UNIT 4 FUNDAMENTALS OF NEURAL NETWORKS

Structure **Page No**

4.1 INTRODUCTION

Research in the field of neural networks has been attracting increasing attention in recent years. Since 1943, when Warren McCulloch and Walter Pitts presented the first model of artificial neurons, new and more sophisticated proposals have been made from decade to decade. Mathematical analysis has solved some of the mysteries posed by the new models but has left many questions open for future investigations. Needless to say, the study of neurons, their interconnections, and their role as the brain's elementary building blocks is one of the most dynamic and important research fields in modern biology.

We shall begin this unit by defining the introduction to neural networks in Sec. 4.2 called neuron, which is the simplest kind of computing units used to build artificial neural networks.

Several associative neural memory models have been proposed over the last two decades e.g., Amari, 1972a; Anderson, 1972; Nakano, 1972; Kohonen, 1972 and 1974; Kohonen and Ruohonen, 1973; Hopfield, 1982; Kosko, 1987; Okajima et al., 1987; Kanerva, 1988; Chiueh and Goodman, 1988; Baird, 1990. These memory models can be classified in various ways depending on their architecture (static versus recurrent), their retrieval mode (synchronous versus asynchronous), the nature of the stored associations (autoassociative versus heteroassociative), the complexity and capability of the memory storage/recording algorithm, etc. A simple static synchronous associative memory is presented along with appropriate memory storage recipes. Then, this simple associative memory is extended into a recurrent auto associative memory by employing feedback. We shall describe these models in Sec. 4.3.

Computing elements neurons are a generalization of the common logic gates used in conventional computing and, since they operate by comparing their total input with a threshold, this field of research is known as threshold logic. We shall discuss threshold logic in Sec. 4.4.

Error correction rules were initially proposed as ad hoc rules for single unit training. These rules essentially drive the output error of a given unit to zero. In Sec. 4.5, we shall start with the classical perceptron learning rule and give a proof for its convergence.

Objectives

After studying the unit, you should be able to:

- define with artificial neural networks, relate between neurons and artificial computing units;
- describe the simplest kind of computing units used to build artificial neural networks;
- generalise the common logic gates used in conventional computing.

4.2 NEURAL NETWORKS

Works on artificial neural networks, commonly referred to as 'neural networks'.

Artificial neural networks are an attempt to do the modeling of the information processing capabilities of nervous systems. Thus, first of all, it is necessary to consider the essential properties of biological neural networks from the viewpoint of information processing. By this, we can design abstract models of artificial neural networks, which can then be simulated and analyzed.

Neural networks theory always emphasizes that the human brain computers in an entirely different way from the conventional digital computer. Although the models which have been proposed to explain the structure of the brain and the nervous systems of some animals are different in many respects, there is a general consensus that the essence of the operation of neural ensembles is "control through communication". Animal nervous systems are composed of thousands or millions of interconnected cells. Each one of them is a very complex arrangement which deals with incoming signals in many different ways. However, neurons are rather slow when compared to electronic logic gates. These can achieve switching times of a few nanoseconds, whereas neurons need several milliseconds to react to a stimulus. Nevertheless the brain is capable of solving problems which no digital computer can yet efficiently deal with. Massive and hierarchical networking of the brain seems to be the fundamental precondition for the emergence of consciousness and complex behavior. So far, however, biologists and neurologists have concentrated their research on uncovering the properties of individual neurons. Today, the mechanisms for the production and transport of signals from one neuron to the other are well-understood physiological phenomena, but how these individual systems cooperate to form complex and massively parallel systems capable of incredible information processing feats has not yet been completely elucidated. Mathematics, physics, and computer science can provide invaluable help in the study of these complex systems. It is not surprising that the study of the brain has become one of the most interdisciplinary areas of scientific research in recent years. However, we should be careful with the metaphors and paradigms commonly introduced when dealing with the nervous system. It seems to be a constant in the history of science that the brain has always been compared to the most complicated contemporary artifact produced by human industry.

In ancient times the brain was compared to a pneumatic machine, in the Renaissance to clockwork, and at the end of the last century to the telephone network. There are some today who consider computers the paradigm par excellence of a nervous system. It is rather paradoxical that when John von Neumann wrote his classical description of future universal computers, he tried to choose terms that would describe computers in terms of brains, not brains in terms of computers. The nervous system of an animal is an information processing totality. The sensory inputs, i.e., signals from the environment, are coded and processed to evoke the appropriate response. Biological neural networks are just one of many possible solutions to the problem of processing information. The main difference between neural networks and conventional computer systems is the massive parallelism and redundancy which they exploit in order to deal with the unreliability of the individual computing units. Moreover, biological neural networks are self-organizing systems and each individual neuron is also a delicate self-organizing structure capable of processing information in many different ways.

We can cite an example of human vision. Human vision is the function of visual system, which provides the information we need to interact with the environment. It is an information-processing task.

In this way, we can generalised neural network as a machine which is designed to model the way in which the brain performs a particular task or function of interest. A neural network has a complex interconnection of simple computing cells. These cells are known as processing units or 'neurons'. The benefits of neural networks are as follows:

i) The most important property of a neuron is non-linearity. All neurons are interconnected in non linear form.

ii) Neural networks offer input-output mapping, that is a unique input signal has a corresponding desired output.

iii) Neural networks are adaptive to changes in the surrounding environment.

iv) Neural networks are used to provide the information not only to select a particular pattern, but also to find the confidence in the decision made.

v) A neural network can be designed to deal with a contextual information.

vi) Neural network degrades gracefully under adverse operating conditions.

vii) Neural network is well suited for implementations using VLSI (Very large scale integrated) technology.

viii) Neural networks enjoy uniformity of analysis and design.

ix) Neural networks are applicable for the interpretation of neural biological phenomenon as research tool.

Models of Computation

Artificial neural networks can be considered as just another approach to the problem of computation. The first formal definitions of computability were proposed in the 1930s and '40s and at least five different alternatives were studied at the time. The computer era was started, not with one single approach, but with a contest of alternative computing models. The five principal models of computation are described below.

1. The Mathematical Model

Mathematicians avoided dealing with the problem of a function's computability until the beginning of this century. This happened not just because existence theorems were considered sufficient to deal with functions, but mainly because nobody had come up with a satisfactory definition of computability, certainly a relative concept which depends on the specific tools that can be used. For example, the general solution for an algebraic equations of degree six cannot be formulated using only algebraic functions, yet this can be done if a more general class of functions is allowed as computational primitives. Another example, the squaring of the circle, is impossible using ruler and compass, but it has a trivial real solution. We can start to specify the available tools with the idea that some primitive functions and composition rules are "obviously" computable. All other functions which can be expressed in terms of these primitives and composition rules are then also computable. David Hilbert, the famous German mathematician, was the first to state the conjecture that a certain class of functions contains all intuitively computable functions. Hilbert was referring to the primitive recursive functions, the class of functions which can be constructed from the zero and successor function using composition, projection, and a deterministic number of iterations (primitive recursion). However, in 1928, Wilhelm Ackermann was able to find a computable function which is not primitive recursive. This led to the definition of the general recursive functions. In this formalism, a new composition rule has to be introduced, the so-called μ operator, which is equivalent to an indeterminate recursion

or a lookup in an infinite table. At the same time Alonzo Church and collaborators developed the lambda calculus, another alternative to the mathematical definition of the computability concept. In 1936, Church and Kleene were able to show that the general recursive functions can be expressed in the formalism of the lambda calculus. This led to the Church thesis that computable functions are the general recursive functions.

2. The Logic-Operational Model (Turing Machines)

Alan Turing introduced another kind of computing model known as turning machine. The advantage of his approach is that it consists in an operational, mechanical model of computability. A Turing machine is composed of an infinite tape, in which symbols can be stored and read again. A read-write head can move to the left or to the right according to its internal state, which is updated at each step. The Turing thesis states that computable functions are those which can be computed with this kind of device. It was formulated concurrently with the Church thesis and Turing was able to show almost immediately that they are equivalent. The Turing approach made clear for the first time what "programming" means, curiously enough at a time when no computer had yet been built.

3. The Computer Model

The first electronic computing devices were developed in the 1930s and '40s. Since then, "computation-with-the-computer" has been regarded as computability itself. However the first engineers developing computers were for the most part unaware of Turing's or Church's research. Konrad Zuse, for ex-ample, developed in Berlin between 1938 and 1944 the computing machines Z1 and Z3 which were programmable but not universal, because they could not reach the whole space of the computable functions. Zuse's machines were able to process a sequence of instructions but could not iterate. Other computers of the time, like the Mark I built at Harvard, could iterate a constant number of times but were incapable of executing open-ended iterations (WHILE loops). Therefore the Mark I could compute the primitive but not the general recursive functions. Also the ENIAC, which is usually hailed as the world's first electronic computer, was incapable of dealing with open-ended loops, since iterations were determined by specific connections between modules of the machine. First computer was able to cover all computable functions by making use of conditional branching and self-modifying programs, which is one possible way of implementing indexed addressing.

4. Cellular Automata

In machines like the Mark I and the ENIAC there was no clear separation between memory and processor, and both functional elements were intertwined. Some machines still worked with base 10 and not 2, some were sequential and others parallel. John von Neumann, who played a major role in defining the architecture of sequential machines, analyzed at that time a new computational model which he called cellular automata. Such automata operate in a "computing space" in which all data can be processed simultaneously. The main problem for cellular automata is communication and coordination between all the computing cells. This can be guaranteed through certain algorithms and conventions. It is not difficult to show that all computable functions, in the sense of Turing, can also be computed with cellular automata, even of the one-dimensional type, possessing only a few states. Turing himself considered this kind of computing model at one point in his career. Cellular automata as computing model resemble massively parallel multi-processor systems of the kind that has attracted considerable interest recently.

5. The Biological Model (Neural Networks)

The explanation of important aspects of the physiology of neurons set the stage for the formulation of artificial neural network models which do not operate sequentially, as Turing machines do. Neural networks have a hierarchical multilayered structure which sets them apart from cellular automata, so that information is transmitted not only to the immediate neighbors but also to more distant units. In artificial neural networks one can connect each unit to any other. In contrast to conventional computers, no program is handed over to the hardware-such a program has to be created, that is, the free parameters of the network have to be found adaptively. Although neural networks and cellular automata are potentially more efficient than conventional computers in certain application areas, at the time of their conception they were not yet ready to take center stage. The necessary theory for harnessing the dynamics of complex parallel systems is still being developed right before our eyes. In the meantime, conventional computer technology has made great strides. There is no better illustration for the simultaneous and related emergence of these various computability models than the life and work of John von Neumann himself. He participated in the definition and development of at least three of these models: in the architecture of sequential computers, the theory of cellular automata and the first neural network models. He also collaborated with Church and Turing in Princeton.

Artificial neural networks have, as initial motivation, the structure of biological systems, and constitute an alternative computability paradigm.

Elements of a Computing Model

What are the elementary components of any conceivable computing model? In the theory of general recursive functions, for example, it is possible to reduce any computable function to some composition rules and a small set of primitive functions. For a universal computer, we ask about the existence of a minimal and sufficient instruction set. For an arbitrary computing model, metaphoric expression has been proposed and is given in Fig. 1.

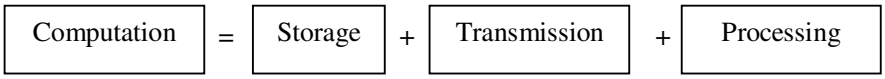


Fig. 1: Metaphoric expression for computing

The mechanical computation of a function presupposes that these three elements are present, that is, that data can be stored, communicated to the functional units of the model and transformed. It is implicitly assumed that a certain coding of the data has been agreed upon. Coding plays an important role in information processing because, as Claude Shannon showed in 1948, when noise is present information can still be transmitted without loss, if the right code with the right amount of redundancy is chosen.

Modern computers transform storage of information into a form of information transmission. Static memory chips store a bit as a circulating current until the bit is read. Turing machines store information in an infinite tape, whereas transmission is performed by the read-write head. Cellular automata store information in each cell, which at the same time is a small processor.

In biological neural networks information is stored at the contact points between different neurons, the so-called **synapses**. Other forms of storage are also known, because neurons are themselves complex systems of self-organizing signaling.

Nervous systems possess global architectures of variable complexity, but all are composed of similar building blocks, the neural cells or **neurons**. They can perform

different functions, which in turn leads to a very variable morphology. If we analyze the human cortex under a microscope, we can find several different types of neurons. Although the neurons have very different forms, it is possible to recognize a hierarchical structure of different layers. Each one has specific functional characteristics. Neurons receive signals and produce a response. The general structure of a generic neuron has branches to the left which are the transmission channels for incoming information and are called **dendrites**. Dendrites receive the signals at the contact regions with other cells. Some animals have neurons with a very different morphology. In insects, for example, the dendrites go directly into the axon and the cell body is located far from them. Organelles in the body of the cell produce all necessary chemicals for the continuous working of the neuron. The mitochondria is a part used to supply energy of the cell, since they produce chemicals which are consumed by other cell structures. The output signals are transmitted by the axon, of which each cell has at most one. Some cells do not have an axon, because their task is only to set some cells in contact with others. These four elements, dendrites, synapses, cell body, and axon are the minimal structure we will adopt from the biological model. Artificial neurons for computing will have input channels, a cell body and an output channel. Synapses will be simulated by contact points between the cell body and input or output connections; a weight will be associated with these points.

Now try the following exercises.

E1) Write all the elements of a neurons.

E2) Describe all the models of computation.

In the following section, we shall discuss the model of neuron.

4.3 MODELS OF A NEURON

An artificial neuron takes input to produce output with the help of the transfer function or activation function. Therefore, different types of transfer functions are in use, such as threshold function, sigmoid function and others. These functions are discussed below.

i) **Threshold function:** Threshold functions is also known as Heaviside function or Hard-limit transfer function. In this function, the output is 0 if the input is less than 0 otherwise the output is 1. Generally, it is used in a perceptron neuron. This is defined as

$$\phi(u) = \begin{cases} 0, & \text{if } u < 0 \\ 1, & \text{otherwise} \end{cases}$$

This function can be described diagrammatically as shown in Fig 2. The output of a neuron with threshold function can be expressed as

$$y_k = \begin{cases} 0 & \text{if } u_k < 0 \\ 1 & \text{otherwise} \end{cases}$$

where $u_k = \sum_{i=1}^{n} w_{k_i} x_i + b_k$, which is the induced local field.

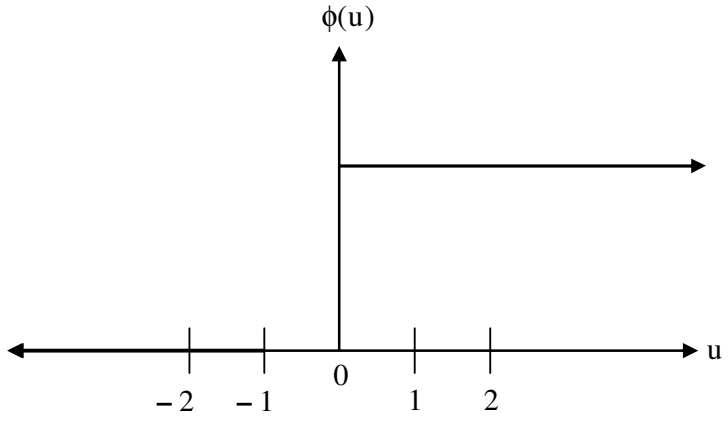


Fig. 2: Threshold function

ii) **Linear Transfer function:** In this function, the output is same as the input within the range from –1 to 1. This is expressed as

$$\phi(u) = u, \text{ where } -1 \le \text{or} \le 1$$

This is generally used in linear filter. The graph of this function is shown in Fig 3.

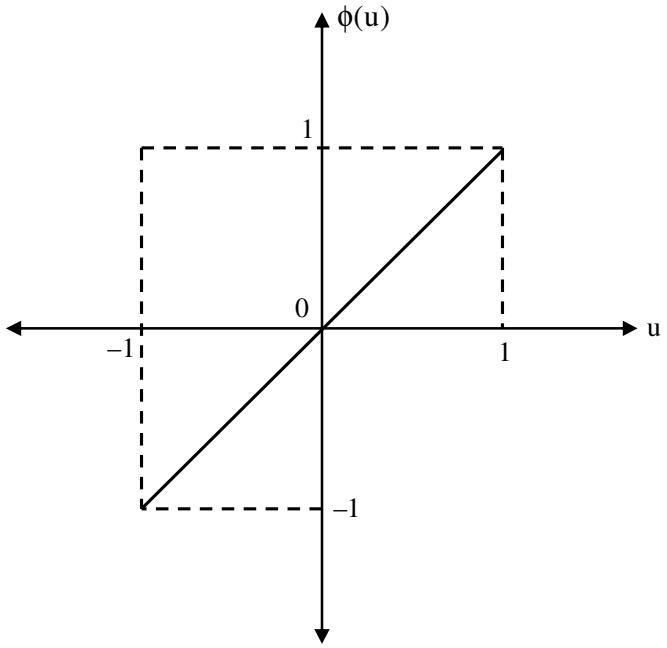


Fig. 3: Linear transfer function

iii) **Sigmoid Function:** The range of the ouput of log-sigmoid is from 0 to 1. It is given by

$$\phi(u) = \frac{1}{1 + e^{-au}}$$

where a is the slope parameter of the log-sigmoid function depends on the value of a. This function is used in a Back-propagation neural network. The graph of a sigmoid function is always S-shaped. It can be seen that for the infinite value of a, it becomes threshold function, which assumes the values 0 and 1. A sigmoid function varies from 0 to 1. Another example of sigmoid function is tan-sigmoid transfer function, which is defined as

$$\phi(u) = \tanh(u) = \frac{e^{au} - e^{-au}}{e^{au} + e^{-au}}$$

The output varies from –1 to 1. The graph of log-sigmoid and tan-sigmoid functions are shown in Fig. 4 and Fig. 5, respectively.

