5.0 INTRODUCTION

In the previous chapters, pattern recognition techniques have been presented which use known statistical information about the patterns. It was shown that when the apriori probabilities of the pattern classes, the class conditional probability density functions, and performance measures, like probability of error, are specified that the optimal classifier can be found. The optimum structure for the two pattern class problem involved the likelihood ratio awhile the optimum structure for a multiple pattern class problem for the fixed 0,1 cost function was a parallel configuration containing evaluation of the likelihood functions for the classes followed by a maximum operation.

If the statistical information is not available it was shown in Chapter 3 that a linear classifier and a generalized linear classifier could be developed in a sequential manner by using available patterns with known classification as "training" samples. Representative structures of these type of classifiers along with the statistically based classifiers are shown in Figure 5.1 for the two class and multiple class cases.
Figure 5-1 Examples of statistical classifiers, (a) two class likelihood ratio test case (b) two class linear (perceptron) (c) two class Generalized Linear and (d) K class case (Maximum $p(x | C_k)$).

Notice that all of these structures calculate one or more functions of the pattern vector and go through either a threshold device or a maximum or a minimum operation.

In this chapter a structure known as a neural network will be explored for the purpose of pattern classification. In a sense they resemble the structures of Fig. 5-1; however, they differ in that the structure is usually of a fixed form with a set of variable parameters. The parameters are changed by a training algorithm and thus are functions of a training set of pattern vectors with known class affiliation. In this way the pattern classifier is "learned" by the network.

Central to the concept of neural networks will be the topic of training. Various training methods will be explored and a few representative examples will be presented. Some guidelines for selecting structures and parameters of the neural networks will also be presented. Currently neural networks are being successfully applied to problems in character recognition, medical diagnosis, financial planning, weather prediction, etc.

5.1 HISTORICAL PERSPECTIVE

Researchers have for a long time looked for methods and circuits that mimic living organisms, humans and insects. A human baby at the age of a couple of months can recognize their mother and family members voices and later can recognize several different toys. A mosquito can recognize the difference between a blood source and other objects, whereas a computer vision system including camera and computer may be over burdened to perform the same job. Therefore it is not unreasonable to look to the biological sciences
especially neuroanatomy and neurophysiology for structures and methods for pattern recognition.

Models for neurons have been topics of research for many years in the medical sciences arena and will continue to be. There is no intent in this chapter to survey and review all this research as the main purpose will be to identify several milestones and developments that changed the research directions and provided the foundation of what are called a neural networks today.

The first identifiable development was the neuron model by McCulloch-Pitts presented in 1943. This model and later modifications to it are still fundamental to research in neural networks today. Fifteen years later, in 1958, the perceptron algorithm was introduced by Rosenblatt, followed by further research in training of linear machines and various connections of linear machines. Typical developments include the ADALINE and MADALINE of Widrow and Hoff (1962). Finding algorithms for multiple linear machines was hindered by the use of threshold or unit step like devices. Later a book by Minsky and Papert [22] in 1969 expressing the limitations of perceptron type networks, slowed research in the neural network area. In 1972 the backpropagation algorithm was developed by Weberos[ ] for training multiple layer neural networks with continuous "activation functions". This algorithm was later modified and popularized in the early eighties by Grossberg, Hopfield, and Rumelhart causing a resurgence of research in neural structures, training methods, and applications. Like any new tool it was used and continues to be used on all types of problems, some successfully and others not. Currently, guidelines are being sorted out for meaningful applications.

5.2 A REAL NEURON

The investigation of neuronal activity in humans and various forms of life has been intriguing area for study. It was felt that understanding human and animal neuronal activity would allow us to "build" machines that would mimic human responses. Experiments on neuronal activity in earthworms, limulus crabs, and other mammals in the last 50 years have given us a fundamental understanding of neuronal activity yet falls short of being able to explain how we think and what is the underlying human mechanisms that give us the ability to recognize complex patterns and organize our behavior.

5.2.1 Neuronal Anatomy

It is commonly agreed that the elemental anatomic unit of the human nervous system is a neuron which can be viewed anatomically as shown in Figure 5.2. The nervous system is then considered a collection of neurons.
Figure 5-2 Schematic view of a neuron, the elemental anatomic unit of the human nervous system.

The area in the upper left of the figure consists of a **cell body** or soma which has a width of around 150 µm, a nucleus, and an active region called the **axon Hillock** which is about 30µm in diameter. Various extensions run in all directions off the soma. The short arm like extensions (shown to the left in Figure 5.2), about 200-300 µm in length, are called **dendrites**. They serve to conduct electrical activity to the cell body and axon Hillock region for communication between neurons. The number of dendrites can vary from 2000, for smaller neurons, to as many as 16000 for larger neurons. The relatively long thin extension from the axon Hillock region is called the **axon** and can be as short as 50 µm or in some cases as long as 1 meter. The axon has a diameter of approximately 20 µm. The axon serves as a "transmission line" that allows communication between neurons. The axon of the longer neurons are encased in a **myelinated shield** to facilitate the conduction and provide isolation from surrounding neuronal activity. For the human neuron the **conduction speed** of electrical transmission is approximately 120 meters/sec which is about 400 ft/sec. The end of the axon is spread out in many small branches providing electrical contact with hundreds of other neurons through their dendrites.
The electrical contact between axon and dendrites is illustrated in Figure 5.3. At the end of a each branch of the axon is an axon terminal or end bulb which is separated from a dendrite of another neuron by the synaptic gap. The electrical information from the axon is transferred to the receptor of the dendrite of another neuron by chemical means through a transmitter substance. The transmitter can be either + or - ions, the most common being acetocholine and ??

5.2.2 Neuron Communication

Information is transferred between neurons via the synaptic junctions between axon and dendrites and created at the axon hillock region. Raises in potentials caused by electrical activities integrated from all dendritic connections causes an increase in potential at the axon hillock region. If the potential at the hillock exceeds a threshold a sequence of impulses or action potentials is created for conduction down the axon to other neurons. The typical shape of an action potential is shown in Figure 5.4.

The potential between inside and outside of the axon for an action potential is shown in Figure 5.4 and is composed of a threshold potential, a latency region, a refractory region, and an action potential. The latency time is an inherent delay before the spike of the action potential. In the refractory period the neuron can not respond to any other activity because of physical limitations of the neuron. If the "input" to the axon hillock is below the threshold the axon potential tracks the soma potential as shown in Figure 5.5. It is also been experimentally shown that as the axon hillock's potential is increased above the threshold a series of spikes or impulses is produced where the frequency of occurrence is proportional to the potential above the threshold. As long as the potential exceeds the threshold the impulses continue. The higher the potential the higher the frequency of impulses up to a point where the refractory and latency limit restrict production of action potentials. The rate of impulses can vary from 10 to 500 pulses/sec for various human neurons. Thus it is seen that the axon hillock membrane potential information is frequency coded for transmission of information down the axon to the axon terminals and dendrites of other neurons.

Figure 5-3 Synapse between axon and dendrite

Figure 5-4 Typical shape of an action potential.
Once these impulses reach the axon terminal they release ions of a particular sign either positive(+) or negative(-) but fixed for each terminus, for conduction across the synaptic gap. The travel of these ions across the gap induces a voltage at the dendrite membrane which eventually will have an effect on the potential at the axon hillock region of another neuron. The potential at the dendrite membrane is roughly proportional to the frequency of impulses reaching the terminus of the connecting axon. As the chemical transmission is much slower in nature than the impulse frequency the **dendritic potential** is much slower as shown in figure 5.6.

**Figure 5-5** Relationship between axon hillock membrane potential and axon potential.

The dendritic membrane potential is sometimes called a "slow potential" and its value is directly proportional to the rate or frequency of the axon impulse frequency, thus
transferring the original soma potential or information to another neuron.

5.3 MCCOLLOCH-PITTS NEURONAL MODEL

A simple model used for trying to duplicate results corresponding to real neurons was proposed by McCollough and Pitts in 195??. The basic model is shown in Figure 5.7 and is seen to be composed of four basic components: a sensory input vector, a plus or minus weighting, a summer, and a threshold unit.

For the sensory input vector the components $x_1, x_2, \ldots, x_n$ are binary variables representing input stimuli and take on only a one or zero value corresponding to whether the stimulus is present or not present respectively. The weights $w_1, w_2, \ldots, w_n$ are assigned either a plus one or minus one corresponding to excitatory or inhibitory operation respectively. The summer adds all the weighted sensory inputs while the threshold device gives an output of one if the weighted sum of the vector components of the input $x$ is greater than or equal $T$ and a zero otherwise. The power of a model like this, in terms of operations it can perform, is remarkable considering the simplicity of its structure. First of all since the input can be considered a binary vector and the output a binary output the model can produce basic binary logic functions. With the proper selection of weights and threshold this model can be shown to perform the logical AND, OR, and NOT functions as shown in Figure 5.8.

Since each neural element can produce the above specified logical operations, a connection of McCollough-Pitts neurons can produce any binary logic function. An example showing the implementation of a specific binary logic expression using the "AND", "OR", and "NOT" McCollough neurons is given in the following example.

Figure 5-7 McCollough-Pitts neuron model.
Figure 5-8 Basic logic functions using McCollough-Pitts neuron model (a) AND, (b) OR, (c) NOT.

EXAMPLE 5.1
Use the basic McCollough-Pitts neuron model to generate the logical expression

\[ f(x_1, x_2, x_3, x_4) = x_1'x_2x_3 + x_1x_2'x_3'x_4 \]

where the ' represents the compliment of the variable.

Solution:
Each of the terms in the logical expression \( f(x) \) can be formed with an AND operation with the proper input variables which are then combined using an OR operation. If a first layer is used to produce the NOTs then the three layer network given in Figure 5.9 produces the required logical expression.

Figure 5-9 Using the McCollough-Pitts basic neural element to implement the binary logic expression \( f(x_1, x_2, x_3, x_4) = x_1x_2'x_3 + x_1'x_2x_3'x_4 \) of Example 5.1
In general using a NOT bank of neurons followed by two layers implementing AND and OR operations, a sum of minterms can be obtained thus implementing any binary logic expression. If we include a memory device then all sequential binary logic expressions can be implemented using multilayer networks containing only McCulloch-Pitts neurons.

Since multilevel logic functions can be written in terms of binary logic functions, multiple McCollough-Pitts neurons are capable of generating all multilevel logic functions. Furthermore by defining a memory element, the Mc-Collough-Pitts neuron can be used to implement any finite-state machine or sequential logic function.

The power obtained using this simple neural model is obvious from the above discussion. If you have the logical expressions then the configuration can be composed of the basic elements. However, the process of training such a network to implement an unknown logical expression becomes almost a trial and error exhaustive type search. Another limitation is that inputs are binary or multilevel discrete variables. A generalization of the McCulloch Pitts model for the case of continuous variables that will allow convenient training algorithms is the basic neural element described in the following section. This generalization will also increase the power of layered networks to be able to approximate any real valued function rather than just binary or multi-level logic functions and thus be universal approximators.

5.4 ARTIFICIAL NEURAL ELEMENT

The McCollough-Pitts neural element uses only binary vector inputs and plus or minus one weights with a "hard" thresholding operation. It seems reasonable to therefore extend the model to include continuous value inputs, arbitrary continuous plus and minus weights, and "soft" or continuous nonlinearities instead of using just a thresholding operation. With this in mind the basic Artificial Neural Element(ANE) or neuron used throughout this presentation is introduced in Figure 5.11.

