = C_j \frac{dv_j}{dt}\ , \quad \therefore \frac{dE}{dt} = -\sum_{j=1}^{n} C_j \left(\frac{dv_j}{dt}\right)\frac{dx_j}{dt} \qquad (12.10)$$

We can write v_j in terms of x_j, then we can get

$$\frac{dE}{dt} = -\sum_{j=1}^{n} C_j \left[\frac{d}{dt} f_j^{-1}(x_j)\right]\frac{dx_j}{dt} = -\sum_{j=1}^{n} C_j \left(\frac{dx_j}{dt}\right)^2 \left[\frac{d}{dx_j} f^{-1}(x_j)\right] \qquad (12.11)$$

From Fig. (12.2b), it shows that the inverse output-input relation $f_j^{-1}(x_j)$ is a monotonically increasing function of the output x_j.

$$\frac{d}{dx_j} f_j^{-1}(x_j) \geq 0 \quad \text{for all } x_j \qquad (12.12)$$

Also $$\left(\frac{dx_j}{dt}\right)^2 \geq 0 \quad \text{for all } x_j \qquad (12.13)$$

Hence, all the factors in sum are positive. Therefore, for the energy function E defined in equation (12.7), we have $\frac{dE}{dt} \leq 0$, therefore it is bounded. From this, we can observe the following two statements:

1. The energy function E is a Liapunov function of the continuous Hopfield model.
2. The model is stable according to the Liapunov stability theorem.

Also, the time evolution of the continuous Hopfield model is described by system of nonlinear first-order differential equations (12.6) represents a trajectory in state space, which seeks out the minima of the energy function E and comes to a stop at fixed point.

The derivative $\frac{dE}{dt}$ vanishes only if $\frac{dx_j}{dt} = 0$ for all j and

also we can write $\frac{dE}{dt} < 0$ except at a fixed point (12.14)

The above equations provide the basis for the following theorem [1]. The energy (Lyapunov) function E of a Hopfield network is a monotonically decreasing as function of time. Accordingly, the Hopfield network is globally asymptotically stable; the attractor fixed-points are the minima of the energy function, and vice versa.

12.3.0 Discrete-Time Hopfield Networks

Consider the single-layer feedback neural network shown in Fig. 12.3. It consists of n neurons having threshold values, θ_i. The total input to i^{th} neuron can be expressed as

$$v_i = net_i = \sum_{\substack{j=1 \\ j\neq 1}}^{n} w_{ij}\, x_j + i_i - \theta_i \qquad \text{for } i = 1, 2, \ldots n \qquad (12.15)$$

where i_i is the external input, w_{ij} is weight value connecting the output of the j^{th} neuron with the input of the i^{th} neuron. Equation (12.15) also can be written in vector form as

$$net_i = W_i\, X + i_i - \theta_i \qquad \text{for } i = 1,2,\ldots n. \qquad (12.26)$$

where $W_i = \begin{bmatrix} w_{i1} \\ w_{i2} \\ . \\ . \\ w_{in} \end{bmatrix}^T$; $X = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix}$

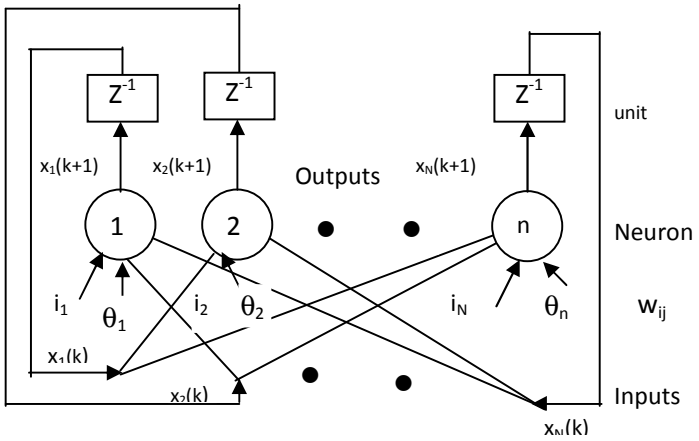


Fig.12.3 Discrete Hopfield Network

The complete matrix description of the above equation (12.16) can be written as

$$NET = WX + I - \theta \qquad (12.17)$$

where, $NET = \begin{bmatrix} net_1 \\ net_2 \\ . \\ . \\ net_n \end{bmatrix}$; $I = \begin{bmatrix} i_1 \\ i_2 \\ . \\ . \\ i_n \end{bmatrix}$; $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ . \\ . \\ \theta_n \end{bmatrix}$ and $W = \begin{bmatrix} W_1 \\ W_2 \\ . \\ . \\ W_n \end{bmatrix} = \begin{bmatrix} 0 & w_{12} & w_{13} & \ldots & w_{1n} \\ w_{21} & 0 & w_{23} & \ldots & w_{2n} \\ w_{31} & w_{32} & 0 & \ldots & w_{3n} \\ . & . & . & . & . \\ . & . & . & . & . \\ w_{n1} & w_{n2} & w_{n3} & \ldots & 0 \end{bmatrix}$

Also assume that, the weight matrix W is symmetrical, i..e $w_{ij}=w_{ji}$ and with the diagonal elements are zero. Physically, this is equivalent to, there is no self feedback in nonlinear dynamical system of Fig.12.3. Let us assume that the neuron's activation function is sgn(·). Then the output or update of each, i^{th} neuron can written as

$$\left.\begin{array}{ll} x_i \to -1 & \text{if } \text{net}_i < 0 \\ x_i \to +1 & \text{if } \text{net}_i > 0 \end{array}\right\} \qquad (12.18)$$

Transitions indicated by right arrows based on the update rule (12.18) are taking place at certain time. That is, if the total input to a particular neuron gathered additively as a weighted sum of outputs plus the external input applied is below the neurons threshold (θ_i), the neuron will have to move to, or remain in, the inhibited state. The rule (12.18) for updating the output of the i^{th} neuron is applied in an asynchronous fashion. This means that for a given time, only a single neuron is allowed to update its output, and only one entry in vector X is allowed to change. Therefore the update algorithm (12.18) for a discrete-time recurrent network can be written as

$$x_i^{k+1} = \text{sgn}\left(W_i X^k + i_i - \theta_i\right) \qquad \text{for } i = 1, 2, \ldots n \qquad (12.19a)$$

where superscript k denotes the index of recursive update. The update scheme in (12.19a) is understood to be asynchronous, thus taking place only for one value of i at a time. The update superscript of x_i^{k+1} refers to the discrete time instant k+1, it also can be written as $x_i[(k+1)T]$, where T denotes the neurons update interval. The recursion starts at x_i^o, which is the output vector corresponding to the initial pattern submitted. The first iteration for k=1, results in x_i^1 where the neuron number i is random. The other updates are also for random node number j, resulting in updates $x_j^1,\ j \neq i$ until all update elements of the vector X^1 are obtained based on X^0. This particular update algorithm is referred to as an asynchronous stochastic recursion of the Hopfield model network. The alternative expressions for equation (12. 19a) in matrix form can be written as

$$X^{k+1} = F\left(WX^k + I - \theta\right) \qquad \text{for } k = 0, 1, 2, \ldots \qquad (12.19b)$$

The discrete Hopfield model also called as a Content-Addressable memory. To evaluate the stability property of the dynamical system, let us consider the computational energy function.

12.3.1 Stability of Discrete -time Hopfield Networks

The scalar-valued energy function, has the matrix form

$$E = \frac{-1}{2} X^T\ WX - I^T\ X + \theta^T X \qquad (12.20a)$$

The expanded form can be written as $E = \frac{-1}{2} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i \neq j}}^{n} w_{ij}\ x_i\ x_j - \sum_{i=1}^{n} i_i\ x_i + \sum_{i=1}^{n} \theta_i\ x_i$ (12.20b)

Comparing the gradient vector, we obtain from (12.26a)

$$\nabla E = \frac{-1}{2}\left(W^T + W\right)X - I^T + \theta^T \qquad (12.21a)$$

which reduces for symmetrical matrix W for which $W^T = W$ to the form

$$\nabla E = -WX - I^T + \theta^T \qquad (12.21b)$$

The energy increment becomes equal. $\Delta E = (\nabla E)\,\Delta X$ (12.21c)

Since only the i^{th} output is updated, we have $\Delta X = \begin{bmatrix} 0 \\ . \\ \Delta x_i \\ . \\ 0 \end{bmatrix}$

and the energy increment (12.21c) reduces to the form

$$\Delta E = \left(-W_i^T X - I_i^T + \theta_i\right)\Delta x_i \qquad \text{for } j \neq i \qquad (12.21d)$$

This can be rewritten as $\Delta E = -\left(\sum_{j=1}^{n} w_{ij}\, x_j + i_i - \theta_i\right)\Delta x_i, \text{ for } j \neq i$ (12.21e)

or briefly $\Delta E = -\,net_i\, \Delta x_i$

Inspecting the update value (12.18), we see that the expression in parameters in (12.21d) and (12.21e), which is equal to net_i, and the current update values of the output node, Δx_i , relate as follows :

$$\left.\begin{array}{lll} \text{i)} & \text{when } net_i < 0, & \text{then } \Delta x_i \leq 0 \text{ and} \\ \text{ii)} & \text{when } net_i > 0, & \text{then } \Delta x_i \geq 0 \end{array}\right\} \qquad (12.22)$$

From the above relations, we can state that under the update algorithm discussed, the product term $net_i\, \Delta x_i$ is always nonnegative. Thus, any corresponding energy changes ΔE in (12.21d) and (12.21e) is non-positive. We therefore can conclude that the neural network undergoing transitions will either decrease or retain its energy E as a result of each individual update. The non-increasing property of the energy function E is valid only when $w_{ij}=w_{ji}$. Otherwise, the proof of non-positive energy increments does not hold entirely.

12.3.2 Relation between the discrete and continuous versions of the Hopfield Network

The Hopfield network may be operated in a continuous mode or discrete mode, depending on the model adopted for describing the neuron. The continuous mode of operation is based on an additive model, where as the discrete mode of

operation is based on the McCulloch Pitts model. We can establish relationship between the stable states of the continuous Hopfield model and those of the corresponding discrete Hopfield model by defining the input-output relation for a neuron such that we may satisfy two simplifying characteristics:

1.The output of a neuron has the asymptotic values

$$x_j = \begin{cases} +1 & \text{for } v_j = \infty \\ -1 & \text{for } v_j = -\infty \end{cases} \qquad \textbf{(12.23)}$$

2.The midpoint of the activation function of neuron lies at the origin, as shown by

$$f_j(0) = 0 \qquad (12.24)$$

Correspondingly, we may set the bias θ_j equal to zero for all j. In formulating the energy function E for a continuous Hopfield model, the neurons are permitted to have self-loops. A discrete model on the other hand need not have self-loops i..e $w_{ji} = 0$; for all in both models. From the above observation, we may redefine the energy function of a continuous Hopfield model give in equation (12.7) as follows:

$$E = \frac{-1}{2} \sum_{\substack{i=1 \\ i \neq j}}^{n} \sum_{j=1}^{n} w_{ji}\, x_i\, x_j + \sum_{j=1}^{n} \frac{1}{R_j} \int_0^{x_j} f_j^{-1}(z)dz \qquad (12.25)$$

we have the relation for inverse function $f_j^{-1}(x)$ as $f_j^{-1}(x) = \frac{1}{a_j} f^{-1}(x)$

By using above relation and in equation (12.25) we can rewrite as

$$E = \frac{-1}{2} \sum_{\substack{i=1 \\ i \neq j}}^{n} \sum_{j=1}^{n} w_{ji}\, x_i\, x_j + \sum_{j=1}^{n} \frac{1}{a_j R_j} \int_0^{x_j} f^{-1}(z)dz \qquad (12.26)$$

The integral $\int_0^{x_j} f^{-1}(z)dz$ has the standard form plotted in Fig.12.4.

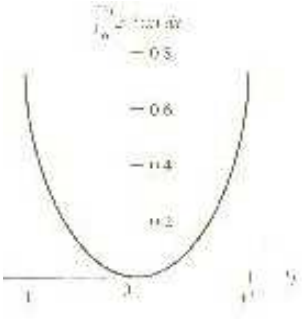


Fig. 12.4

Its value is zero for x_j=0 and positive otherwise. It assumes a very large value as x_j approaches ± 1. If, however, the gain a_j of neuron j become infinitely large (i.e., the

sigmoidal nonlinearity approaches the idealized hard-limiting form), the second term of equation (12.26) becomes negligibly small. In the limiting case when $a_j = \infty$ for all j, the maxima and minima of the continuous Hopfield model become identical with those of the corresponding discrete Hopfield model. In the later case, the energy (Lyapunov) function is defined simply by

$$E = \frac{-1}{2} \sum_{\substack{i=1 \\ i \neq j}}^{n} \sum_{j=1}^{n} w_{ji}\, x_i\, x_j \qquad (12.27)$$

where the j^{th} neuron state $x_j \pm 1$. We conclude, therefore, that the only stable points of the very high-gain, continuous, deterministic Hopfield model correspond to the stable points of the discrete stochastic Hopfield model.

12.4.0 The Energy Landscape

The energy landscape has hollows that represent the patterns stored in the network. An unknown input pattern represents a particular point in the energy landscape, and as the network iterates its way to a solution, the point moves through the landscape towards one of the hollows. These basins of attraction represent the stable states of the network. The solution from the net occurs when the point moves into the lowest region of the basin; from there, energy where else in the close vicinity is uphill, and so it will stay where it is.

The energy function for the perceptron was $E = -\frac{1}{2}\sum\left(t_{pj} - O_j\right)^2$, but it depends on knowledge of the required output as well as the actual output of the net. For the Hopfiled net, which steps its way towards a solution the required intermediate steps are not known, and we therefore need something more applicable to this architecture. The weight values in the network must affect the energy, as must the actual patterns presented, so the energy function must reflect these requirements. We can identify a suitable energy function for the Hopfild net as

$$E = -\frac{1}{2}\sum_i \sum_{j \neq i} w_{ij} x_i x_j + \sum_i x_i \theta_i \qquad (12.28)$$

where w_{ij} represents the weight between node i and node j of the network and x_i represents the output from node i. The threshold value of node i is represented as θ_i. As the output is fed back into the net, the outputs at any one time represents the next set of inputs, and so both the weights and the inputs are explicitly represented as required. The weights in the network contain the pattern information, and so all the patterns are included in this energy function. Nodes are not connected directly to themselves, and so the terms w_{ii} are zero. Since the connections are symmetric $w_{ij} = w_{ji}$

If we make our patterns occupy the low points in the energy landscape, then we can perform gradient descent on the energy function in order to end up in one of these minima, which will represent our solution.

12.5.0 Storing the patterns in the Hopfield Network

Storing the pattern is nothing but training of the network. To store a pattern, we need to minimize the value of the energy function for that particular pattern so that it occupies a minimum point in the energy landscape. However we also want to leave any previous stored pattern in their hollows, so that adding new patterns doesn't destroy all the previous information. The weight matrix contains the information about the stored patterns. Let us find an expression for the weight values that will produce a minimum in the energy function.

Considering this in terms of the energy function, $E = -\frac{1}{2}\sum_i \sum_{j \neq i} w_{ij} x_i x_j + \sum_i x_i \theta_i$

to minimize for a particular pattern S that has set of input elements (x_0, x_1, ...x_{n-1}).

We want each term to be negative, and so we require $\sum x_i \theta_i$ to be negative. This can be achieved by setting θ_i to the opposite sign of x_i for a particular pattern. However, a different pattern will have different values of x_i and then the threshold term may well increase the value of E. In order to avoid this, the best that we can do is to set the threshold to zero.