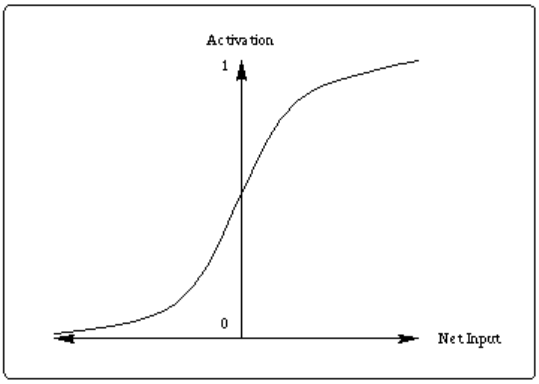


Fig. 4: Log-sigmoid function

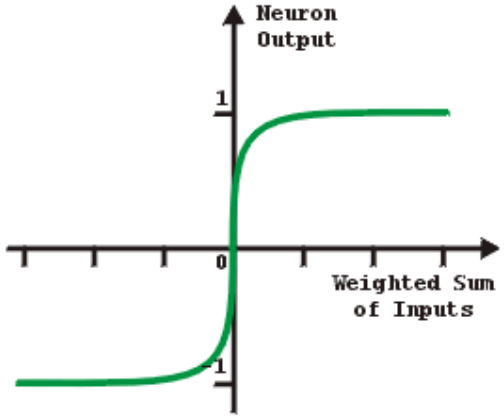


Fig. 5: Tan-sigmoid function

E3) What is the role of an activation function in neural networks?

So far, we introduced the neural networks. Now let us discuss the networks with primitive functions.

Networks of primitive functions

The structure of an abstract neuron is shown in Fig. 6 with n inputs. Each input channel i can transmit a real value x_i. The primitive function f computed in the body of the abstract neuron can be selected arbitrarily. Usually the input channels have an associated weight, which means that the incoming information x_i is multiplied by the corresponding weight w_i. The transmitted information is integrated at the neuron (usually just by adding the different signals) and the primitive function is then evaluated.

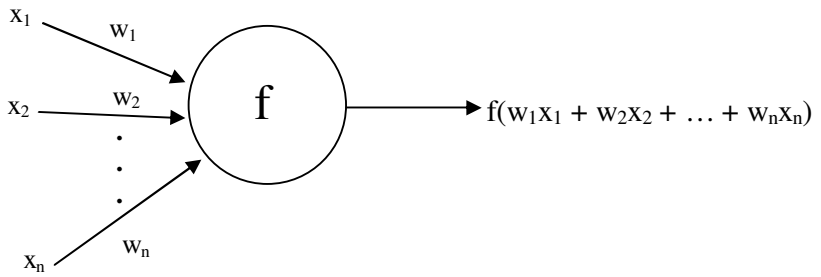


Fig. 6: An abstract neuron

In an artificial neural network, if we consider each node as a primitive function capable of transforming its input in a precisely defined output, then artificial neural networks are nothing but networks of primitive functions. Different models of artificial neural networks differ mainly in the assumptions about the primitive functions used, the interconnection pattern, and the timing of the transmission of information.

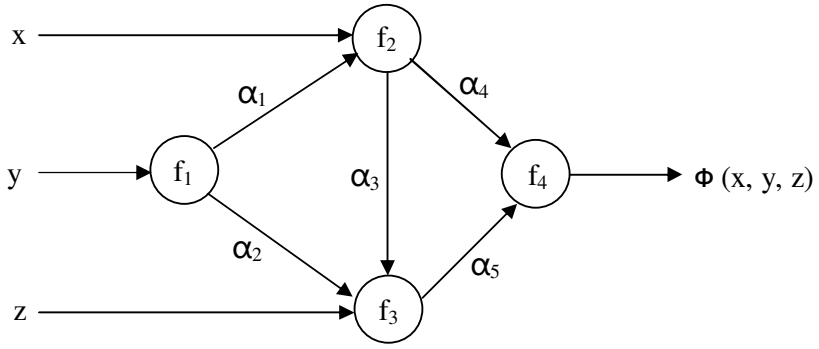


Fig. 7: Functional model of an artificial neural network

The digraph of a typical artificial neural networks have the structure shown in Fig. 7. The network can be thought of as a function Φ which is evaluated at the point (x, y, z). The nodes implement the primitive functions f_1, f_2, f_3, f_4 which are combined to produce Φ and arewritten at vertices. Vertices are connected by the edges. The function Φ implemented by a neural network will be called the **network function** by mentioning weights at different edges. Different selections of the weights $\alpha_1, \alpha_2, \ldots, \alpha_5$ produce different network functions.

Therefore, tree elements are particularly important in any model of artificial neural networks:

- The structure of the nodes,
- The topology of the network,
- The learning algorithm used to find the weights of the network.

The neural architecture may be classified in the three fundamentally different classes of networks, which are shown in Fig. 8.

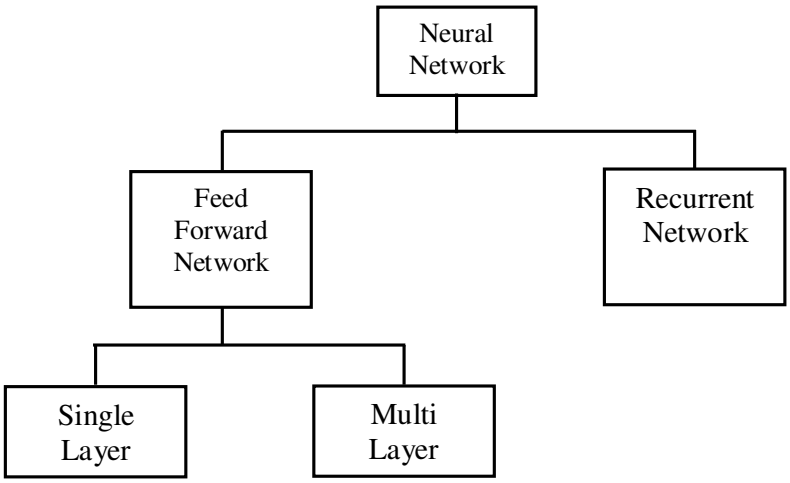
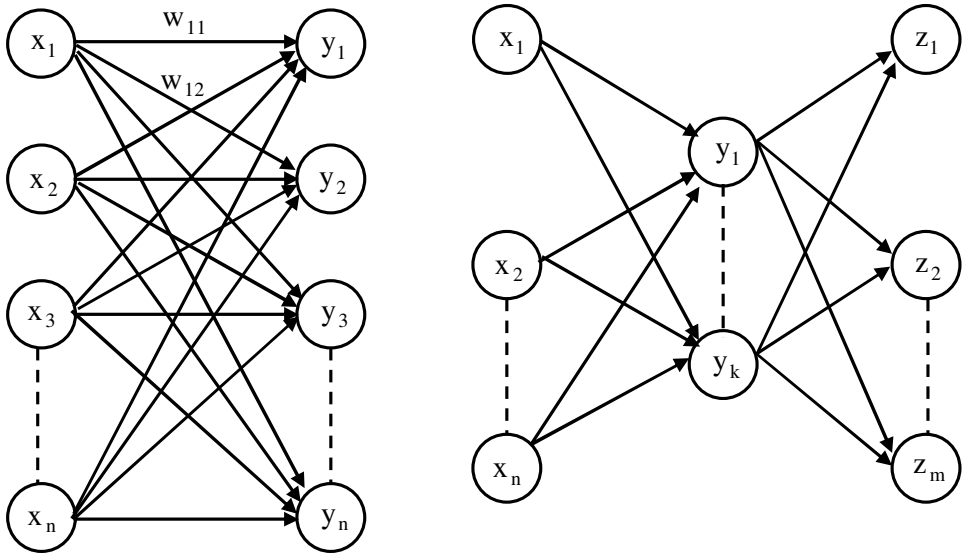


Fig. 8: Types of Neural Networks

For this, consider a input layer made up of input neurons $x_1, x_2, \cdots, x_n$ which receives the input signals and a output layer made up of output neurons $y_1, y_2, \ldots, y_n$, receives output signals. The edges, which carry the synaptic links as weights connect every i^{th} input neurons to the j^{th} output neurons are lebelled by weight w_{ij}, not vice-versa. If in a network the input layer is connected with a output layer in a acyclic way, then such a network is known as feed forward network. In feed forward network, only the output layer performs computation, therefore it is named as single layer feed forward network. In case of multilayer feed forward network, one or more hidden layers are connected to process. These hidden layers perform the useful intermediary computations, before directing the input to the output layer. These networks are shown in Fig. 9.



Single layer **Multi layer**

Fig. 9: Single layer and multi layer perceptrons

Recurrent networks are slightly different from the feed forward networks, in the sense that they work in cyclic way. In recurrent networks, the output layer is feedback into itself as input. These are shown in Fig. 10.

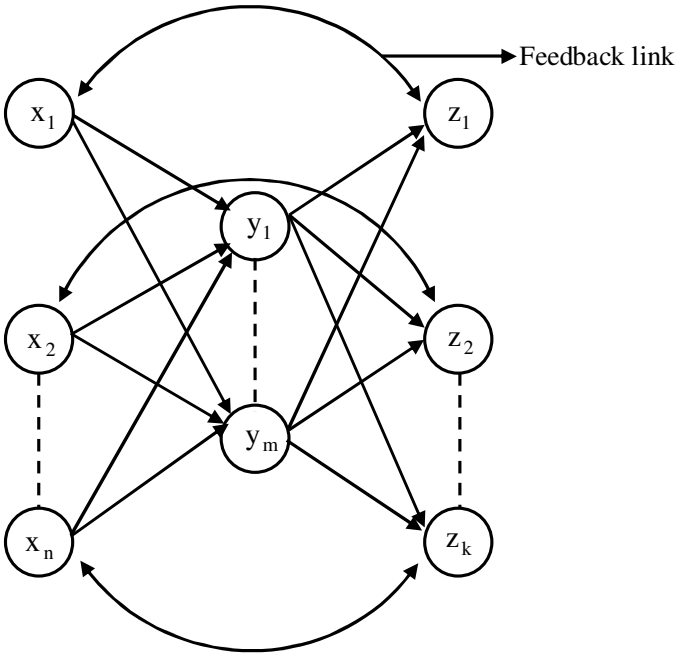


Fig. 10: Recurrent Networks

At this point we must discuss the limitation in the theory of artificial neural networks, we do not consider the whole complexity of real biological neurons. We only abstract some general principles and content ourselves with different levels of detail when simulating neural ensembles. The general approach is to conceive each neuron as a primitive function producing numerical results at some points in time. However we can also think of artificial neurons as computing units which produce pulse trains in the way that biological neurons do. We can then simulate this behavior and look at the output of simple networks. This kind of approach, although more closely related to the biological paradigm, is still a very rough approximation of the biological processes.

There are three aspects to the construction of a neural network:

1. **Structure –** the architecture and topology of the neural network.
2. **Encoding –** the method of changing weights/modifying weights.
3. **Recall –** the method and capacity to retrieve information.

Let's cover the first one – structure. This relates to how many layers the network should contain, and what their functions are, such as for input, for output, or for feature extraction. Structrue also encompasses how interconnections are made between neurons in the network, and what their functions are.

The second aspect is encoding. Encoding refers to the paradigm used for the determination of and changing of weights on the connections between neurons. In the case of the multilayer feed-forward neural network, we initially can define weights by randomization. Subsequently, in the process of training, we can apply the backpropagation algorithm, which is a mean of updating weights starting from the output backwards. When the training is completed in the multilayer feed-forward neural network, the encoding ends since weights do not change after training is completed.

Finally, recall is also an important aspect of a neural network. Recall refers to getting an expected output for a given input. If the same input as before is presented to the network, the same corresponding output as before should result. The type of recall can characterize the network as being autoassociative or heteroassociative.

Autoassociation is the phenomenon of associative an input vector with itself as the output, whereas heteroassociation is that of recalling a related vector given an input vector.

The three aspects to the construction of a neural network mentioned above essentially distinguish between different neural network and are part of their design process.

Now, try the following exercises.

E4) Express the network function ϕ given in Fig. 7, in terms of the primitive functions f_1, f_2, f_3 and f_4 and of the weights $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ and α_5.

E5) Differentiate between the biological neural networks and the artificial neural networks.

So far, we have introduced the neural networks. In the following section, we shall discuss the threshold logic.

4.4 THRESHOLD LOGIC

The computing elements are a generalization of the common logic gates used in conventional computing and, since they operate by comparing their total input with a threshold, which is shown in Fig. 11.

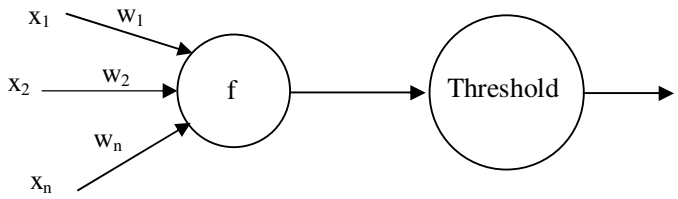


Fig. 11: Threshold logic

The first model we consider was proposed in 1943, by warren McCulloch and Walter Pitt. McCulloch-Pitts networks are simpler, because they use solely binary signals, i.e., ones or zeros. The nodes produce only binary results and the edges transmit exclusively ones or zeros. The networks are composed of directed unweighted edges of excitatory or of inhibitory type. Each McCulloch-Pitts unit is assigned a certain threshold value θ.

Therefore, the McCulloch-Pitts model seems very limited. Since only binary information can be produced and transmitted. But it already contains all necessary features to implement the more complex models. In Fig. 12, following Minsky it will be represented as a circle with a black half. Incoming edges arrive at the white half, outgoing edges leave from the black half. Outgoing edges can fan out any number of times.

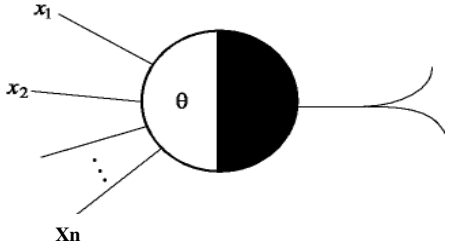


Fig. 12: Diagram of a McCulloch–Pitts unit

The rule for evaluating the input to a McCulloch-Pitts unit is given in the following steps:

Step 1: Assume that a McCulloch-Pitts unit gets an input $x_1, x_2, \ldots, x_n$ through n excitatory edges and an input $y_1, y_2, \ldots, y_m$ through m inhibitory edges.

Step 2: If $m \geq 1$ and at least one of the signals $y_1, y_2, \ldots, y_m$ is 1, the unit is inhibited and the result of the computation is 0. Otherwise the total excitation $x = x_1 + x_2 + \cdots + x_n$ is computed and compared with the threshold θ of the unit (if $n = 0$ then $x = 0$). If $x \geq \theta$ the unit fires a 1, if $x < \theta$ the result of the computation is 0.

This rule implies that a McCulloch-Pitts unit can be inactivated by a single inhibitory signal, as is the case with some real neurons. When no inhibitory signals are present, the units act as a threshold gate capable of implementing many other logical functions of n arguments.

An active unit follows step function. This function changes discontinuously from zero to one at θ. When θ is zero and no inhibitory signals are present, we have the case of a unit producing the constant output one. If θ is greater than the number of incoming excitatory edges, the unit will never fire.

Now, let us define the conjunction, disjunction and negation in case of these units. Simple logical functions can be implemented directly with a single McCulloch-Pitts unit. The output value 1 represents the logical value true and 0, the logical value false. A single unit can compute the disjunction or the conjunction of n arguments. The logic elements can reduce the complexity of the circuit used to implement a given logical function.

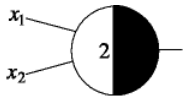
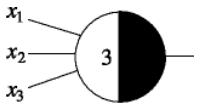
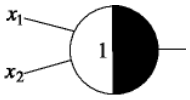
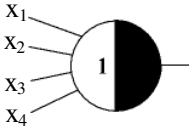
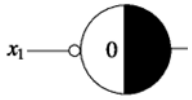
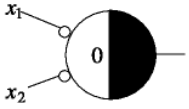
Gates	Implementation	Generalized
Disjunction (AND)		
Conjunction (OR)		
Negation (NOT)		

Fig. 13: Logical gates

Try the following exercise.

E6) Find polynomial expressions corresponding to the OR, AND and NOT for x_1 and x_2.

Geometric Interpretation

It is not easy to visualize the kind of functions that can be computed with McCulloch-Pitts cells by using a diagram. For this, the eight vertices of a three-dimensional unit cube can be considered. Each of the three logical variables x_1, x_2 and x_3 can assume one of two possible binary values. There are eight possible combinations of x_1, x_2 and x_3 which can be represented by the vertices of the cube. A logical function is just an assignment of a 0 or a 1 to each of the vertices. Fig. 13 shows one of these assignments. In the case of n variables, the cube consists of 2^n vertices and admits 2^{2n} different binary assignments.

McCulloch-Pitts units divide the input space into two half-spaces. For a given input (x_1, x_2, x_3) and a threshold θ the condition $x_1 + x_2 + x_3 \geq \theta$ is tested, which is true for all points to one side of the plane with the equation $x_1 + x_2 + x_3 = \theta$ and false for all points to the other side (without including the plane itself in this case). Fig. 14 shows function values of a logical function of x_1, x_2 and x_3.

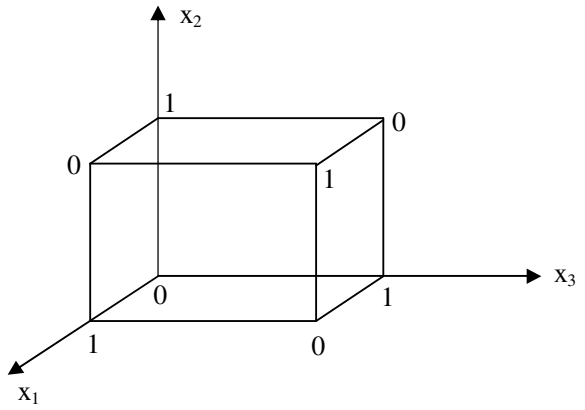


Fig. 14: Function values of a logical function of x_1, x_2 and x_3

Every logical function of n variables can be written in tabular form. The value of the function is written down for every one of the possible binary combinations of the n inputs. If we want to build a network to compute this function, it should have n inputs and one output. The network must associate each input vector with the correct output value. If the number of computing units is not limited in some way, it is always possible to build or synthesize a network which computes this function. The constructive proof of this proposition profits from the fact that McCulloch-Pitts units can be used as binary decoders.

Example 1: Consider the vector $(1, 0, 1)$. It is the only one which fulfills the condition $x_1 \wedge \neg x_2 \wedge x_3$. This condition can be tested by a single computing unit.

Assume that a function F of three arguments has been defined according to the following table: To compute this function it is only necessary to decode all those vectors for which the function's value is 1. Fig. 15 shows a network capable of computing the function F.