![Figure 5-10 Basic Artificial Neural Element (ANE).](image)

The input to the ANE is a vector of real valued components $x_1, x_2, \ldots, x_n$ along with a 1, and thus can be thought of as an augmented pattern vector. The weights $w_1, w_2, \ldots, w_{n+1}$, also real valued, can be thought of as a vector of multipliers. In the neuron model "net" is defined as the output of the weighted inputs which in vector form can be thought as a dot product or a matrix multiplication. The input to the nonlinear function is a single real value and produces a single real valued output $y$. Analytically the input output relationship can be
expressed as follows

\[ \text{net} = \sum w_i x_i \]  
\[ y = f(\text{net}) = f(\mathbf{w}^T \mathbf{x}) \]  

The function \( f(.) \) is called an **activation function**. Using the vector notation just described, two short hand notations for the Basic Artificial Neural Element are shown in Figure 5-11(b) and (c). The nodal representation is shown in Fig. 5-11(b) where the circle stands for summation followed by a nonlinear activation function. In 5-11(c) the double sided input arrow represents a vector input and the \( \mathbf{w} \) above means the dot product with the input vector \( \mathbf{x} \) and \( \mathbf{w} \). The circle indicates the nonlinear operation while the output arrow is indicative of a single scalar output.

**Figure 5-11** Notational conventions for basic artificial neural network.

The activation function in general has no conditions placed on it although it is conventional to think of the function as nondecreasing and bounded between -1 and 1. An activation function is called **unipolar** if its output is limited to values greater than or equal to zero and **bipolar** if its output values contain both positive and negative values. Examples of continuous, discrete, and mixed unipolar and bipolar typical activation functions are shown in Fig. 5-12.

The four most commonly used activation functions are the logistic function, the hyperbolic tangent function, the unit step function, and the signum function as illustrated in Fig. 5-12. Suggestions for selection of the activation function for given problems will be discussed later. It will be shown that the choice of the activation function will impact the formulas, mechanics, and convergence times of the training algorithm; however, it does not have a significant impact on the overall results.

Other types of nonlinear functions, not necessarily monotonic, have been successfully used. One of these, the radial basis functions, will be presented later in this chapter and others are given in the problems section.
Figure 5-12 Typical activation functions: (a) logistic, (b) step function, (c) unipolar saturated ramp, (d) hyperbolic tangent, (e) signum function, (f) bipolar saturated ramp.

5.4.1 Logistic function

The **logistic function** is an example of a unipolar continuous activation function and its functional relationship is given as follows

\[
\frac{1}{1 + e^{\lambda \text{net}}}
\]

The value of the function at 0 is seen to be 0.5 regardless of the value of the parameter \( \lambda \) while \( \lambda \) controls the steepness of the activation function at the origin. The general shape of the function is shown in Figure 5.13 as the parameter lambda is increased from a small positive value to a large positive value. As \( \lambda \) approaches infinity the curve approaches the unit step function.
5.4.2 Hyperbolic activation function

The hyperbolic tangent function is a commonly used bipolar continuous activation function and its functional relationship and equivalent forms are given as follows

\[
f(\text{net}) = \tanh(\lambda \text{net}/2) \quad \frac{\exp(\lambda \text{net}/2) \& \exp(\text{net}/2)}{\exp(\lambda \text{net}/2) \& \exp(\text{net}/2)} \quad \frac{1 \& \exp(\text{net})}{1 \& \exp(\text{net})} = 1
\]

The value of the \( f(\text{net}) \) at the origin is 0 and if \( \lambda \) is very large \( f(\text{net}) \) approaches the signum function. Plots of the hyperbolic tangent activation function for different values of the parameter lambda are shown in Figure 5.14.

5.4.3 Unit step activation function
The unit step activation function is a unipolar binary function given by

\[
f(\text{net}) = \begin{cases} 
1 & \text{for net } \geq 0 \\
0 & \text{for net } < 0
\end{cases}
\]  

(5-5)

**5.4.4 Signum function activation function**

A commonly used bipolar binary activation function is the signum function defined as follows

\[
f(\text{net}) = \begin{cases} 
1 & \text{for net } \geq 0 \\
\&1 & \text{for net } < 0
\end{cases}
\]  

(5-6)

**5.4.5 Linear activation function**

The linear activation function is simply given by

\[
f(\text{net}) = \text{net}
\]  

(5-7)

**5.4.6 Non monotonic activation Functions**

The activation function for neural elements has primarily been chosen as a monotonic function. This does not mean that only monotonic functions should be used. It has been shown by various researchers that gaussian functions and other unimodal functions can be used as activation functions with success\[?\]. In many cases such activation functions actually mimic the solution that would be obtained in the statistical approach. A special class of nonmonotonic activation functions are the **radial basis functions** which will be defined in a later section on radial basis neural networks.

By using a nonmonotonic activation function a single neural element can create decision regions that are not just one side of a hyperplane. The following example shows a multiply connected decision region for the two class pattern recognition problem using a single neural element and a nonmonotonic activation function.

**EXAMPLE**

(similar to HW 5.7)
5.5 GENERAL CONCEPT OF TRAINING A SYSTEM

Before training procedures are developed for training a network to perform the particular job of pattern recognition the general concept of training a system will be discussed. It will be convenient to think of the training to be composed of several major parts: (1) the problem formulation and job statement, (2) the system to be trained, (3) the training samples and their associated desired outputs (4) the performance measure, and (5) the training algorithm.

5.5.1 Problem Formulation

The first essential part for the training of a neural network is the job of problem statement. This must contain a clear precise statement of exactly what you wish the system to be trained to do. For example the system is to be trained to classify a presented patterns into one of several different classes (pattern classification), or to determine a separation of the samples into pattern classes (clustering), or to approximate a particular function of pattern vectors (approximation), or to identify an unknown system (identification).

5.5.2 System To Be Trained

The system which is to be trained to perform the job specified in the problem formulation must be selected. Usually it will be a particular type of system with a given structure that contains a set of parameters that can be adjusted in some way by using a training set and a training algorithm. For example the system could be a linear time invariant system with exponential impulse response but with unknown amplitude and exponential factor as parameters. The perceptron discussed earlier represents a system that calculates the output for a particular input as a weighted sum of the input vectors components. The weights correspond to the parameter set that is to be adjusted by the training procedure.

The emphasis in this chapter is on nonlinear systems that are combinations of basic parametric nonlinear building blocks called artificial neural elements. During the process of training, the system changes its parameters by a specified algorithm. After the training the system parameters are, in most cases, frozen and the system performs the job it was supposed to do.

The system selected for training could be linear or nonlinear, time varying or stationary, memory or memory less, etc. The choice is up to the user and usually is based on experience and in many problems is nothing more than a a glorified trial and error process.

5.5.3 Training Samples

It is assumed that a set of training samples representative to the problem you wish to solve are available. Also the desired output for each of the input training samples is usually specified but not necessarily. For the pattern classification problem, samples from each of the classes could be given along with their correct classification.

It is usually good practice to divide the training samples into two subsets, one for the training of the system and the other for evaluating the performance of the trained system. It is hoped that the training samples used to train the network are representative of the classes and lead to a solution that correctly classifies the validation set of samples and other new presented samples.
5.5.4 Performance Measure

A performance measure serves as a yardstick for measuring the performance of the trained system and sometimes as a basis for the training algorithm. For example in pattern classification, a few common performance measures are probability of misclassification, mean squared error from desired output, and the average sum of absolute value of difference between desired and actual output. It will be seen that the choice of a performance measure may facilitate the training by yielding a simple training algorithm.

5.5.5 Training Algorithm

The training algorithm provides the procedure for adapting or changing the system parameters to minimize or maximize the specified performance measure by using the training samples. It describes how the training samples will be processed and how the parameters of the system will be changed. A training algorithm will be called "supervised", or with a teacher, if it uses the desired outputs for the training samples and "unsupervised", or without a teacher, if it does not use the training sample outputs. A rough block diagram for supervised and unsupervised training is shown in Figure 5.7. Unsupervised training may use additional information about the patterns but does not use the desired outputs. Also shown in Fig. 5-7 is the case of imperfect supervised training where the algorithm uses the desired outputs that may have been corrupted. Supervised training corresponds to closed loop adaption, while unsupervised training is comparable to open loop adaption in the control system vernacular.

If the change of the parameters occurs after each presentation of a training sample the algorithm is said to be trained "by sample" or sequentially trained, whereas if parameters are adjusted after presentation of the entire training set the algorithm is said to be trained "by epoch" or batch trained. For both cases the processing of all the patterns in the training set will be called one pass through the training samples. The perceptron algorithm from Chapter 4 is an example of a training by sample procedure while the Least mean squared error algorithm is an example of a training by epoch algorithm.

Let the training set be represented by \( N_{ts} \) ordered pairs of sample vector \( \mathbf{x}_j \) and desired output \( \mathbf{d}_j \) as follows

![Figure 5-15](image)
\[ T = \{ (x_j, d_j) : j = 1 \text{ to } N_{ts} \} \] (5-8)

If \( w \) represents the parameter vector and \( w(k) \) the parameter vector at the \( k \)th iteration and \( w(k+1) \) at the \( k+1 \) iteration, the training algorithm parameter update equation for training by sample can be written as a function \( g(.) \) of \( w(k), x(k), \) and \( y(k) \) as follows

\[ w(k+1) = g(w(k), x(k), d(k), y(k)) \] (5-9)

In the above equation \( x(k) \) is the presented training sample which comes from the training set \( T \), \( y(k) \) is the output vector for that \( x(k) \), and \( w(k+1) \) is the new or updated parameter vector as the result of training sample \( x(k) \), and \( d(k) \) is the desired output for the training sample \( x(k) \).

The **weight update algorithm** for the case of training by epoch can be written in terms of all the training samples, the desired output vectors for those samples and the output vectors for the system as

\[ w(k+1) = g(w(k), x_1, x_2, ..., x_{N_{ts}}, d_1, d_2, ..., d_{N_{ts}}, y_1(k), y_2(k), ..., y_{N_{ts}}(k)) \] (5-10)

### 5.6 TRAINING AN ARTIFICIAL NEURAL ELEMENT

In this section training of neural elements for solving approximation and pattern recognition problems will be explored. The discrete perceptron algorithm will be revisited with format changed to appear in the standard learning format which leads to the continuous perceptron algorithm. In particular the delta algorithm for neural elements with arbitrary differentiable activation functions is presented.

Usually the artificial neural element has a fixed nonlinear activation function and training means that the training samples will be used to change the weight vector to perform a particular job. The training algorithms for various nonlinear activation functions are now presented.