We write x_i^s to mean element i of input pattern S, which can be either +1 or –1. Now, w_{ij} is the weight between nodes i and j as before, and contains the pattern information from all the taught patterns. This means that we can split the weight matrix into two parts, one which represents the effects of all the pattern except the S^{th} one, denoted by $w_{ij}^{'}$ and a second which is the contribution made by the s^{th} pattern alone, shown as w_{ij}^s.

$$\therefore E = -\frac{1}{2}\sum_i \sum_{j \neq i} w_{ij}^1 x_i x_j - \frac{1}{2}\sum_i \sum_{j \neq i} w_{ij}^s x_i^s x_j^s \quad = E_{\text{all except, s}} + E_s \qquad (12.29)$$

This can be thought of as viewing the energy as a "signal" plus a "node" term; the "signal" as the energy due to the pattern S, whilst the "noise" is due to the contributions from all the other patterns. Storing this pattern corresponds to making the energy function as small as possible. To store pattern S, we want to minimize the contribution to the energy function from S^{th} pattern energy term, and so make

$$E_s = -\frac{1}{2}\sum_i \sum_{j \neq i} w_{ij}^s x_i^s x_j^s \qquad (12.30)$$

as small as possible. This corresponds to making $\sum_i \sum_{j \neq i} w_{ij}^s x_i^s x_j^s$ as large as possible, due to the minus sign. Now, the elements in x_i are either +1 or –1; however, x_i^2 is always positive, so if we make the energy term depend on $x_i^2 x_j^2$ it will always be positive, and so the sum will be as large as possible. We write this most simply by equating

$$\sum_i \sum_{j \neq i} w_{ij}^s x_i x_j = \sum_i \sum_{j \neq i} x_i^2 x_j^2$$

and noticing that all we have to do is to make the weight term $w_{ij}^s = x_i x_j$

This means that we have our result; setting the values of the weights $w_{ij}^s = x_i x_j$ minimizes the energy function for patterns. This relation is equivalent to that of Hebbian learning principle. The Hopfield has no alterative learning algorithm as such; patterns are so stored are simply by lowering their energies. The network has no hidden units, and so is unable to encode the data. With this we can conclude that, the storing of the pattern in Hopfield network can be done using Hebbian learning principle. That is, the storing of the patterns in Hopfield network is nothing but learning of the network. Suppose that we wish to store a set of n-dimensional vectors denoted $\{x_k|_{k=1,2,\quad M}\}$. We call these M vectors fundamental memories, representing the patterns to be memorized by the network. According to the outer product rule of storage, that is generalization of Hebb's postulate of learning, the synaptic weights of the network can be computed as

$$w_{ij} = \begin{cases} \dfrac{1}{N} \displaystyle\sum_{\mu=1}^{M} x_{\mu,j} x_{\mu,i} & i \neq j \\ 0 & i = j, \end{cases}$$

where w_{ij} is the connection weight between node i and node j, and element of the exemplar pattern x_μ equal to either +1 or –1. The thresholds of the units are assumed to be zero.

12.6.0 Recalling the Patterns from Hopfield Network

Having stored our patterns in the net, we now need to be able to recall them. This can be accomplished if we perform gradient descent on our energy function. Considering our energy function in equation (12.28) we need to calculate the contribution that a particular nodes value makes to the energy, and then we can cycle around the net, reducing each nodes contribution until the energy value is at a minimum.

We can express the energy function in two parts, splitting off the contribution made by the k^{th} node.

$$E = -\frac{1}{2} \sum_{i \neq k} \sum_{j \neq k} w_{ij} x_i x_j + \sum_{i \neq k} x_i \theta_i - \frac{1}{2} x_k \sum_j x_j w_{kj} - \frac{1}{2} x_k \sum_i x_i w_{ik} + x_k \theta_k \quad (12.31)$$

The k^{th} neuron changes output state from x_{k1} to x_{k2}.

$\therefore$ The difference in energy $\Delta E = E_2 - E_1$ caused by the state change $\Delta x_k = x_{k2} - x_{k1}$

$$\therefore \Delta E = -\frac{1}{2} \left[\Delta x_k \sum_j x_j w_{kj} + \Delta x_k \sum_i x_i w_{ik} \right] + \Delta x_k \theta_k \qquad (12.32)$$

Since the matrix w_{2j} is symmetric, we can interchange the indices as simplify the expression to $\Delta E = -\Delta x_k \left[\sum_j x_j w_{kj} - \theta_k \right]$. The term $\sum_j x_j w_{kj}$ is the weighted sum of the inputs to node k, and θ_k is the threshold of unit k. Now, the threshold of every node was set to zero in the storage phase, in order to ensure that the patterns occupied the minima in the energy function. Remembering that the nodes output is either a +1 or –1, decreasing ΔE_k will mean outputting a +1 of the weighted sum is greater than zero, and –1 of it is less than zero, since this will always serve to reduce the value of ΔE_k.

$$\sum_{j \neq k} w_{ij} x_i \begin{cases} > 0 & x_i \rightarrow +1 \\ = 0 & \text{remains in previous state} \\ < 0 & x_i \rightarrow -1 \end{cases}$$

We can see that the update function performs this operation, and so implementing gradient descent in E. This allows us to recall our patterns from the net by cycling through a succession of states, each of which has a lower energy than the previous one. This relaxation in lower energy is continuous until a steady state of low energy is reached, when the net has found its way into a minimum and so produced the pattern.

12.7.0 Summary of the Hopfield Algorithms

1. **Learning** : Suppose that we wish to store a set of N-dimensional vectors denoted $\{\xi_k |_{k=1,2,\ M}\}$. We call these M vectors fundamental memories, representing the patterns to be memorized by the network. According to the outer product rule of storage, that is generalization of Hebb's postulate of learning, the synaptic weights of the network can be computed as

$$w_{ij} = \begin{cases} \frac{1}{N} \sum_{\mu=1}^{M} \xi_{\mu,j} \xi_{\mu,i} & i \neq j \\ 0 & i = j, \end{cases}$$

Where w_{ij} is the connection weight between node i and node j, and element of the exemplar pattern ξ_μ equal to either +1 or –1. The thresholds of the units are assumed to be zero.

2. **Initialization** :Let ξ_{probe} denote an unknown N-dimensional input vector (probe vector) presented to the network. The algorithm initialized by setting

$$x_j(0) = \xi_{j,\,probe}\ ,\ j=1,2,\ldots N$$

where $x_j(0)$ is the state of neuron j at time *l*=0 and $\xi_{j,\,probe}$ is the j^{th} element of the probe vector ξ_{probe}.

3. **Iterate until convergence**

$$x_j(l+1) = f_h\left[\sum_{i=1}^{N} w_{ij} x_i(l)\right] \qquad 0 \le j \le N$$

The function f_h is the hard liming non-linearity, or step function. Repeat the iteration until the outputs from the nodes remain unchanged.

4 **Outputting**

Let X_{fixed} denote the fixed point (stable state) computed at the end of step 3. The resulting output vector O of the network is $O = X_{fixed}$

Step 1 is the storage phase and Step 2 through 4 constitutes the retrieval phase.

The weights between the neurons are set using the equation given in the algorithm, from exemplar patterns of all classes. This is the teaching stage of the algorithm, and associates each pattern with itself. The recognition stage occurs when the output of the net is forced to match that of a composed unknown pattern at time zero. The net is then allowed to iterate freely in discrete time steps, until it reaches a stable situation when the output remains unchanged, the net thus converges on the solution.

The autoassociation of patterns should mean that the presentation of a corrupt input pattern will result in the reproduction of the perfect pattern as the output of the network therefore acts as a content-addressable memory.

Example: For the following given vectors, construct the weight matrix for the associate Hopfield network. $X_1 = (1, -1, -1)^T$, $X_2 = (-1, 1, -1)^T$ and $X_3 = (-1, -1, 1)^T$. Recall the fundamental vector stored in network by using X_1 as probe vector.

For the associate Hopfield net, let us use the outer products to construct the weight matrix as follows: $$W = \sum_i \left(X_i X_i^T - I_n\right)$$

where X_i is the n-dimensional bipolar (-1 and 1) vector to be stored in the net and I_n is an n×n identity matrix.

Given Compute the weight matrix.

$$W = \left(X_1 X_1^T - I_3\right) + \left(X_2 X_2^T - I_3\right) + \left(X_3 X_3^T - I_3\right)$$

$$= \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}^{[-1 \quad -1 \quad -1]} + \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}^{-[1 \quad 1 \quad -1]} + \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}^{[-1 \quad -1 \quad 1]} - \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 0 & -1 & -1 \\ -1 & 0 & -1 \\ -1 & -1 & 0 \end{bmatrix}$$

Next, we use X_1 is the probe vector.

$F_h(X_1 W) = F_h([2 \;\; 0 \;\; 0]) = [1 \;\; -1 \;\; -1]$

which is exactly X_1. In other cases, more than one iterations may be necessary for reaching equilibrium. Whether all stored vectors can be retrieved correctly depends on the capacity of the net.

It is also a symmetrically weighted network, since the weights on the link from one node to another are the same in the both directions. Each node has, like the single-layer perceptron, a threshold and a step-function, and the nodes calculate the weighted sum of their inputs minus the threshold value, passing that through the step function to determine their output state. The net takes only 2-state inputs these can be binary (0, 1) or bipolar (-1, +1).

12.8.0 Capacity of the Hopfield Nets

The Hopfield net can be activated either synchronously or asynchronously. In the synchronous modes the weighs are updated simultaneously. In asynchronous operation, the network updates only one neuron at a time. The neuron is selected from n neurons with a probability of 1/n. To study the memory capacity of the Hopfield net, let us define what is a memory in the neural associative network.

If a binary n-dimensional vector X is a memory, then for each component (neuron) i = 1, . . . , n.

$$X_i = F_h\left(\sum_{j=1}^{n} w_{ij} x_j\right)$$

where F_h is the hard limiting function. Intuitively, this means that X_i is a memory if the network is stable at that point. The memory also tends to act as attractor because it will pull states that are similar to map onto the memory. This has the effect of correcting some or all the errors in a probe vector. A probe vector refers to the vector we use to perform association.

Suppose we try to store ***m*** memory vectors X's of a n dimensions. Each component in X is either +1 or –1. The ***m*** vectors are called fundamental memories. The outer product construction method is used to arrive at the appropriate weight matrix W. The weight matrix so constructed grantees that the network will converge to a stable state. Additionally, if ***m*** << n, then the system will behave will.

In some empirical work in his 1982 paper, Hopfield showed that about half the memories were stored accurately in a net of n nodes if ***m***=0.15n. The other patterns did not get stored as stable states. In a more rigorous piece of analysis McCliece et al. (1987) showed theoretically that, if we require almost all the required memories to be stored accurately, then the maximum number of patterns ***m*** is $\frac{n}{2\ln n}$. For n=100 this gives m=11.

It is desirable that fundamental memories are fixed points. In the Hopfield net, a fixed point X satisfies $X = F_h(XW)$. Once the network settles in a fundamental memory, it will not leave. The fundamental memory should have the capability to attract states that are close to it.

- There are three possibilities for convergence.
- The first possibility is direct convergence. In asynchronous operation, the probe vector lies within the basin of attraction and leads directly to the fundamental memory at the center. In the synchronous mode, this means convergence to the memory in one step.
- In second possibility, a random step will be in the correct direction with high probability. After several random steps, the probe vector will converge to the fundamental memory. This process can be thought of as a two-iteration convergence in synchronous operation.
- In third possibility, the probe vector moves back and forth but gets closer to the fundamental memory. After a finite number of iterations, the systems will settle in a fixed point either on or close to the fundamental memory.
- It has been conjectured that the maximum number of memories ***m*** that can be stored in a network of n neurons and recalled exactly is less than Cn^2 where C is a positive constant greater than one. In general, this bound is overly optimistic.
- Hopfield should experimentally that the general capacity limit was about 0.15n.
- If the ***m*** memories are chosen at random and if the input pattern is less than n/2 away in Hamming distance from a stored pattern, the maximum asymptotic value of ***m*** for which all the fundamental memories can be correctly recalled can be shown as $m \leq \frac{n}{4 \ln n}$.

DHM Example (contd...)

- **Example:**
  - A 4 node network, stores 2 patterns (1 1 1 1) and (-1 -1 -1 -1)
  - Weights: $w_{\ell,j} = 1$, for $\ell \neq j$, and $w_{j,j} = 0$ for all j

- Corrupted input pattern: (1 1 1 -1)

Node selection		output pattern
node 2:	$w_{2,1}x_1 + w_{2,3}x_3 + w_{2,4}x_4 + I_2 = 1 + 1 - 1 + 1 = 2$	(1 1 1 -1)
node 4:	$1+1+1-1 = 2$	(1 1 1 1)

No more change of state will occur, the correct pattern is recovered

- Equal distance: (1 1 -1 -1)

node 2:	net = 0, no change	(1 1 -1 -1)
node 3:	net = 0, change state from -1 to 1	(1 1 1 -1)
node 4:	net = 0, change state from -1 to 1	(1 1 1 1)

No more change of state will occur, the correct pattern is recovered

If a different node selection order is used, the stored pattern (-1 -1 -1 -1) may be recalled

DHM Example (contd…)

- Missing input element: (1 0 -1 -1)

Node selection	output pattern
node 2: $w_{12}x_1+w_{32}x_3+w_{42}x_4+I_2 = 1-1-1+0 < 0$	(1 -1 -1 -1)
node 1: net = -3, change state to -1	(-1 -1 -1 -1)

No more change of state will occur, the correct pattern is recovered

- Missing input element: (0 0 0 -1)

  the correct pattern (-1 -1 -1 -1) is recovered

  This is because the AM has only 2 attractors

  (1 1 1 1) and (-1 -1 -1 -1)

  When spurious attractors exist (with more memories), pattern completion may be incorrect

DHM Example (contd…)

- Missing input element: (1 0 -1 -1)

Node selection		output pattern
node 2:	$w_{12}x_1+w_{32}x_3+w_{42}x_4+I_2 = 1-1-1+0 < 0$	(1 -1 -1 -1)
node 1: net = -3, change state to -1		(-1 -1 -1 -1)

No more change of state will occur, the correct pattern is recovered

- Missing input element: (0 0 0 -1)

  the correct pattern (-1 -1 -1 -1) is recovered

  This is because the AM has only 2 attractors

  (1 1 1 1) and (-1 -1 -1 -1)

  When spurious attractors exist (with more memories), pattern completion may be incorrect

Convergence Analysis of DHM

- **Two questions:**
  1. Will Hopfield AM converge (stop) with any given recall input?
  2. Will Hopfield AM converge to the stored pattern that is **closest** to the recall input ?
- Hopfield provides answer to the first question
  - By introducing an **energy function** to this model,
  - No satisfactory answer to the second question so far.
- **Energy function**:
  - Notion in thermo-dynamic physical systems. The system has a tendency to move toward lower energy state.
  - Also known as Lyapunov function. After Lyapunov theorem for the stability of a system of differential equations.

Convergence Comments

Comments:

1. Why converge?
   - Each time, E is either unchanged or decreases by an amount.
   - E is bounded from below.
   - There is a limit to how much E may decrease. After finite number of steps, E will stop decreasing no matter what unit is selected for update
2. The state the system converges to is a stable state.