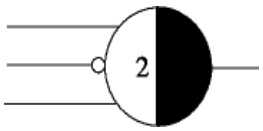


Fig. 15: Decoder for the vector (1, 0, 1)

Input vectors	F
(0, 0, 1)	1
(0, 1, 0)	1
All others	0

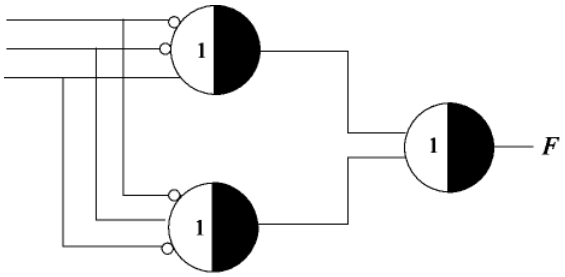


Fig. 16: Synthesis of the function F

The individual units in the first layer of the composite network are decoders. For each vector for which F is 1 a decoder is used. In our case we need just two decoders. Components of each vector which must be 0 are transmitted with inhibitory edges, components which must be 1 with excitatory ones. The threshold of each unit is equal to the number of bits equal to 1 that must be present in the desired input vector. The last unit to the right is a disjunction: if any one of the specified vectors can be decoded this unit fires a 1.

Example 2: Assume that three weighted edges converge on the unit shown in Fig. 17(a). The unit computes Equivalent networks $0.3x_1 + 0.2x_2 + 0.4x_3 \geq 0.7$ or $2x_1 + 2x_2 + 4x_3 \geq 7$.

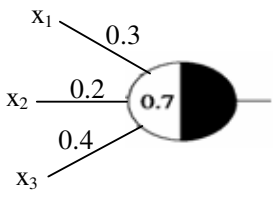


Fig. 17 (a): Weighted units

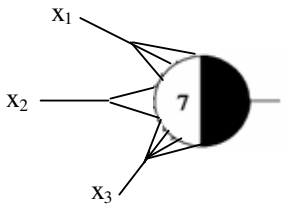


Fig. 17 (b): Equivalent computing unit

The Fig. 17 shows that positive rational weights can be simulated by simply fanning-out the edges of the network required number of times. This means that we can either use weighted edges or go for a more complex topology of the network, with many redundant edges.

The same can be done in the case of irrational weights if the number of input vectors is finite.

Applications of Threshold Logic

Threshold units can be used in any application in which we want to reduce the

execution time of a logic operation to possibly just two layers of computational delay without employing a huge number of computing elements. It has been shown that the parity and majority functions, for example, cannot be implemented in a fixed number of layers of computation without using an exponentially growing number of conventional logic gates, even when unbounded fan-in is used. The majority function k out of n is a threshold function implementable with just a single McCulloch-Pitts unit. Although circuits built from n threshold units can be built using a polynomial number $P(n)$ of conventional gates the main difference is that conventional circuits cannot guarantee a constant delay. With threshold elements we can build multiplication or division circuits that guarantee a constant delay for 32 or 64-bit operands. Any symmetric Boolean function of n bits can in fact be built from two layers of computing units using $n+1$ gates. Some authors have developed circuits of threshold networks for fast multiplication and division, which are capable of operating with constant delay for a variable number of data bits. Threshold logic offers thus the possibility of harnessing parallelism at the level of the basic arithmetic operations.

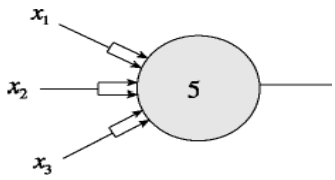


Fig. 18: Fault-tolerant gate

Threshold logic also offers a simpler way to achieve fault-tolerance. Fig. 18 shows an example of a unit that can be used to compute the conjunction of three inputs with inherent fault tolerance. Assume that three inputs x_1, x_2, x_3 can be transmitted, each with probability p of error. The probability of a false result when x_1, x_2 and x_3 are equal, and we are computing the conjunction of the three inputs, is $3p$, since we assume that all three values are transmitted independently of each other. But assume that we transmit each value using two independent lines. The gate of Fig. 18 has a threshold of 5, that is, it will produce the correct result even in the case where an input value is transmitted with an error.

The probability that exactly two ones arrive as zeros is $p2$ and, since there are 15 combinations of two out of six lines, the probability of getting the wrong answer is $15p2$ in this case. If p is small enough then $15p2 < 3p$ and the performance of the gate is improved for this combination of input values. Other combinations can be analyzed in a similar way. If threshold units are more reliable than the communication channels redundancy can be exploited to increase the reliability of any computing system

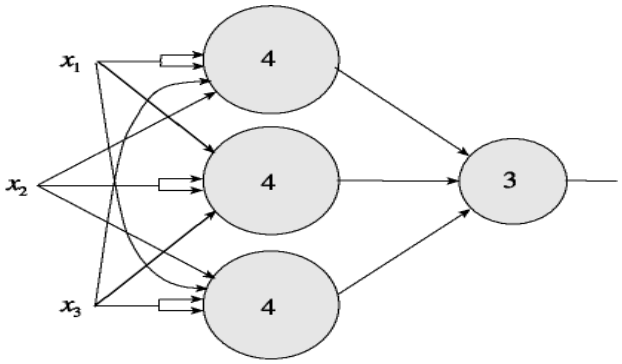


Fig. 19: A fault-tolerant AND built of noisy components

When the computing units are unreliable, fault tolerance is achieved using redundant

networks. Fig. 19 is an example of a network built using four units. Assume that the first three units connected directly to the three bits of input x_1, x_2, x_3 all fire with probability 1 when the total excitation is greater than or equal to the threshold θ but also with probability p when it is $\theta - 1$. The duplicated connections add redundancy to the transmitted bit, but in such a way that all three units fire with probability one when the three bits are 1. Each unit also fires with probability p if two out of three inputs are 1. However each unit reacts to a different combination. The last unit, finally, is also noisy and fires any time the three units in the first level fire and also with probability p when two of them fire. Since, in the first level, at most one unit fires when just two inputs are set to 1, the third unit will only fire when all three inputs are 1. This makes the logical circuit, the AND function of three inputs, built out of unreliable components error-proof.

Now, in the following section, we shall discuss error correction learning.

4.5 ERROR-CORRECTION LEARNING

Neural networks also learn in the same way as we learn from our surroundings. The learning in neural networks may be through a teacher or without a teacher. The learning with a teacher is referred as supervised learning while the learning without a teacher may be categorized as unsupervised learning and reinforced learning. All these learning are performed on neural networks as performed on human learning. In case of supervised learning, the teacher has the knowledge of environment. Here the teacher and the knowledge are considered a set of input-output examples. However, the environment is unknown to the neural networks. The block diagram of this process is given in Fig. 20.

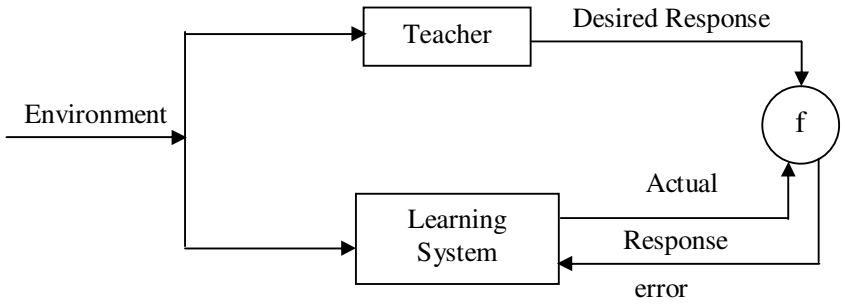


Fig. 20: Block diagram of learning process

If the teacher is not present, the system learns of its own by discovering and adapting to structural features in the input pattern, then the learning is unsupervised learning. While in turn, if the available teacher does not present the expected answer but only helps to identify whether the computed output is correct or incorrect, then the learning is reinforced learning. In this process of learning a reward is given for a correct answer and a panelty for wrong answer.

Consider that a teacher and a neural network are to state a training vector taken from the same environment. In this the teacher has the desired ability about the training vector to provide to the neural network. The response to be performed by the neural network is the optimum action, which is adjusted in between the training vector and the error signal. Therefore,

The error signal = Desired response – the actual response

The adjustment is made iteratively. By this, the knowledge of the teacher about environment is transferred to the neural networks with the help of training. This knowledge is stored in the form of fixed synaptic weights, in the neural networks,

which are represented as long-term memory. In this way, the neural networks are able to perform completely with the environment by itself. This type of supervised learning forms the basis of error-correction learning. In the block diagram of supervised learning, it is clear that the environment, which is not known, is outside the closed loop feedback system. The performance measures of such systems are in terms of mean square error, or the sum of squared errors over the training sample. These are defined as a function of synaptic weights of the system.

In this function, the free parameters are considered as coordinates, and this function is a multi-dimensional error surface or simply error surface. The true error surface is distributed uniformally over all the possible sets of input-output examples. Each point on the surface is a representatives of the given operation under the supervision of a teacher. The point has to move down iteratively to reach towards a minimum point of the error surface. This minimum point is a local minimum. The supervised learning system performs this with the help of the information about the gradient of the error surface. The gradient at any point is a vector quantity. The example of such error surface is a "random walk".

Now the following tasks can be performed with the help of such learning processes.

i) The most important task of a learning process is pattern recognition. Senses receive and then recognize the source of the data by making patterns like faces, voice on the telephone, smelling, etc.

ii) Another important task is pattern association. In this associative memory plays an important role, that learn by associations.

iii) Another important learning task is the approximation of functions.

iv) Control is the learning task for neural networks. Control is of a plant, which may be a process of any system or critical part of the system. This part or process or plant is maintained in a controlled condition.

v) The task of beamforming is done by the learning processes in neural networks. The spatial properties of a target signal and the background noise are distinguished throught beamforming.

Error correction rules were initially proposed as ad hoc rules for single unit training. These rules essentially drive the output error of a given unit to zero. We start with the classical perceptron learning rule and give a proof for its convergence.

Throughout this section, an attempt is made to point out criterion functions that are minimized by using each rule. We will also cast these learning rules as relaxation rules, thus unifying them with the other gradient-based search rules.

i) Perceptron Learning Rule

Consider the following version of a linear threshold gate shown in Fig. 21. We will refer to it as the perceptron. The perceptron maps an input vector $x = x_1\ x_2 \ldots x_{n+1}{}^T$ to a bipolar binary output y, and thus it may be viewed as a simple two-class classifier. The input signal x_{n+1} is usually set to 1 and plays the role of a bias to the perceptron. We will denote by w the vector $[w = w_1\ w_2 \ldots w_{n+1}]^T\ R_{n+1}$ consisting of the free parameters (weights) of the perceptron. The input/output relation for the perceptron is given by $y = \text{sgn}(x^T w)$, where sgn is the "sign" function which returns $+1$ or -1 depending on whether the sign of its scalar argument is positive or negative, respectively.

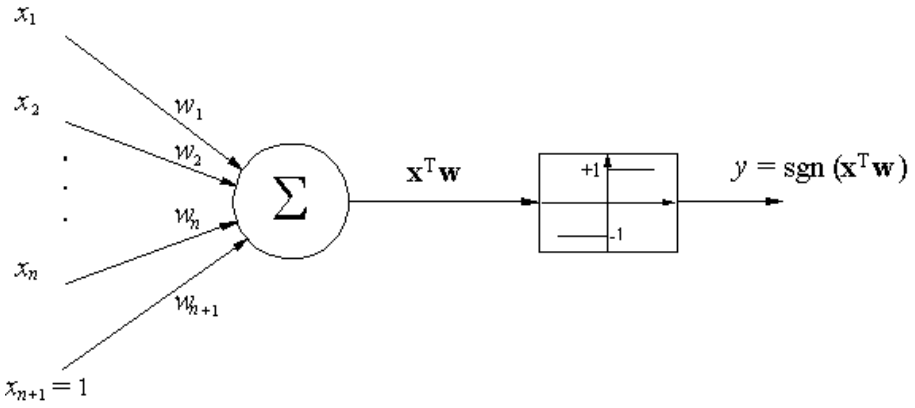


Fig. 21: The perceptron computational unit.

Assume we are training the above perceptron to load (learn) the training pairs: $\{x_1, d_1\}, \{x_2, d_2\}, \ldots, \{x_m, d_m\}$ where $x^k \in R^{n+1}$ is the kth input vector and $d^k \in \{-1, +1\}$, $k = 1, 2, \ldots, m$, is the desired target for the kth input vector (usually, the order of these training pairs is random). The entire collection of these pairs is called the training set. The goal, then, is to design a perceptron such that for each input vector x_k of the training set, the perceptron output yk matches the desired target dk ; that is, we require $y_k = \text{sgn}(w^T x_k) = d_k$ for each $k = 1, 2, \ldots, m$. In this case, we say that the perceptron correctly classifies the training set. Of course, "designing" an appropriate perceptron to correctly classify the training set amounts to determining a weight vector w^* such that the following relations are satisfied:

$$\begin{cases} (x^k)^T w^* > 0 & \text{if} \quad d^k = +1 \\ (x^k)^T W^* < 0 & \text{if} \quad d^k = -1 \end{cases} \tag{1}$$

Recall that the set of all x which satisfy $x^T w^* = 0$ defines a hyperplane in R^n. Thus, in the context of the above discussion, finding a solution vector w^* to Eqn. (32) is equivalent to finding a separating hyperplane which correctly classifies all vectors x_k, $k = 1, 2, \ldots, m$. In other words, we desire a hyperplane $x^T w^* = 0$ which partitions the input space into two distinct regions, one containing all points x_k with $d_k = +1$ and the other region containing all points x_k with $d_k = \sim 1$.

One possible incremental method for arriving at a solution w* is to invoke the perceptron learning rule (Rosenblatt, 1962):

$$\begin{cases} w^1 \quad \text{is set arbitrarily} \\ w^{k+1} = w^k + \rho(d^k - y^k)x^k \quad k = 1, 2, \ldots \end{cases} \tag{2}$$

where k is a positive constant, called the learning rate. The incremental learning process given in Eqn. (2) proceeds as follows. First, an initial weight vector w_1 is selected (usually at random) to begin the process. Then the m pairs $\{x_k, d_k\}$ of the training set are used to successively update the weight vector until (hopefully) a solution w^* is found which correctly classifies the training set. This process of sequentially presenting the training patterns is usually referred to as "cycling" through the training set, and a complete presentation of the m training pairs is referred to as a cycle (or pass) through the training set. In general, more than one cycle through the training set is required to determine an appropriate solution vector. Hence, in Eqn. (2), k in w_k refers to the iteration number. On the other hand, k in x_k (and d_k) is the label of the training pair presented at the kth iteration. To be more precise, if the

number of training pairs, m, is finite, then the superscripts in x_k and d_k should be replaced by $(k \sim 1) \bmod m] + 1$. Here, a mod b returns the remainder of the division of a by b (e.g., $5 \bmod 8 = 5$, and $19 \bmod 8 = 3$). This observation is valid for all incremental learning rules presented here.

Notice that for $= 0.5$, the perceptron learning rule can be written as:

$$\begin{cases} w^1 \text{ is set arbitrarily} \\ w^{k+1} = w^k + z^k \quad \text{if } (z^k)^T w^k \le 0 \\ w^{k+1} = w^k \qquad \text{otherwise} \end{cases} \tag{3}$$

where

$$z^k = \begin{cases} +x^k \quad \text{if } d^k = +1 \\ -x^k \quad \text{if } d^k = -1 \end{cases} \tag{4}$$

That is, a correction is done if and only if a misclassification, indicated by

$$(z^k)^T w^k \le 0 \tag{5}$$

occurs. The addition of vector z_k to w_k in Eqn. (3) moves the weight vector directly toward and perhaps across the hyperplane $(z^k)^T W^k = 0$. The new inner product $(z^k)^T W^{k+1}$ is larger than $(z^k)^T W^k$ by the amount of $\| z^k \|^2$ and the correction $w_k = w_k + \tilde{1} w_k$ is clearly moving w_k in a good direction, the direction of increasing $(z^k)^T W^k$, as can be seen from Fig. 22. Thus, the perceptron learning rule attempts to find a solution w^* for the following system of inequalities

$$(z^k)^T w > 0 \text{ for } k = 1, 2, \ldots, m \tag{6}$$

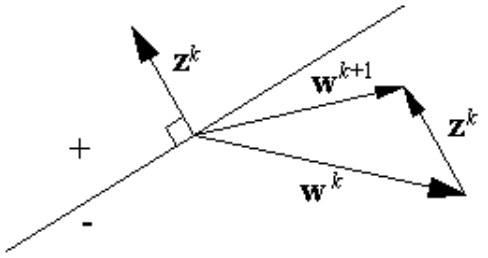


Fig. 22: Geometric representation of the perceptron learning rule with = 0.5.

In an analysis of any learning algorithm, there are two main issues to consider:

1. The existence of solutions and
2. Convergence of the algorithm to the desired solutions (if they exist).

In the case of the perceptron, it is clear that a solution vector (i.e., a vector w^* which correctly classifies the training set) exists if and only if the given training set is linearly separable. Assuming, then, that the training set is linearly separable, we may proceed to show that the perceptron learning rule converges to a solution (Novikoff, 1962; Ridgway, 1962; Nilsson, 1965) as follows. Let w^* be any solution vector, so that

$$(z^k)^T w^* > 0 \text{ for } k = 1, 2, \ldots, m \tag{7}$$

Then, from Eqn. (3), if the kth pattern is misclassified we may write

$$wk+\tilde{1}w^* = wk\ w^* + zk \tag{8}$$

Where k is a positive scale factor, and hence

$$\| w^{k+1} - \alpha w^* \|^2 = \| w^k - \alpha w^* \|^2 + 2(z^k)^T (w^k - \alpha w^*) + \| z^k \|^2 \tag{9}$$

Since zk is misclassified, we have $(z^k)^T W^k \le 0$, and thus

$$\| w^{k+1} - \alpha w^* \|^2 \le \| w^k - \alpha w^* \|^2 - 2\alpha (z^k)^T w^* + \| z^k \|^2 \tag{10}$$

Now, let $\beta^2 = \max_i \| z^i \|^2$ and $\gamma = \min_i (z^i)^T w^*$ (is positive since $(z^i)^T w^* > 0$) and substitute into Eqn. (10) to get

$$\| w^{k+1} - \alpha w^* \|^2 \le \| w^k - \alpha w^* \|^2 - 2\alpha\gamma + \beta^2 \tag{11}$$

If we choose sufficiently large, in particular $\alpha = \frac{\beta^2}{\gamma}$, we obtain

$$\| w^{k+1} - \alpha w^* \|^2 \le \| w^k - \alpha w^* \|^2 - \beta^2 \tag{12}$$

Thus, the square distance between wk and w^* is reduced by at least 2 at each correction, and after k corrections we may write Eqn. (12) as

$$0 \le \| w^{k+1} - \alpha w^* \|^2 \le \| w^1 - \alpha w^* \|^2 - k\beta^2 \tag{13}$$

It follows that the sequence of corrections must terminate after no more than k_0 corrections, where

$$k_0 = \frac{\| w^1 - \alpha w^* \|^2}{\beta^2} \tag{14}$$

Therefore, if a solution exists, it is achieved in a finite number of iterations. When corrections cease, the resulting weight vector must classify all the samples correctly since a correction occurs whenever a sample is misclassified and since each sample appears infinitely often in the sequence. In general, a linearly separable problem admits an infinite number of solutions. The perceptron learning rule in Eqn. (2) converges to one of these solutions. This solution, though, is sensitive to the value of the learning rate, used and to the order of presentation of the training pairs. This sensitivity is responsible for the varying quality of the perceptron generated separating surface observed in simulations.