#### 5.6.1 Signum function activation training algorithm

The basic neural element with a signum activation function is equivalent to the discrete perceptron discussed in Chapter 4 and is shown in Figure 5.16.
It is assumed that there are available vector training samples and that they have been augmented to give vectors $\mathbf{x}_i$ of size $n+1$ by 1. Considering the two pattern classes, $C_1$ and $C_2$, the discrete perceptron is desired that will yield an output of 1 if the pattern is from $C_1$ and an output of -1 if the pattern is from $C_2$. Let $d[\mathbf{x}_i]$, either + or -1, represent the desired output for training vector $\mathbf{x}_i$, then the discrete perceptron algorithm, given in Chapter 4, for the $p^{th}$ iteration, can be rewritten in the following form

$$w(p) \% 1 \rightarrow w(p) \% ^c \left(d[\mathbf{x}(p)] \otimes \text{sgn}(w^T(p) \mathbf{x}(p))\right) \mathbf{x}(p) \quad (5-11)$$

where $\mathbf{x}(p)$ is one of the training samples and $d[\mathbf{x}(p)]$ is the desired output for that $p^{th}$ training sample. Note that if the sample is incorrectly classified the term of (5-11) in parentheses (13) will be either +2 or -2 and thus the total correction scale factor is either +c or -c. The iterations are continued until one entire pass through the training data set does not cause a change in the weight vector. The procedure has been shown to converge in a finite number of trials only if the pattern classes are linearly separable.

5.6.2 The Delta training algorithm

If the signum function used as the nonlinearity is replaced with an arbitrary differentiable activation function $f(.)$, a basic neural element sometimes called the continuous perceptron element is specified and is shown in Fig. 5.17.

A common training algorithm for a continuous activation function is the delta training algorithm. It is based on the concept that the weight change at each iteration of the algorithm moves the new weight vector to minimize the squared distance between the actual and desired output.
Let \( x(p) \) be a training vector selected from the training set \( \{ x_1, x_2, \ldots, x_K \} \) and \( d[x(p)] \) be the corresponding desired output from the set \( \{ d[x_1], d[x_2], \ldots, d[x_K] \} \). Define the performance measure \( E_p \) to be the squared error at the \( p \)th iteration for the \( p \)th training sample \( x(p) \) as

\[
E_p = \frac{1}{2} \left( d[x(p)] \& f(w^T x(p)) \right)^2
\]  
(5-12)

Suppose we look at solving numerically for the \( w \) which gives the minimal value of \( E_p \) by using the method of steepest descent. The gradient algorithm changes or updates the weight vector \( w \) by the following where the subscript \( k \) represents the iteration number

\[
w_{k+1} = w_k - \eta \frac{\partial}{\partial w} E_p
\]  
(5-13)

The \( \eta \) is a prespecified constant and \( \frac{\partial}{\partial w} E_p \) is the gradient vector for \( E_p \) given by

\[
\frac{\partial}{\partial w} E_p = \left[ \frac{M_{E_p}}{M_{w_1}}, \frac{M_{E_p}}{M_{w_2}}, \ldots, \frac{M_{E_p}}{M_{w_n}} \right]^T
\]  
(5-14)

Each entry in the gradient can be obtained from (5-12) as follows

\[
\frac{M_{E_p}}{M_{w_j}} = \frac{1}{2} \left( d[x(p)] \& f(w^T x(p)) \right) \left( \frac{\partial f(x^T x(p))}{\partial w_j} \right) x_j(p)
\]  
(5-15)

where \( x_j(p) \) is the \( j \)th component of the training sample \( x(p) \). If the above partials are substituted into the gradient and if further the iteration number for the gradient algorithm is assumed to be \( p \), the iteration number for the training algorithm, Eq. (5-13) becomes

\[
\frac{M_{E_p}}{M_{w_j}} = \frac{1}{2} \left( d[x(p)] \& f(w^T x(p)) \right) \left( \frac{\partial f(x^T x(p))}{\partial w_j} \right) x_j(p)
\]  
(5-15)
This vector weight update equation is known as the **local delta algorithm** for training artificial neural elements with continuous activation functions \( f(\text{net}) \). Notice that this is a local optimization procedure as it uses only the current training sample, the current weight vector and the desired output for the current training sample to determine the new or next weight vector. Thus this form of the delta algorithm is a supervised training by sample procedure.

The **global delta algorithm** can be developed by defining a global performance measure, \( E_{\text{TOT}} \), as the sum of the individual errors squared for all training samples in the training set. Thus using the individual errors defined in Eq. (5.12), the total error becomes

\[
E_{\text{TOT}} = \frac{1}{2} \sum_{k=1}^{N_S} (d[x_k] \& f(w^T x_k))^2
\]

where \( x_k \) and \( d[x_k] \), \( k=1,2, \ldots, N_S \) are the known training vectors and desired outputs respectively. If the gradient algorithm is again used to perform a numerical solution for the minimization of \( E_{\text{TOT}} \), a global delta algorithm can be developed. The gradient of \( E_{\text{TOT}} \) now contains terms involving all samples and is given by

\[
L(E_{\text{TOT}}) = \frac{\partial E_{\text{TOT}}}{\partial w} = \sum_{k=1}^{N_S} (d[x_k] \& f(w^T x_k)) f'(w^T x_k) x_k
\]

Using the above expression for the gradient and the gradient algorithm (5-13), the vector weight update equation for the \( p \)th iteration can be shown to be

\[
w(p+1) = w(p) - \eta \left[ \sum_{k'=1}^{N_S} (d[x_k] \& f(w^T x_k)) f'(w^T x_k) x_k \right]
\]

Eq. (5-19) is called the **global weight update equation** for the global delta algorithm and it changes the current weight vector \( w(p) \) to the new weight vector \( w(p+1) \) by using all training vectors at each iteration and thus is a training by epoch method. Since it uses desired outputs it is a supervised training procedure. The algorithm is considered global since it uses all the training vectors at each iteration.

Eq. (5-19), the global delta algorithm, is an example of a **batch update**, while Eq. (5-16), the local delta algorithm, is an example of a **sequential update**. The batch update can be implemented by computing each term in the sum by using the same \( w(p) \) and accumulating the results before making the update.

### 5.6.3 Continuous Perceptron Training Algorithm

Assume that training samples are available from two pattern classes and that the pattern class is known for each of the training samples. The delta training rule is as follows
\[ w(p_\perp) \rightarrow w(p) \; \% \eta \{ \; d[x(p)] \; \& f(w^T(p)x(p)) \; \} \; f'(w^T(p)x(p)) \; x(p) \] (5-20)

On the left side of the equation is the new weight vector and it equals the sum of the old weight vector and a correction vector. The correction vector is seen to be the product of a training factor, \( \eta \), the difference between the output of the \( p \)th training sample and the desired output for that sample, the derivative of the activation function evaluated at the product of the \( p \)th training sample and the \( p \)th weight vector, and the current training pattern. Different activation functions will give different correction terms but the overall structure is just the addition of a scaled version of the training sample similar to the discrete perceptron. The continuous perceptron does not necessarily guarantee convergence but provides separation of classes most of the time if the patterns are linearly separable.

### 5.6.4 Specific Delta Training Algorithms

Although similar in form the delta training algorithms for specific nonlinearities are different. The training algorithms for the logistic, hyperbolic, and linear activation functions are now presented.

**Logistic activation function training algorithm**

The logistic activation function is unipolar with output as follows

\[ f(\text{net}) = \frac{1}{1 + \exp(\lambda \text{net})} \] (5-21)

Taking the derivative of this \( f(\text{net}) \) with respect to \( \text{net} \) yields the derivative \( f'(\text{net}) \) as

\[ f'(\text{net}) = \frac{\lambda \exp(\lambda \text{net})}{[1 + \exp(\lambda \text{net})]^2} = \frac{\lambda}{1 + \exp(\lambda \text{net})} \frac{\exp(\lambda \text{net})}{1 + \exp(\lambda \text{net})} = \frac{1}{1 + \exp(\lambda \text{net})} f(\text{net}) (1 - f(\text{net})) \] (5-22)

The last step above writes the derivative of \( f(\text{net}) \) in terms of the activation function given in (5-21). Substituting the derivative, (5-22), into (5-16), the training algorithm becomes

\[ w(p_\perp) \rightarrow w(p) \; \% \eta \{ \; d[x(p)] \; \& f(\text{net}) \; \} \; f(\text{net}) (1 - f(\text{net})) \; x(p) \] (5-23)

where \( \text{net} = w^T(p)x(p) \)

**Hyperbolic Tangent Activation Function Training Algorithm**

The hyperbolic tangent activation function is bipolar and its input-output relationship is written in the following two equivalent forms

\[ f(\text{net}) = \tanh(\lambda \text{net}/2) \]

\[ = \frac{2}{1 + \exp(\lambda \text{net})} \& 1 \] (5-24)
Taking the derivative of \( f(\text{net}) \) in (5-24) and substituting the results into the delta algorithm of (5-16) gives the following weight update algorithm for the hyperbolic activation function

\[
\mathbf{w}(p)^{\prime} = \mathbf{w}(p) - \eta \left( \frac{d[x(p)] & f(\text{net})}{(1 & f^2(\text{net}))} \right) \mathbf{x}(p)
\]

where \( \text{net}' = \mathbf{w}^T(p) \mathbf{x}(p) \) (5-25)

**Linear Activation Function training algorithm**

Assume that the activation function is a linear function of net and given by

\[
f(\text{net}) = \text{net}
\]

Using the linear activation function and the general delta algorithm Eq. (5-16) the weight update equation is easily seen to be

\[
\mathbf{w}(p)^{\prime} = \mathbf{w}(p) - \eta \left( \frac{d[x(p)] & f(\text{net})}{(1 & f^2(\text{net}))} \right) \mathbf{x}(p)
\]

where \( \text{net}' = \mathbf{w}^T(p) \mathbf{x}(p) \) (5-27)

This weight update equation (5-27) is the same as that used by Widrow and Hoff and so it is sometimes referred to as the **Widrow-Hoff training rule**. Convergence occurs only if the patterns are linearly separable.

### 5.6.5 Scale Factors in Training

Notice that the weight change depends upon a scale factor and the input to the node. Thus if either one of them is zero then there will be no change in the weight vector for that iteration. The scale factor used to multiply the input \( x_j \) for the \( \tanh(\lambda \text{net}/2) \) activation function is seen to be

\[
SF = \eta \left( \frac{d[x(p)] & f(\text{net})}{(1 & f^2(\text{net}))} \right)
\]

If the desired output \( d[x(p)] = 1 \) then the scale factor becomes

\[
SF_1 = \eta \left( \frac{d[x(p)] & f(\text{net})}{(1 & f^2(\text{net}))} \right)
\]

This SF is plotted in Figure 5.18 (a). Thus if the output \( f(\text{net}) \) is close to 1, the SF is small and close to zero. Notice however if the output \( f(\text{net}) \) is close to -1 , which is not what we desire for the output, then the scale factor is also close to zero. Thus for this activation function, even though the output is wrong, no correction will be made regardless of the input, certainly an undesirable property.
Figure 5-18  Scale factors for desired outputs of 1 and -1.

Similarly if the desired output is a -1, the SF-1 becomes

\[ SF_{-1} = \eta \left( \frac{\partial f}{\partial f} \right)(1 - \frac{1}{f^{2}(net)}) \]

and is seen plotted in Figure 5-18(b). For correct value of -1 the SF-1 is zero and close to zero for values close to -1 which is desirable. However if the output is wrong, \( f(\text{net})=1 \), the scale factor is zero and for output values close to zero the SF-1 is close to zero thus resulting in little or no correction to weight vector. The largest changes occur at the peaks of the scale factor which are at \( f(\text{net}) = 1/3 \) or \(-1/3\).