   Will return to this state after some small perturbation. It is called an **attractor** (with different attraction basin)
3. Error function of BP learning is another example of energy/Lyapunov function. Because
   - It is bounded from below (E>0)
   - It is monotonically non-increasing (W updates along gradient descent of E)

Convergence Example

Example:

- A 4 node network, stores 3 patterns

  (1 1 -1 -1), (1 1 1 1) and (-1 -1 1 1)
- Weights:

$$\begin{pmatrix} 0 & 1 & -1/3 & -1/3 \\ 1 & 0 & -1/3 & -1/3 \\ -1/3 & -1/3 & 0 & 1 \\ -1/3 & -1/3 & 1 & 0 \end{pmatrix}$$

Corrupted input pattern: (-1 -1 -1 -1) (Pattern Recovery)

- If node 4 is selected:

  (-1/3 -1/3 1 0) (-1 -1 -1 -1) + (-1)

  = 1/3 + 1/3 –1 –1 = -4/3,
- No change of state for node 4
- Same for all other nodes, net stabilized at (-1 -1 -1 -1)
- A spurious state/attractor is recalled

Convergence (contd…)

- For input pattern (-1 -1 -1 0) (Pattern Recovery + Completion)
  - If node 4 is selected first,
    (-1/3 -1/3 1 0) (-1 -1 -1 0) + (0)
    = 1/3 + 1/3 – 1 – 0 – 0 = –1/3,
    node 4 changes state to -1, then the same as in the previous example, network will stabilize at (-1 -1 -1 -1)
  - However, for the same input pattern (-1 -1 -1 0), if the node selection sequence is 1, 2, 3, 4, the net will stabilize at state (-1 -1 1 1), a genuine attractor

update				
Node 1	-1	-1	-1	0
Node 2	-1	-1	-1	0
Node 3	-1	-1	1	0
Node 4	-1	-1	1	1
	-1	-1	1	1
	-1	-1	1	1
	-1	-1	1	1
	-1	-1	1	1

Stable

$$W = \begin{pmatrix} 0 & 1 & -1/3 & -1/3 \\ 1 & 0 & -1/3 & -1/3 \\ -1/3 & -1/3 & 0 & 1 \\ -1/3 & -1/3 & 1 & 0 \end{pmatrix}$$

Capacity Analysis of DHM

- ***P***: maximum number of random patterns of dimension ***n*** can be stored in a DHM of ***n*** nodes
- Hopfield's observation: $P \approx 0.15n, \quad \frac{P}{n} \approx 0.15$ ≈15%
- Theoretical analysis:

$$\text{capacity } C = \frac{P}{n} \le \frac{1}{4 \ln n} \approx \frac{1}{2 \log_2 n}$$

P/n decreases because larger n leads to more interference between stored patterns (stronger cross-talks).

- Some work to modify HM to increase its capacity to close to ***n***, *W* is trained (not computed by Hebbian rule).
- Another limitation: full connectivity leads to excessive connections for patterns with large dimensions

Grossberg Layer

The Grossberg layer functions in a familiar manner. Its NET output is the weighted sum of the Kohonen layer outputs k_1, k_2, ...k_n, forming the vector K. The connecting weight vector designated V consists of the weights v_{ij}, v_{2j}, ...v_{nj}. The NET output of each Grossberg neuron is then

$$NET_j = \sum_i k_i v_{ij}$$

where NET_j is the output of Grossberg neuron j, or in vector form

Y = KV

Where

Y = The Grossberg layer output vector

K = The Kohonen layer output vector

V = The Grossberg layer weight matrix.

If the Kohonen layer is operated such that only one neuron's NET is at one and all others are at zero, only one element of the K vector is nonzero, and the calculation is simple. In

fact, the only action of each neuron in the Grossberg layer is to output the value of the weight that connects it to the single nonzero Kohonen neuron.

TRAINING THE GROSSBERG LAYER

The Grossberg layer is relatively simple to train. An input vector is applied, the Kohonen output(s) are established, and the Grossberg outputs are calculated as in normal operation. Next, each weight is adjusted only if it connects to a Kohonen neuron having a nonzeon output. The amount of the weight adjustment is proportional to the difference between the weight and the desired output of the Grossberg neuron to which it connects.

In symbols

$$v_{ij\ \text{new}} = v_{ij\ \text{old}} + \beta\left(y_j - v_{ij\ \text{old}}\right)k_i$$

where

k_i = the output of Kohonen neuron i (only one Kohonen neuron is nonzero).

y_j = component j of the vector of desired outputs.

Initially β is set to approximately 0.1 and is gradually reduced as training progress. From this it may be seen that the weights of the Grossberg layer will converge to the average values of the desired outputs, whereas the weights of the Kohonen layer are trained to the overage values of the inputs.

Grossberg training is supervised; the algorithm has a desired output to which it trains. The unsupervised, self-organizing operation of the Kohonen layer produces outputs at indeterminate positions; these are mapped to the desired outputs by the Grossberg layer.

KOHONEN SELF-ORGANIZING NETWORKS

17.0.0 Introduction: The Self-organizing map (SOM), sometimes referred to as "Kohonen map" due to its invention by Professor Teuvo Kohonen, is an unsupervised learning technique that reduces the dimensionality of data through the use of a self-organizing neural network. A probabilistic version of SOM is the Generative Topographic Map (GTM) of Bishop, Svensen and Williams.

Self-organization systems are considering a special class of artificial neural networks known as self-organizing maps. These networks are based competitive learning, the output of neurons in the network compete among themselves to be activated or fired with the result that only one output neuron, or one neuron per group, is on at any time. An output neuron that wins the competition is called winner-takes –all neuron or simply a winning neuron. In a self-organizing map the neurons are placed at the nodes of a lattice that is usually one-or-two dimensional. Higher dimensional maps are also possible but not as common. The neurons become selectively tuned to various input patterns (stimuli) or classes of input patterns in the course of a competitive learning process. The locations of the neurons so tuned (i.e., the winning neurons) become ordered with respect to each other in such a way that a meaningful coordinate system for different input features is

created over the lattice. A self-organizing map is therefore characterized by the formation of a topographic map of the input patterns in which the spatial locations (i.e., coordinates) of the neurons in the lattice are indicative of intrinsic statistical features contained in the input patterns, hence the name “self-organizing map”.

Counter propagation is combinations of two well know algorithms: the self-organizing of Kohonen and the Grossberg outstar.

It has been postulated that the brain uses spatial mapping to model complex data structure internally. Kohonen uses this idea to get good advantage in his network because it allows him to perform data compression on the vectors to be stored in the network, using a technique known as vector quantization.

Data compression means that multi dimensional data can be represented in a much lower dimensional space. Much of the cerebral cortex is arranged as a two dimensional plane of interconnected neurons but it is able to deal with concepts in much higher dimensions. The implementations of Kohonen’s algorithm are also predominantly two-dimensional.

A typical network showed in Fig. (1) is a one layer two-dimensional Kohonen network. The most obvious point to note is that the neurons are not arranged in layers as in the multiplayer perception (input, hidden, output) but rather on a flat grid. All inputs connect to every node in the network. Feedback is restricted to lateral interconnections to immediate neighboring nodes. Note that there is no separate output layer each of the nodes in the grid is itself an output node.

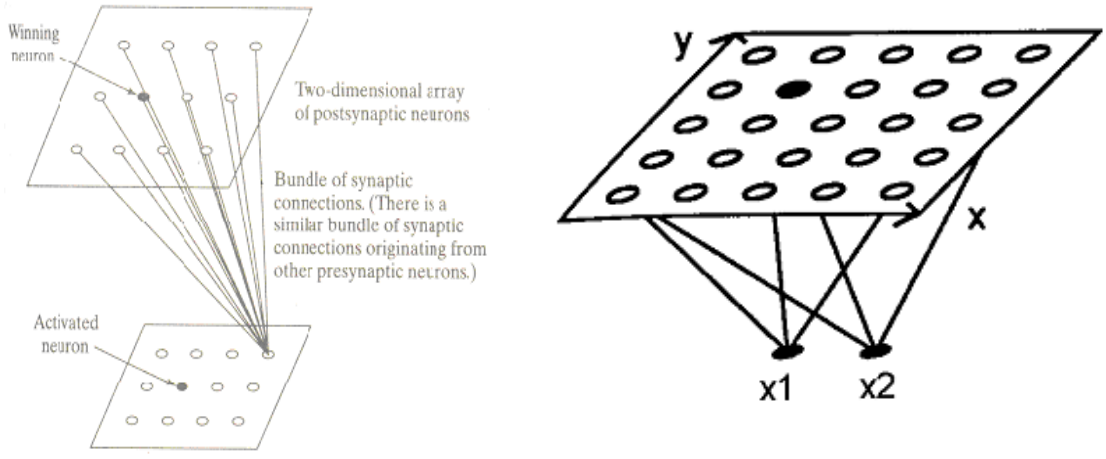


Fig. (1) A Kohonen feature map note that there is only one layer of neurons and all inputs are connected to all nodes.

Neighbourhoods

This idea is an introduced by Kohonen and is a dynamically changing boundary that defines how many nodes surrounding the winning node will be effected with weight modifications during the training process. Initially each node in the network will be assigned a large neighborhood (where “large” can imply every node in the network).

When a node is selected as the closest match to an input it will have its weights adaptive to tune it to the input signal. However, all the nodes in the neighborhood will also be adapted by a similar amount. As training processes the size of the neighbourhood is slowly decreased to a predefined limit. The key notion is the neighborhood and that forms the topology of the map/output. The most common topologies are rectangular and hexagonal.

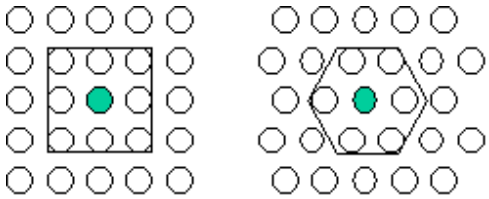


Fig. 17.2

A parameter decides the radius R of the grid. The picture above indicates R = 1. Then when an output unit i becomes the winner, all of its neighbors (including i itself) update the weights.

The activation of an output unit is usually calculated as the Euclidean distance from the input vector.

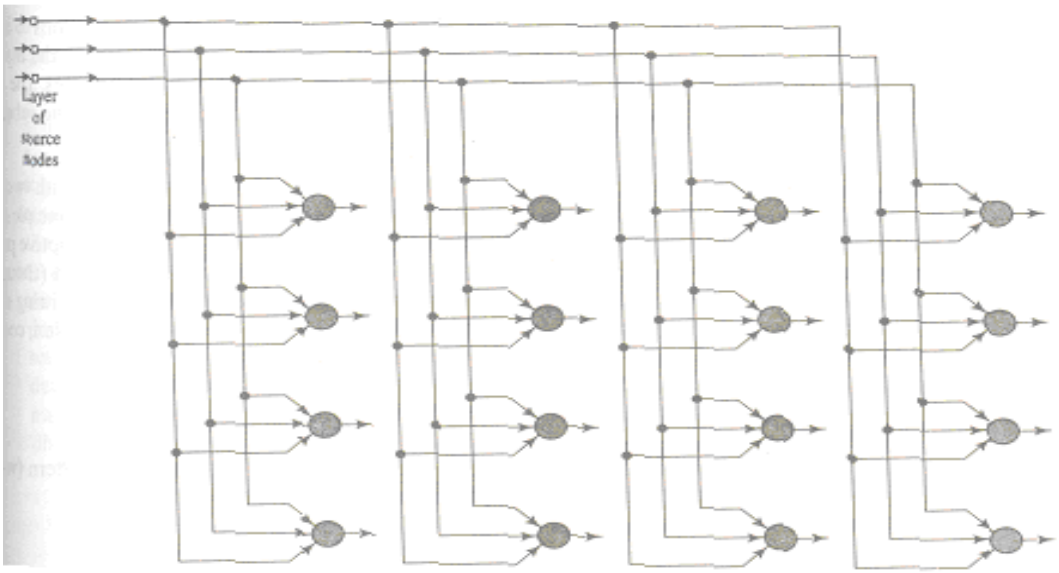
$$y_j = \sum_i (w_{ji} - x_i)^2$$

Algorithm : In order to use the neighborhood concept the following step are used to update the weights.

1. (a) Initialize weights w_{ij}.
   (b) Set topological neighborhood parameter R.
   (c) Set the learning rate eta.
   1. Do while the terminating condition is false,
   2. For each input pattern x, do
   3. For each output unit y,
   4. Compute the activation.
   5. Find the output unit j that has the **minimum** activation.
   6. For all unit k which is in the neighborhood of j, do
   7. Update all weights connected from k
   8. (Possibly) reduce the radius of neighborhood.
9. (Possibly) update the learning rate.

And the weight update for a neighbor unit k is

$$w_{ki} \leftarrow w_{ki} + \eta \cdot [x_i - w_{ki}]$$



Kohonen Layer

In its simplest form, the Kohonen layer functions in a "Winner-Take-all fashion' that is, for a given input vector, one and only one Kohonen neuron outputs a logical one; all other output a zero. One can think of the Kohonen neurons as a series of light bulbs, only one of which comes on for a given input vector. Associated with each Kohonen is a set of weighs connecting it to each input.

Let the input signal in input layers are x_1, x_2, ...x_m; comprising the input weights w_{1j}, w_{2j}, . . .w_{mj} comprising a weight vector w_1.

As with neurons in most networks, the NET output of each Kohonen neuron as simple the summation of the weighted inputs. This may be express as follows:

$$NET_j = w_{1j}\, x_1 + w_{2j}\, x_2 + \ldots\ldots + w_{mj}\, x_m \qquad (1)$$

Where NET_j is the NET output of Kohone neuron j

$$NET_j = \sum_i x_i w_{ij} \qquad (2)$$

or in vector notation

$$N = XW \qquad (3)$$

Where N is the vector of Kohonen layer NET outputs.

The Kohonen neuron with the largest NET value is the "Winner". Its output is set to one; all others are set zero.

The Kohonen Algorithm

The learning algorithm organizes the nodes in the grid into local neighborhoods that act as feature classifies on the input data. The topographic map is autonomously organized by a cyclic process of comparing input patterns to vectors "stored" at each node. No

training response is specified for any training input. Where inputs match the node vectors, that area of the map is selectively optimized to represent an average of the training data for that class. From a randomly organized set of nodes the grid settles into a feature map that has local representation and its self-organized. The algorithm itself is rotationally very simple.

Kohonen training is self-organizing algorithm that operates on the unsupervised node. For this reason, it as difficult (an unnecessary) predict which specific Kohonen neuron well be activated for a given input vector. It is only necessary to ensure that training separates dissimilar input vectors.

Preprocessing the Input vectors

It is highly desirable (but not mandatory) to normalize all input vectors before applying them to the network. This is done by dividing each component of an input vector by that vectors length. This length is found by taking the square root of the sum of the squares of all of the vectors components. In symbols

$$x_i^1 = x_i / \left(x_1^2 + x_2^2 + \text{----} + \text{x}_\text{m}^2\right)^{1/2} \qquad (4)$$

This converters an input vector into a unit vector pointing in the same direction; that is, a vector of unit length in n-dimensional space.

To train the Kohonen layer, an input vector is applied and its dot product is calculated with the weight vector associated with each Kohonen neuron. The neuron with the highest dot product is declared the **"winner"** and its weights are adjusted. Because the dot product operation used to calculate the NET values is a measure of similarity between the input and weight vectors, the training procedure actually consists of selecting he Kohonen neuron whose weight vector is most similar to the input vector, and making it still more similar. Note again that is unsupervised training there is no teacher. The network self-organises so that a given Kohonen neuron has maximum output for a given input vector. The training equation that follows as used

$$W_{new} = W_{old} + \alpha (x - W_{old}) \qquad (5)$$

Where

W_{new} = the new value of a weight connecting an input component x to the winning neuron.