The bound on the number of corrections, k_0, given by Eqn. (14) depends on the choice of the initial weight vector w_1. If $w_1 = 0$, we get

$$k_0 = \frac{\alpha^2 \| w^* \|^2}{\beta^2} = \frac{\beta^2 \| w^* \|^2}{\gamma^2}$$

or

$$k_0 = \frac{\alpha^2 \| W^* \|^2}{\beta^2} = \frac{\beta^2 \| w^* \|^2}{\gamma^2} \tag{15}$$

Here, k_0 is a function of the initially unknown solution weight vector w^*. Therefore, Eqn. (15) is of no help for predicting the maximum number of corrections. However, the denominator of Eqn. (15) implies that the difficulty of the problem is essentially determined by the samples most nearly orthogonal to the solution vector.

ii) Generalizations of the Perceptron Learning Rule

The perceptron learning rule may be generalized to include a variable increment k and a fixed, positive margin b. This generalized learning rule updates the weight vector whenever $(z^k)^T W^k$ fails to exceed the margin b. Here, the algorithm for weight vector update is given by

$$\begin{cases} w^1 \quad \text{arbitrary} \\ w^{k+1} = w^k + \rho^k z^k & \text{if } (z^k)^T w^k \le b \\ w^{k+1} = w^k & \text{otherwise} \end{cases} \tag{16}$$

The margin b is useful because it gives a dead-zone robustness to the decision boundary. That is, the perceptron's decision hyperplane is constrained to lie in a region between the two classes such that sufficient clearance is realized between this hyperplane and the extreme points (boundary patterns) of the training set. This makes the perceptron robust with respect to noisy inputs. It can be shown (Duda and Hart, 1973) that if the training set is linearly separable and if the following three conditions are satisfied:

1. $\rho^k \ge 0$ (17)

2. $\lim_{m\to\infty} \sum_{k=1}^{m} \rho^k = \infty$ (18)

3. $\lim_{m\to\infty} \dfrac{\sum_{k=1}^{m} (\rho^k)^2}{\left(\sum_{k=1}^{m} \rho^k \right)^2} = 0$ (19)

(e.g., $k = \dfrac{\rho}{k}$ or even $k = k$) then w_k converges to a solution w^* which satisfies $(z^i)^T W^* > b$ for $i = 1, 2, \ldots, m$. Furthermore, when k is fixed at a positive constant, this learning rule converges in finite time.

Another variant of the perceptron learning rule is given by the "batch" update procedure

$$\begin{cases} w^1 \quad \text{arbitrary} \\ w^{k+1} = w^k + \rho \sum_{z \in Z(w^k)} z \end{cases} \tag{20}$$

where $z(w^k)$ is the set of patterns z misclassified by w^k. Here, the weight vector change $\Delta w = w^{k+1} - w^k$ is along the direction of the resultant vector of all misclassified patterns. In general, this update procedure converges faster than the perceptron rule, but it requires more storage.

In the nonlinearly separable case, the above algorithms do not converge. Few theoretical results are available on the behavior of these algorithms for nonlinearly separable problems [see Minsky and Papert (1969) and Block and Levin (1970) for some preliminary results]. For example, it is known that the length of w in the perceptron rule is bounded, i.e., tends to fluctuate near some limiting value $\|w^*\|$ (Efron, 1964). This information may be used to terminate the search for w^*. Another approach is to average the weight vectors near the fluctuation point w^*. Butz (1967) proposed the use of a reinforcement factor, $0 1$, in the perceptron learning rule. This reinforcement places w in a region that tends to minimize the probability of error for nonlinearly separable cases. Butz's rule is as follows

$$\begin{cases} w^1 \quad \text{arbitrary} \\ w^{k+1} = w^k + \rho z^k \quad \text{if } (z^k)^T w \le 0 \\ w^{k+1} = w^k + \rho\gamma z^k \quad \text{if } (z^k)^T w > 0 \end{cases} \tag{21}$$

iii) The Perceptron Criterion Function

It is interesting to see how the above error correction rules can be derived by a gradient descent on an appropriate criterion (objective) function. For the perceptron, we may define the following criterion function (Duda and Hart, 1973):

$$J(w) = -\sum_{z \in Z(w)} z^T w \tag{22}$$

where $Z(w)$ is the set of samples misclassified by w (i.e., $z^T w \ 0$). Note that if $z(w)$ is empty then $J(w) = 0$, otherwise $J(w) > 0$. Geometrically, $J(w)$ is proportional to the sum of the distances from the misclassified samples to the decision boundary. The smaller J is, the better the weight vector w will be.

Given this objective function $J(w)$, we can incrementally improve the search point w^k at each iteration by sliding downhill on the surface defined by $J(w)$ in w space. Specifically, we may use J to perform a discrete gradient descent search, which updates w^k so that a step is taken downhill in the "steepest" direction along the search surface $J(w)$ at w^k. This can be achieved by making w^k proportional to the gradient of J at the present location w^k, formally we may write

$$w^{k+1} = w^k - \rho \nabla J(w)|_{w=w^k} = w^k - \rho \left[\frac{\partial J}{\partial w_1} \frac{\partial J}{\partial w_2} \cdots \frac{\partial J}{\partial w_{n+1}} \right]^T \Bigg|_{w=w^k} \tag{23}$$

Here, the initial search point, w^1, and the learning rate (step size) are to be specified by the user. We will refer to Eqn. (23) as the steepest gradient descent search rule or, simply, gradient descent. Next, substituting the gradient

$$\nabla J(w^k) = -\sum_{z \in Z(w^k)} z \tag{24}$$

in Eqn. (20), leads to the weight update rule

$$w^{k+1} = w^k + \rho \sum_{z \in Z(w^k)} z \tag{25}$$

The learning rule given in Eqn. (25) is identical to the multiple-sample (batch) perceptron rule of Eqn. (20). The original perceptron learning rule of Eqn. (3) can be thought of as an "incremental" gradient descent search rule for minimizing the perceptron criterion function in Eqn. (22). Following a similar procedure as in Eqns. (23) through (24), it can be shown that

$$J(w) = -\sum_{z^T w \le b} (z^T w - b) \tag{26}$$

is the appropriate criterion function for the modified perceptron rule.

Here, it may be noted that the gradient of J in Eqn. (24) is not mathematically precise. Due to the piecewise linear nature of J, sudden changes in the gradient of J occur every time the perceptron output y goes through a transition at $(z^k)^T w = 0$. Therefore, the gradient of J is not defined at "transition" points w satisfying $(z^k)^T w = 0, k = 1, 2, \ldots, m$. However, because of the discrete nature of Eqn. (23), the

likelihood of w^k overlapping with one of these transition points is negligible, and thus we may still express J as in Eqn. (24).

iv) Mays Learning Rule

The criterion functions in Eqns. (22) and (26) are by no means the only functions we can construct that are minimized when w is a solution vector. For example, an alternative function is the quadratic function

$$J(w) = \frac{1}{2} \sum_{z^T w \le b} (z^T w - b)^2 \tag{27}$$

where b is a positive constant margin. Like the previous criterion functions, the function $J(w)$ in Eqn. (27) focuses attention on the misclassified samples. Its major difference is that its gradient is continuous, whereas the gradient of the perceptron criterion function, with or without the use of margin, is not. Unfortunately, the present function can be dominated by the input vectors with the largest magnitudes. We may eliminate this undesirable effect by dividing by $\| z \|^2$:

$$J(w) = \frac{1}{2} \sum_{z^T w \le b} \frac{(z^T w - b)^2}{\| z \|^2} \tag{28}$$

The gradient of $J(w)$ in Eqn (28) is given by

$$\nabla J(w) = \sum_{z^T w \le b} \frac{z^T w - b}{\| z \|^2} z \tag{29}$$

which, upon substituting in Eqn. (23), leads to the following learning rule

$$\begin{cases} w^1 \quad \text{arbitrary} \\ w^{k+1} = w^k + \rho \sum_{z^T w \le b} \dfrac{b - z^T w^k}{\| z \|^2} z \end{cases} \tag{30}$$

If we consider the incremental update version of Eqn. (30), we arrive at Mays' rule (Mays, 1964):

$$\begin{cases} w^1 \quad \text{arbitrary} \\ w^{k+1} = w^k + \rho \dfrac{b - (z^k)^T w^k}{\| z^k \|^2} z^k & \text{if } (z^k)^T w^k \le b \\ w^{k+1} = w^k & \text{otherwise} \end{cases} \tag{31}$$

If the training set is linearly separable, Mays' rule converges in a finite number of iterations, for $0 < \pi < 2$ (Duda and Hart, 1973). In the case of a nonlinearly separable training set, the training procedure in Eqn. (31) will never converge. To fix this problem a decreasing learning rate such as $k = \dfrac{\rho}{k}$ may be used to force convergence to some approximate separating surface (Duda and Singleton, 1964).

v) Widrow-Hoff (-LMS) Learning Rule

Another example of an error correcting rule with a quadratic criterion function is the Widrow-Hoff rule (Widrow and Hoff, 1960). This rule was originally used to train the linear unit, also known as the adaptive linear combiner element (ADALINE), shown in Fig. 23. In this case, the output of the linear unit in response to the input x_k is simply $y_k = (x_k)Tw$. The Widrow-Hoff rule was originally proposed as an ad hoc rule, which embodies the so-called minimal disturbance principle. Later, it was

discovered (Widrow and Stearns, 1985) that this rule converges in the mean square to the solution w^* which corresponds to the least-mean-square (LMS) output error, if all input patterns are of the same length (i.e., $\| x_k \|$ is the same for all k). Therefore, this rule is sometimes referred to as the -LMS rule.

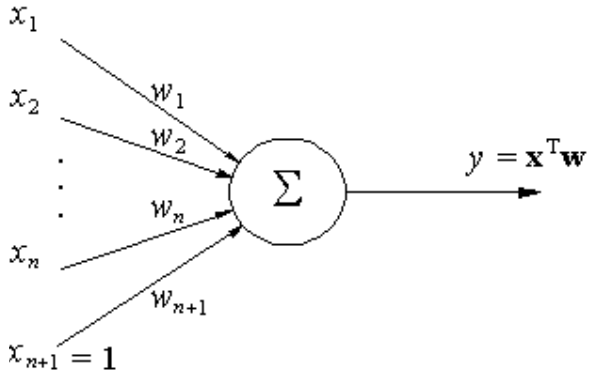


Fig. 23: Adaptive linear combiner element (ADALINE).

The -LMS rule is given by

$$\begin{cases} w^1 = 0 \quad \text{or arbitrary} \\ w^{k+1} = w^k + \alpha(d^k - y^k)\dfrac{x^k}{\| x^k \|^2} \end{cases} \tag{32}$$

where d^k is the desired response, and > 0. Eqn. (32) is similar to the perceptron rule if we set, in Eqn. (2), as

$$\rho = \rho^k \frac{\alpha}{\| x^k \|^2} \tag{33}$$

Though, the error in Eqn. (32) is measured at the linear output; not after the nonlinearity as in the perceptron. The constant ρ controls the stability and speed of convergence (Widrow and Stearns, 1985; Widrow and Lehr, 1990). If the input vectors are independent over time, stability is insured for most practical purposes.

As for May's rule, this rule is self-normalizing in the sense that the choice does not depend on the magnitude of the input vectors. Since the -LMS rule selects w^k to be collinear with x^k, the desired error correction is achieved with a weight change of the smallest possible magnitude. Thus, when adapting to learn a new training sample, the responses to previous training samples are minimally disturbed, on the average. This is the basic idea behind the minimal disturbance principle on which the -LMS is founded. Alternatively, one can show that the -LMS learning rule is a gradient descent minimizer of an appropriate quadratic criterion function.

Try the following exercises.

E7) Define error-correction learning.

E8) Differentiate supervised and unsupervised learning by citing examples of each.

Now, let us summarize the unit.

4.6 SUMMARY

In this unit, we have covered the following points.

1. Mathematical analysis has solved some of the mysteries posed by the new models but has left many questions open for future investigations. Needless to say, the study of neurons, their interconnections, and their role as the brain's elementary building blocks is one of the most dynamic and important research fields in modern biology.
2. The simplest kind of computing units are used to build artificial neural networks. These computing elements are a generalization of the common logic gates used in conventional computing and, since they operate by comparing their total input with a threshold, this field of research is known as threshold logic.
3. A simple static synchronous associative memory is presented along with appropriate memory storage recipes. Then, this simple associative memory is extended into a recurrent auto associative memory by employing feedback.
4. Error correction rules were initially proposed as ad hoc rules for single unit training. These rules essentially drive the output error of a given unit to zero. We start with the classical perceptron learning rule and give a proof for its convergence.

4.7 SOLUTIONS/ANSWERS

E1) The elements of a neuron are dendrites, synapses, cell body and axis.

E2) The five models of computations are given below

i) The mathematical model

ii) The logic operational model

iii) The computer model

iv) Cellular automate

v) The biological model.

E3) The activation function (also known as the transfer function) performs the task of axon and synpse. The output of a neuron longly depends on its activation function. Different types of activation function are in use, such as hard-limit, linear log sigmoid, tan sigmoid and others.

E4) $\phi = \alpha_4 f_2 + \alpha_5 f_3$

$f_2 = \alpha_1 f_1$

$f_3 = f_1\alpha_2 + f_2\alpha_3$

E5) A biological neuron model is a mathematical description of the properties of nerve cells, or neurons, that is designed to accurately describe and predict biological processes. This is in contrast to the artificial neuron, which arise for computational effectiveness, although these goals sometimes overlap.

E6) $x_1 \wedge x_2$

$x_1 \vee x_2$

$\neg x_1$ or $\neg x_2$

E8) Supervised networks are a little more straightforward to conceptualize than unsupervised networks. We can apply the inputs to the supervised network along with an expected response, much like the Pavlovian conditioned stimulus and response regimen. We can mold the network with stimulus-response pairs. A stock market forecaster may present economic data (the stimulus) along with metrics of stock market performance (the response) to the neural network to the presenter and attempt to predict the future once training is complete.

We provide unsupervised networks with only stimulus. For example, we want an unsupervised network to correctly classify parts from a conveyor belt into part numbers, providing an image of each part to do the classification (the stimulus). The unsupervised network in this case would act like a look-up memory that is indexed by its contents, or a Content-Addressable-Memory (CAM).

4.8 REFERENCES

1. Simon S Haykin and Simon Haykin, (1998), *Neural Networks: A Comprehensive Foundation*, Pearson Education.
2. Raul Rojas, (1996), *Neural Networks: A Systematic Introduction.*
3. S. Rajasekaran and G.A. Vijayalakshmi Pai, (2010), *Neural Netwoks: Fuzzy Logic, and Genetic Algorithms.*
4. D.K. Pratihar, (2008), *Soft Computing.*
5. Valluru B. Rao and Hayagriva V. Rao, *C + + Neural Networks & Fuzzy Logic*.

UNIT 5 SINGLE LAYER PERCEPTRONS

Structure **Page No.**

5.1 INTRODUCTION

We have discussed model of McCulloch-Pitts units. Now the question of how to find the parameters adequate for a given task was left open. If two sets of points have to be separated linearly with a perceptron, adequate weights for the computing unit must be found. The perceptron learning algorithm deals with this problem. These are discussed in Section 5.2.

We arrived at the conclusion that McCulloch-Pitts units can be used to build networks capable of computing any logical function and of simulating any finite automaton. From the biological point of view, however, the types of network that can be built are not very relevant. The computing units are too similar to conventional logic gates and the network must be completely specified before it can be used. In this connection, we shall discuss XOR problem in Section 5.3.

Objectives

After studying the unit you should be able to:

- define single layer perceptron;
- write the perecptron algorithm;
- apply the perceptron algorithm in different situation;
- describe the XOR problem.

5.2 SINGLE LAYER PERCEPTRON ALGORITHM AND ITS CONVERGENCE

Let us first start by defining the perceptron learning algorithms for Neural Networks. A learning algorithm is an adaptive method by which a network of computing units self-organizes to implement the desired behavior. This is done in some learning algorithms by presenting some examples of the desired input-output mapping to the network. A correction step is executed iteratively until the network learns to produce the desired response. The learning algorithm is a closed loop of presentation of examples and of corrections to the network parameters, as shown in Fig. 1. In some simple cases the weights for the computing units can be found through a sequential test of stochastically generated numerical combinations. However, such algorithms which look blindly for a solution do not qualify as “learning”. A learning algorithm must adapt the network parameters according to previous experience until a solution is found, if it exists.

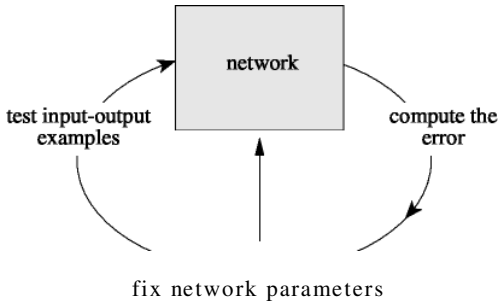


Fig. 1: Learning Process in a parametric system

Learning algorithms can be divided into supervised and unsupervised methods. Supervised learning denotes a method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured. The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm. This kind of learning is also called learning with a teacher, since a control process knows the correct answer for the set of selected input vectors.

Unsupervised learning is used when, for a given input, the exact numerical output a network should produce is unknown. Assume, for example, that some points in two-dimensional space are to be classified into three clusters. For this task we can use a classifier network with three output lines, one for each class (Fig. 2). Each of the three computing units at the output must specialize by firing only for inputs corresponding to elements of each cluster. If one unit fires, the others must keep silent. In this case we do not know a priori which unit is going to specialize on which cluster. Generally we do not even know how many well-defined clusters are present. Since no "teacher" is available, the network must organize itself in order to be able to associate clusters with units.