We thus see if the output of the neuron is +1 or -1, which means that net is in the saturation region of the activation function, no corrections to the weights occur even though the outputs are wrong. These lock up situations can bring the training of a neural element to a standstill.

Biasing the scale factors by selecting \( d[x(p)] \) to be 0.9 or -0.9 gives the scale factors as

\[
SF_{0.9} = \eta \left( 0.9 \frac{\partial f}{\partial f(\text{net})} \right)(1 - \frac{1}{f^{2}(\text{net})})
\]

\[
SF_{-0.9} = \eta \left( -0.9 \frac{\partial f}{\partial f(\text{net})} \right)(1 - \frac{1}{f^{2}(\text{net})})
\]

This choice or activation function does not cure the problem, it just moves the zero crossings of the curves as shown in Figure 5-19 for desired outputs as 0.9 and -0.9.

Figure 5-19  Scale factors for desired outputs of .9 and -.9.

5.7 MULTILAYER NEURAL NETWORKS
A neural network is defined as any connection of various basic neural elements. A common multilayer neural network structure is the feed-forward network, an example of which for three layers is shown in Figure 5-20.

![Figure 5-20 Feedforward multilayer neural network.](image)

The first layer of neural elements calculates their outputs $y_i^{(1)}$ and feeds these values forward to the second layer of neural elements that calculate their outputs $y_i^{(2)}$ which are fed forward to the next layer of neural elements, etc. The network shown is composed of three layers with three, four, and two basic neural elements per layer respectively. Implementation in hardware and software can be done in many different ways. The most straightforward method would be parallel calculations to get the outputs of each layer and sequential batch transmission between layers.

In a feedforward neural network no basic neural element's output serves as the input to any node in its layer or any node in any previous layers. If any of the outputs are connected to current layer or previous layers it is called a feedback neural network. Examples of feedback neural networks including the Hopfield Net are shown in Figure 5-21. The analysis and synthesis of such systems will be presented in Chapter 7.
Figure 5-21. Examples of feedback single and two layer neural networks.

5.7.1 Feedforward structures for pattern recognition

One use of feedforward neural networks is for pattern recognition and three basic structures predominate for the multiclass problem. The two main structures are single node output and multi-nodal output while another less common structure is the multi-nodal coded output. Examples of each type are shown in Figure 5.22 for discrimination between four pattern classes.

Figure 5-22. Structures of feedforward neural networks for pattern recognition: (a) Single node output, (b) Multi-nodal output, (c) Coded multi-nodal output.

**Single node output.** The single node output has a single neuron as the output layer of the neural network and performs classification by partitioning the range of the single output node into regions corresponding to each pattern class. For example in a four class problem if y represents the single-node output of the feedforward neural network with range -1 to +1, a commonly used partitioning is as follows.

\[
\begin{align*}
&\text{if } \&1 \# y \# 5 \text{ then decide class } C_1 \\
&\&5 < y \# 0 \quad \text{then decide class } C_2 \\
&0 < y \# .5 \quad \text{then decide class } C_3 \\
&.5 < y \# 1 \quad \text{then decide class } C_4
\end{align*}
\]  

\( (5-32) \)
It should be made clear that the regions do not necessarily have to be equal in size or for that manner simply connected. Normally simply connected and equally spaced regions are selected for convenience.

**Multi-node output** The multinode output uses an output node for each of the pattern classes. For the four pattern case with identical neural elements in the output layer. If \( y_1, y_2, y_3, \) and \( y_4 \) are the outputs from the last layer, one way of implementing the decision rule is to decide \( C_j \) if \( y_j \) is greater than all other \( y_k \). Resolve ties by random selection among the tied classes. This structure is similar to that shown in Chapter 2, Figure 2-12 which is a maximum likelihood classifier and the multiclass perceptron algorithm described in Chapter 4.

**Multi-node coded output** In the multi-node coded output the number of output nodes is selected as the number of bits required to write the number of classes minus 1 in binary. Thus for the 4 class we require two output nodes. If each of the output nodes has a range from -1 to +1 a classification rule could be expressed as follows

\[
\begin{align*}
& \text{if } \&d \ # y_1 \ # 0 \ \text{AND} \ \&d \ # y_2 \ # 0 \ \text{then decide class } C_0 \\
& \&d \ < y_1 \ # 0 \ \text{AND} \ 0 < y_2 \ # 1 \ \text{then decide class } C_1 \\
& 0 < y_1 \ # 1 \ \text{AND} \ &d \ # y_2 \ # 0 \ \text{then decide class } C_2 \\
& 0 < y_1 \ # 1 \ \text{AND} \ 0 < y_2 \ # 1 \ \text{then decide class } C_3
\end{align*}
\] (5-33)

Thus the classification rule assigns the class number corresponding to the binary representation of the output variables.

**5.7.2 Analysis of multilayer neural networks**

In most cases the analysis of a neural network borders on impossibility, however, certain neural networks with discrete neural elements in simple two and three layer structures can be analyzed explicitly. By analysis it is meant that the output is calculated for the range of all possible input vectors or patterns. As a result for some problems it becomes possible to identify the decision regions in the pattern space. When the pattern space is higher than three dimensions such a description is almost impossible to visualize, yet the decision regions still may be able to be described in terms of hyperplanes in these higher dimensional spaces.

The following example illustrates the analysis of a certain type of neural network that is constructed of two layers, the first layer specifying hyperplanes in the pattern space that we could associate with features and the second layer being a logical "AND" operation.

**EXAMPLE 5.2**

You are asked to analyze the neural network shown in Figure 5.23.
Figure 5-23. Neural network to analyze in Example 5.2.

Solution:

Analysis of this network consists of calculating the output $y_1^{(i)}$ for all values of the input pattern space. One way of doing this is simple to substitute in a grid of possible input vectors and determine the output, in other words an exhaustive procedure. Because of the simple structure of this problem it is possible to determine the output by analyzing each neural element and combining the results. The outputs of the neural elements in the first layer can be written as follows

\[
y_1^{(1)} = \begin{cases} 1 & \text{if } & \& x_1 \% 1 > 0 \\ & \text{&} & < 0 \end{cases}
\]

\[
y_2^{(1)} = \begin{cases} 1 & \text{if } & x_1 \% x_2 > 0 \\ & \text{&} & < 0 \end{cases}
\]

\[
y_3^{(1)} = \begin{cases} 1 & \text{if } & \& x_2 \% 1 > 0 \\ & \text{&} & < 0 \end{cases}
\]

The output of the neural element in the second layer is easily seen to perform the AND operation and as it is similar to the McCollough Pitts model for logical AND and given by

\[
y_1^{(2)} = \begin{cases} 1 & \text{if } & \text{net}_1^{(2)} + y_1^{(1)} \% y_2^{(1)} \% y_3^{(1)} > 0 \\ & \text{&} & < 2 \end{cases}
\]

If $y_1^{(1)}$, $y_2^{(1)}$, and $y_3^{(1)}$ are all one then the output $y_1^{(2)}$ will be 1, otherwise it will be -1 since the sum is less than the threshold of the output neuron. If the outputs are all 1, then the following inequalities in the original pattern space are determined.
Thus the boundaries are hyperplanes. The values of $\text{net}_i^{(2)}$ in different areas of the pattern space $[x_1, x_2]$ are shown in Fig. 5-24(a) while the output $y_i^{(2)}$ for all input vectors $[x_1, x_2]^T$ is shown in Figure 5-24(b).

5.7.3 Synthesis of multilayer feedforward neural networks

Certain multilayer feedforward neural networks can be synthesized (constructed) from a set of regions specified by hyperplane boundaries by using a three layer neural network with unit step activation functions. The first layer specifies the hyperplane boundaries, the second acts like a logical AND to determine the regions, and finally the third layer, acts like a logical OR, to combine multiple regions for a particular class. Multi-input AND and OR neural elements are shown in Figure 5-25 using a unit step activation function.
The following example illustrates a procedure for synthesis of a pattern recognizer whose classification regions in the pattern space are specified with hyperplanes by using the hyperplane neural element together with the neural equivalent of the logical AND and OR operations.

**EXAMPLE 5.3**

The decision regions for a pattern classifier are shown in Figure 5-26. You are asked to determine (synthesize) a neural network that will implement the classifier using basic neural elements with unit step activation functions.

**Solution:**
Notice that the region for pattern class 1 is not simply connected. Each portion of the total region for class 1 will contribute a neural element or node in the second layer of the network while each of the hyperplane boundaries contributes a neural element to the first layer. The third and final layer is a single neural element that performs a logical OR operation. The weights for the first node in the first layer are -1,-1,3 which come from

\[ g_1(x) = -x_1 - x_2 + 3 \]

This line or in general a hyperplane has a positive side including the shaded area shown. Each of the other hyperplanes shown in Figure 5-26 contributes a node in the first layer as shown in Figure 5-27 with weights generated by its \( g_k(x) \).

5.8 TRAINING A MULTILAYER FEEDFORWARD NEURAL NETWORK

In general it is difficult to synthesize a neural network to solve a pattern recognition problem. First the size of the pattern space makes it impossible to set up the regions necessary to proceed using the synthesis procedure. Second even if the regions can be found they may not have hyperplane boundaries and approximation of those boundaries with hyperplanes may expand the number of nodes to an unacceptable number. The procedure shown in Example 5.3 uses only signum or step function activation functions thus limiting the flexibility of the network attributed to the continuous nonlinear activation functions.
Another severe limitation of the synthesis procedure is that it requires the decision regions to be known and for most problems this is not the case as we may have only training samples available. Training procedures for multilayer neural networks with step and signum function nonlinearities are not available thus general procedures for synthesis can not be formulated. Extension of the procedures used in section 5.6 to neural networks with continuous monotonic activation function is blocked by the continuous nature of the element output the very property that gives them extra flexibility.

The purpose of this section is to present a procedure for training multilayer feedforward neural networks whose basic neural elements have continuous monotonic activation functions. Its origin will be the extension of the delta training algorithm and thus is sometimes referred to as the generalized delta algorithm. As in the delta algorithm the procedure will be derived by using a numerical minimization procedure to minimize a squared error performance index.

The structure of the feedforward neural network to be trained is shown in Figure 5-28 and great care in notation is required to describe the update equations. A subscripted and superscripted notation for the output of each node is used where the superscript identifies the layer and the subscript indicates the node number in the layer. Thus \( y^{(3)}_2 \) would indicate the output of node 2 in layer 3.

![Figure 5-28](image)

**Figure 5-28.** Notational representation used for the training of a feedforward neural network.

The weights at each neuron and the outputs of each layer can be expressed in vector notation as follows
With the superscript \((L)\), a double subscript \(i,j\) is used to identify the weight connecting the \(j\)th input to the \(i\)th node of layer \((L)\). The standard backpropagation algorithm for sequential update is derived by using the gradient algorithm to locally minimize the performance measure \(E_p\) given by

\[
E_p = \frac{1}{2} \sum_{m=1}^{N_L} \left[ d_m(x(p)) \odot y^{(L)}_m \right]^2
\]

where \(y^{(L)}_m\) is the output of the \(m\)th node in layer \(L\)

\(N_L\) equals the number of nodes in layer \(L\), and

\(d_m(x(p))\) is the desired output at the \(m\)th node for the \(p\)th training pattern vector \(x(p)\).