W_{old} = the previous value of this weight

α = a training rate coefficient that may vary during the training process

Each weight associated with the winning Kohonen neuron is changed by an amount proportional to the difference between its value and the value of the input to which it connections. The direction of the change minimizes the difference between the weight and its input.

The variable α is a training rate coefficient that usually starts out at about 0.7 and may be gradually reduced during training. This allows large initial steps for rapid, coarse training and smaller steps as the final value is approached.

If only one input vector were to be associated with each Kohonen neuron, the Kohonen layer could be trained with a single calculation per weight. The weights of a winning neuron would be made equal to the components of the training vector ($\alpha = 1$).

Usually the training set includes many input vectors that are similar, and the network should be trained to activate the same Kohonen neuron for each of them. In this case, the weights of that neuron should be the average of the input vectors that will activate it. Setting α to a low value will reduce the effect of each training step, making the final value an average of the input vectors to which it was trained. In this way, the weights associated with a neuron will assume a value near the "center" of the input vectors for which that neuron is the "winner".

Initializing the weight vectors

For Kohonen training, randomized weight vectors should be normalized. After training, the weight vectors must end up equal to normalized input vectors. Therefore, prenormalization to unit vectors will start weight vectors closer to their final state and there by shorten the training process.

Randomizing the Kohonen layer weights can cause serious training problems, as it will uniformly distribute the weight vectors around the hypersphere. Because the input vectors are usually not evenly distributed and tend to be grouped on a relatively small portion of the hypesphere surface, most of the weight vectors will be so far away from the any input vector that they will never be best match. These Kohonen neurons will always have an output of zero and will be wasted. Furthermore, the remaining weights that do become the best match may be too few in number to allow separation of input vector categories that are close together on the surface of the hypersphere. Suppose there are several sets of input vectors all of which are similar, yet most be separated into different categories. The network should be trained to activate a different Kohonen neuron for each category. If the initial density of weight vectors is too low in the vicinity of the training vectors; it may be impossible to separate similar categories, there by not be enough weight vectors in the vicinity to assign one to each input vector category.

The most desirable solution is to distribute the weight vectors according to the density of input vectors that must be separated, thereby placing more weight vectors in the vicinity of large number input vectors. This is impractical to implement directly, but several techniques approximate its effect.

- One solution, called the convex combination methods, sets all the weights to the same value $1/\sqrt{n}$**,** where **n** is the number of inputs and hence, the number of components in each weight vector. This makes the weight vectors of unit length and all coincident.
- Another method add noise to the input vectors. This causes them to move randomly, eventually capturing a weight vectors. This method also works, but it is even slower than convex combination.

A third method starts with randomized weights but in the initial stages of the training process adjust all of the weights, not just those associated with the winning Kohonen neuron. This moves the weights vectors around to the region of input vectors. As training progresses, weight adjustments, are restricted to those Kohonen neurons that are nearest to the winner. This radius of adjustment as gradually decreased until only those weights are adjusted that are associated with the Kohonen neuron.

Algorithm

1. Initialize network

Define $w_{ij}(t)$ $(0 \le i \le n-1)$ to be the weight from input i to node j at time t. Initialize weights from the n inputs to the nodes to small random values. Set the initial radius of the neighbourhood around node j, $N_i(0)$, to be large.

2. Present input

Present input $x_0(t)$, $x_1(t)$, ...$x_n(t)$, where $x_i(t)$ is the input to node i at time t.

3. Calculate distances

Compute the distance d_j between the input and each output node i, given by

$$d_j = \sum_{i=0}^{n-1} \left(x_i(t) - w_{ij}(t) \right)^2$$

4. Select minimum distance

Designate the output node with minimum d_j to be j*.

5. Update weights

Update weights for node j* and its neighbours, define by the neighbourhood in $N_{j*}(t)$. New weights are

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)\left(x_i(t) - w_{ij}(t) \right)$$

for j in $N_{j*}(t)$, $0 \le i \le n-1$

The term $\eta(t)$ is a gain term $(0 < \eta(t) < 1)$ that decreases in time, so slowing the weight adoption. Notice that the neghbourhood $N_{j*}(t)$ decrease in size as time goes on, thus localising area of maximum activity.

6. Repent by going to step 2.

WEIGHT TRAINING

There is no derivative process involved in adapting the weights for the Kohonen network. Referring to the algorithm, we can see that the change in the weight value is proportional to the difference between the input vector and the weight vector.

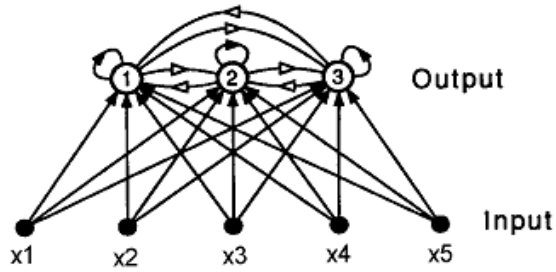
$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_i(t) - w_{ij}(t))$$

where w_{ij} is the i^{th} component of weight vector to node j, for j in the neighbourhood $N_{j*}(t)$ $(0 \le i \le n-1)$

The unit of proportionality is $\eta(t)$, the learning rate coefficient, where $0 < \eta(t) < 1$. This term decreases the adaptation rate with time (where by “time” we mean the number of passes through the training set).

3. Competitive Learning

- A type of unsupervised learning.
- Every output unit is connected to all of input units
- Output units are laterally connected to each other via pre-wired, **negative**/inhibitory connections.
- This way, output units compete for being excited ==> Only one output unit is active at a time (**winner-takes-all** mechanism).



Note that winner-take-all mechanism is essentially the same as selecting the output unit with the highest value. The merits are mostly:

  - accomplishing the selection using the network (i.e., by architecture rather than procedure)
  - as a prologue to Kohonen network
- **CORRECTION** **10/14**

  Activation of the output nodes is in general $y_j = y_j - \varepsilon \sum_{i,j \neq j} y_i$

  where yi is

1. In a simple competitive learning network,

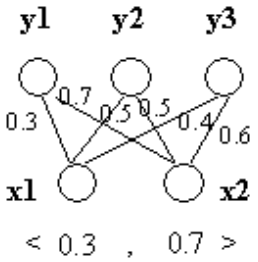
$$y_j = \sum_i w_{ij} \cdot x_i$$

2. In a network aimed for **clustering**,

$$y_j = \sum_i (w_{ij} - x_i)^2$$

, which is the **Euclidean distance** between the weight vector and the input vector

Note for this kind of network, the winner output node is essentially the one whose (incoming) weight vector is the closest to the input vector.



Algorithm:

1. Repeat until the weight changes become very small:
2. Present an input vector
3. Calculate the initial activation for each output unit
4. Let the output units fight until only one is active
5. Adjust the weights on the input lines that lead to the single active output unit:

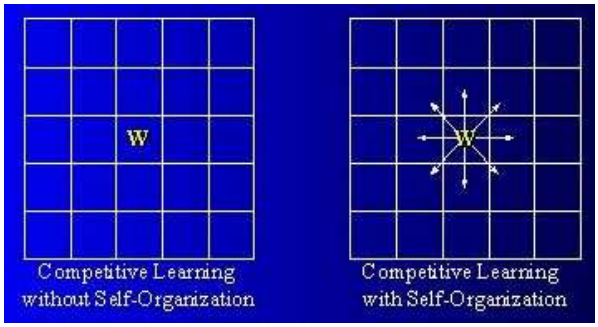
$$\Delta w_j \leftarrow \eta \cdot \frac{x_j}{m} - \eta \cdot w_j$$

where $\mathbf{w_j}$ is the (old) weight on the connection from input unit j to the active output unit, and **m** is the number of input units that are active in the input vector that was chosen at step 1.

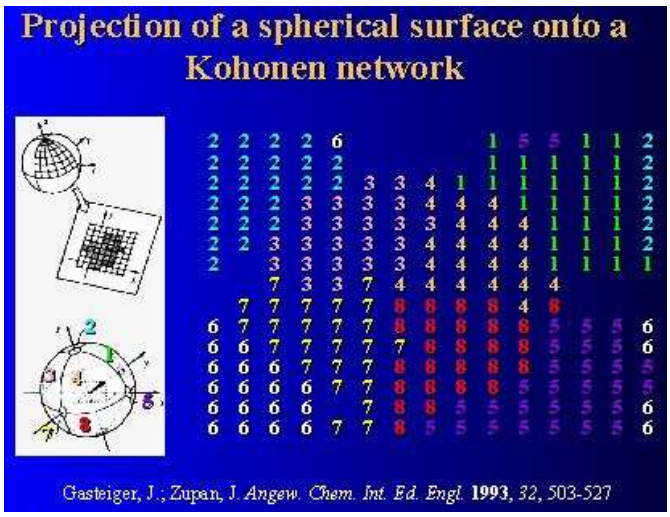
- Another usefulness is to **cluster/group input vectors which activates the same output unit** (similar in notion to k-means clustering)
- But some problems -- Sometimes one unit always wins, particularly when clusters are close together.
- NOTE:
  There is no notion of "error" here because unsupervised learning does not have "target output".

4. Kohonen Self-Organising Maps

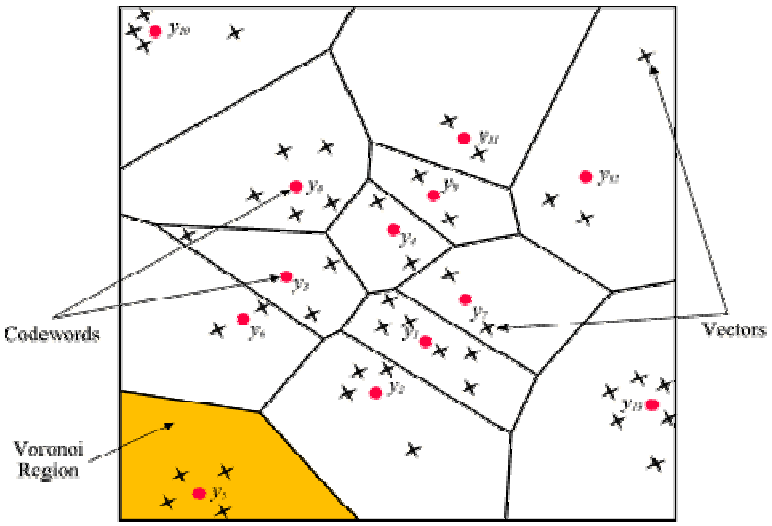
- A competitive learning network.
- But differences:
  - no lateral connections in the output units
  - output units formed in a structure defining **neighborhood**



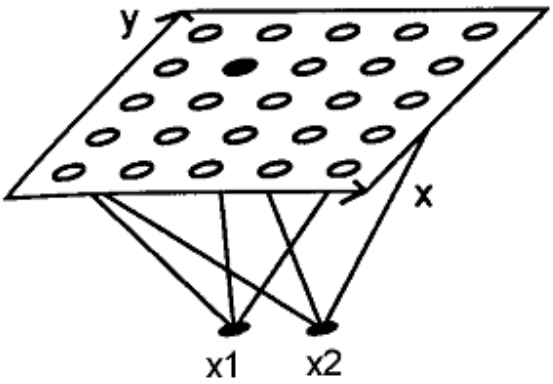
- Self-organizing maps **preserve topology** of the input vectors (in n-dimension) in the output vector (in m-dimension).
    1. When **m < n**, the networks performs **dimensionality reduction.**



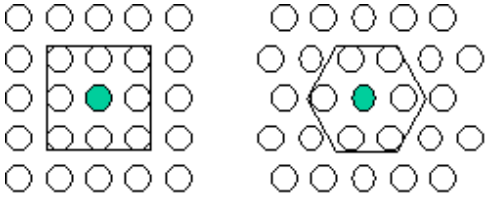
or **vector quantization** (through training) and **clustering**.



2. When m >= n, the output solves a **mixture model** (with latent variables).



In any case, the key notion is the neighborhood and that forms the topology of the map/output. The most common topologies are rectangular and hexagonal.



A parameter decides the radius R of the grid. The picture above indicates R = 1. Then when an output unit i becomes the winner, all of its neighbors (including i itself) update the weights.

The activation of an output unit is usually calculated as the Euclidean distance from the input vector.

$$y_j = \sum_i (w_{ji} - x_i)^2$$

<== CORRECTION 10/14

Algorithm:

0. Initialize weights wij.
   Set topological neighborhood parameter R.
   Set the learning rate eta.
1. Do while the terminating condition is false,
2. For each input pattern x, do
3. For each output unit y,
4. Compute the activation.
5. Find the output unit j that has the **minimum** activation.
6. For all unit k which is in the neighborhood of j, do
7. Update all weights connected from k
8. (Possibly) reduce the radius of neighborhood.

9. (Possibly) update the learning rate.

And the weight update for a neighbor unit k is

$$w_{ki} \leftarrow w_{ki} + \eta \cdot [x_i - w_{ki}]$$

There are many applications of Kohonen network, mostly in the areas of exploratory data analysis.

0. Vector quantization
1. Speech recognition, character recognition
2. Many dimensionality reduction problems

Example:

- 2-dimensional random inputs -- weights in the weight space settle close to inputs (see p. 8-9 in Gurney handout, section7)

Projects at Neural Network Research Center at Univ. of Helsinki, http://www.cis.hut.fi/research/

- ***Topographic map***
  - a mapping that preserves neighborhood relations between input vectors, (topology preserving or feature preserving).
  - if i_1 and i_2 are two neighboring input vectors (by some distance metrics),
    - their corresponding winning output nodes (classes), i and j must also be close to each other in some fashion
  - one dimensional: line or ring, node i has neighbors $i\pm1$ or $i\pm1 \bmod n$.
  - two dimensional: grid.
    rectangular: node(i, j) has neighbors:

    $(i, j \pm 1), (i \pm 1, j)$, (or additional $(i \pm 1, j \pm 1)$)

    hexagonal: 6 neighbors

- **Biological motivation**
  - Mapping two dimensional continuous inputs from sensory organ (eyes, ears, skin, etc) to two dimensional discrete outputs in the nerve system.
    - Retinotopic map: from eye (retina) to the visual cortex.
    - Tonotopic map: from the ear to the auditory cortex
  - These maps preserve topographic orders of input.
  - Biological evidence shows that the connections in these maps are not entirely "pre-programmed" or "pre-wired" at birth. **Learning must occur** after the birth to create the necessary connections for appropriate topographic mapping.

SOM Architecture

Two layer network:

- Output layer:
  - Each node represents a class (of inputs)
  - Node function: $o_j = i_l \cdot w_j = \sum_k w_{j,k} \cdot i_{l,k}$
  - Neighborhood relation is defined over these nodes
    - $Nj(t)$: set of nodes within distance $D(t)$ to node j.
  - Each node cooperates with all its neighbors and competes with all other output nodes.
  - Cooperation and competition of these nodes can be realized by Mexican Hat model

$D = 0$: all nodes are competitors (no cooperative)

→ random map

$D > 0$: → topology preserving map

Algorithm Self Organize:

- Select network topology (neighborhood relation)
- Initialize weights randomly, and select D(0) >0
- While computational bounds are not exceeded, do

1. Select an input sample i_l
2. Find the output node j^* with minimum $\sum_{k=1}^{n} (i_{l,k}(t) - w_{j,k}(t))^2$
3. Update weights to all nodes within a topological distance of D(t) from j^*, using $w_j(t+1) = w_j(t)+\eta(t)(i_l(t)-w_j(t))$, where $0<\eta(t)\leq \eta(t-1) \leq 1$.
4. increment t;
   End-while

Note:

1. Initial weights: small random value from (-e, e)
2. Reduction of η :
   Linear: $\eta(t+1) = \eta(t) - \Delta\eta$

   Geometric: $\eta(t+1) = \eta(t) \cdot \beta$ where $0 < \beta < 1$
3. Reduction of D: $D(t+\Delta t) = D(t) - 1$ while $D(t) > 0$
   should be much slower than η reduction.