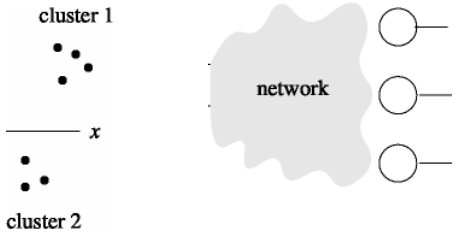


Fig. 2: Three clusters in a classifier network

Supervised learning is further divided into methods which use reinforcement or error correction. Reinforcement learning is used when after each presentation of an input-output example we only know whether the network produces the desired result or not. The weights are updated based on this information (that is, the Boolean values true or false) so that only the input vector can be used for weight correction. In learning with error correction, the magnitude of the error, together with the input vector, determines the magnitude of the corrections to the weights, and in many cases we try to eliminate the error in a single correction step.

An example of supervised learning is the perceptron learning algorithm with reinforcement. Some of its variants use supervised learning with error correction (corrective learning).

Now, we shall deal with learning methods for perceptrons. To simplify the notation we adopt the following conventions. For this, consider the input $(x_1, x_2, \ldots, x_n)$ to the

perceptron which is called the input vector. If the weights of the perceptron are the real numbers $w_1, w_2, \ldots, w_n$ and the threshold is θ, we call $w = (w_1, w_2, \ldots, w_{n+1})$ with $w_{n+1} = -\theta$ **the extended weight vector** of the perceptron and $(x_1, x_2, \ldots, x_n, 1)$, **the extended input vector**. The threshold computation of a perceptron will be expressed using scalar products. The arithmetic test computed by the perceptron is thus

$$w.x \geq \theta,$$

if w and x are the weight and input vectors, and

$$w.x \geq \theta$$

if w and x are the extended weight and input vectors. It will always be clear from the context whether normal or extended vectors are being used.

Example 1: We are looking for the weights and threshold needed to implement the AND function with a perceptron, the input vectors and their associated outputs are given in Table 1.

Table 1

Input	Output
0, 0	0
0, 1	0
1, 0	0
1, 1	1

If a perceptron with threshold zero is used, the input vectors must be extended and the desired mappings are given in Table 2.

Table 2

Input	Output
0, 0, 1	0
0, 1, 1	0
1, 0, 1	0
1, 1, 1	1

A perceptron with three still unknown weights (w_1, w_2, w_3) can carry out this task.

The proof of convergence of the perceptron learning algorithm assumes that each perceptron performs the test $w \cdot x > 0$. So far we have been working with perceptrons which perform the test $w.x \geq 0$. We must just show that both classes of computing units are equivalent when the training set is finite, as is always the case in learning problems. Not let us define the following definitions.

Definition 1: Two sets A and B of points in an n-dimensional space are called absolutely linearly separable if $n+1$ real numbers $w_1, \ldots. w_{n+1}$ exist such that every point $(x_1, x_2, \ldots, x_n) \in A$ satisfies $\sum_{i=1}^{n} w_1 x_1 > w_{n+1}$ and every point $(x_1, x_2, \ldots, x_n) \in B$ satisfies $\sum_{i=1}^{n} w_1 x_1 < w_{n+1}$.

If a perceptron with threshold zero can linearly separate two finite sets of input vectors, then only a small adjustment to its weights is needed to obtain an absolute linear separation.

A usual approach for starting the learning algorithm is to initialize the network weights randomly and to improve these initial parameters, looking at each step to see whether a better separation of the training set can be achieved. In this section we identify points $(x_1, x_2, \ldots, x_n)$ in n-dimensional space with the vector x with the same coordinates.

Definition 2: The open (closed) positive half-space associated with the n-dimensional weight vector w is the set of all points $x \in R^n$ for which $w.x > 0$ $(w.x \geq 0)$. The open (closed) negative half-space associated with w is the set of all points $x \in R^n$ for which $w.x < 0$ $(w.x \geq 0)$.

We omit the adjectives "closed" or "open" whenever it is clear from the context which kind of linear separation is being used.

Example 2: Consider three weights w_1, w_2, and $w_3 = -\theta$ are needed to implement the desired separation with a generic perceptron. The first step is to extend the input vectors with a third coordinate $x_3 = 1$ and to write down the four inequalities that must be fulfilled:

$$(0, 0, 1).(w_1, w_2, w_3) < 0 \quad (1)$$

$$(1, 0,1).(w_1, w_2, w_3) < 0 \quad (2)$$

$$(0, 1, 1).(w_1, w_2, w_3) < 0 \quad (3)$$

$$(1, 1, 1).(w_1, w_2, w_3) > 0 \quad (4)$$

This can be written as

$$Aw > 0$$

where A is the 4×3 matrix and w the weight vector (written as a column vector) given as $A \begin{bmatrix} 0 & 0 & -1 \\ -1 & 0 & -1 \\ 0 & -1 & -1 \\ 1 & 1 & 1 \end{bmatrix}$ and $w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$. The learning problem is to find the appropriate weight vector w.

Now, let us introduce the perceptron learning algorithm. For this, consider two training sets, P and N, in n-dimensional extended input space, and a weight vector w. These are selected such that all vectors in P belong to the open positive half-space and all vectors in N to the open negative half-space of the linear separation.

Algorithm Perceptron

Step 1: Initialize the weight vector w_0 randomly and set $x = 0$.

Step 2: Select a vector $x \in P \cup N$ randomly.

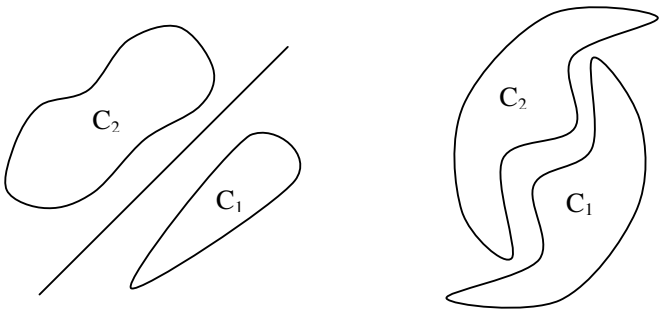
Step 3: If $x \in P$ and $w_t.x > 0$, or if $x \in N$ and $w_t.x < 0$, then go to step 2. Otherwise if $x \in N$ and $w_t.x \geq 0$, then set $w_{t+1} = w_t + \alpha x$ and $t = t + 1$ and go to step 2 otherwise if $x \in P$ and $w_t.x \leq 0$, then set $w_{t+1} = w_t - \alpha x$ and $t = t + 1$, and go to step 2.

Here α is the learning rate parameter. w(uc) ith the help of this algorithm, the weight vectors can be corrected in case of incorrect classification of the

selected vectors P and N. Here it is clear that if P and N are separable, the vector w is updated by the finite number of iterations.

The iterations are stopped by getting the all vectors correctly. It is clear that the weight w_{t+1} is the new weight, which has been adjusted by the old weight w_t with the input vector x. If the value of x is fixed, then the learning algorithm is termed as fixed increment algorithm. This perceptron is one of the important achievement due to Rosemblatt. Further Minsky and Papert contributed the linearly separable solution space and an illustration as XOR problem.

Perceptrons can handle only linearly separable tasks. In two dimensional space linearly separability them by a straight line.



(a) Linearly separable patterns **(b) Non-linearly separable patterns**

Fig. 3

Although the perceptron learning algorithm converges to a solution, the number of iterations can be very large if the input vectors are not normalized and are arranged in an unfavorable way.

There are faster methods to find the weight vector appropriate for a given problem. When the perceptron learning algorithm makes a correction, an input vector x is added or subtracted from the weight vector w. The search direction is given by the vector x. Each input vector corresponds to the border of one region of the error function defined on weight space. The direction of x is orthogonal to the step defined by x on the error surface. The weight vector is displaced in the direction of x until it "falls" into a region with smaller error.

Example 3: (The dynamics of perceptron learning using the error surface for the OR function). The input $(1, 1)$ must produce the output 1 (for simplicity we fix the threshold of the perceptron to 1). The two weights w_1 and w_2 must fulfill the inequality $w_1 + w_2 \geq 1$. Any other combination produces an error. The contribution to the total error is shown in Fig. 4 as a step in the error surface. If the initial weight vector lies in the triangular region with error 1, it must be brought up to the verge of the region with error 0. This can be done by adding the vector $(1, 1)$ to w. However, if the input vector is, for example, $(0.1, 0.1)$, it should be added a few times before the weight combination (w_1, w_2) falls to the region of error 0. In this case we would like to make the correction in a single iteration.

These considerations provide an improvement for the perceptron learning algorithm: if at iteration t the input vector $x \in P$ is classified erroneously, then we have $w_t \not\geq 0$.

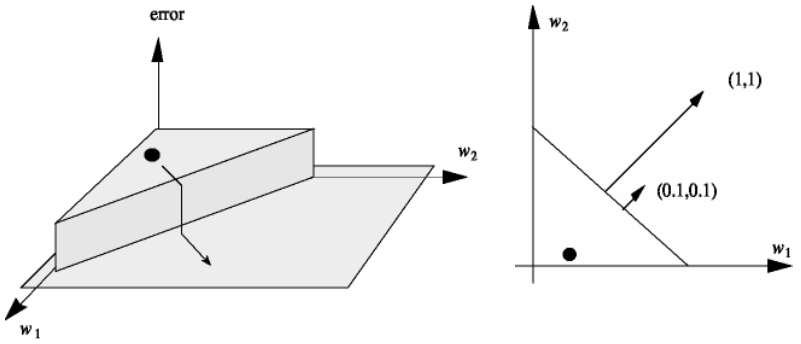


Fig. 4: A step on the error surface

The number ε guarantees that the new weight vector just barely skips over the border of the region with a higher error. The constant ε should be made small enough to avoid skipping to another region whose error is higher than the current one. When $x \in N$ the correction step is made similarly, but using the factor $\delta - \varepsilon$ instead of $\delta + \varepsilon$.

The accelerated algorithm is an example of corrective learning: We do not just "reinforce" the weight vector, but completely correct the error that has been made. A variant of this rule is correction of the weight vector using a proportionality constant γ as the learning factor, in such a way that at each update the vector $\gamma(\delta + \varepsilon)x$ is added to w. The learning constant falls to zero as learning progresses.

The perceptron learning algorithm selects a search direction in weight space according to the incorrect classification of the last tested vector and does not make use of global information about the shape of the error function. It is a greedy, local algorithm. This can lead to an exponential number of updates of the weight vector. Fig. 5 shows the different error regions in a worst case scenario. The region with error 0 is bounded by two lines which meet at a small angle. Starting the learning algorithm at point $w0$, the weight updates will lead to a search path similar to the one shown in the figure. In each new iteration a new weight vector is computed, in such a way that one of two vectors is classified correctly. However, each of these corrections leads to the other vector being incorrectly classified. The iteration jumps from one region with error 1 to the other one. The algorithm converges only after a certain number of iterations, which can be made arbitrarily large by adjusting the angle at which the lines meet.

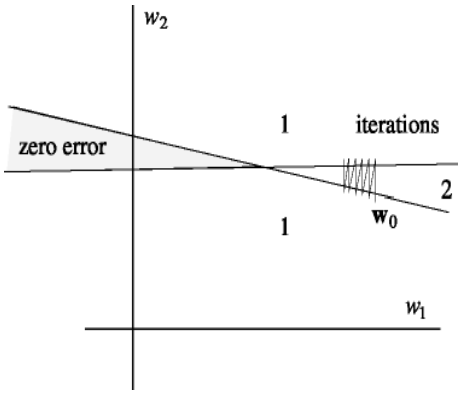


Fig. 5: Worst case for perceptron learning (weight space)

Fig. 5 corresponds to the case in which two almost antiparallel vectors are to be classified in the same half-space (Fig. 6). An algorithm which rotates the separation line in one of the two directions (like perceptron learning) will require more and more time when the angle between the two vectors approaches 180 degrees. This example is

a good illustration of the advantages of visualizing learning algorithms in both the input space and its dual, weight space. Fig. 6 shows the concrete problem and Fig. 5 illustrates why?

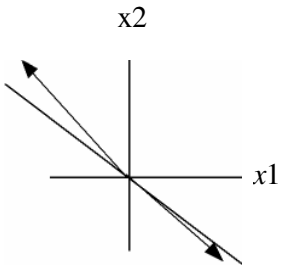


Fig. 6: Worst case for perceptron learning (input space)

Now let us discuss the single layer perceptron in the following.

A neural network with a single layer is also capable of processing for some important applications, such as integrated circuit implementations for assembly line control. The most common capability of the different models of neural networks is pattern recognition. But one network, called the Brain-State-in-a-Box, which is a single-layer neural network, can do pattern completion. Adaline is a network with A and B fields of neurons, but aggregation or processing of input signals is done only by the field B neurons.

The Hopfield network is a single-layer neural network. The Hopfield network makes an association between different pattern (heteroassociation) or associates a pattern with itself (autoassociation). We may characterize this as being able to recognize a given pattern. The idea of viewing it as a case of pattern recognition becomes more relevant if a pattern is presented with some noise, meaning that there is some slight deformation in the pattern, and if the network is able to relate it to the correct pattern.

The Perceptron technically has two layers, but has only one group of weights. We therefore still refer to it as a single-layer network. The second layer consists solely of the output neuron, and the first layer consists of the neurons that receive input(s). Also, the neurons in the same layer, the input layer in this case, are not interconnected, that is, no connections are made between two neurons in that same layer. On the other hand, in the Hopfield network, there is no separate output layer, and hence, it is strictly a single-layer network. In addition, the neurons are all fully connected with one another.

The ADALINE network is abbreviated for **Adaptive Linear Neural Element Network**. This network was framed by Bernard Widrow of Stanford University, makes use of supervised learning. A simple ADALINE network is shown in Fig. 7. Here, there is only one output neuron and the output values are bipolar $(-1 \text{ or } +1)$, and the inputs x_i may be binary, bipolar or real valued. The bias weight is w_0 with an input link of $x_0 = +1$. If $\sum_{i=0}^{n} x_i w_i \geq 0$, then the output is 1 otherwise it is -1.

The supervised learning algorithm adopted by the network is similar to the perceptron learning algorithm, devised by Widrow-Hoff (1960). The learning algorithm is also known as the Least Mean Square (LMS) or Delta rule. The rule is given by

$$w_{n+1} = w_n + \alpha(t - y)x_i$$

where α is the learning coefficient, t is the target, y is the computed output, and x_i is the input.

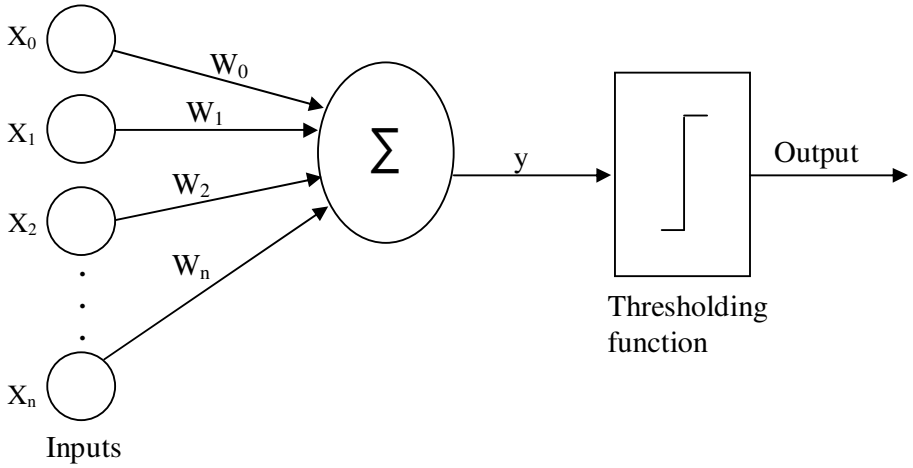


Fig. 7: A simple ADALINE network

ADALINE network is used virtually in all high speed modems and telephone switching systems to cancel the echo in long distance communication circuits.

Try the following exercises.

E1) The input to a single-input neuron is 2.0, its weight is 2.3 and its bias is – 3.

i) What is the net input to the transfer function?

ii) What is the neuron output for the following transfer function
   a) Hard limit,
   b) Linear,
   c) Log-sigmoid.

E2) Given a two-input neuron with the following parameters: b = 1.2, W = [3 2] and $x = \begin{bmatrix} -5 & 6 \end{bmatrix}^T$, calculate the neuron output for the following transfer functions:

i) A symmetrical hard limit transfer function.

ii) A linear transfer function.

iii) A hyperbolic tangent sigmoid (tansig) transfer function.

E3) A single-layer neural network is to have six inputs and two outputs. The outputs are to be limited to and continuous over the range 0 to 1. What can you tell about the network architecture? Specifically:

i) How many neurons are required?

ii) What are the dimensions of the weight matrix?

iii) What kind of transfer functions could be used?

iv) Is a bias required?

In the following section, we shall discuss the XOR problem.

5.3 XOR PROBLEM

The perceptron cannot find weights for classification type of problems that are not linearly separable. An example is the XOR (exclusive OR) problem. The ability of a Perceptron in evaluating functions was brought into question when Minsky and Papert

proved that a simple function like XOR (the logical function exclusive or) could not be correctly evaluated by a Perceptron. XOR is a logical operation which can be described by its truth table. The truth table is presented in Table 3.

Table 3

Inputs (x_1)	**Inputs (x_2)**	**Output $f(x_1, x_2) = XOR(x_1, x_2)$**
0	0	0
0	1	1
1	0	1
1	1	0

Here, it is clear that the behaviour of the XOR, if both inputs are the same value, the output is 0, otherwise the output is 1.

The problem for the artificial neural network is to classify the inputs as odd parity or even parity. Here odd parity is the odd number of 1 bits in the inputs and even parity is the even number of 1 bit in the inputs. This is not possible, since as is evident by the perceptron, we are unable to find a little separating even parity input patterns from the odd parity input patterns.

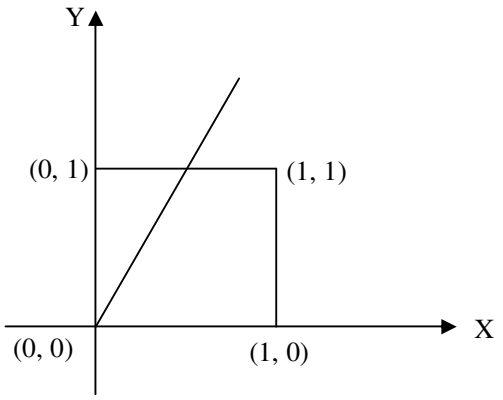


Fig. 8: Patterns of the XOR problem (non-linear separable)

Now, the question arises, that why is these perceptron unable to find weights for non-linearly separable classification problems? This can be explained by means of a simple instance. For this, consider a perceptron network with two inputs x_1 and x_2 and bias $x_0 = 1$ (refer Fig. 9).