The sum of the squares over all \(N_L\) output nodes gives the Euclidean distance squared between the desired or target vector output and the actual output using the networks present weight matrices and the current input training pattern vector \(x(p)\). The gradient algorithm can be determined by calculating the partial of \(E_p\) with respect to all components of the weight matrices. As the formulae are seen to depend upon the layer we will first find the partial derivatives with respect to the weights in the last or \(L\)th layer and follow that with derivations for the next to last and each preceding layer.

**5.8.1 Rule #1 (Last layer \(L\))**

The partial of \(E_p\) with respect to \(w^{(L)}_{kj}\) is given by

\[
\frac{\partial E_p}{\partial w^{(L)}_{kj}} = \sum_{m=1}^{N_L} d_m(x(p)) \odot y^{(L)}_m \odot \left[ \frac{\partial y^{(L)}_m}{\partial w^{(L)}_{kj}} \right]
\]

where \(y^{(L)}_m = \sum_{i=1}^{N_m} \left( w^{(L)}_{mi} \odot y^{(L)}_i \right)\)

The only term in the sum that contains \(w_{kj}\) is \(w^{(L)}_k\), the weight vector of output node \(k\), thus when taking the partial all other terms are zero and the partial for weights in layer \(L\), node \(k\) becomes

\[
\frac{\partial E_p}{\partial w^{(L)}_k} = \sum_{m=1}^{N_L} d_m(x(p)) \odot y^{(L)}_m \odot \left( \sum_{i=1}^{N_m} w^{(L)}_{mi} \right)
\]

The sum of the squares over all \(N_L\) output nodes gives the Euclidean distance squared between the desired or target vector output and the actual output using the networks present weight matrices and the current input training pattern vector \(x(p)\). The gradient algorithm can be determined by calculating the partial of \(E_p\) with respect to all components of the weight matrices. As the formulae are seen to depend upon the layer we will first find the partial derivatives with respect to the weights in the last or \(L\)th layer and follow that with derivations for the next to last and each preceding layer.

**5.8.1 Rule #1 (Last layer \(L\))**

The partial of \(E_p\) with respect to \(w^{(L)}_{kj}\) is given by

\[
\frac{\partial E_p}{\partial w^{(L)}_{kj}} = \sum_{m=1}^{N_L} d_m(x(p)) \odot y^{(L)}_m \odot \left[ \frac{\partial y^{(L)}_m}{\partial w^{(L)}_{kj}} \right]
\]

where \(y^{(L)}_m = \sum_{i=1}^{N_m} \left( w^{(L)}_{mi} \odot y^{(L)}_i \right)\)

The only term in the sum that contains \(w_{kj}\) is \(w^{(L)}_k\), the weight vector of output node \(k\), thus when taking the partial all other terms are zero and the partial for weights in layer \(L\), node \(k\) becomes
Using the gradient algorithm (5-13) and the partials above for each k and j, the training rule, called **Rule #1**, for the training of the weights in the output layer becomes

\[
\frac{\Delta w_{kj}^{(L)}}{\Delta w_{kj}^{(L)}} (p) = \frac{w_{kj}^{(L)}}{w_{kj}^{(L)}} (p) \% \Delta w_{kj}^{(L)} (p)
\]

\[
\Delta w_{kj}^{(L)} (p) = \eta \left[ \frac{d_{k}(x(p)) \& y_{k}^{(L)}}{d_{k}(x(p)) \& y_{k}^{(L)}} \right] \frac{d f_{k}^{(L)}(.)}{d(.)} \left[ w_{k}^{(L)} y_{k}^{(L)} \right]_{y_{k}^{(L)} \& y_{k}^{(L)}}^{(L,\delta)}
\]

(5-40)

where \( k = 1, 2, ..., N_L \) and \( j = 1, 2, ..., N_{L \delta} \).

### 5.8.2 Rule #2 (layer L-1)

For weights in the layer (L-1), the partial derivatives of \( E_p \) with respect to the (L-1) layer weights is given by

\[
\frac{\Delta w_{m}^{(L)}}{\Delta w_{m}^{(L)}} (p) = \left[ \frac{d_{m}(x(p)) \& y_{m}^{(L)}}{d_{m}(x(p)) \& y_{m}^{(L)}} \right] \frac{d f_{m}^{(L)}(.)}{d(.)} \left[ w_{m}^{(L)} y_{m}^{(L)} \right]_{y_{m}^{(L)} \& y_{m}^{(L)}}^{(L,\delta)}
\]

(5-41)

where \( y_{m}^{(L)} = f_{m}^{(L)} \left( w_{m}^{(L)} y_{m}^{(L)} \right) \).

To evaluate the partials in the sum, the chain rule can be applied and since every output node in the Lth layer is function of those weights the partials will occur in each term of the sum. The partials for the jth node and the ith input, as shown in Figure 5-28, are

\[
\frac{\Delta w_{m}^{(L)}}{\Delta w_{m}^{(L)}} \left[ \frac{d_{m}(x(p)) \& y_{m}^{(L)}}{d_{m}(x(p)) \& y_{m}^{(L)}} \right] \frac{d f_{m}^{(L)}(.)}{d(.)} \left[ w_{m}^{(L)} y_{m}^{(L)} \right]_{y_{m}^{(L)} \& y_{m}^{(L)}}^{(L,\delta)}
\]

(5-42)

The second step above is because \( w_{m}^{(L)} \) is not a function of \( w_{ji}^{(L-1)} \) while \( y^{(L-1)} \) is. But \( y^{(L-1)} \) has \( w_{ji}^{(L-1)} \) only in the jth component. The partial in eq. (5-36) becomes

\[
\frac{\Delta w_{m}^{(L)}}{\Delta w_{m}^{(L)}} \left[ \frac{d_{m}(x(p)) \& y_{m}^{(L)}}{d_{m}(x(p)) \& y_{m}^{(L)}} \right] \frac{d f_{m}^{(L)}(.)}{d(.)} \left[ w_{m}^{(L)} y_{m}^{(L)} \right]_{y_{m}^{(L)} \& y_{m}^{(L)}}^{(L,\delta)}
\]

(5-43)
Using the gradient algorithm (5-13) and combining (5-40), (5-41) and (5-42), the following weight update equation for weights in layer L-1, called **Rule #2**, can be derived.

\[
w_{ji}^{(L\&d)}(p) % \eta \frac{\partial E}{\partial w_{ji}^{(L\&d)}(p)} \quad \text{where}
\]

\[
v_{ji}^{(L\&d)}(p) = \sum_{m=1}^{N_L} \left[ d_m(x(p)) \Delta y_m^{(L)} \frac{d f_m^{(L)}(\cdot)}{d(\cdot)} \right] \\
\frac{\partial E}{\partial w_{ji}^{(L\&d)}(p)} \frac{d f_j^{(L\&d)}(\cdot)}{d(\cdot)} \frac{d f_j^{(L\&d)}(\cdot)}{d(\cdot)} \\
\frac{d f_j^{(L\&d)}(\cdot)}{d(\cdot)} \frac{d f_j^{(L\&d)}(\cdot)}{d(\cdot)} \\
\frac{d f_j^{(L\&d)}(\cdot)}{d(\cdot)} \frac{d f_j^{(L\&d)}(\cdot)}{d(\cdot)} \\
\frac{d f_j^{(L\&d)}(\cdot)}{d(\cdot)} \frac{d f_j^{(L\&d)}(\cdot)}{d(\cdot)}
\] (5-44)

Rules #1 and #2 for the logistic and tanh nonlinearities are now presented using the weight update equations (5-39) and (5-43) which constitute the backbone of the backpropagation algorithm.

**5.8.3 Specific Rules for Various Nonlinearities**

Using the above rules #1 and #2 the weight update equations can be found for the logistic and hyperbolic activation functions. Note that the \( \lambda \) will be incorporated into the \( \eta \) in all formulas that follow in this section.

**Logistic activation function**

The logistic activation function and its derivative are given by

\[
f(\text{net}) = \frac{1}{1 + \exp(\text{\( \lambda \) \text{net}})}
\]

\[
\frac{d f(\text{net})}{d \text{net}} = \lambda \text{net}(1 - \text{net})
\] (5-45)

Using this function and its derivative the following rules are easily established from Eq. (5-35) and (5-37) for the logistic activation function.

**Rule #1 for logistic activation function**

\[
w_{kj}^{(L)}(p) % \eta \left[ d_k(x(p)) \Delta y_k^{(L)} \right] \frac{1}{1 + \exp(\text{\( \lambda \) \text{net}})}
\]

\[
\text{where} \quad \delta_k^{(L)} = y_k^{(L)} (1 - y_k^{(L)})
\] (5-46)

for \( k = 1, 2, \ldots, N_L \quad j = 1, 2, \ldots, N_{L\&d} \)

Notice that the weight update equations for the output layer weights are the same as the delta algorithm for a single neuron. This is not true in the preceding layers as the desired output of each of the preceding layer nodes is not known.

**Rule #2 for logistic activation function**
\[ v_{ji}^{(L\&d)}(p \&d) \cdot w_{ji}^{(L\&d)}(p) \% \eta \delta_j^{(L\&d)} \left( \sum_{m=1}^{N_L} w_{mj}^{(L)}(p) \left[d_m(x(p)) \& y_m^{(L)} \right] \delta_m^{(L)} \right) y_i^{(L\&d)} \]

where \( \delta_j^{(L\&d)} \cdot y_j^{(L\&d)}(1 \& y_j^{(L\&d)}) \) and \( \delta_m^{(L)} \cdot y_m^{(L)}(1 \& y_m^{(L)}) \)
for \( k' = 1, 2, ..., N_L \quad j' = 1, 2, ..., N_{L\&d} \)

**Hyperbolic activation function**

The hyperbolic activation function and its derivative are given by

\[
\begin{align*}
    f(\text{net}) &= \tanh(\lambda \text{net}/2) \\
    \frac{df(\text{net})}{d\text{net}} &= \frac{\lambda}{2}(1 \& f^2(\text{net}))
\end{align*}
\]

With \( y_i^0 = x_i \) (the input pattern vector), the weight update rules #1 and #2 become

**Rule #1 for hyperbolic tangent activation function**

\[
\begin{align*}
    w_{kj}^{(L)}(p \&d) \cdot w_{kj}^{(L)}(p) \% \eta [d_k(x(p)) \& y_k^{(L)}] \delta_k^{(L)} y_j^{(L\&d)} \\
    \text{where} \quad \delta_k^{(L)} = 1 \& (y_k^{(L)})^2
\end{align*}
\]
for \( k' = 1, 2, ..., N_L \quad j' = 1, 2, ..., N_{L\&d} \)

**Rule #2 for hyperbolic tangent activation function**

\[
\begin{align*}
    v_{ji}^{(L\&d)}(p \&d) \cdot w_{ji}^{(L\&d)}(p) \% \eta \delta_j^{(L\&d)} \left( \sum_{m=1}^{N_L} w_{mj}^{(L)}(p) \left[d_m(x(p)) \& y_m^{(L)} \right] \delta_m^{(L)} \right) y_i^{(L\&d)} \\
    \text{where} \quad \delta_j^{(L\&d)} = 1 \& (y_j^{(L\&d)})^2 \quad \text{and} \quad \delta_m^{(L)} = 1 \& (y_m^{(L)})^2
\end{align*}
\]
for \( k' = 1, 2, ..., N_L \quad j' = 1, 2, ..., N_{L\&d} \)