   D can be a constant through out the learning.
4. Effect of learning
   For each input ***i***, not only the weight vector of winner j^* is pulled closer to ***i***, but also the weights of j^*'s close neighbors (within the radius of ***D***).
5. Eventually, w_j becomes close (similar) to $w_{j\pm1}$. The classes they represent are also similar.
6. May need large initial ***D*** in order to establish topological order of all nodes.

7. Find j^* for a given input $i_{l:}$
   - With minimum distance between w_j and i_l.

   - Distance: $\text{dist}(w_j, i_l) = \left\| w_j - i_l \right\|_2 = \sum_{k=1}^{n} (i_{l,k} - w_{j,k})$

   - Minimizing dist(w_j, i_l) can be realized by maximizing

$$o_j = i_l \cdot w_j = \sum_k w_{j,k} \cdot i_{l,k}$$

$$-\text{dist}(w_j, i_l) = -\sum_{k=1}^{n} (i_{l,k}^2 + w_{j,k}^2 - 2 i_{l,k} \cdot w_{j,k})$$

$$= -\sum_{k=1}^{n} i_{l,k}^2 - \sum_{k=1}^{n} w_{j,k}^2 + 2\sum_{k=1}^{n} i_{l,k} \cdot w_{j,k}$$

$$= 2\sum_{k=1}^{n} i_{l,k} \cdot w_{j,k} - 2$$

$$= 2 i_l \cdot w_j - 2$$

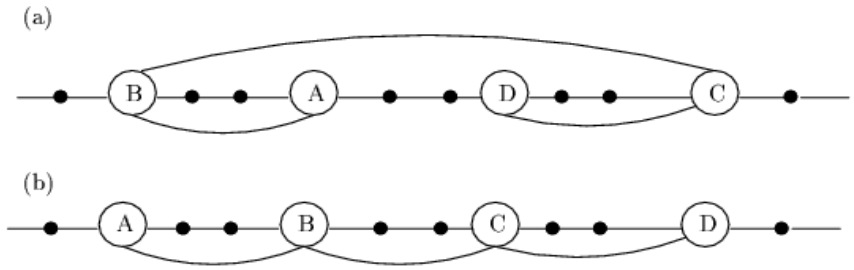


Fig.Emergence of an “ordered” map.

The dark circles represent data points in one-dimensional space, and nodes are identified as A, B, C, and D, with connections indicating linear topology. Figure (a) depicts initial positions of nodes before applying SOM learning algorithm, and figure (b) shows final positions of nodes with topologically adjacent nodes having similar weights.

Counter Propagation Networks

17.0Introduction

Robert Hecht-Nielsen [1] developed the *counter-propagation network* as a means to combine an unsupervised Kohonen layer with a teachable output layer. The operation for the counter-propagation network is similar to that of the Learning Vector Quantization network in that the middle Kohonen layer and finding the closest fit to an input stimulus

and outputting its equivalent mapping. The counter propagation network thus performs a direct mapping of the input to the output.

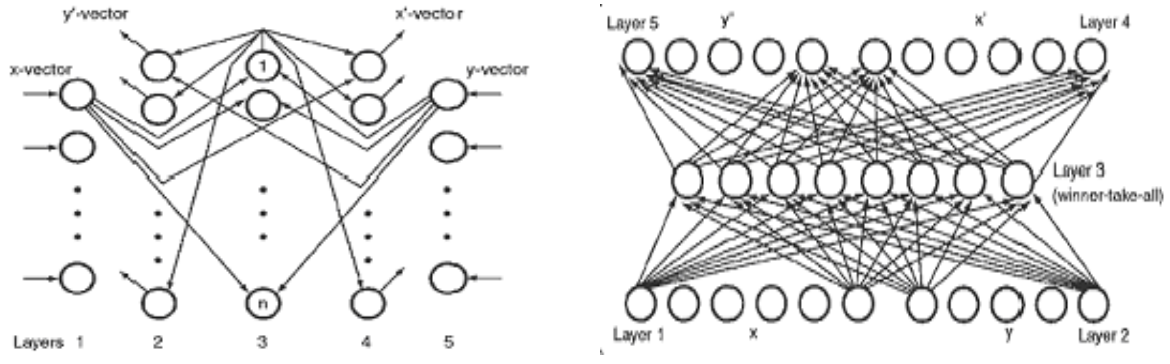
Although most mapping problems are best attacked by the back-propagation method, counter propagation holds promise on those occasions in which it is necessary to handle statistically equi-probable feature vectors. Image-processing applications such as statistical data compression via vector quantization, the recognition of continuously varying patterns, and the estimation of multidimensional probability density functions are examples of problems that appear suitable for counter propagation networks.

The CPN is a multilayer networks based on the various combining structures of input, clustering, and output layers. Compared to the back propagation network, it reduces the time by one hundred times. CPN is different from the backpropagation, in the sense that it provides solution for those applications which can not have larger iterations. As a result, CPN can be used for data compression, function approximation, pattern association and signal enhancement applications.

17.1 CPN Models

This network differs from other architectures in that is uses vectors and normalizes all inputs (the length of each vector must be the same). The CPN functions as a lookup table capable of generalization. The training process consisting of two stages: (1) The input vectors are clustered on the basis of Euclidian distances or by the dot product method, and (2) the desired response is obtained by adopting the weights from the cluster units to the output units. The CPN is classified in two types, namely (1) Full counter propagation network and (2) Forward only counter propagation network.

The first counter-propagation network consisted of a *bi-directional mapping* between the input and output layers [1] as shown in Fig.17.1. In essence, while data is presented to the input layer to generate a classification pattern on the output layer, the output layer in turn would accept an additional input vector and generate an output classification on the network's input layer. The network got its name from this counter-posing flow of information through its structure. The network has five layers, two input layers (1 and 5), one hidden layer (3) and two output layers (2 and 4).



(a) (b)

Fig. 17.1 The Full Counter Propagation Network

While in the second type of CPN the data flows in uni-direction only and it has three layers, as shown in Fig.17.2. If the inputs are not already normalized before they

enter the network. A fourth layer is sometimes added. The main layers include an input buffer layer, a self-organizing Kohonen layer, and an output layer, which uses the Delta Rule to modify its incoming connection weights. Sometimes this layer is called a Gross berg Outstar layer.

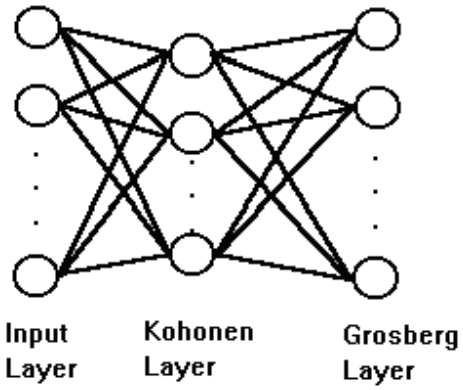


Fig.17.2 Forward only Counter Propagation Network

The counter propagation network consists of an input layer, a hidden layer called Kohonen, and an output layer called Grossberg layer. The Kohonen layer works in winner-takes all fashion. The training process of this network consists of two steps: first, an unsupervised competitive learning is performed by the Kohonen layer, then after the Kohonen layer is stable a supervised learning is performed by the Grossberg layer. In normal operation, when an input is presented to the network, the Kohonen layer classifies it and the winner node activates the appropriate output nodes in the Grossberg layer.

The size of the input layer depends upon how many separable parameters define the problem. For the network to operate properly, the input vector must be normalized. This means that for every combination of input values, the total "length" of the input vector must add up to one. This can be done with a preprocessor, before the data is entered into the counter-propagation network. Or, a normalization layer can be added between the input and Kohonen layers. The normalization layer requires one processing element for each input, plus one more for a balancing element.

This layer modifies the input set before going to the Kohonen layer to guarantee that all input sets combine to the same total. Normalization of the inputs is necessary to insure that the Kohonen layer finds the correct class for the problem. Without normalization, larger input vectors bias many of the Kohonen processing elements such that weaker value input sets cannot be properly classified. Because of the competitive nature of the Kohonen layer, the larger value input vectors overpower the smaller vectors.

As a summary the CPN has four major components: an input layer that performs some processing on the input data, a processing element called an **instar**, a layer of instars known as a competitive network, and a structure known as an **outstar**.

17.2Features of CPN

In normal operation, when an input is presented to the network, the Kohonen layer classifies it and the winner node activates the appropriate output nodes in the Grossberg layer. Now let's consider the size of the middle layer not in the context of

training issues but in terms of the accuracy of the networks response. If the data patterns are evenly distributed throughout the unit circle, we expect that the weight vectors will also be equally distributed after training. The situation becomes more complex if the input patterns are not evenly distributed about the unit circle.

In this case the weight vectors are clustered during training about the areas of the unit circle most likely to contain input vectors. Regions outside the most common input areas may end up with very few weight vectors in their region. Thus the occasional input vector that occurs in one of these sparsely populated regions may end up being approximated by a weight vector that is only a gross estimate of the actual input vector. On the other hand, input vectors that do occur in the area densely clustered with weight vectors will be quite closely approximated.

In reality, a no uniform distribution of input patterns is much more likely in a real application than a uniform one. This means that the accuracy of the network's mapping is better in those parts of the unit circle that are more likely to contain input vectors. Rather than having a uniform accuracy, counter propagation networks have higher accuracy.

17.3The Instar

The hidden layer of the CPN comprises a set of processing elements known as *instars*. The k^{th} processing unit in hidden layer can shown as in Fig. 17.3:

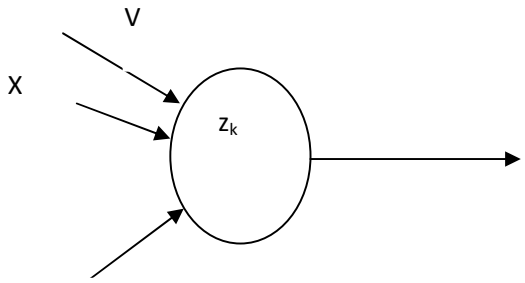


Fig.17.3 processing element in hidden layer

1. Select an input vector X_l from the given training set (X_l, Y_l), $l=1, 2…L$.

2. Normalize the input vector and apply it to the CPN competitive layer.

3. Determine the unit that wins the competition by determining the unit k whose Vector W_k is closest to the given input.

4. Update the winning unit's weight vectors

$v_{ki}\ (new)=v_{ki}\ (old)+ \alpha\,(x_i-v_{ki}\ (old))$

where $\alpha < 1$

5. Repeat step 1 through 4 until all the input vectors are grouped properly by applying the training vector several times.

After successfully training the weight vector leading to each hidden unit represents the average of the input vectors corresponding to the group represented by unit.

17.4The Outstar

1. After training the instars apply a normalized input vector X_l to the input layer and corresponding desired output Y_l to the output layer.
2. Determine the winning unit k in the competitive layer

3. Update the weights on the connections from the winning competitive unit to the output units as

$$u_{jk}\ (new) = u_{jk}\ (old) + a\ (y_k - u_{jk}\ (old))$$

4. Repeat steps 1 through 3 until all the vector pairs in the training data are mapped satisfactorily.

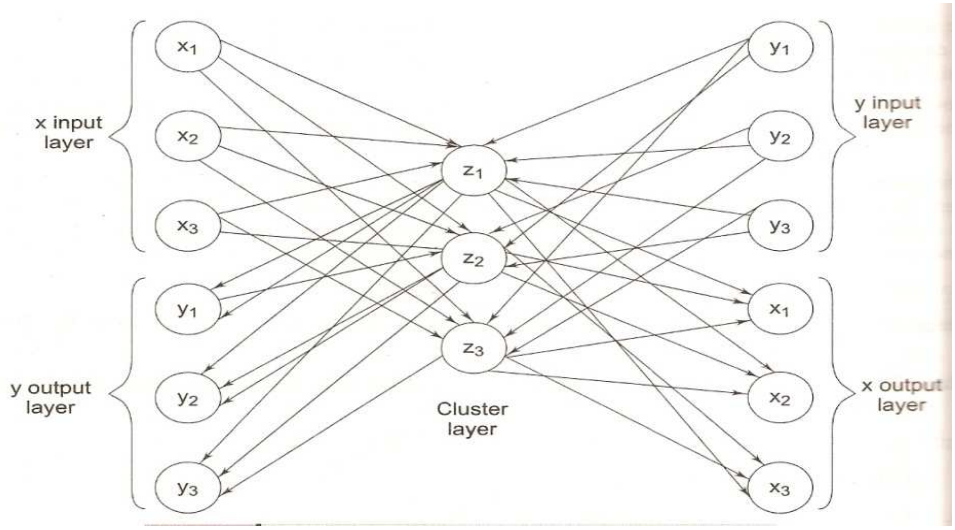
After successful training the outstar weight vector for each unit in the hidden competitive layer represents the average of the subset of the output vectors corresponding to the input vectors belonging to that unit.

17.5Full Counter Processing Network

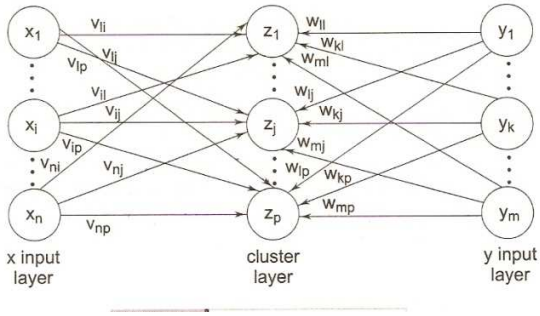
The full CPN possesses the generalization capability which allows it to produce a correct output even when it is given an input vector that is partially incomplete or partially incorrect. The Full CPN can represent large number of vector pairs, X:Y by constructing a lookup table. It operates in two stages; during first stage, the training vector pairs are used to form clusters. Clustering can be done either by dot product or Euclidean distance. During the second stage, the weights are adjusted between the cluster units and the output units.

17.5.1 Architecture of Full CPN

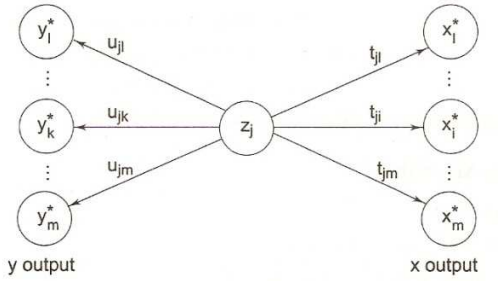
The architecture of the Full CPN is shown in Fig. 17.4. The given vector pairs are X:Y and the approximated vector pairs may be X^* : Y^*. The CPN functions in two modes: Normal mode, where the input vector is accepted and output vector is produced and training mode, where the input vector is applied and the weights are adjusted to obtain desired output vector.