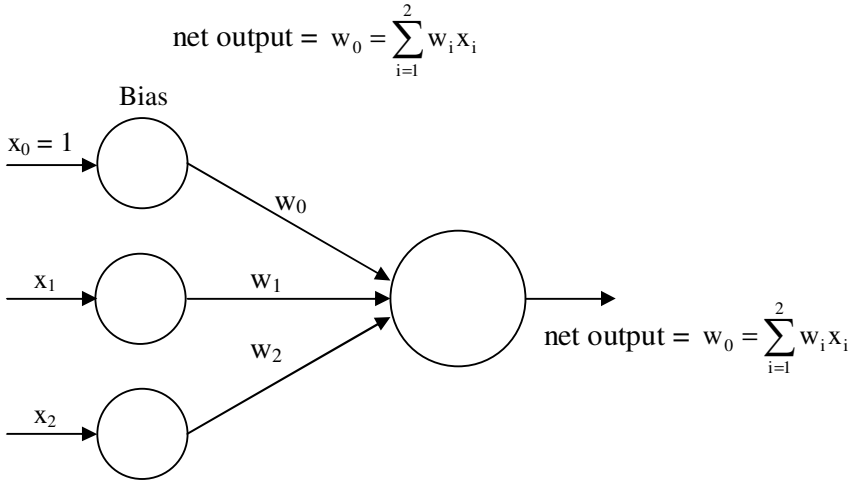


Fig. 9: A perceptron model

which represents the equation of a straight line, which acts as a decision boundary separating the points into classes C_1 and C_2, above and below the line respectively. This is shown in Fig. 10.

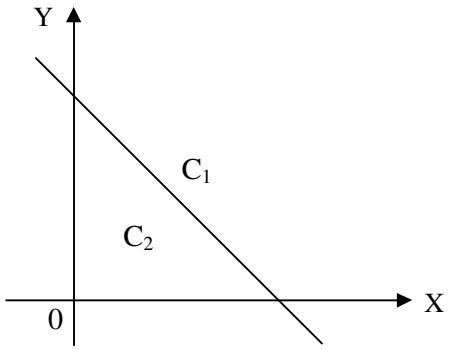


Fig. 10: A 2-classsification problem

This gives the answer that is the perceptron aims to do for a problem when it is able to obtain their weights.

Minsky and Papert showed that it is impossible to come up with the proper set of weights for the neurons in the single layer of a simple Perceptron to evaluate the XOR function. The reason for this is that such a Perceptron, one with a single layer of neurons, requires the function to be evaluated, to be linearly separable by means of the function values.

Since there are two arguments for the XOR function, there would be two neurons in the input layer, and since the function's value is one number, there would be one output neuron. Therefore, we need two weights w_1 and w_2, and a threshold value θ. Let us now look at the conditions to be satisfied by the w's and the θ so that the outputs corresponding to given inputs would be as for the XOR function.

First the output should be 0 if inputs are 0 and 0. The activation works out as 0. To get an output of 0, you need $0 < \theta$. This is your first condition. Table 4 shows this and two other conditions we need, and why.

Table 4: Conditions on Weights

Input	Activation	Output	Needed Condition
0, 0	0	0	$0 < \theta$
1, 0	w_1	1	$w_1 > \theta$
0, 1	w_2	1	$w_2 > \theta$
1, 1	$w_1 + w_2$	0	$w_1 + w_2 < \theta$

From the first three conditions, we can deduce that the sum of the two weights has to be greater than θ, which has to be positive itself. Line 4 is inconsistent with lines 1, 2, and 3, since line 4 requires the sum of the two weights to be less than θ. This affirms the contention that it is not possible to compute the XOR function with a simple perceptron.

Geometrically, the reason for this failure is that the inputs (0, 1) and (1, 0) with which we want output 1, are situated diagonally opposite each other, when plotted as points in the plane.

We can't separate the T's and the F's with a straight line. This means that we cannot draw a line in the plane in such a way that neither (1, 1) nor (0, 0) is on the same side

of the line as (0, 1) and (1, 0).

Implementation of Logical Functions

In each of the previous sections a threshold element was associated with a whole set of predicates or a network of computing elements. From now on, we will deal with perceptrons as isolated threshold elements which compute their output without delay.

Definition 3: A simple perceptron is a computing unit with threshold θ which, when receiving the n real inputs $x_1, x_2, \ldots, x_n$ through edges with the associated weights $w_1, w_2, \ldots, w_n$, outputs 1 if the inequality $\sum_{i=1}^{n} w_i x_i \geq \theta$ holds and otherwise θ.

The origin of the inputs is not important, whether they come from other perceptrons or another class of computing units. The geometric interpretation of the processing performed by perceptrons is the same as with McCulloch-Pitts elements. A perceptron separates the input space into two half-spaces. For points belonging to one half-space the result of the computation is 0, for points belonging to the other it is 1.

It is possible to generate arbitrary separations of input space by adjusting the parameters of this example. In many cases it is more convenient to deal with perceptrons of threshold zero only. This corresponds to linear separations which are forced to go through the origin of the input space. The two perceptrons shown in Fig. 11 are equivalent. The threshold of the perceptron to the left has been converted into the weight $-\theta$ of an additional input channel connected to the constant 1. This extra weight connected to a constant is called the bias of the element.

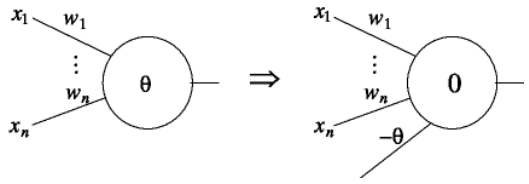


Fig. 11: A perceptron with a bias

Most learning algorithms can be stated more concisely by transforming thresholds into biases. The input vector $(x_1, x_2, \ldots, x_n)$ must be extended with an additional 1 and the resulting $(n+1)$-dimensional vector $(x_1, x_2, \ldots, x_n, 1)$ is called the extended input vector. The extended weight vector associated with this perceptron is $w_1, \ldots, w_n, w_{n+1})$, whereby $w_{n+1} = \theta$.

We can now deal with the problem of determining which logical functions can be implemented with a single perceptron. A perceptron network is capable of computing any logical function, since perceptrons are even more powerful than unweighted McCulloch-Pitts elements. If we reduce the network to a single element, which functions are still computable?

Example 4: Consider a function of two variables. Table 5 shows all 16 possible Boolean functions of two variables f_0 to f_{15}. Each column f_1 shows the value of the function for each combination of the two variables x_1 and x_2. The function f_0, for example, is the zero function whereas f_{14} is the OR-function.

Table 5

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
0	0	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0

x_1	x_2	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Perceptron-computable functions are those for which the points whose function value is 0 can be separated from the points whose function value is 1 using a line. Fig. 12 shows two possible separations to compute the OR and the AND functions.

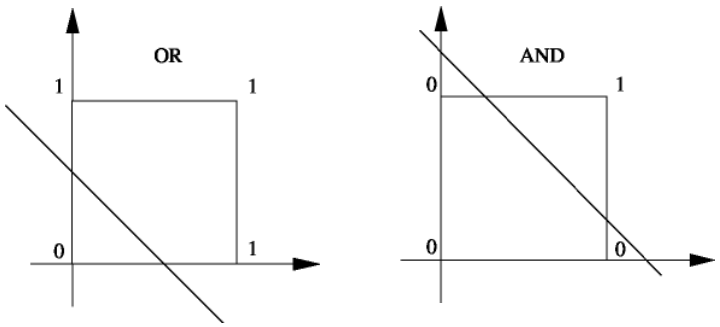


Fig. 12: Separations of input space corresponding to OR and AND

It is clear that two of the functions in the table cannot be computed in this way. They are the function XOR and identity (f_6 and f_9). It is intuitively evident that no line can produce the necessary separation of the input space. This can also be shown analytically.

Let w_1 and w_2 be the weights of a perceptron with two inputs, and θ its threshold. If the perceptron computes the XOR function the following four inequalities must be fulfilled:

$$x_1 = 0\ x_2 = 0\ w_1x_1 + w_2x_2 = 0 \Rightarrow 0 < \theta$$
$$x_1 = 1\ x_2 = 0\ w_1x_1 + w_2x_2 = w_1 \Rightarrow w_1 \geq \theta$$
$$x_1 = 0\ x_2 = 1\ w_1x_1 + w_2x_2 = w_2 \Rightarrow w_2 \geq 0$$
$$x_1 = 1,\ x_2 = 1,\ w_1 + w_2 \Rightarrow w_1 + w_2 < \theta$$

Since θ is positive, according to the first inequality, w_1 and w_2 are positive too, according to the second and third inequalities. Therefore the inequality $w_1 + w_2 < \theta$ cannot be true. This contradiction implies that no perceptron capable of computing the XOR function exists.

Try the following exercises.

E4) Consider the ADALINE filter with three neurons in the input layer having weights $w_{11} = 2$, $w_{12} = -1$ and $w_{13} = 3$ and the input sequences is $\{\cdots, 0, 0, 0, 5, -4, 0, 0, 0, \cdots\}$

i) What is the filter output just prior to 0 ?

ii) What is the filter output from 0 to 5 ?

E5) The following is a training set for a 2-classification problem. Iterate the perceptron through the training set and obtain the weights.

Inputs		Classification
X1	**X2**	**0/1**
0.25	0.353	0
0.25	0.471	1
0.5	0.353	0
0.5	0.647	1
0.75	0.705	0
0.75	0.882	1
1	0.705	0
1	1	1

E6) Attempt solving the XOR problem using the above implementation. Record the weights. What are your observations?

Now, let us summarize the unit.

5.4 SUMMARY

In this unit we have covered the following

1. Learning algorithm in the perceptron is performed for a finite number of iterations and their steps.
2. Single layer perceptrons are the cases where the computation layer consists of a single neuron.
3. The classical architecture of neural network is the Rosenblatt's perceptron. The major application of this is ADALINE.
4. The computing units are too similar to conventional logic gates and the network must be completely specified before it can be used.

5.5 SOLUTIONS/ANSWERS

E1) i) The net input

$$= wx + b = (2.3)\,(2) + (-3) = 1.6$$

ii) For the hard limit transfer function;

Output = hardlim (1.6) = 1.0

For the linear transfer function;

Output = purelin (1.6) = 1.6

For the log-sigmoid transfer function;

$$\text{Output} = \text{logsig}\,(1.6) = \frac{1}{1+e^{-1.6}} = 0.8320$$

E2) First calculate the net input n:

$$n = Wx + b = \begin{bmatrix}3 & 2\end{bmatrix}\begin{bmatrix}-5\\6\end{bmatrix} + (1.2) = -1.8.$$

Now find the outputs for each of the transfer functions.

i) a = hardlims (–1.8) = – 1

ii) a = lin (–1.8) = 0

iii) a = tansig (–1.8) = – 0.9468

E3) The problem specifications allow you to say the following about the network.

i) Two neurons, one for each output, are required.

ii) The weight matrix has two rows corresponding to the two neurons and six columns corresponding to the six inputs. (The product $\mathbf{W_p}$ is a two-element vector.)

iii) Of the transfer functions we have discussed, the logsig transfer function would be most appropriate.

iv) Not enough information is given to determine if a bias is required.

E4) i) Just prior to 0 three zeros have entered the filter, and the output is zero.

ii) At 0 the digit "5" has entered the filter, and it will be multiplied by w_{11}, which has the value 2, so that output is 10. This can be viewed as the matrix operation:

$$\text{Output} = Wx(0) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix} = 10.$$

Similarly, we can calculate the next outputs as

$$Wx(1) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} -4 \\ 5 \\ 0 \end{bmatrix} = -13$$

$$Wx(2) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ -4 \\ 5 \end{bmatrix} = 19$$

$$Wx(3) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -4 \end{bmatrix} = -12$$

and

$$Wx(4) = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 0.$$

All remaining outputs will be zero.

5.6 PRACTICAL EXERCISES

Session 6

1) Implement the perceptron learning algorithm in the computer. Find the weights for an edge detection operator using this program. The input-output examples can be taken from a digitized picture of an object and another one in which only the edges of the object have been kept.

2) Implement using C, the fixed increment perceptron learning algorithm. [**Hint:** for fixed increment perceptron learning algorithm α is fixed.]

5.7 REFERENCES

1. Simon S Haykin and Simon Haykin, (1998), *Neural Networks: A Comprehensive Foundation*, Pearson Education.
2. Raul Rojas, (1996), *Neural Networks: A Systematic Introduction.*
3. S. Rajasekaran and G.A. Vijayalakshmi Pai, (2010), *Neural Netwoks: Fuzzy Logic, and Genetic Algorithms*.
4. D.K. Pratihar, (2008), *Soft Computing*.
5. Valluru B. Rao and Hayagriva V. Rao, *C + + Neural Networks & Fuzzy Logic*.

UNIT 6 MULTI-LAYER PERCEPTRON

Structure **Page No**

6.1 INTRODUCTION

In this unit, we extend the Single Layer Neural Network architecture to the Multilayer Feedforward (MLFF) network with backpropagation (BP) learning. This network is also called multilayer perceptron. As its name suggests that this perceptron network has more than one layer. First, we briefly review the perceptron model discussed in unit 4 to show how this is altered to form MLFF networks in Sec. 6.2. In Section 6.3, we derive the generalized delta (backpropagation) learning rule and see how it is implemented in practice. Also, we shall examine the variations in the learning process to improve the efficiency, and ways to avoid some potential problems that can arise during training are described.

Objectives

After studying the unit you should be able to:

- define the multi-layer perceptron;
- formulate the multi-layer model for the given activation functions of input, hidden and output layers;
- implement the backpropagation algorithm.

6.2 MULTI LAYER PERCEPTRON

In Unit 5, Rosenblatts' perceptron i.e. single layer perceptron was introduced and the limitation of this with regard to the solution of linearly inseparable (or nonlinearly separable) problems was discussed.

The first approach to solve such linearly inseparable problems was to have more than one perceptron, each set up identifying small linearly separable sections of the inputs. Then, combining their outputs into another perceptron would produce a final indication of the class to which the input belongs.

Let us start with an example of XOR problem. The combination of perceptrons to solve the XOR problem is represented in Fig. 1. It looks that the arrangement shown in Fig. 1 can solve the problem. On investigation, it is formed that this arrangement of perceptrons in layers will be unable to learn. It is known that each neuron in the architecture takes the weighted sum of inputs, thresholds it and output 1/0. For any perceptron, in the first layer, the inputs come from the actual inputs of the problem, while for the perceptron in the second layer the inputs are nothing but the outputs of the first layer. The perceptrons of the second layer do not know which of the real inputs from the first layer were on or off.

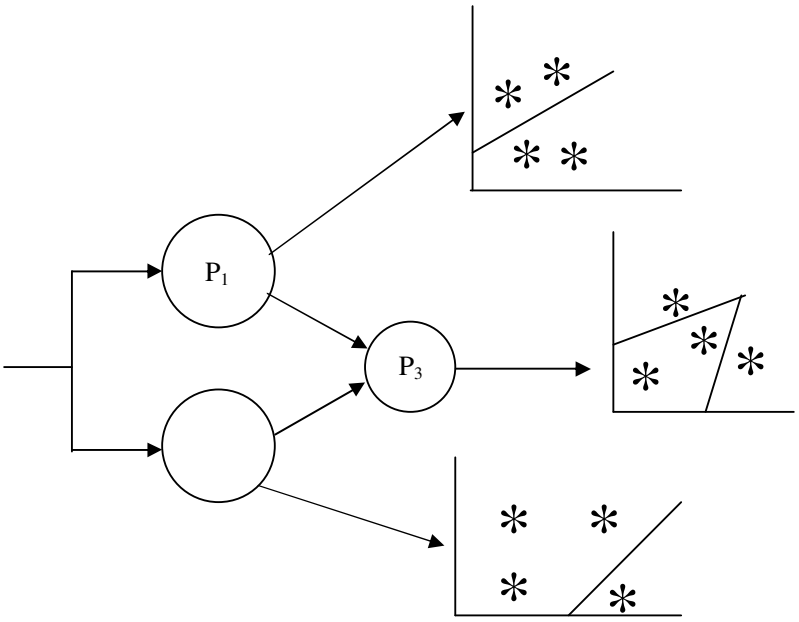


Fig. 1: Combination of perceptrons to solve XOR problem

It is impossible to strengthen the connections between active inputs and strengthen the correct parts of the network. The actual inputs are effectively masked off from the output units by the intermediate layer. The two states of neuron being on or off do not give us any indication of the scale by which we have to adjust the weights. These are shown in Fig. 2, where the threshold is adjusted at θ and at 0. The hard-hitting threshold functions remove the information that is needed if the network is to successfully learn. Hence, the network is not able to find which of the input weights are increased and which one are not and so. Therefore, the network is unable to work to produce a better solution next time. The way to go around the difficulty using the step function as the thresholding process is to adjust it slightly and to use a slightly different nonlinearity.

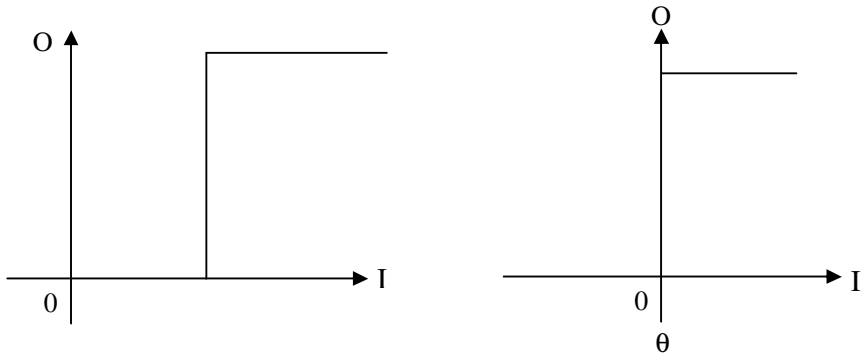


Fig. 2: "Step" or "Heaviside" functions

By smoothing the threshold function, we reach so that it turns on or off as before. It should have a sloping region in the middle that provides some information on the inputs. By this, strengthen or weaken the relevant weights can be determined. Hence, the network can learn as required. Here, $0 = 1$ if $I > \theta = 0$, $I < \theta$ where 0 is the output and I is the input. Also $1 < 0 < 1$ if $I = \theta$ and $u(t) = \sum_{i=1}^{n} w_i I_i$ notations of input and output the model of a neuron is shown in Fig. 3.