**5.9 BACKPROPAGATION TRAINING ALGORITHMS**

The backpropagation training algorithms provide a systematic way of adjusting the weights to minimize \( E_p \), the sample error squared. The algorithms adjust the weights according to two different rules, one for the weights in the last or output layer and one for the other layers sometimes called hidden layers, to emphasize the fact that the required output of these layers are not known to us. A conceptual diagram for the neural network to be trained is shown in Figure 5-29.
In the figure three layers of neural elements are shown with weight matrices $W^{(L)}$, $W^{(L-1)}$, $W^{(L-2)}$, etc. These matrices are composed of columns of the weight vectors from each node in that layer as shown below:

$$W^{(m)} = \begin{bmatrix}
    w_1^{(m)} & w_2^{(m)} & \ldots & w_N^{(m)}
\end{bmatrix}
$$

The input $x(p)$ represents the $p^{th}$ training sample taken from a set of training samples with known classification. The output vector $y^{(L)}$ is the result from the input of the training sample $x(p)$ and $d(X(p))$ is the desired output vector for that $x(p)$. The desired output and actual output vectors serve as inputs to Rule #1 which will change the weight matrix $W^{(L)}$. Parts of the calculations for Rule #1 together with the output vectors at layers 2 and 3, and the desired output at layer 3 are used to change the weight matrix $W^{(L-1)}$ by using rule #2. Similarly rule #2 changes the weight matrix $W^{(L-2)}$ as a result of the desired output of the last layer and actual outputs and inputs of the first layer along with calculations from the previous rule applications. The general approach to the training is one of calculating outputs of the respective layers in a forward direction then calculating weight corrections for each layer in a backward direction followed by a block weight update for all layers, and repeating this process with other training samples until a defined performance measure is satisfied. The steps in the backpropagation procedure are now presented.

### 5.9.1 Backpropagation Algorithm

Assume that we have available a set of training samples

$$\{ x(p) : p = 1, 2, \ldots, N_s \}$$
and a set of corresponding target or desired vectors as

\[ \{ \mathbf{d}(x(p)) : p = 1, 2, \ldots, N_s \} \]

Assume also that a neural net structure is specified by number of layers and number of nodes for each layer. For purpose of illustration a three layer net will be assumed similar to Fig. 5-28. Also assume that the nonlinearities (activation functions) are specified for each node and that a sequential or training by sample procedure is to be used. The backpropagation algorithm is described by the following steps and the flow diagram is shown in Figure 5.30.

**Backpropagation Training Algorithm**

**(step 0) Initialization**

After a neural net structure with a certain type of activation functions and a specific number of nodes and layers has been specified the components of the initial weight matrices \( W^{(1)}(1), W^{(2)}(1), \) and \( W^{(3)}(1) \) are selected as random samples from a known probability distribution. Usually the samples are from a uniform density with a small range of possible outputs, perhaps from \([-1, 1]\). Select training parameters, \( \eta \), momentum factor, \( m \), error tolerance, \( e \), and maximum number of iterations, \( \text{MAX} \). Set \( p = 1 \) to be the iteration number and go to step 1.

**(Step 1) Select one of the training samples**

**Figure 5-30.** Flow diagram for the backpropagation algorithm for training feedforward neural networks.
Select one of the training samples \( \mathbf{x}(p) \) either by some specified order of the training samples or by some random selection procedure. Go to Step 2.

**Step 2** Forward calculation of layer output vectors
Using the existing weight matrices \( W^{(1)}(p) \), \( W^{(2)}(p) \), and \( W^{(3)}(p) \) and the \( p \)-th training sample \( \mathbf{x}(p) \), calculate in turn the vector outputs of each layer \( y^{(1)}(p) \), \( y^{(2)}(p) \), and \( y^{(3)}(p) \). Go to step 3.

**Step 3** Calculation of performance
The overall performance in terms of the sum of the Euclidean squared distances between desired vectors and actual outputs over all training samples, is a good measure of performance; however, it places considerable computational burden on the procedure. To reduce this computational expense the errors are summed over one pass through the data with errors for each sample determined from previous weights thus providing only an approximation of the total performance error. When the error gets small this performance measure gets close to the more accurate determination with the same weight matrices. Go to step 4.

**Step 4** Branch on error tolerance
If the total error is greater than the tolerance, tol, then go to step 5. If the error tolerance is satisfied a solution has been obtained and the current weight matrices are fixed and thus the design is specified. The design would then be ready to test on data that was not in the training set to see how good the network extrapolates.

**Step 5** Branch on number of iterations
If the current number of iterations is greater than the maximum number of iterations specified then quit. The weight matrices could be frozen but with the understanding that the error tolerance is not satisfied with these values. The algorithm is said to have not converged. This does not mean necessarily that a design with the given structure does not exist, it just means that you didn't find one and have chosen to quit. These weight matrices could be further used as initial conditions for beginning the procedure again or be a vision for changing the original assumed structure or training parameters.

If maximum number of iterations has not been reached then go to step 6.

**Step 6** Calculation of weight change matrices
Using the output vectors from step 2 and the desired output for the \( p \)-th training sample \( \mathbf{x}(p) \), calculate the changes in the output layer by using the Rule#1 for the particular type of activation functions assumed. Then the changes in the next to last or second layer are computed using the proper Rule #2 and finally the weight changes for the first layer are determined using an extended version of rule #2. Then go to step 7.

**Step 7** Updating the weight matrices
Add the weight changes calculated in step 3 to the weight matrices to obtain the new weight matrices \( W^{(1)}(p+1) \), \( W^{(2)}(p+1) \), and \( W^{(3)}(p+1) \). If training by epoch is used the weight changes will be accumulated through one entire pass through the training samples before being updated. Then \( p \) is increased by one and we return to step 1.
5.9.2 Backpropagation with Momentum Algorithm

Research has shown that a modification to the backpropagation algorithm called **backpropagation with momentum**, if used properly, can significantly speed up the training process and reduce the probability of locking up on a local minimum. All steps of the backpropagation algorithm remain the same except that the weight update equations have an additional correction term which is a weighted version (by \( \alpha \)) of the previous delta weight update as given by

\[
\text{Regular Correction} \quad \text{Momentum Term}
\]

\[
\begin{align*}
\text{Rule } #1: \quad & w_{kj}^{(L)}(p) \%\Delta w_{kj}^{(L)}(p) \%\alpha \Delta w_{kj}^{(L)}(p \& d) \\
\text{Rule } #2: \quad & w_{ji}^{(L \& d)}(p) \%\Delta w_{ji}^{(L \& d)}(p) \%\alpha \Delta w_{ji}^{(L \& d)}(p \& d)
\end{align*}
\]

(5-51)

Normally \( \alpha \) is selected to be between 0 and 1. This additional correction or momentum term gives added "momentum" in the direction the weights are changing in the hopes that getting trapped in local minimums can be avoided. Too large of a value of \( \alpha \), close to 1, can have an adverse affect as weight fluctuations become large and saturation or oscillations can occur. Two small of a values for \( \alpha \), close to 0, offer little correction and thus no performance gain or resistance to locking up in local minimums. The mechanical analog is that of pushing a marble down an inclined curved surface representing the performance measure, with small valleys and hills as shown in Figure 5.31. If the marble is pushed just hard enough there will be enough momentum to roll out of the local minimum represented by the small valleys and not roll out of the valley representing the global minimum.

![Figure 5-31](image)

**Figure 5-31** Illustration of mechanical analog for the momentum correction to the backpropagation algorithm.

5.10 BACKPROPAGATION ALGORITHM PARAMETER SELECTION

The success of applying the backpropagation algorithm depends on the selection of many parameters for the neural network including number of layers, number of nodes for each layer, training constant, initial conditions, types of nonlinearities, target values, training by epoch or sample, etc. Precise choices for these parameters and properties can not be given.
as they are highly problem dependent; however, guidelines for their selection can be given and are now presented.

5.10.1 Selection of Initial Conditions for Weights

The initialization of the weights can be accomplished in several different ways and it is well known from experience that the number of iterations to a solution can be reduced by proper selection. With no outside or additional information as to what the initial conditions should be the weights are usually selected at random from a uniform probability density function on the interval -1 to 1.

Ng and Widrow [?] have shown that the average number of iterations for convergence can be reduced by selecting the weights at random from uniform distributions whose intervals depend upon the number of inputs to the neural element (fan in) defined by \( N_{\text{fan in}} \) for the node or neural element under consideration in the following way.

\[
&\frac{2.4}{N_{\text{fan in}}} < w_{ij} < \frac{2.4}{N_{\text{fan in}}}
\]

Thus the uniform intervals can be of different size for nodes in different locations throughout the network.

Other research has indicated that marginal improvement in convergence rate can be obtained by using more sophisticated and complicated selection of the initial conditions[?].

5.10.2 Selection of Training Patterns

The main criterion for selection of the training patterns is to choose patterns that are representative of the classes of samples being considered. In a way if you knew what properties of the samples make the patterns representative the problem of pattern recognition by any technique would be significantly simplified. In general these properties are unknown and the selection of training patterns becomes more subjective and usually modified by past experience and practical considerations. For example if some of the patterns in a two dimensional space were straight lines and some were quadratics, leaving all samples from either the quadratic or the linear group out of the training set would most likely result in poor performance.

5.10.3 Number of Training Samples Required

In general the number of training samples depends on the complexity of the patterns and the number of classes. The only basic rule is that enough training samples must be selected to be representative of each of the classes. This is a very vague rule; however, we could not expect to get a reasonable design if a subset of one of the pattern classes was not represented in the training set. Choosing too many training samples may lead to an unnecessarily complicated solution, while selection of too few could lead to poor generalization and performance.

5.10.4 Sample Ordering for Training

When using the training by sample procedure it is usually good practice to select the training samples in random order during each pass through the data samples. Ordering in this fashion will usually reduce the number of iterations required for convergence. If random selection is used for the first pass and the same order for the remaining passes reduction of
the number of iterations over no random ordering has also been observed. Selection of orders that use all samples from one class, then all samples from the next class, etc, have been shown on the average to increase the number of iterations required for convergence over the number for random order selection.

When training by epoch, the order of presentation of samples will not affect the convergence rate since the weight update equation uses the sum of the corrections from all training vectors and thus is insensitive to the order of presentation.

5.10.5 Normalization of Training Samples

There is a connection between the parameters of the nonlinear activation function, the training parameter and the amplitudes and number of inputs to the nodes. If the combination of these parameters are such that the operation of the nonlinearities is in their saturation regions the ability of the net to learn is significantly reduced and in many cases produces lock up situations. A lockup occurs when the weights connected to a node cease to be able to be changed because its inputs are such that their weighted sum is in the saturation region of the nonlinearity. It is therefore convenient to normalize the input pattern vectors and adjust the nonlinearity parameters and training factors accordingly to avoid probable lock up. In normalizing the samples all components of the pattern vectors must be scaled by the same number and usually scaling is performed such that the new coordinate values, except for a few outliers, are bounded between -1 and 1. If training samples are scaled or normalized during the training phase, the non training samples must also be normalized or scaled by the same factor before being processed and classified. If training samples are not normalized, the training parameters, especially $\eta$, must be reduced if a significant number of the training vector components are much greater than one.