17.4 Architecture of Full CPN



(a) First Phase of Full CPN



(b) Second Phase of Full CPN

Fig. 17.5

The architecture of a CPN resembles an *instar* and *outstar* model. Basically, it has two input layers (as shown in Fig 17.5a) and two output layers(as shown in Fig 17.5b) with common hidden layer. The model which connects the input layers to the hidden layers is called Instar model and the model which connects the hidden layer to output layer is called Outstar model. The weights are updated both in the Instar and Outstar model. The Instar performs the first phase of training and the Outstar model performs the second phase of training.

17.5.1 Training of Full CPN

The full CPN training is achieved in two phases.

First Phase

This phase of training is known as an Instar modeled training. The active units here are the units in the x- input, z- cluster and y- input layers.

Generally in CPN, the cluster unit does not assume any topology, but the winning unit is allowed to learn. This winning unit uses our standard Kohonen learning rule for its weight updation. The rule is given by

$$v_{ij}(new) = v_{ij}(old) + \alpha(x_i - v_{ij}(old))$$

$$= (1-\alpha)\, v_{ij}(m) + \alpha x_i \qquad i=1 \text{ to } n$$

$$w_{kj}(new) = w_{kj}(old) + \beta(y_k - w_{kj}(old))$$

$$= (1-\beta) w_{kj}(old) + \beta y_k \qquad k=1 \text{ to } m$$

Second Phase

In this phase, we can find only the J unit remaining active in the cluster layer. The weights from the winning cluster unit J to the output units are adjusted, so that vector of activation of the units in the y output layer, y*, is approximation of input vector y; and x* is an approximation of input vector x.this phase may be called as outstar modeled learning. In out star learning, no competition is assumed among the units, and the lrearing occurs fro all the units in a particular layer. The weight updation rule is given as,

$$u_{jk}(new) = u_{jk}(old) + a\,(y_k - u_{jk}(old))$$

$$= (1\text{-}a)\, u_{jk}(old) + a y_k \qquad k=1 \text{ to } m$$

$$t_{ji}(new) = t_{ji}(old) + b\,(x_i - t_{ji}(old))$$

$$= (1\text{-}b)\, t_{ji}(old) + b\, x_i \qquad i=1 \text{ to } n$$

The weight change indicated is the learning rate times the error.

17.5.2 Training Algorithm of Full CPN

The parameters used are

x	- Input training Vector
y	- target vector
z_j	- activation of cluster unit Z_j
x*	- approximation to vector x.
y*	- approximation to vector y.
V_{ij}	- weight from x input layer to Z cluster layer.
W_{jk}	- weight from y input layer to Z- cluster layer.
T_{ji}	- weight from cluster layer to X- output layer.
U_{jk}	- weight from cluster to Y- output later.
α, β	- learning rates during Kohonen learning.
a, b	- learning rates during Grossberg learning.

The algorithm uses the Euclidean distance or dot product method for calculation of the winner unit. The winner unit is during both first and second phase of training. In the first phase of training for weight updation Kohonen learning rule is used and for second phase of learning Grossberg learning rile is used.

The algorithm for the full CPN is given by,

Step 1: Initialize the weights, learning rats etc.

Step 2: While stopping condition for phase training is false, perform Steps 3-8

Step 3: For each training input pair x:y, do steps 4-6

Step 4: Set X input layer activations to vector x;

Set Y input layer activations to vector y;

Step 5: Find winning cluster unit using Euclidean distance.

Step 6: Update weight for winning unit.

$$v_{ij}(new) = v_{ij}(old) + \alpha\,(x_i - v_{ij}(old)) \qquad i=1, \text{ to } n$$

$$w_{kj}(new) = w_{kj}(old) + \beta(y_k - w_{kj}(old)) \qquad k=1 \text{ to } m$$

Step 7: Reduce learning rates α, and β.

Step 8: Test stopping condition for phase 1 training.

Step 9: While stopping condition for phase 2 training is false, perform Steps 10-16.

Step 10: For each training input pair x:y, do steps 11-14.

Step 11: Set X input layer activations to vector x;

Set Y input layer activations to vector y;

Step 12: Winning cluster unit using Euclidean distance.

Step 13: Updating weights for winning unit, the values of in this phase are constant.

$$v_{ij}\ (new)=v_{ij}\ (old)+\ \alpha\ (x_i-v_{ij}\ (old)) \qquad i=1,\ to\ n$$

$$w_{kj}\ (new)=\ w_{kj}\ (old)+\ \beta(y_k-\ w_{kj}\ (old)) \qquad k=1\ to\ m$$

Step 14: Update weights from unit z_j to the output layers.

$$u_{jk}\ (new)=\ u_{jk}\ (old)+\ a\ (y_k\ -\ u_{jk}\ (old)) \qquad k=1 \text{ to } m$$

$$t_{ji}\ (new)=\ t_{ji}\ (old)+\ b\ (x_i-\ t_{ji}(old)) \qquad i=1 \text{ to } n$$

Step 15: Learning rates a and b are to be reduced.

Step 16: Test the stopping condition for phase 2 training.

The winning unit selection is done either by dot produce or Euclidean distance. The dot product is done by calculating the net input.

$$Z_{-inj} = \sum_i x_i u_{ij} + \sum_k y_k w_{kj}$$

The cluster unit with the largest net input is winner. Here the vectors should be normalized. In Euclidean distance,

$$D_j = \sum_i (x_i - v_{ij})^2 + \sum_i (y_k - w_{kj})^2$$

The square of whose distance from the input vector is smallest is the winner. In case of tie between the selections of the winning unit, the unit with the smallest in desk is selected. The stopping condition may be the number of iteration or the reductions in the learning rate up to a certain level.

17.5.3 Application Procedure

In the learning algorithm, if only one Kohonen neuron is activated for each input vector, this is called the Accretive mode. If a group of Kohonen neurons having the highest outputs is allowed to prevent its outputs to the Grossberg layer, this is called the Interpolative mode. This mode is capable of representing more complex mapping and can produce more accurate results.

The application procedure of the CPN is

Step 1: Initialize weights

Step 2: For each input pair x:y, perform Steps 3-5

Step 3: Set x input layer activations to vector x;

Set y input layer activations to vector y;

Step 4: Find the cluster unit Zj close to the input pair.

Step 5: Compute approximations to x and y:

$x_i^* = t_{Ji}$ $\quad\quad$ $y_i^* = u_{Jk}$

17.5.4 Initializing the Weight Vectors

All the networks weights must be sent to initial values before training starts. It is a common practice with neural networks to randomize the weights to small numbers. The weight vectors in CPN should be distributed according to the density of input vectors that must be separated, thereby placing more weight vectors in the vicinity of a large number of input vectors. This is obtained by several techniques.

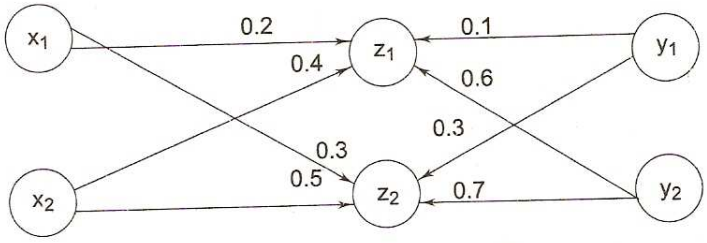
One technique is called, **Convex combine method**, sets all the weights to the same value $1/\sqrt{n}$, where n is the number of inputs and hence, the number of components in each weight vector. This makes all the weight vectors of unit length and coincident. This method operates well but slows the training process, as the weight vectors have to adjust to a target.

Another method **adds noise** to the input vectors. This causes them to move randomly, eventually capturing a weight vector. This method also works, but it is even slower than convex combination.

A third method starts with randomized weights but in the initial stages of the training process adjusts all the weights, not just those associated with the winning Khonoene neuron. This moves the weights vector around, to the region of the input vectors as training starts, weight adjustments are restricted to those Khononen neurons that are nearest to the winner. This radius of adjustment is gradually decreased until only those weights are adjusted that are associate with the winning Khononen neuron.

There is another method, which gives each Khononen neuron a “conscience”. If it has been winning more than its fair share of the time, say 1/k, where k is the number of Khononen neurons, it raises the threshold that educes its chances of winning, thereby allowing the other neurons an opportunity to be trained. Thus weights may be initialized.

Example: Consider the following full CPN using input pair x= (1,1) y= (0,1). Perform first phase of training (one step only). Find the activation of the cluster layer units and update the weights using learning rates of 0.3.



Solution: It is needed to perform first phase of training for one step only. The steps are as follows.

Step 1: Initialize weights

$$v = \begin{bmatrix} 0.2 & 0.3 \\ 0.4 & 0.5 \end{bmatrix},\ w = \begin{bmatrix} 0.1 & 0.3 \\ 0.6 & 0.7 \end{bmatrix}$$

Initialize the following learning rates

$$\alpha = \beta = 0.3$$

Step 2: Begin training.

Step 3: Present the input vector pair.

x = (1,1) and y = (0,1)

Step 4: Set the activations of input and output to x and y.

Step 5: Computing the winning unit, by Euclidean distance,

$$D_j = \sum_i (x_i - v_{ij})^2 + \sum_i (y_k - w_{kj})^2$$

$$D(1) = (1-0.2)^2 + (1-0.4)^2 + (0-0.1)^2 + (1-0.6)^2 = 1.17$$

$$D(2) = (1-0.3)^2 + (1-0.5)^2 + (0-0.3)^2 + (1-0.7)^2 = 0.92$$

$$D(2) \prec D(1)$$

$$J = 2$$

Step 6: Updating the weights on the winning unit. The weight updation is given by

$$V_{ij}(new) = V_{ij}(old) + \alpha(x_i - V_{ij}(old));\ \text{i= 1 to 2}$$

$$V_{12}(n) = V_{12}(0) + \alpha(x_1 - V_{12}(0))$$

$$V_{12}(n) = 0.3 + 0.3(1-0.3) = 0.51$$

$$V_{22}(n) = 0.5 + 0.3(1-0.5) = 0.65$$

Also, $W_{kj}(new) = W_{kj}(old) + \beta(y_k - W_{kj}(old))$; k=1 to 2

$$W_{12}(x) = 0.3 + 0.3(0-0.3) = 0.21$$

$$W_{22}(x) = 0.7 + 0.3(1-0.7) = 0.79$$

Thus the first iteration is preformed. The updated weighs are,

$$u = \begin{bmatrix} 0.2 & 0.51 \\ 0.4 & 0.65 \end{bmatrix} \qquad w = \begin{bmatrix} 0.1 & 0.21 \\ 0.6 & 0.79 \end{bmatrix}$$

These weights may be further used for iterations, depending upon the reduction of learning rates and the stopping conditions.

17.6 The Forward only Counter Propagation Network

The forward CPN is the simplified form of Full CPN. The forward CPN is a feed forward mapping network that combines a portion of the self-organizing map Kohonen at the hidden level and Grossberg's outstar at the output level. The forward only CPN has only one input layer and one output layer, but the training is still performed in two

phases. This is yet another topology to synthesize complex classification problems, while trying to minimize the number of processing elements and training time. The architecture of forward only CPN as show in Fig. 17.6

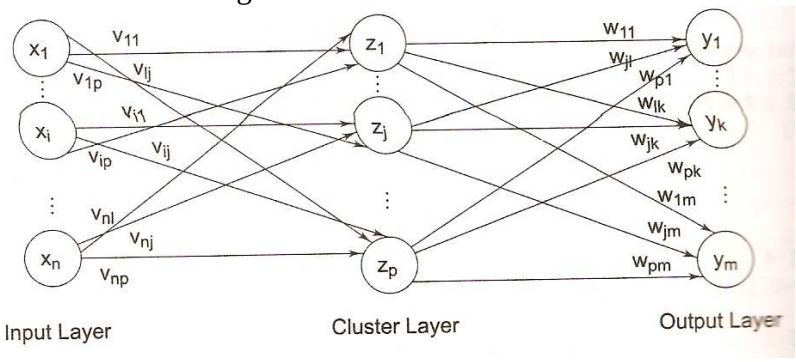


Fig 17.6 Architecture of a Forward only CPN

17.6.1 Training Algorithm

The parameters are similar to that in full CPN, except that some in the full CPN do not exist here.

The training algorithm of forward only CPN is:

Step 0: Initialize the weights, learning rates etc.

Step 1: While stopping condition for phase 1 is false, perform Steps 2-7

Step 2: For each training input x, do steps 3-5.

Step 3: Set x input layer activations to vector x.

Step 4: Find winning cluster unit.

Step 5: Update weight for winning unit.

$$v_{iJ}(new) = v_{iJ}(old) + \alpha(x_i - v_{iJ}(old)) \qquad i=1, \text{ to } n$$

Step 6: Reduce learning rate α.

Step 7: Test stopping condition for phase 1 training.

Step 8: While stopping condition for phase 2 training is false, perform Steps 9-15

Step 9: For each training input pair x:y, do steps 10-13.

Step 10: Set X input layer activations to vector x;

Set Y output layer activations to vector y;

Step 11: Complete the winning cluster unit.

Step 12: Updating weights unit z; here the value of α is a small constant value.

$$v_{iJ}(new) = v_{iJ}(old) + \alpha(x_i - v_{iJ}(old)) \qquad i=1, \text{ to } n$$

Step 13: Update weights from cluster unit to the output unit.

$$w_{Jk}(new) = w_{Jk}(old) + a(y_k - w_{Jk}(old)) \qquad k = 1 \text{ to } m$$

Step 14: Reduce the learning rate a.

Step 15: Test the stopping condition for phase 2 training.

The typical values of α and a, may be α = 0.5 to 0.8, and a= 0 to 1 (mostly it can take initial values of 0.1and 0.6). The winning unit selection is done either by dot product or Euclidean distance method. In dot product, the cluster unit with largest net input is selected. is done by calculating the net input.

$$Z_{-inj} = \sum_i x_i v_{ij}$$

In Euclidean distance,

$D_j = \sum_i (x_i - v_{ij})^2$ is used. The smaller D_j is the winner unit.

In case of tie, the unit with smallest index is chosen, as the winner.

17.6.2 Application Procedure

The application procedure for forward only CPN is as follows.

Step 1: Initialize weights (obtained from training)

Step 2: Present input vector x.

Step 3: Find unit J nearest to vector x.

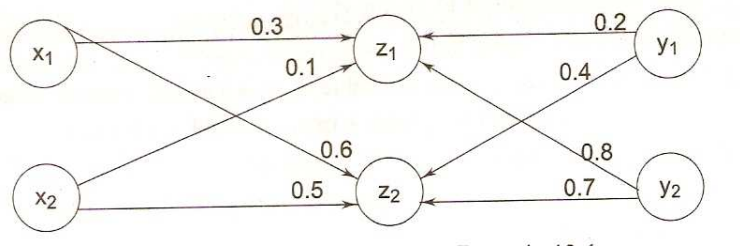
$y_k = w_{jk}$

The activation of the cluster unit is,

$$Z_j = \begin{cases} 1 & \text{if } j = J \\ 0 & \text{otherwise} \end{cases}$$

A forward only CPN can be used in interpolation mode. Here, more than one Kohonen units has a non zero activation. By using interpolation mode, the accuracy is increased.

Example: Consider the forward only CPN shown in the figure below using the input pair x= (1, 0) y= (0, 1) perform the training (one step iteration). Find the activation of cluster layer unit. Update the eights using learning stateα=0.5 and a=0.1



Solution

Step 0: Initialize weights

$$v = \begin{bmatrix} 0.3 & 0.6 \\ 0.1 & 0.5 \end{bmatrix}, \quad w = \begin{bmatrix} 0.2 & 0.8 \\ 0.4 & 0.7 \end{bmatrix}$$

Initialize learning rates

$\alpha = 0.5$, a= 0.4

Step 1: Begin training of the firsr phase.