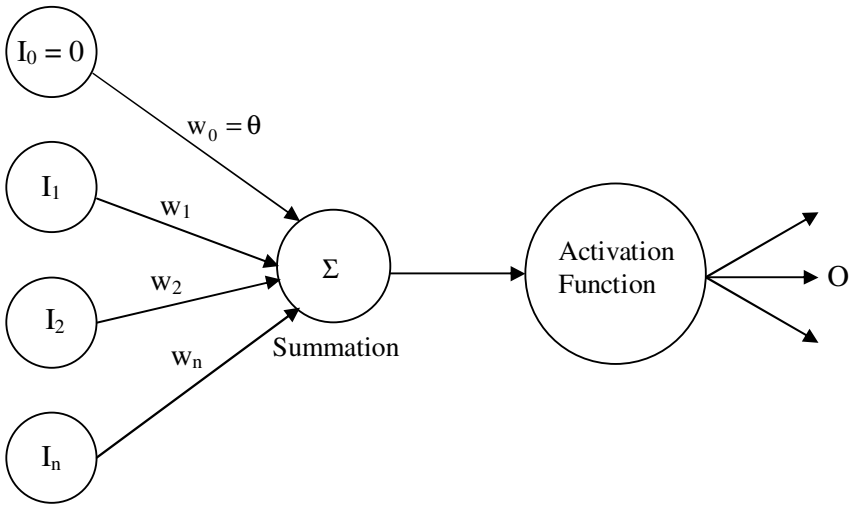


Fig. 3: An artificial neuron

Now let us define few nonlinear activation operators in addition to the earlier defined activation operators.

i) **Linear Function:** The output for linear function is o = gI where $g = \tan \phi$ and ϕ is the angle from the x-axis. The graph is shown in Fig. 4.

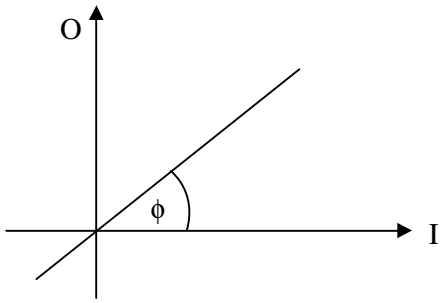


Fig. 4

ii) **Piecewise Linear Function:** The output for piecewise linear functions is defined as $O = \begin{Bmatrix} 1 & \text{if} & mI > 1 \\ gI & \text{if} & |mI < 1| \\ -1 & \text{if} & mI > -1 \end{Bmatrix}$. The corresponding graph is shown in Fig. 5.

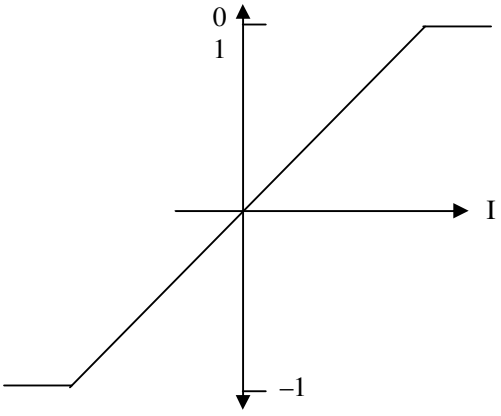


Fig. 5

iii) **Hard Limiter Function:** The output is given as $O = \text{sgn}[I]$, and the corresponding graph is shown in Fig. 6.

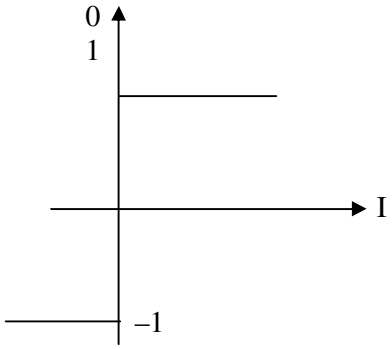


Fig. 6

iv) **Unipolar Sigmoidal and Bipolar Sigmoidal Function:** The output of unipolar sigmoidal function is given as $O = \frac{1}{(1+\exp(-\lambda I))}$ whereas the output of the bipolar function is given as $O = \tanh[\lambda I]$.

v) **Unipolar Multimodal Function:** The output is given as

$$O = \frac{1}{2}\left[1 + \frac{1}{M}\sum_{m=1}^{M} \tanh\left(g^m\left(I - W_O^m\right)\right)\right].$$

The graph is depicted in Fig. 7.

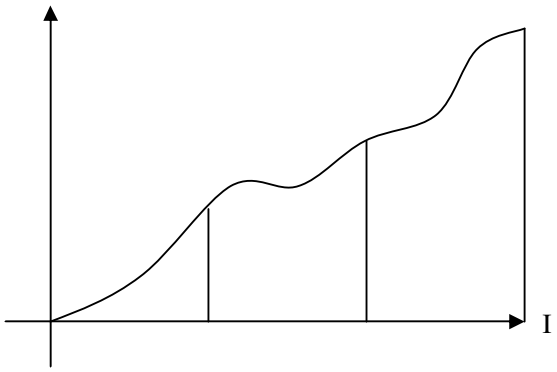


Fig. 7

vi) **Radial Basis Function (RBF):** The output is $O = \exp(I)$ where

$$I = \left[\frac{-\sum_{i=1}^{N} W_i(t) - X_i(t)^2}{2\sigma^2}\right].$$ This curve is same as the normal curve.

Considering threshold θ, the relative input to the neuron is given by

$$u(t) = W_1 I_1 + W_2 I_2 + \cdots + W_n I_n - \theta$$

$$= \sum_{i=0}^{n} W_i I_i \text{ where } W_0 = -\theta;\ I_0 = 1$$

And the output $O = f(u)$ where f is the nonlinear transfer function.

So far, we have discussed the mathematical details of a neuron at a single level. Although a single neuron can perform certain simple pattern detection problems, we need larger networks to offer greater computational capabilities. In order to mimic the layered structure of certain portions of the brain, let us explain the single feedforward neural network as shown in Fig. 8. Consider a single layer feedforward neural network shown in Fig. 8 consisting of an input layer to receive the inputs and an output layer to output the vectors respectively. The input layer consists of 'n' neurons and the output layer consists of 'm' neurons. Indicate the weight of the synapse connecting i^{th} input neuron to the j^{th} output neuron as W_{ij}. Now let us recall the single layer neural network with these notations, which is shown in Fig. 8. The inputs of the input layer and the corresponding outputs of the output layer are given as

$$\text{Consider} \rightarrow I_I = \begin{Bmatrix} I_{I1} \\ I_{I2} \\ \vdots \\ I_{Im} \end{Bmatrix}; \text{ and } O_O = \begin{Bmatrix} O_{O1} \\ O_{O2} \\ \vdots \\ O_{On} \end{Bmatrix}$$

Here input layer consists of m neurons and the output layer consists of n neurons that the input layer use linear transfer function and the output layer use unipolar sigmoidal function. Accordingly, $\underset{m\times 1}{\{O_I\}} = \underset{n\times 1}{\{I_I\}}$ or $I_{O_j} = \sum_{i=I}^{n} w_{ij}\, I_{Ii}$. The input to the output layer can be given as

$$[I_O] = [W]^t [O_I] = [W]^t [I_I]$$
$$n\times 1 \quad n\times m\, m\times 1 \quad n\times m\, m\times 1$$

$$O_{Ok} = \frac{1}{\left(1+e^{-\lambda I_{Ok}}\right)}$$

or $$[O_O] = f[WI]$$

where, λ is sigmoidal gain and $[W]$ is weight matrix and is also known as connection matrix.

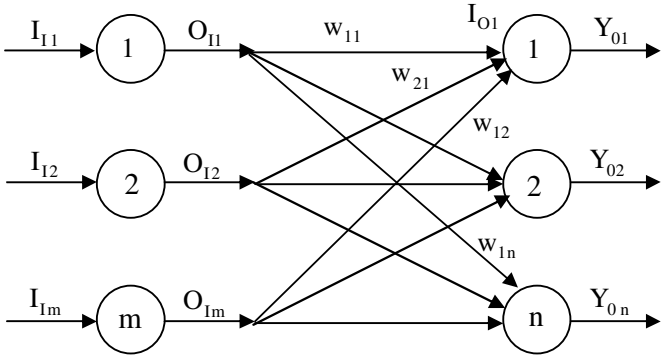


Fig. 8: Single layer feedforward neural network

The nonlinear activation function $f[WI]$ operates component wise on the activation values 'I' of each neuron, where each activation value is in turn a scalar product of the input with respect to weight.

The sigmoidal function is given as

$$f(I) = \frac{1}{\left(1+e^{-\lambda I}\right)}$$

and $$f'(I) = \lambda f(I)\left(I - f(I)\right)$$

Now let us extend the single layer model to multi layer model.

In multi layer perceptron the adapted perceptrons are arranged in layer. This model has three layers; an input layer, and output layer, and a layer in between the input and the output called the hidden layer. We use linear transfer function for the perceptrons in the input layer and sigmoidal or squashed-S functions for hidden layer and the output layer. The input layer does not perform a weighted sum or threshold. In this, we have modified the single layer perceptron by changing the nonlinearity from a step function to a sigmoidal function and added a hidden layer. This type of network recognizes more complex things. The input-output mapping of multilayer perceptron is given by

$$O = N_3\left[N_2\left[N_1[I]\right]\right]$$

where N_1, is the nonlinear mapping provided by input, N_2 is the nonlinear mapping provide by hidden layer and N_3 is the nonlinerar mapping provided by output layer. Multilayer perceptron provides many applications of neural networks, such as functional approximation, learning, generalization, etc. The multilayer network with one hidden layer is shown in Fig. 9. The activity of the output units are described by the activity of neurons in the hidden layer and the weight between the hidden and output layers and the neurons in the hidden layer is determined by the activities of the neurons in the input layer and the connecting weights between input and hidden units. In such networks neurons in the hidden layers are free to construct their own representations of the input.

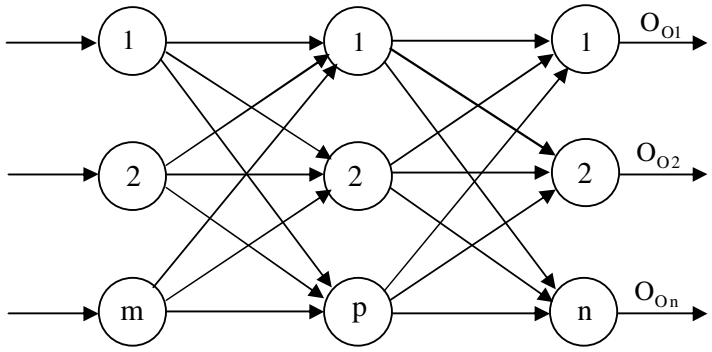


Fig. 9: Multilayer perceptron model

Now, we shall discuss MADALINE network, which is an example of multilayer perceptron. MADALINE is abbreviated for many ADALINE i.e. many ADALINE are connected to create such network. The MADALINE network helps countering the problem of non-linear separability. For example, the MADALINE network with two units can be applied to find a solution of the XOR problem. In its functioning the input bits x_1, x_2 are received by each unit of ADALINE and the bias input is assumed as 1 as its input. The calculated weighted sum of the inputs is passed on to the bipolar threshold units. The final output is obtained by computing the logical 'and' ing (bipolar) of the two threshold outputs. The computation of output is as given in Table 1.

Table 1

Threshold Outputs			Output
x_1	x_2		
+1	+1	Even parity	1
–1	–1		1
+1	–1	Odd parity	–1
–1	+1		–1

The corresponding network is shown in Fig. 10.

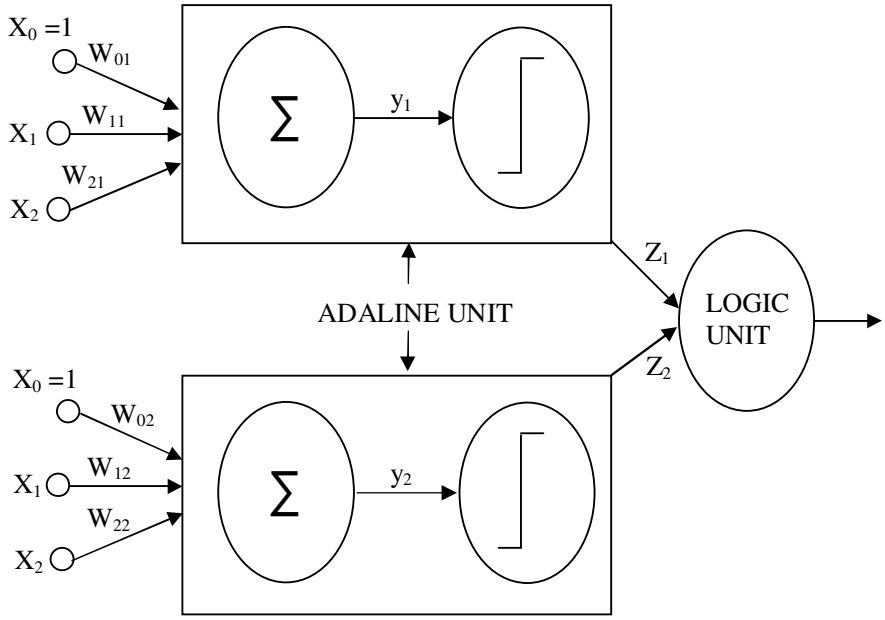


Fig. 10: A MADALINE network

Now, try the following exercises.

E1) Consider the weights w_{11}, w_{12}, w_{21} and w_{22} on connection from the input neurons to the hidden layer neurons and v_1, v_2 be the weights on the connections from the hidden layer neurons to the output neurons with following values. $w_{11} = 0.15$, $w_{12} = 0.3$, $w_{21} = 0.15$, $w_{22} = 0.3$, $v_1 = -0.3$ and $v_2 = 0.3$. Also consider the input (0, 0) generates the output (0, 0), (1,1) generates (1,1) and (1, 0) or (0, 1) generate (0, 1) as the hidden layer output.

i) Write the output of the neuron set at the hidden layer.

ii) Check whether the vectors in this set are linear separable or not. Give reason.

iii) Assume that input (0, 0) and (1, 1) gives 0, and (0, 1) generates 1 as output at outer layer, then draw the diagraph of this network.

iv) Obtain the activation of layers in the network.

E2) Show the decision boundaries of MADALINE to solve XOR problem.

E3) Consider the following table for the connections between the input neurons and the hidden layer neurons.

Input neuron	Hidden layer neurons	Connection weights
1	1	1
1	2	0.1
1	3	–1
2	1	1
2	2	–1
2	3	–1
3	1	0.2
3	2	0.3
3	3	0.6

The connections weights from Hidden layer neurons to the output neurons are 0.6, 0.3 and 0.6 for first, second and third neurons respectively corresponding threshold value for output layer is 0.5 and for hidden layer 1.8, 0.05 and – 0.2 for first, second and third neuron respectively,

i) Draw the diagraph of the network.

ii) Write the results of activation and interpret.

Now, in the following section we shall discuss the backpropagation learning.

6.3 BACKPROPAGATION LEARNING

Consider the network with input, hidden and output neurons, where the corresponding activities are denoted by writing subscript I, H, O for input, hidden and output neurons respectively. Also consider the weighted synapses from i^{th} hidden neurons to j^{th} input neuron, denoted by V_{ij}. The corresponding diagram is represented in Fig. 11.

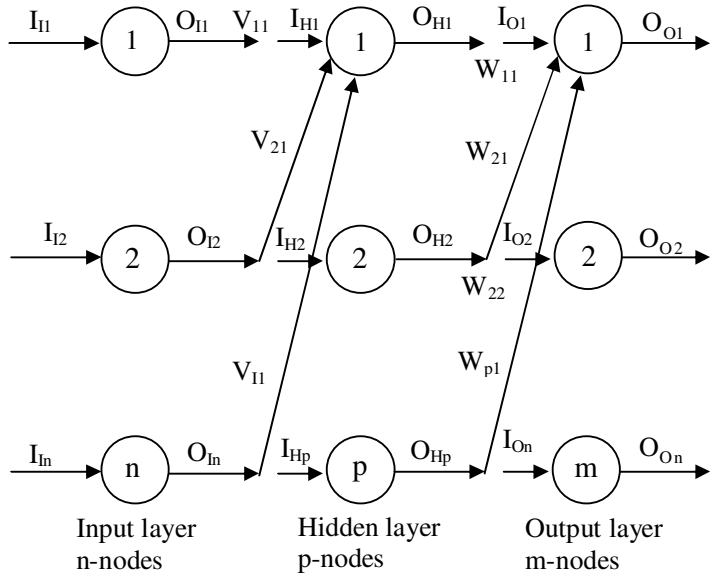


Fig. 11: Multilayer feedforward backpropagation network model

Let us assume the activation function for the output of the input layer to the input of input layer linear for this

$$\underset{n\times 1}{\{O\}_I} = \underset{n\times 1}{\{I\}_I}$$

input to the hidden neuron is the weighted sum of the outputs of the input neurons to get I_{Hp} (i.e. Input to the p^{th} hidden neuron) as

Therefore $$I_{Hp} = \sum_{i=1}^{n} V_{ip} O_{Ii}$$

where $$n = 1, 2, 3, \cdots, p$$

Denoting weight matrix or connectivity matrix between input neurons and hidden neurons as $\underset{l\times m}{[V]}$, we can get an input to the hidden neuron as

In the matrix notation, $\underset{p\times 1}{\{I\}_H} = \underset{p\times n}{[V]^T} \underset{n\times 1}{\{O\}_I}$

Let us again assume the activation function for the output of the p^{th} hidden neuron as sigmoidal function or squashed-S function. Then the output O_{Hp} is given by

$$O_{Hp} = \frac{1}{\left(1+e^{-\lambda\left(I_{Hp}-\theta_{Hp}\right)}\right)}$$

θ_{Hp} is the threshold of the p^{th} neuron. The computation of hidden layer is shown in Fig. 12.