5.10.6 Selection of Training Procedure

In training by epoch the sum of the weight corrections for each training sample will move the weight vector in the direction to minimize the global error, while training by sample, a weight correction at each presentation, moves the weight vectors in the direction to minimize the local error. The training by sample procedure has been shown to be more efficient than training by epoch in terms of rate of convergence in that it usually produces a solution in a fewer number of passes through the data. This improvement is probably due to the fact that training by sample gives you more than just one weight change during each pass through the data.

5.10.7 Choice of Training Parameter

The training parameter $\eta$ has a direct effect on the convergence rate and selection is highly problem and fan independent. Increasing this parameter from very small values will usually reduce the average number of iterations required for satisfying a specified tolerance or performance; however, if increased to much it will increase the number of iterations for convergence and if too large will cause saturation and lock up situations. With normalized training parameters and small pattern vector dimension ball park values are between .001 to 5 vector with default values between .1 and .9.

As the number of inputs to the nodes in each layer could be different, improved convergence can be obtained by having different training parameters for each layer, however in most applications a global selection proves to be satisfactory.
5.10.8 Choice of Node Nonlinearities
In section 5.3 many different types of activation functions for the neural element's nonlinearity were presented. The outputs for unipolar and bipolar activation functions were usually between zero and one and minus one to plus one respectively. The choice of nonlinearity has been shown to have only secondary affects on the convergence rate for the case of monotonic activation functions, with a slight advantage to the bipolar choice.
Choice of nonmonotonic activation functions can result in designs with fewer number of nodes and a reduction in average number of iterations required for convergence. Considerable success has been obtained using radial basis functions which are discussed later in this chapter.

5.10.9 Choice of Momentum Factor
The momentum factor $\alpha$ is a positive value between 0 and 1. If the value is selected too large the network will force the net input to the nonlinearity to be in the saturation region and cause lock up thus reducing its capability to converge to a meaningful solution. Although a matter of trial and error the selection of $\alpha = 0.1$ is a good starting point smaller for large fan in and slightly higher for small fan in.

5.10.10 Number of Layers and Nodes
The number of nodes and layers to use is critical to a successful design. Too large of a number of layers and nodes slows convergence and gives a design that is costly to implement while to small of a number may result in a network incapable of solving the problem at hand with the given performance level. The number of nodes should in a sense match the complexity of the problem being solved. Since the complexity is a poorly defined property a clear cut specification is not available. At best, the selection of the number of nodes and layers is a trial and error process.
Over the past few years researchers have looked at the problem from a function approximation stand point. Kolmogorov[?] gave a result that can serve as providing an upper bound to the problem. Roughly speaking his result states that if the pattern vector has N variables that any function can be approximated by three layers with a total of $N(2N+1)$ nodes using a monotonic increasing activation function.
Another rule of thumb that has been used by ????[ ] usually also gives too large of a number of nodes. This rule states that if d equals the dimension of the pattern vectors and c is the number of classes to be distinguished then the number of layers to use is three with d nodes in the first layer, 2d + 1 nodes in the second layer and c nodes in the third or output layer. Again this should be interpreted as a rough guide and with actual applications this usually gives a much more complicated neural structure than is necessary.
Recent research has presented some surprising results expressed in terms of existence theorems for approximation of functions. A sample of this type of result is the universal approximation theorem given by Cybenko[?] and results by Sobajic[?]

5.10.11 Selection of Error Tolerance
Two types of error have been presented for describing the performance of the resulting trained network. The first was the per training sample error $E_p$ given by
where the sum is over the components of the desired or target vector for the current training sample. The second was the total error over all training samples given by $E_{TOT}$ expressed as

$$E_{TOT} = \sum_{p} E_p$$

(5-54)

where $\mathcal{M}$ is the set of all training patterns. Thus two different tolerances for the design parameters can be specified: a total error and maximum individual error. Values of $E_{TOT}$ too small can cause loss in generality sometimes called memorization or "grandmothering" while values too large may not allow the distinguishing of the classes. Values for $E_{TOT}$ should usually be somewhere in the range from 0.000001 to 0.1 with nominal values around .001. Max individual errors could be selected approximately equal to $E_{TOT}$ divided by the number of training samples. Satisfying the individual errors with these values will automatically satisfy the total error requirement.

5.10.12 Choice of Maximum Number of Iterations

The maximum number of iterations, MAXIT, before the training is stopped, which results in no design that satisfies the performance, is largely a resources problem. Basically it gives the amount of time you are willing to wait to satisfy the performance values given. As it is not unusual to need to take 10,000 to 100,000 passes through the training set for reasonably complicated problems, large values are common. Complex data may require even higher numbers. Smaller values of MAXIT and a trial and error changing of the $\alpha$ the momentum factor, eta the learning parameter, $\varepsilon$ the error tolerance, and number of nodes and layers may result in a faster determination of a final design.

Remember that stopping at MAXIT does not mean that a larger number may or may not result in convergence it simply means we have quit looking.

5.10.13 Selection of Target Values

The selection of desired or target values depends on the number of classes, the number of nodes in the output layer, and the type of nonlinearities in the output layer. For functional approximation the desired or target values are those that the function is to give. For the pattern recognition problem there is more flexibility. When a neural network is used as a pattern recognizer an additional operation is required on the output of the neural network. Roughly we can think of this operation as a quantization since its output value should give the class number. This structure is shown in Figure 5-32.

**Figure 5-32** Neural network structure for pattern recognition.
The "quantization" operation and the selection of target values will be described for three different types of feedforward neural network structures (a) Single output neuron for K classes, (b) Multiple neuron output layer for K classes uncoded, and (c) Multiple neuron output for K classes coded.

(a) Single output- K classes.

For the single output case and K patterns case the single output region must be broken into K parts as shown in Figure 5-33 where R_k means the region where we decide C_k.

![Figure 5-33](image)

Figure 5-33 Quantization and decision boundaries for the single output neuron structure.

Also shown is a corresponding quantizer that could give the class number for its output. In general the target or desired values could be positioned as far apart as possible in the space determined by the output layer. Common targets for the two class case and a single output neuron are -0.9 and 0.9 with a boundary of the quantizer at 0 for a bipolar activation function like the tanh nonlinearity thus giving the following desired outputs

\[
d(x(k))' = \begin{cases} 
0.9 & \text{if } x(k) \in C_1 \\
0.1 & \text{if } x(k) \in C_2 
\end{cases} 
\]

(5-55)

Similarly for the unipolar logistic nonlinearity the choice might be 0.1 and 0.9 with a quantizer boundary of 0.5 thus giving the desired or target values as

\[
d(x(k))' = \begin{cases} 
0.9 & \text{if } x(k) \in C_1 \\
0.1 & \text{if } x(k) \in C_2 
\end{cases} 
\]

(5-56)

The preference is to push the targets closer to the saturation region and have equally spaced targets for the two class case.

For the K-class case of a single output neuron and unipolar activation function that has a range from 0 to 1, one possible way of selecting the regions is equally spaced along the axis with a choice of desired or target values midway within the regions as follows.

\[
d(x(k))' = \begin{cases} 
\frac{1}{2K} & \text{if } x(k) \in C_1 \\
\frac{3}{2K} & \text{if } x(k) \in C_2 \\
\vdots & \text{...} \\
\frac{2K-1}{2K} & \text{if } x(k) \in C_K 
\end{cases} 
\]

(5-57)
For the K-class case of a single output neuron and bipolar activation function that has a range from -1 to 1, one possible way of selecting the regions is to divide the region [-1, 1] into equally spaced regions along the axis with a choice of desired or target values midway within the regions. For example a four class problem could have regions (-1, -0.5), (-0.5,0), (0, 0.5), and (0.5, 1) with corresponding target values for classes $C_1$, $C_2$, $C_3$, and $C_4$ as -0.75, -0.25, 0.25, and 0.75 respectively.

(b) **Multiple output - K classes uncoded.**

If the neural network structure selected has as many output neurons as the number of classes it is common to have each node identify a different class.

For the **two class case** and two output bipolar neurons the target values are vectors given by $d[x(k)]$. A common choice for this problem would be

\[
 d[x(k)] \begin{cases} 
 [0.9, 0.9]^T & x(k) \in C_1 \\
 [0.9, 0.9]^T & x(k) \in C_2 
 \end{cases} \tag{5-58}
\]

Using unipolar output neurons common target vectors are $[0.9, 0.1]^T$ for class $C_1$ and $[0.1, 0.9]^T$ for class $C_2$. We thus spread out the target vectors in the output vector space.

For the **K-class case** and K output unipolar neurons, a common assignment of target vectors is as follows

\[
 d[x(p)] \begin{cases} 
 [0.9, 0.1, 0.1, \ldots, 0.1]^T & x(p) \in C_1 \\
 [0.1, 0.9, 0.1, \ldots, 0.1]^T & x(p) \in C_2 \\
 \vdots & \vdots \\
 [0.1, 0.1, \ldots, 0.1, 0.9]^T & x(p) \in C_K 
 \end{cases} \tag{5-59}
\]

The quantizer can be implemented in several different ways. Perhaps the easiest method is to select the class number that corresponds to the largest entry in the output vector $y$, that is

Select **Class J** if $y_J$ is the largest of $y_k$

Another way is to compute the Euclidean distance between the desired vectors and the output $y$ and select the class corresponding to the smallest distance.

Select **class J** if $z_j^T \ 2y \& d_j(x(p))$ is the smallest $\tag{5-60}$

The overall structures of the output layer for the above methods are shown in Figure 5-34(a) and (b) respectively.
Figure 5-34  Quantizers for the multiple output neuron uncoded case (a) Largest $y_k$
(b) Smallest distance $z_k$.

(c) Multiple output - K classes coded.
If there are K classes and $K=2^M$ then the output layer could be selected to have only
M outputs such that the binary version of the output vector gives the index of the class. For
example for bipolar activation functions in the output layer the desired values might be

$$
\begin{align*}
    d[x(p)]' = \begin{cases}
        [0.9 \ 0.9 \ ... \ 0.9 \ 0.9 \ 0.9] \quad x(p) \in C_0 \\
        [0.9 \ 0.9 \ ... \ 0.9 \ 0.9 \ 0.9] \quad x(p) \in C_1 \\
        [0.9 \ 0.9 \ ... \ 0.9 \ 0.9 \ 0.9] \quad x(p) \in C_2 \\
        \vdots \\
        [0.9 \ 0.9 \ ... \ 0.9 \ 0.9 \ 0.9] \quad x(p) \in C_{K/2} \\
    \end{cases}
\end{align*}
$$

(5-61)

Common unipolar neuron target vectors would replace the -0.9 above with 0.1.

The quantizer could then be defined in terms of selecting the class that the output
vector is closest to the target vector where ties are resolved randomly. Another way for
bipolar activation functions would be to apply a unit step function on the output vector
components and then select class whose number is the binary equivalent of the output vector
as shown in Figure 5-35.
5.11 FUNCTIONAL LINK FEEDFORWARD NETWORKS

Some pattern recognition problems have enough information to select the features as functions of the input pattern vectors and/or knowledge that the decision surfaces are of specific form. For example the problem may be the separation of classes with gaussian densities but with unknown parameters, like mean vectors and covariance matrices. In this case the decision surfaces are known to be piecewise quadratic surfaces as shown in Chapter 2. The structure of this optimum decision rule can be implemented in the two layer structure shown in Fig. 5-36.