Step 2: Present input vector x = (1, 0).

Step 3: Set activations of x = (1, 0)*

Step 4: Calculate winning cluster unit.

$$D_j = \sum_i (x_i - v_{ij})^2 \text{ ; i= 1 to 2.}$$

$$D(1) = \sum_{i=1}^{2} (x_i - v_{i1})^2$$

$$D(1) = (1-0.3)^2 + (0-0.1)^2 = 0.5$$

$$D(2) = (1-0.6)^2 + (0-0.5)^2 = 0.41$$

$$D(2) \prec D(1)$$

$$J = 2$$

Step 5: Updating the weights on the winner unit,

$$V_{ij}(new) = V_{ij}(old) + \alpha(x_i - V_{ij}(old)) \text{ ; i= 1 to 2}$$

$$V_{12}(n) = V_{12}(old) + \alpha(x_1 - V_{12}(0))$$

$$V_{12}(n) = 0.6 + 0.5(1-0.6) = 0.8$$

$$V_{22}(n) = 0.5 + 0.5(0-0.5) = 0.25$$

The updated weights are,

$$\begin{bmatrix} 0.3 & 0.8 \\ 0.1 & 0.25 \end{bmatrix}$$

Step 6 & 7: Keep α – same value for using it in the second phase. If stopping condition is mentioned, check or move to next step.

Step 8: Begin second stage of training.

Step 9: Present input vector pair x and y.

Step 10: Set the activations to, x= (1 0) and y= (0 1)

Step 11: Calculate the winning cluster unit,

$$D_j = \sum_i (x_i - v_{ij})^2$$

$$D(1) = (1-0.3)^2 + (0-0.1)^2 = 0.5$$

$$D(2) = (1-0.8)^2 + (0-0.25)^2 = 0.1025$$

$$D(2) \prec D(1)$$

$$J = 2$$

Step 12: Update the weights into cluster unit

$$V_{in}(new) = V_{iJ}(old) + \alpha(x_i - V_{iJ}(old))\text{ ; I= 1 to 2}$$

$$V_{12}(n) = V_{12}(0) + \alpha(x_i - V_{12}(old))$$

$$V_{12}(n) = 0.8 + 0.5(1 - 0.8) = 0.9$$

$$V_{22}(n) = 0.25 + 0.5(0 - 0.25) = 0.125$$

Step 13: Update the weights from the cluster unit to the output unit

$$W_{21}(n) = W_{21}(0) + 0.1(y_1 - W_{21}(0))$$

$$W_{21}(n) = 0.4 + 0.1(0 - 0.4) = 0.36$$

$$W_{22}(n) = 0.7 + 0.1(1 - 0.7) = 0.73$$

Thus the updated weights of second phase of training are

$$v = \begin{bmatrix} 0.3 & 0.9 \\ 0.1 & 0.125 \end{bmatrix} \quad w = \begin{bmatrix} 0.2 & 0.5 \\ 0.36 & 0.73 \end{bmatrix}$$

The next iteration can be carried out depending on the stopping condition or by further reduction of learning rates.

17.7 Limitations of Counter propagation Network

What do we really have here? The CPN boils down to a simple lookup table. An input pattern is presented to the net, which causes one particular winning neuron in the middle layer to fire. The output layer has learned to reproduce some specific output pattern when it is stimulated by a signal from this winner. Presenting the input stimulus merely causes the network to determine that this stimulus is closest to stored pattern, for example, and the output layer obediently reproduces pattern stored.

The CPN thus performs a *direct mapping* [2] of the input to the output. There is a problem, which could arise with this architecture. The competitive Kohonen layer learns without any supervision. It does not know what class it is responding to. This means that it is possible for a processing element in the Kohonen layer to learn to take responsibility for two or more training inputs, which belong to different classes. When this happens, the output of the network will be ambiguous for any inputs, which activate this processing element.

To alleviate this problem, the processing elements in the Kohonen layer could be pre-conditioned to learn only about a particular class. The name counter propagation derives from the initial presentation of this network as a five-layered network with data flowing inward from both sides, through the middle layer and out the opposite sides. There is literally a counter flow of data through the network. Although this is an accurate picture of the network, it is unnecessarily complex; we can simplify it considerably with no loss of accuracy. In the simpler view of the counter propagation network, it is a three-layered network.

The input layer is a simple fan-out layer presenting the input pattern to every neuron in the middle layer. The middle layer is a straightforward Kohonen layer, using the competitive filter-learning scheme. Such a scheme ensures that the middle layer will categorize the input patterns presented to it and will model the statistical distribution of

the input pattern vectors.

The third, or output layer of the counter propagation network is a simple outstar array. The outstar, you may recall, can be used to associate a stimulus from a single neurode with an output pattern of arbitrary complexity. In operation, an input pattern is presented to the counter propagation network and distributed by the input layer to the middle, Kohonen layer. Here the neurodes compete to determine that neurode with the strongest response (the closest weight vector) to the input pattern vector.

That winning neurode generates a strong output signal (usually a +1) to the next layer; all other neurodes transmit nothing. At the output layer we have a collection of outstar grid neurodes. These are neurodes that have been trained by classical conditioning to generate specific output patterns in response to specific stimuli from the middle layer. The neurode from the middle layer that has fired is the hub neurode of the outstar, and it corresponds to some pattern of outputs.

Because the outstar-layer neurodes have been trained to do so, they obediently reproduce the appropriate pattern at the output layer of the network. In essence then, the counter propagation network is exquisitely simple: the Kohonen layer categorizes each input pattern, and the outstar layer reproduces whatever output pattern is appropriate to that category.

17.8 Applications of CPNs

The potential applications of CPN are data compression, function approximation, pattern association, image classification, and signal enhancement applications. We now discuss two applications such as (1) handwritten character recognition. and (2) Solving power flow problem.

17.8.1 On-Line Handwritten Character Recognition [3]

Two classes of recognition system usually distinguished: on-line systems for which handwriting data are captured during the writing process, which makes available the information on the ordering of the strokes and offline system for which recognition takes place on a static image captured once the writing process is over. On-line handwritten scripts are usually dealt with pen tip traces from pen-down to pen-up positions. Let us present example of on-line handwritten character recognition using CPN as given in [4], in which upper case English alphabets are considered for recognition.

17.8.1.1 System Overview

The method proposed in [3] is a simple technique and is a writer-independent system based on the neural network. Conventionally, the data obtained needs a lot of preprocessing including filtering, smoothing, slant removing and size normalization before recognition process. A block diagram of on-line recognition system of isolated roman characters is shown in Fig. 17.7 . the flow of data during training is shown by the dashed line arrows, while the data flow during recognition is shown by solid line arrows. The input to the system is a sequence of handwritten character patterns. After receiving input from tablet the extreme coordinates i.e., left, right, top, bottom are calculated. Then character is captured in a grid as shown in Fig. 17.8 and sensing the character pixels in grid boxes, the character is digitized in a binary strring. This binary string is applied at the input of the CPN for training and recognition. The grid sizes may be selected to capture the properties, for example 14x8 (ie.e 14 rows and 8 columns).

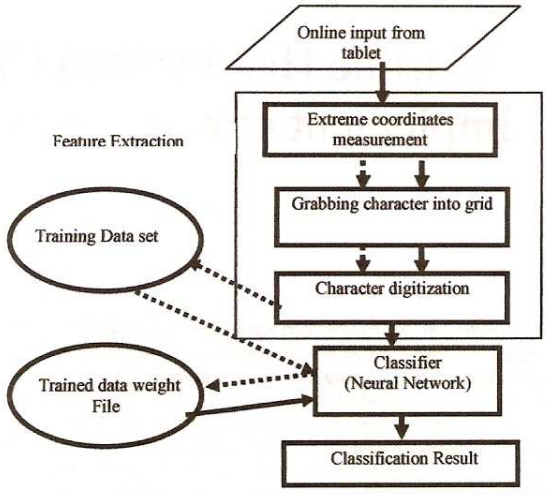


Fig. 17.7 Block diagram of the system

The system consisting of following major components:

A. Data Acquisition

Tablet SummaSketch III was used to take samples. No restriction was imposed on the content or style of writing; the only exception was the stipulation on the isolation of characters.

B. Character Detection

It searches from left to right for white pixels starting from left-top corner of the area specified for writing. A trace of a white pixel is the indication of the presence of a character.

C. Calculating the number of RowsI

The algorithm searches for the presence and absence of white pixels going from top to bottom. The continuous absence of white pixels (or presence of black pixels) could be a gap between two rows. To make sure whether it is a gap, algorithm searches from left to right against every black pixel, if there is no trace of white pixel for the entire row, the gap is confirmed.

D. Character Boundary Calculation

The algorithm checks from left to right and top to bottom for left, top, right and bottom boundaries of the character. While going from left to right, the first white pixel is the left boundary and last white pixel is the right boundary of the character. Similarly from top to bottom, first white pixel is the top boundary and last white pixel is the bottom boundary of the character. If there is a vertical gap between two portion of a same character, e.g., ‘ H’ then the algorithm also check from top to bottom for that particular area. The presence of white pixel will eliminate the doubt of a true gap.

17.8.1.2 Feature Extraction

In general the feature extraction consists of three steps: extreme coordinates measurement, grabbing character into grid, and character digitization. The handwritten character is captured by its extreme coordinates from left /right and top/bottom and is subdivided into a rectangular grid of specific rows and columns. The algorithm automatically adjusts the size of grid and its constituents according to the dimensions of the character. Then it searches the presence of character pixels in every box of the grid. The boxes found with character pixels are considered “on” and the rest are marked “off”. A binary string of each character is formed locating the “on” and “off” boxes (named as character digitization) and presented to the neural network input for training and recognition purposes. The total number of grid boxes represented the number of binary inputs. A 14x8 grid thus resulted in 112 inputs to the recognition model.

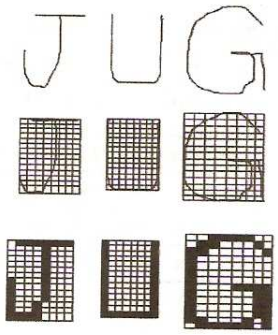


Fig. 17.8 Steps in Feature Extraction

17.8.1.3 Neural Net Approach

The underlying principle for CPN is for a given independent variable vector **I** not present in the available data set, find the independent variable vector in the data set closest to **I**. The criterion of closeness in the n-dimensional Euclidean space can either be distance based (minimum Euclidean distance) or can be angle based (minimum angle between vectors of normalized lengths). If $\mathbf{X_k}$ is the vector found closest in the data set , then the value of $f(\mathbf{I})$ can be approximated as the dependent variable value corresponding to $\mathbf{X_k}$. This technique runs into problems when the data set becomes very large. During training, these "correct" values are supplied to the units of the final layer from, the input units of layer 1. During training the network is exposed to examples of the mapping *f*. After each $\mathbf{x}_k$ is selected, $\mathbf{y}_k = f(\mathbf{x}_k)$ is determined and both $\mathbf{x}_k$ and $\mathbf{y}_k$ are input to the network. An important component of training in the CPN is reduction of the data set into a respective data set of lesser, specified size. This is achieved and the estimates of the dependent variable values corresponding to the new (and reduced in number) independent variable vectors can also be calculated. Thus CPN actually operates as a closest-match lookup table and training a CPN is an attempt to appropriately reduce the size of the lookup table.

For a CPN in its final form, each PE in the hidden layer represents an independent variable entry in the (reduced) lookup table; the weights to one such PE in the hidden layer from all the PEs in the input layer represent the components of the corresponding independent variable vector. The hidden layer PE whose incoming weights are closest to

an input vector “wins” the competition and provides an output value of +1; all other hidden layer PEs supply zero outputs. The weights from the hidden layer PEs to the output layer PEs represent the dependent variable values. A CPN with one linear PE in the output layer thus behaves as estimating one function. With multiple PEs in the output layer, the CPN becomes an estimator of more than one functions.

For a three layer CPN, the size of input layer would be 112 nodes. The look-up table grows with increase in training samples. Instead of using Kohonen’s learning algorithm for reducing the size of the look-up table, a much simpler technique was employed. Since there were k samples for a character in a particular model, why not reduce the k vectors to one vector by taking the average of the sample vectors? Each component of the resultant averaged vector was average of the corresponding components of the k vectors.

17.8.2 Power Flow Problem [4]

POWER flow or load flow analysis is performed to determine the steady state operating condition of a power system, by solving the static load flow equations (SLFE) for a given network. *The main objective of power flow (PF) studies is to determine the bus voltage magnitude with its angle at all the buses, real and reactive power flows (line flows) in different lines and the transmission losses occurring in a power system.* Power flow study is the most frequently carried out study performed by power utilities and it is required to be performed at almost all the stages of power system planning, optimization, operation and control.

Fig.17.9 shows the architecture of the proposed counter propagation neural network. The composition of the input variables for the proposed neural network has been selected to emulate the solution process of a conventional power flow program. A feed-forward counterpropagation neural network (CPNN) can be used to solve the power flow problem (A case study on IEEE 14- bus system is considered).

The Fig. 17.9 show the CPN for power flow study and its input vector consists of the electric network parameters represented by the diagonal elements of the bus conductance and susceptance matrix, voltage magnitudes *Vg* of generation and slack buses, the active power generations *Pg* of PV buses. In order to speed up the neural network training, the conductance and susceptance are normalised between 0.1 and 0.9. For this CPNN based power flow model, the system loads, active and reactive power components are represented like constant admittance and they are included into the diagonal of the bus admittance matrix [Y]=[G]+j[B], where [G] and [B] are the bus conductance and susceptance matrices respectively.

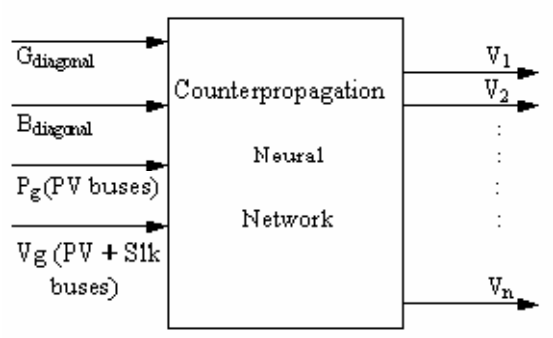


Fig. 17.9 The CPNN Architecture

The objective of power flow study is to determine the voltage and its angle at each bus, real and reactive power flow in each line and line losses in the power system for specified bus or terminal conditions. The power flow studies are conducted for the purpose of planning (viz. short, medium and long range planning), operation and control. For the purpose of power flow studies, it is assumed that the three-phase power system is balanced and also mutual coupling between elements is neglected. Variable associated with each bus of the power system include four quantities viz. voltage magnitude *Vi*, its phase angle δ_i , real power P_i and reactive power Q_i total 4*n* variable for n buses system. At every bus two variables are specified, the remaining two can be found by solving the 2*n* power flow equations. Out of these four quantities only two are generally specified at a few bus and depending upon which two are specified, we have three categories of buses, namely Swing Bus or Reference Bus, Generator Bus or *PV* Bus and Load Bus or *PQ* Bus.

Counterpropagation Neural Network

The CPNN model shown in Fig. 17.10, involves both supervised and unsupervised learning. A large number of load patterns are generated randomly by perturbing the load at all the buses in wide range, voltage magnitude at PV and slack buses and real power generation at PV buses and transformer tap setting. Single line outages are considered as contingencies. Newton-Raphson (NR) power flow program is used to generate training / testing patterns for different load scenarios and for all the single-line outage contingencies.