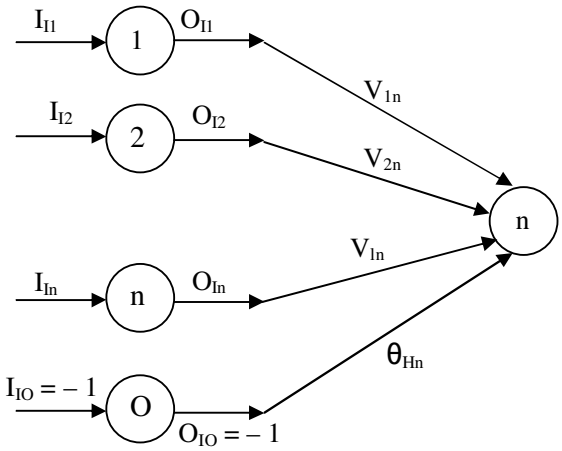


Fig. 12: Threshold in hidden layer

Here, the output of each component to the hidden neuron is given by

$$[O]_H = \left\{ \begin{array}{c} - \\ - \\ \frac{1}{\left(1+e^{-\lambda\left(I_{Hn}-\theta_{Hn}\right)}\right)} \\ - \\ - \end{array} \right\}$$

The input to the i^{th} output neuron

$$I_{Oi} = \sum_{j=1}^{i} W_{ji} O_{Hj} \qquad i = 1,2,3,\cdots m$$

The matrix notation is

$$\underset{m\times 1}{[I]_O} = \underset{m\times p}{[W]^T} \underset{p\times 1}{[O]_H}$$

Considering the output of the q^{th} output neuron O_{Oi} is given by

$$O_{Oi} = \frac{1}{\left(1+e^{-\lambda\left(I_{Oi}-\theta_{Oi}\right)}\right)} \text{ (considering the sigmoidal function)}$$

where θ_{Oq} is the threshold of the q^{th} neuron. This threshold is shown in Fig. 13.

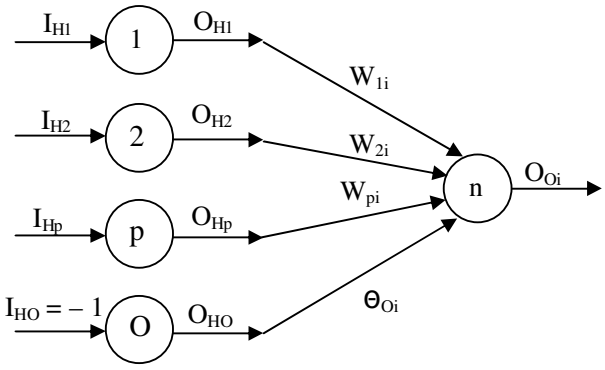


Fig. 13: Threshold in output layer

Similarly, the outputs of output neurons are given by

$$[O]_H = \begin{Bmatrix} - \\ - \\ \dfrac{1}{\left(1+e^{-\lambda\left(I_{Oi}-\theta'_{Oi}\right)}\right)} \\ - \\ - \end{Bmatrix}$$

Now let us calculate the error.

The Euclidean norm of error E_1^0 for the first training pattern is given by

$$\sum_{i=1}^{n} E_{1i} = \frac{1}{2} \sum_{i=1}^{n} \left(O_T - O_C\right)^2$$

where E_{1i} is the error in the i^{th} neuron for the first training patterns, and O_T is the target output and O_C is the calculated output.

Using the error, let us now write the modified values of weight vectors using steepes descent method.

$$[V]^{t+1} = [V]^t + [\Delta V]^{t+1}$$
$$[W]^{t+1} = [W]^t + [\Delta W]^{t+1}$$

where

$$[\Delta w]^{t+1} = \alpha[\Delta w]^t + \eta[y],$$
$$p\times m \qquad p\times m \qquad p\times m$$
$$[\Delta v]^{t+1} = \alpha[\Delta v]^t + \eta[x], \quad \text{here } \eta \text{ is learning rate.}$$
$$n\times p \qquad n\times p \qquad n\times p$$
$$[y] = [o]_H \quad \left[d_y\right]'$$
$$p\times m \quad p\times 1 \quad 1\times m'$$
$$\left[d_y\right]_{m\times 1} = \begin{bmatrix} - \\ - \\ \left(O_T - O_{Ok}\right) O_{Ok}\left(1-O_{Ok}\right) \\ - \end{bmatrix}$$

$$\underset{1\times p}{[X]} = \underset{1\times 1}{[o]_I} \quad \underset{1\times p}{[d_x]} = \underset{1\times 1}{[I]_I} \quad \underset{1\times p}{[d_x]'}$$

$$[d_x]_{p\times 1} = \begin{bmatrix} - \\ - \\ e_i [O_{Hi}](1-O_{Hi}) \\ - \\ - \end{bmatrix}$$

and

$$[e]_{p\times 1} = [w]_{p\times m} \; [d_y]_{m\times 1}$$

The weights and threshold may be updated as

$$[W]^{t+1} = [W]^t + [\Delta W]^{t+1}$$
$$[V]^{t+1} = [V]^t + [\Delta V]^{t+1}$$
$$[\theta]_O^{t+1} = [\theta]_O^t + [\Delta\theta]_O^{t+1}$$
$$[\theta]_H^{t+1} = [\theta]_H^t + [\Delta\theta]_H^{t+1}$$

Now, after defining the backpropagation network let us take the backpropagation algorithm for this. Let us consider the three-layer network with input layer having 'n' nodes, hidden layer having 'p' nodes, and an output layer with 'm' nodes. We consider sigmoidal functions for activation functions for the hidden and output layers and linear activation function for input layer. Now let us write the backpropagation algorithm stepwise.

Step 1: Initialize the weights V and W usually from –1 to 1.

Step 2: For each training pair, assume there are 'n' inputs given by $\underset{n\times 1}{[I]_I}$ and 'm' output $\underset{m\times 1}{[O]_O}$ in a normalized form.

Step 3: Set the number of neurons in the hidden layer to lie between $1 < p < 21$.

Step 4: Set $\lambda = 1$ and threshold value 0.

Step 5: Compute the output of the input layer.

Step 6: Compute the inputs and outputs to the hidden layer.

Step 7: Compute the inputs and outputs to the output layer.

Step 8: Calculate the error.

Step 9: Find

$$[V]^{t+1} = [V]^t + [\Delta V]^{t+1}$$
$$[W]^{t+1} = [W]^t + [\Delta W]^{t+1}$$

Step 10: Repeat steps 5 to 9 until the convergence in the error rate is less than the tolerance value.

Let us cite the following example to understand the algorithm.

Example 1: Consider the three training sets given in the following Table 2,

Table 2

Inputs		Output
I_1	I_1	O
0.3	– 0.2	0.2
0.4	0.6	0.3
0.6	– 0.2	0.1

with the initial vectors $[W]^0 = \begin{bmatrix} 0.2 \\ -0.5 \end{bmatrix}$ and $[V]^0 = \begin{bmatrix} 0.1 & 0.4 \\ -0.2 & 0.2 \end{bmatrix}$.

Solution: Let us find the improved weights. The architecture of this model is given in Fig. 14.

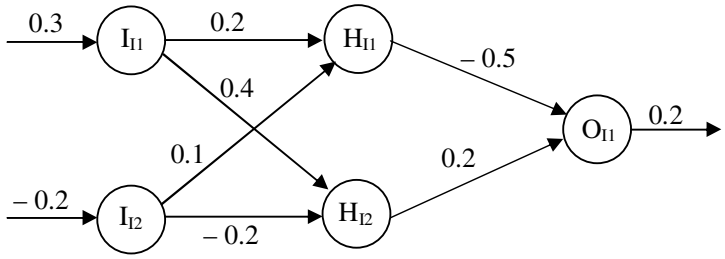


Fig. 14: Multilayer architecture

$$[I]_H = [V]^T \quad [O]_I$$

$$= \begin{bmatrix} 0.2 & 0.1 \\ 0.4 & -0.2 \end{bmatrix} \begin{bmatrix} 0.3 \\ -0.2 \end{bmatrix} = \begin{bmatrix} 0.04 \\ 0.16 \end{bmatrix}$$

$$[O_C]_H = \begin{bmatrix} \frac{1}{1+e^{-0.04}} \\ \frac{1}{1+1e^{-0.16}} \end{bmatrix} = \begin{bmatrix} 0.51 \\ 0.54 \end{bmatrix}$$

$$[I]_O = [w]^t [O]_H = [-0.5 \quad 0.2] \begin{bmatrix} 0.51 \\ 0.52 \end{bmatrix}$$

$$= -0.15$$

$$[O_C]_O = \frac{1}{1+e^{0.151}} = 0.462$$

$$\text{error} = [O_{TO} - O_{CO}]^2$$

$$= [0.2 - 0.462]^2 = 0.069$$

$$d = (O_{TO} - O_{CO1})(O_{CO1})$$

$$= (0.2 - 0.462)(0.462)(1 - 0.462) = -0.065$$

$$[O_C]_H [d] = \begin{bmatrix} -0.033 \\ -0.035 \end{bmatrix}$$

Here assume that $\alpha = 1$ and $\eta = 0.5$

$$[\Delta w]^1 = \alpha [\Delta w]^0 + \eta [y]$$

$$= \begin{bmatrix} -0.517 \\ 0.182 \end{bmatrix}$$

$$e = [w][d] = \begin{bmatrix} 0.032 \\ -0.013 \end{bmatrix}$$

$$d^* = \begin{bmatrix} -0.001 \\ 0.0004 \end{bmatrix}$$

$$X = [O]_i \left[d^*\right]^1 = \begin{bmatrix} 0.0003 & 0.0001 \\ 0.0002 & -0.00009 \end{bmatrix}$$

$$[V]^1 = \begin{bmatrix} 0.1998 & 0.40007 \\ 0.1001 & -0.20004 \end{bmatrix}$$

and $[W]^1 = \begin{bmatrix} -1.01 \\ 0.38 \end{bmatrix}$

Using these modified $[V]^1$ and $[w]^1$, error is calculated and then can be processed for next training set.

Now, try the following exercises.

E4) Find the modified weights for the training set having input $I_1 = 0.3$, $I_2 = -0.5$ and output = 0.1 with $[V]^0 = \begin{bmatrix} 0.1 & 0.4 \\ -0.2 & 0.2 \end{bmatrix}$ and $[W]^0 = \begin{bmatrix} 0.2 \\ -0.5 \end{bmatrix}$.

E5) How many Layers are there in Multi layer Neural Network?

E6) What is cycle to modify the weight values?

Now, let us summarize the unit.

6.4 SUMMARY

In this unit, we have covered the following points.

i) The neural network architectures are broadly classified as single layer feed forward networks and multilayer feed forward networks. If only input and output layers are present, then the network is single layer network. In turn, if in addition to the input and output layer one or more intermediate layer exists, then the network is multilayer network.

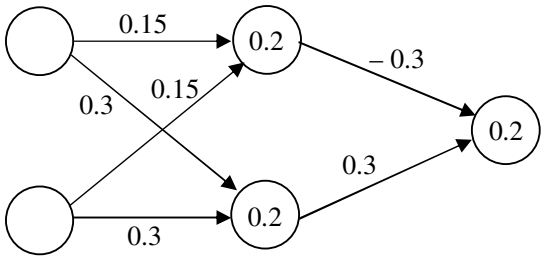
ii) Backpropagation is a systematic method of training multilayer neural networks. It is built on high mathematical foundation and has very good application potential.

6.5 SOLUTIONS/ANSWERS

E1) i) Output set = {(0, 0), (1, 1), (0, 1)}

ii) These three vectors given in (i) are separable with (0, 0) and (1, 1) on one side of the separating line, while (0, 1) is on the other side.

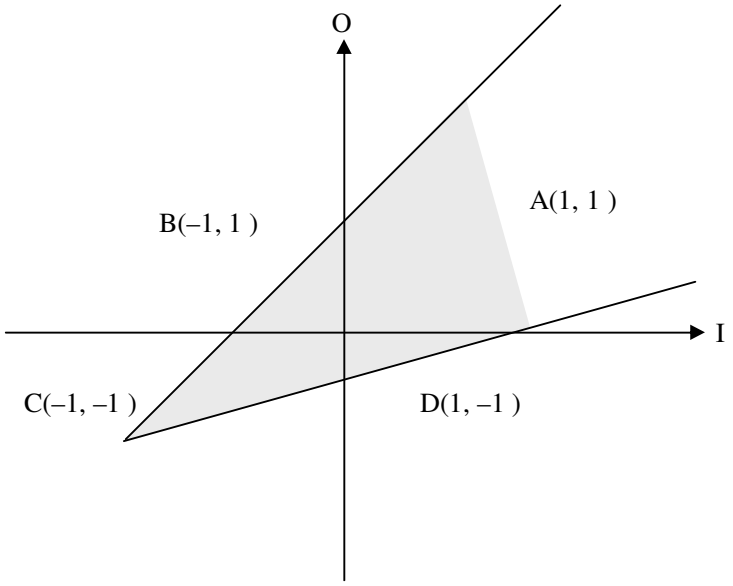
iii)



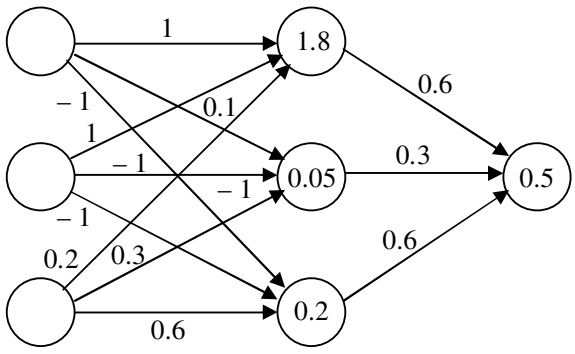
iv)

Input	Activation (Hidden layer)	Output (Hidden layer)	Output neuron activation	Output of the network
(0, 0)	(0, 0)	(0, 0)	0	0
(1, 1)	(0.3, 0.6)	(1, 1)	0	0
(0, 1)	(0.15, 0.3)	(0, 1)	0.3	1
(1, 0)	(0.15, 0.3)	(0, 1)	0.3	1

E2)



E3) i)



ii) Let us consider the vertexes of a cube be O, A, B, C, D, E, F and G be (0, 0, 0,), (0, 0, 1), (0, 1, 0), (0, 1, 1,), (1, 0, 0), (1, 0, 1), (1, 1, 0) and (1, 1, 1).

Vertex/ Coordinates	Hidden Layer Neuron	Weighted Sum	Comment	Activation	Contribution to output	Sum
O: 0, 0, 0	1	0+0+0 = 0	< 1.8	0	0	
	2	0+0+0 = 0	< 0.05	0	0	
	3	0+0+0 = 0	> –0.2	1	0.6	0.6*
A: 0, 0, 1	1	0+0+0.2 = 0.2	< 1.8	0	0	
	2	0+0+0.3 = 0.3	> 0.05	1	0.3	
	3	0+0+0.6 = 0.6	> –0.2	1	0.6	0.9*
B: 0, 1, 0	1	0+1+0 = 1	< 1.8	0	0	
	2	0–1+0 = –1	< 0.05	0	0	
	3	0–1+0 = –1	< –0.2	0	0	0
C: 0, 1, 1	1	0+1+0.2 = 1.2	< 1.8	0	0	
	2	0+0.1+0.2 = 0.2	> 0.05	1	0.3	
	3	0–1+0.6 = – 0.4	< – 0.2	0	0	0.3
D: 1, 0, 0	1	1+0+0 = 1	< 1.8	0	0	
	2	0.1+0+0 = 0.1	> 0.05	1	0.3	
	3	–1+0+0=–1	< –0.2	0	0	0.3
E: 1, 0, 1	1	1–0+0.2= 1.2	< 1.8	0	0	
	2	0.1+0+0.2 = 0.4	> 0.05	1	0.3	
	3	–1+0+0.6 = –0.4	< –0.2	0	0	0.3
F: 1, 1, 0	1	1+1+0 = 2	> 1.8	1	0.6	
	2	0.1–1+0 = – 0.9	< 0.05	0	0	
	3	–1–1+0 = –2	< – 0.2	0	0	0.6*
G: 1, 1, 1	1	1+1+0.2 = 2.2	> 1.8	1	0.6	
	2	0.1–1+0.3 = – 0.8	< 0.05	0	0	
	3	–1–1+0.6 = – 1.4	< – 0.2	0	0	0.6*

* The output neuron fires, as this value is greater than 0.5 (the threshold value); the function value is + 1.

E4) The following values are obtained:

$$[V]^0 = \begin{bmatrix} 0.1 & 0.4 \\ -0.2 & 0.2 \end{bmatrix}$$

$$[W]^0 = \begin{bmatrix} 0.2 \\ -0.5 \end{bmatrix}$$

$$[O]_I = \begin{bmatrix} 0.3 \\ -0.5 \end{bmatrix}$$

$$[I]_H = \begin{bmatrix} 0.13 \\ 0.02 \end{bmatrix}$$

$$[O_C]_H = \begin{bmatrix} 0.53 \\ 0.50 \end{bmatrix}$$

$$[I] = [-0.146]$$

$[O_C]_O = 0.464$

error = 0.132

assuming $[\alpha = 0.5, \eta = 0.5]$

d = – 0.090

$$[Y] = \begin{bmatrix} -0.048 \\ -0.0456 \end{bmatrix}$$

$$[\Delta W]^1 = \begin{bmatrix} 0.076 \\ -0.273 \end{bmatrix}$$

$$e = \begin{bmatrix} -0.018 \\ 0.045 \end{bmatrix}$$

$$d^* = \begin{bmatrix} -0.001 \\ -0.002 \end{bmatrix}$$

$$X = \begin{bmatrix} -0.0003 & -0.0006 \\ 0.0005 & 0.0012 \end{bmatrix}$$

$$[V]^1 = \begin{bmatrix} 0.0498 & 0.1997 \\ -0.0997 & 0.1005 \end{bmatrix}$$

$$[W]^1 = \begin{bmatrix} 0.276 \\ -0.773 \end{bmatrix}$$

6.6 PRACTICAL ASSIGNMENT

Session 7

Write a program in 'C' language to implement the backpropagation algorithm. Show step by step output to input, hidden and output neurons as well as errors. How the weights W and V modified? Use data given in Example 1.

6.7 REFERENCES

1. Simon S Haykin and Simon Haykin, (1998), *Neural Networks: A Comprehensive Foundation*, Pearson Education.
2. Raul Rojas, (1996), *Neural Networks: A Systematic Introduction*.
3. S. Rajasekaran and G.A. Vijayalakshmi Pai, (2010), *Neural Netwoks: Fuzzy Logic, and Genetic Algorithms*.
4. D.K. Pratihar, (2008), *Soft Computing*.
5. Valluru B. Rao and Hayagriva V. Rao, *C + + Neural Networks & Fuzzy Logic*.