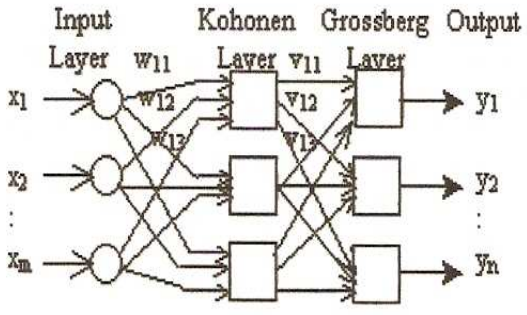


Fig. 17.10 Feed-forward Counterpropagation Network

Two counterpropagation neural networks are used, one (CPNN1) for computation of bus voltage magnitudes at all the *PQ* type buses, while the other (CPNN2) for

computation of bus voltage angles at *PV* type and *PQ* type buses. After training, the knowledge about the voltage magnitudes at all the *PQ* buses and voltage angle at different *PV* and *PQ* buses for various contingencies under different system operating conditions (training patterns) are stored in the structured memory by the trained CPNNs.

B. CPNN Solution Algorithm

The solution algorithm for power flow problem using CPNN is as follows:

(i) A large number of load patterns are generated randomly by perturbing the load at all the buses, real power generation at the generator buses, voltage magnitudes at *PV* & slack buses and transformer tap settings.

(ii) AC power flow (NR) programs are run for all the load patterns and also for contingency cases to calculate bus voltage magnitudes at all the *PQ* type buses and voltage angle at all the *PV* and *PQ* type buses except the slack bus.

(iii) The diagonal elements of the bus conductance and susceptance matrix (active and reactive loads added to it), voltage magnitudes at PV and slack buses and real power generations at PV buses are selected as input features.

(iv) All the input vectors and the initial randomized weights for Kohonen' layer are normalized before applying to the counterpropagation network.

(v) Train the Kohonen clustering network by applying it to the CPNN competitive layer. Set iteration count C =1.

(vi) Determine the unit (neuron) that wins the competition by determining the unit k whose vector wk is closest to the given input.

(vii) Update the winning unit's weight vector as

$$w_k(C+1) = w_k(C) + \eta\,(x_i - w_k(C))$$

(viii) Repeat Steps (v) through (vii) until all input vectors are applied.

(ix) Increase the iteration count by one ($C = C+1$) and repeat Steps (v) through (viii) until all input vectors are grouped properly by applying the training vectors several times.

(x) After training the kohonen's layer, apply a normalized input vector x_i to the input layer and the corresponding desired output vector y_i to the output layer. (xi) Determine the winning neuron k in the competitive layer.

(xii) Update the weights on the connections from the winning competitive unit to the output units

$$v_k(C+1) = v_k(C) + \eta\,(x_i - v_k(C))$$

(xiii) Repeat Steps (x) through (xiii) until all the input-output pairs in the training data are mapped satisfactorily

A. Simulation results

The IEEE-14 bus system, which is composed of 14 buses and 20 lines, has been used.. The data for IEEE-14-bus system were taken, with buses renumbered to make bus-1 as slack bus having prespecified voltage as $1.06\angle 0°$ p.u., buses 2-5 as *PV* buses and buses 6-14 as load (*PQ*) buses. One CPNN model (CPNN1) was trained to provide bus voltage magnitude at all the *PQ* buses, while the other neural network (CPNN2) was trained to compute the bus voltage angles at all the *PV* and *PQ* type buses.

The total number of inputs is 29, including diagonal values of G and B, real and reactive loads, real bus power generation at bus no. 2, bus voltage magnitudes at 4 PV and the slack buses. For training and testing of CPNN, 25 load scenarios were generated by perturbing the load at all the buses in the range of 50% to 150%, *PV* bus voltage

magnitude between 0.9 to 1.10, real power generation between 80% to 120%, transformer tap setting between 0.9 to 1.10. Single-line outages were considered as contingencies. Newton-Raphson (NR) power flow program was used to generate training / testing patterns for 25 load scenarios and for all the single-line outage contingencies.

The NR method converged for different loading conditions and for 19 line outage cases i.e. for 500 cases. Out of 500 generated patterns, 400 patterns corresponding to 20 load scenarios were arbitrarily selected and used for training of the CPNN, while 100 patterns corresponding to 5 load scenarios were used for testing the performance of the trained counter propagation neural networks.

Two CPNNs were used, one for computation of bus voltage magnitudes at all the 9 *PQ* type buses, while the other for computation of bus voltage angle at 4 *PV* type buses and 9 *PQ* type buses (total 13). The number of hidden neurons (nodes) could be decided using some trial and error method.

Adaptive Resonance theory

Introduction: The adaptive resonance theory (ART), proposed by Carpenter and Grossberg. The ART is useful for classifying patterns by discovering invariant patterns that are crucial for classification. ART recognizes that much of learning occurs without teacher, although ART can be used in a supervised mode also. Of the two versions of ART, ART1 refers to a network that handles binary inputs and ART2 refers to one that can handle analog inputs.

- **ART1**: for binary patterns; **ART2**: for continuous patterns
- Motivations: Previous methods have the following problems:
    1. Number of class nodes is pre-determined and fixed.
        - Under- and over- classification may result from training
        - Some nodes may have empty classes.
        - no control of the degree of similarity of inputs grouped in one class.
    2. Training is non-incremental:
        - with a fixed set of samples,
        - adding new samples often requires re-train the network with the enlarged training set until a new stable state is reached.
- Ideas of ART model:
    - suppose the input samples have been appropriately classified into k clusters (say by some fashion of competitive learning).
    - each weight vector is a representative (average) of all samples in that cluster.
    - when a new input vector ***x*** arrives
        1. Find the winner *j** among all k cluster nodes
        2. Compare w_{j^*} with ***x***

      if they are sufficiently similar (***x*** resonates with class j*),

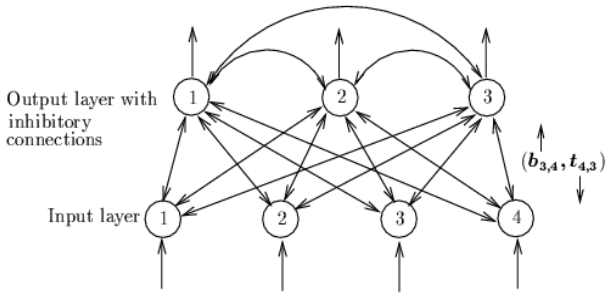
      then update w_{j^*} based on $|x - w_{j^*}|$

      else, find/create a free class node and make ***x*** as its

first member.

- To achieve these, we need:
  - a mechanism for testing and determining (dis)similarity between x and w_{j*}
  .a control for finding/creating new class nodes.
  - need to have all operations implemented by units of local computation.
- Only the basic ideas are presented
  - Simplified from the original ART model
  - Some of the control mechanisms realized by various specialized neurons are done by logic statements of the algorithm

ART1 Architecture



x: input (input vectors)

y: output (classes)

$b_{j,i}$: bottom up weights from x_i to y_j (real values)

$t_{i,j}$: top down weights from y_j to x_i (binary/bipolar)

ρ: vigilance parameter for similarity comparison ($0 < \rho < 1$)

Working of ART1

- 3 phases after each input vector ***x*** is applied
- ***Recognition phase***: determine the winner cluster for ***x***
  - Using bottom-up weights ***b***
  - Winner j^* with max $yj^* = bj^* \odot x$
  - x is tentatively classified to cluster j^*
  - the winner may be far away from x (e.g., $|tj^* - x|$ is unacceptably large)
- ***Comparison phase***:
  - Compute similarity using top-down weights ***t***:

vector: $s^* = (s_1^*, ..., s_n^*)$ where $s_i^* = t_{l,j^*} \cdot x_l$

$$s_i^* = \begin{cases} 1 & \text{if both } t_{l,j^*} \text{ and } x_l \text{ are } 1 \\ 0 & \text{otherwise} \end{cases}$$

  - If (# of 1's in s)|/(# of 1's in x) > ρ, accept the classification, update bj^* and tj^*

– else: remove *j** from further consideration, look for other potential winner or create a new node with *x* as its first patter.

- ***Weight update/adaptive phase***
  – Initial weight: (no bias)

bottom up: $b_{j,l}(0) = 1/(n+1)$ top down: $t_{l,j}(0) = 1$

– When a resonance occurs with node j^*, update b_{j^*} and t_{j^*}

$$b_{j^*,l}(new) = \frac{s_l^*}{0.5 + \sum_{l=1}^{n} s_l^*} = \frac{t_{l,j^*}(old)x_l}{0.5 + \sum_{l=1}^{n} t_{l,j^*}(old)x_l}$$

– If k sample patterns are clustered to node *j* then
t_j = pattern whose 1's are common to all these k samples

$$t_j(\text{new}) = t_j(0) \wedge x(1) \wedge x(2).....x(k) = x(1) \wedge x(2).....x(k)$$

$b_{j,l}(\text{new}) \neq 0$ iff $s_l \neq 0$ only if $x_l(i) \neq 0$, b_j is a normalized t_j

Algorithm of ATR1: Intialize each $t_{l,j}(0) = 1$, $b_{j,l} = \frac{1}{n+1}$

1. Let A contains all nodes.
2. For random chosen input vector X, compute $y_j = b_j * x$ fro each $j \in$ A.
3. Repeat
   (a) Let j^* be a node in A with largest y_j.
   (b) Compute $s^* = (s_1^*, ..., s_n^*)$ where $s_l^* = t_{l,j} * x_l$
   (c) If $\frac{\sum_{l=1}^{n} s_l^*}{\sum_{l=1}^{n} x_l} \leq \rho$ then remove j^* from set A
   Else associate x with node j^* and update weights:

$$b_{j^*,l}(new) = \frac{t_{l,j^*}(old)x_l}{0.5 + \sum_{l=1}^{n} t_{l,j^*}(old)x_l}$$

$$t_{l,j^*}(new) = t_{l,j^*}(old)x_l$$

until A is empty or x is associated with some node;

4. If A is empty, create, create new node with weight vector
   end - while.

Example: ρ=0.7, n=7; input patterns
$x(1) = (1, 1, 0, 0, 0, 0, 1)$ $x(2) = (0, 0, 1, 1, 1, 1, 0)$ $x(3) = (1, 0, 1, 1, 1, 1, 0)$ $x(4) = (0, 0, 0, 1, 1, 1, 0)$
$x(5) = (1, 1, 0, 1, 1, 1, 0)$
initially: $t_{l,1}(0) = 1,\ b_{1,l}(0) = 1/8$

For input x(1); $y_1 = \frac{1}{8}\times 1 + \frac{1}{8}\times 1 + \frac{1}{8}\times 0 + \ldots\ldots \frac{1}{8}\times 0 + \frac{1}{8}\times 1 = \frac{3}{8}$

Node 1 wins $\frac{\sum_{l=1}^{7} t_{l,1} x_l}{\sum_{l=1}^{7} x_l} = \frac{3}{3} = 1 > 0.7$

$$b_{1,l}(1) = \begin{cases} \frac{1}{0.5+3} = \frac{1}{3.5} & for\ l = 1,2,7; \\ 0 & \text{otherwise} \end{cases}$$

$$t_{l,1}(1) = t_{l,1}(0) x_l$$

$$B(1) = \begin{bmatrix} \frac{1}{3.5} & \frac{1}{3.5} & 0 & 0 & 0 & 0 & \frac{1}{3.5} \end{bmatrix}$$

$$T(1) = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

For the second sample (0,0,1,1,1,1,0), y_1=0, but, $\frac{\sum_{l=1}^{7} t_{l,1} x_l}{\sum_{l=1}^{7} x_l} = 0 < \rho$, A new node is generated with

$$B(2) = \begin{bmatrix} \frac{1}{3.5} & \frac{1}{3.5} & 0 & 0 & 0 & 0 & \frac{1}{3.5} \\ 0 & 0 & \frac{1}{4.5} & \frac{1}{4.5} & \frac{1}{4.5} & \frac{1}{4.5} & 0 \end{bmatrix}$$

and

$$T(2) = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

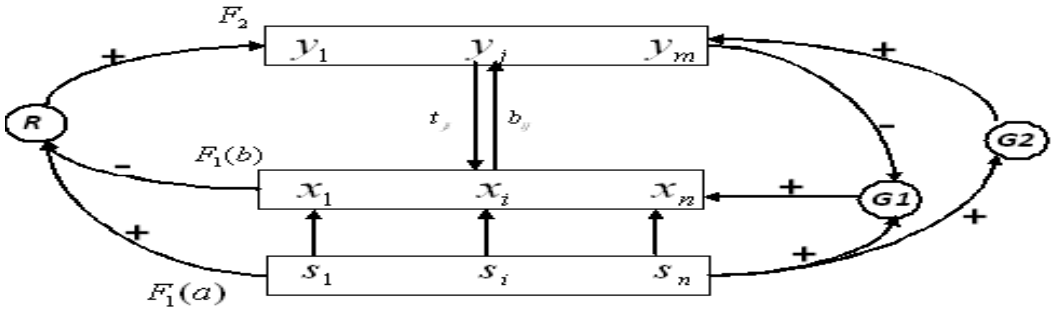
The network stabilizes with three nodes, in two presentations of each input vector.

Notes:

1. Classification as a search process
2. No two classes have the same ***b*** and ***t***
3. Outliers that do not belong to any cluster will be assigned separate nodes

4. Different ordering of sample input presentations may result in different classification.
5. Increase of r increases # of classes learned, and decreases the average class size.
6. Classification may shift during search, will reach stability eventually.
7. There are different versions of ART1 with minor variations
8. ART2 is the same in spirit but different in details.

ART1 Architecture



F_1(a) : input units, $F_1(b)$: interface units , F_2 cluster units, F_1(a) to F_1(b) pair-wise connection between F_2 and $F_1(b)$: full connection, b_{ij}: bottomup weights from x_i to y_j (real value). t_{ji} topdown weights from y_j to x_i (representing class j binary/bipolar).

- F_2 cluster units: competitive, receive input vector ***x*** through weights ***b:*** to determine winner ***j***.
- $F_1(a)$ input units: placeholder or external inputs
- $F_1(b)$ interface units:
  - pass ***s*** to ***x*** as input vector for classification by F_2
  - compare ***x*** and t_{j^*} (projection from winner y_j)
  - controlled by gain control unit ***G1***

  Nodes both F_1(b) and F_2 obey 2/3 rule (output 1 if two of the three inputs are 1).

  Input to F_1(b):s_i, G1,t_{ji} ; Input to F_2: t_{ji}, G2, R

- Needs to sequence the three phases (by control units ***G1, G2,*** and ***R***)

$$G_1 = \begin{cases} 1 & \text{if } s \neq 0 \text{ and } y = 0 \\ 0 & \text{otherwise} \end{cases}$$

$G_1 = 1$: $F_1(b)$ open to receive $s \neq 0$

$G_1 = 1$: $F_1(b)$ open to for t_{j^*}

$$G_2 = \begin{cases} 1 & \text{if } s \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

G_2 =1 signals the start of a new classification for a new input.

$$R = \begin{cases} 0 & \text{if } \dfrac{\|x\|}{\|s\|} \geq \rho \\ 1 & \text{otherwise} \end{cases} \qquad 0<\rho<1 \text{ vigilance parameter.}$$

R = 0: resonance occurs, update and ***R*** = 1: fails similarity test, inhibits ***J*** from further computation