5.11.1 Functional Link Structure

If the first layer of Fig. 5-36 is incorporated into nonlinear functions the functional link structure shown in Fig. 5-37 is defined. It is a "flat" structure in that the neural network is composed of only a single output layer.
The choice of the nonlinear functions, \( g_{i}(x) \), is arbitrary but should be motivated by the problem that is to be solved. For example, the \( g_{i}(x) \) could be second-order polynomials, Fourier basis functions, orthonormal basis functions, etc. This selection can be considered to be equivalent to selecting the features of the data that facilitate the solution of the pattern recognition problem and will be discussed in more detail in Chapter 8.

The quadratic functional link is shown in Figure 5-38. Notice that the output vector contains linear, squares and cross products of all the components in the pattern vector \( x \) given by \( x' \ [x_1, x_2, \ldots, x_n]^T \).

---

**Figure 5-37** Functional Link feedforward neural network.

**Figure 5-38** Quadratic functional link.
The **Fourier transform functional link** for the pattern vector \( x' = [x_1, x_2, \ldots, x_n]^T \) uses as its \( g_i(x) \) the following summations which are recognized as the coefficients of the Discrete Fourier Transform.

\[
\begin{align*}
    f_1(x) &= \frac{1}{N} \sum_{i=1}^{N} x_i \\
    f_2(x) &= \frac{1}{N} \sum_{i=1}^{N} x_i \sin(2\pi i/N) \\
    f_3(x) &= \frac{1}{N} \sum_{i=1}^{N} x_i \cos(2\pi i/N) \\
    \vdots \\
    f_k(x) &= \frac{1}{N} \sum_{i=1}^{N} x_i \cos(\pi i)
\end{align*}
\]

(5-62)

The functional link structure reduces considerably the number of neurons needed for its implementation, minimizes hardware and software, simplifies the training algorithm for training since it contains only one layer, thus using only the simpler rule #1, and with proper selection of training parameters, speeds up the training process. It is actually an implementation of the generalized linear decision rule. The transformation from input vector to output of the functional link does not give any new information just a different representation. In many cases information is lost in going through the link, for example using only the lowest two Fourier coefficients does not preserve information about higher frequencies. In most cases the transformation gives an output vector of a larger size than the original pattern vector so as to facilitate a linear separability of the patterns. For the functional link net to perform satisfactorily a judicious choice of the functions and the number of functions is required. Also, in most cases, the functional link implementation requires careful selection of the neural network training parameters as the training seems more sensitive than that for the multi-layer feedforward neural net implementation.

### 5.11.2 Solutions for the Parity 2 Problem

The parity two problem will give a one output if the components of the pattern vector \( x_1 \) and \( x_2 \) are the same. Shown in Fig 5-39 are the parity-2 patterns and their desired outputs for the four pattern vectors casewith values of 1 for class \( C_1 \) and 0 for class \( C_2 \).
Figure 5-39  Parity 2 problem (a) Class pattern vectors (b) desired outputs.

The objective is to find two neural networks that will give a desired value of 1 when $x_1 \neq x_2$ and a desired value of 0 if $x_1 = x_2$. The problem will be solved first using a standard feedforward neural network and then by using a functional link network with the structure shown in Figure 5-37.

**Neural Network Implementation:** A two layer neural network structure shown in Figure 5-40 was selected to solve the parity two problem. Note that it is two layers with two neural elements in the first layer and one neural element in the output layer. Since the patterns are not linearly separable, a one layer feedforward neural network would be insufficient to solve the problem. Using logistic activation functions for the neurons, this network was trained with the four pattern vectors shown, resulting in the neural network shown in Figure 5-40.

![Neural Network Diagram](image)

Figure 5-40  Neural network solution for the parity 2 problem.

**Functional Link Implementation:** A functional link implementation would be a flat neural network composed of a functional link and one output neural element. The functional link chosen to implement the parity two problem can take many forms; however, the simplest uses only functions $x_1$, $x_2$, $x_1x_2$, and 1. Thus the output neuron will have only four inputs as shown in Figure 5-41.

![Functional Link Diagram](image)

5-49
The output of the functional link produces, for the pattern classes, new training patterns which are easily seen to be as follows.

Class $C_1$: \( \left\{ \begin{array}{c} \left[ \begin{array}{c} \& \& 1 \ 1 \end{array} \right]^T, \left[ \begin{array}{c} 1 \ 1 \ 1 \ 1 \end{array} \right]^T \end{array} \right\} \)

Class $C_2$: \( \left\{ \begin{array}{c} \left[ \begin{array}{c} \& \& \& \& 1 \ 1 \end{array} \right]^T, \left[ \begin{array}{c} 1 \& \& \& \& 1 \ 1 \end{array} \right]^T \end{array} \right\} \)

A neural net was trained using these extended training patterns resulting in the weights as shown in Figure 5-42.

Notice that the straight neural network implementation required two more neural elements than the functional link implementation. The functional link net essentially replaces the first layer of the neural network implementation without a functional link.

Calculation of the functional link functions, the product $x_1x_2$ for this problem, creates an increase in overhead (number of computations) for its implementation. In general, this increase must be considered as part of the complexity and overhead of the functional link implementation. These calculations could be carried out in many different hardware and software implementations when realizing the classifier.

5.12 PERFORMANCE FOR NEURAL NETWORK DESIGNS

During the training of a feedforward neural network common performance criteria were the partial mean squared error or total mean squared error over all training patterns. Other performance measures include number of training samples correctly and incorrectly classified and the ratio of correct classification to total number classified. Each of these
criteria can lead to a realistic design for the problem at hand. The design should then be validated by using samples that were not in the training set. This will test the ability of the network to extrapolate to samples not in the training set. The overall procedure is shown conceptually in Figure 5-43.

**Figure 5-43** Conceptual diagram for evaluating the performance of a neural network design.

Common measures of performance for a neural network designs after training with known classification, include the confusion matrix, the total probability of error and probability of each type of error, and a cost per decision similar to Bayes risk resulting from patterns not included in the training set.

### 5.12.1 Confusion Matrix

The confusion matrix is a table that gives the number of classifications of each type for each pattern class in the validation pattern set. It can also be used for checking the performance of the designed system on the training samples. The entries in the row for correct class, corresponding to the row number, give the number of the evaluation set that were classified to the class indicated on the top column index. An example of a confusion matrix is shown in Figure 5-44.
From the table it is seen that if the samples known to be from class C\textsubscript{2} are classified using the neural net design 18 of them will be confused or classified as class 1, while 57 of them will be classified correctly as coming from class C\textsubscript{2}, none confused with class 3, 7 classified incorrectly or confused with class 4, etc. The numbers in each row will add up to the total number of samples in the class corresponding to that row. If the network design is successful the matrix of values will have high values along the diagonal, indicating correct classification, and low numbers off diagonal indicating errors in classification.

5.12.2 Probabilities of Error

Estimates of the probabilities of error and probabilities of correct decision for each class provide useful information about the performance of a neural network design. These estimates can be determined from the confusion matrix but in a sense do not provide as much information as the confusion matrix as it does not tell how the patterns are misclassified. For each class the estimate of the probability of correct classification can be obtained by dividing the number of patterns classified correctly by the total number of patterns classified from that class. Estimates of the probability of error for each class is simply 1 minus the probability of being correct. For example using the confusion matrix shown in Figure 5-44, the probability of being correct conditioned on class C\textsubscript{2} is 57 over 100 or 0.57 with corresponding conditional probability of error as 0.43. Similarly the following probabilities of error can be found by using the confusion matrix as

\[
\begin{align*}
\hat{P}(\text{correct|C}_1) & \sim \hat{P}(\text{error|C}_1) \sim 0.82 & \hat{P}(\text{correct|C}_2) & \sim \hat{P}(\text{error|C}_2) \sim 0.57 \sim 0.43 \\
\hat{P}(\text{correct|C}_5) & \sim \hat{P}(\text{error|C}_5) \sim 0.99 & \hat{P}(\text{correct|C}_5) & \sim \hat{P}(\text{error|C}_5) \sim 0.01
\end{align*}
\]
As an overall measure of performance, the total probability of correct classification and error can be determined by estimating the apriori probabilities of each class. If the testing samples are representative of the pattern vectors, the apriori probabilities can be estimated by the ratio of the number of samples in a particular class divided by the total number of patterns classified. For the table shown in Fig. 5-44 the $\hat{P}(C_j)$ can be estimated by $50/450$.

With the estimates of the other apriori probabilities the total probability of error can be estimated by

$$\hat{P}(\text{error}) \sum_{i=1}^{N_C} \hat{P}(\text{error*}C_i) P(C_i)$$

where $N_C$ is the number of classes in verification set.

For the example of Fig. 5-44 the total probability of error is seen to be

$$\hat{P}(\text{error}) \begin{array}{cccc}
0.18 & 100/450 & 100/450 & 100/450 \\
0.28 & 100/450 & 50/450 & 100/450 \\
0.8 & 100/450 & 0.01 & 100/450 \\
0.43 & 100/450 & 0.28 & 100/450 \\
0.01 & 100/450 & 0.2888 & \\
\end{array}$$

5.12.3 Cost per Decision

If $C_{ij}$ represents the cost in deciding class $C_i$ when the true class is $C_j$, then the average cost per decision $J$ is defined by

$$J^\circ \frac{\text{Average Cost}}{\text{decision}} \sum_{i=1}^{N_C} \sum_{j=1}^{N_C} C_{ij} \hat{P}(\text{decide } C_i* C_j) \hat{P}(C_j)$$

where the probabilities given are estimated using the confusion matrix and the costs $C_{ij}$ have been subjectively assigned. In this way errors of a certain kind can be weighted heavier or lighter based on perceived costs of making errors which is the basic Bayes performance measure. This measure can be used even though your design was based on Epstein or $E_{\text{TOT}}$, however your design may not be optimal with respect to Bayes performance.

5.13 RADIAL BASIS FUNCTION NEURAL NETWORKS

A radial basis function neural network is a two layer neural network with non-monotonic activation functions as shown in Figure 5-45.
The output $F(\mathbf{x})$ of the radial basis function neural network is composed of a linear function of the outputs from a bank of neural elements with non monotonic activation functions $\varphi(.)$ and characterized by the following equation

$$F(x_1, x_2, ..., x_n) = \sum_{j=1}^{M} \alpha_j \varphi(j, \mathbf{z}_j)$$

(5-68)

where $2\mathbf{z}_j \Delta \mathbf{z}_j$ is a norm that may be taken as the standard Euclidean norm as follows

$$2\mathbf{z}_j \Delta \mathbf{z}_j = \sqrt{\sum_{i=1}^{n} (x_i \Delta z_{j_i})^2}$$

(5-69)

The $\{ \mathbf{z}_j : j = 1, 2, ..., M \}$ are assumed to be known vectors which in some representations could correspond to cluster centers. The $\varphi(r)$ is also assumed known and can take many forms, the most common of which are those which are monotonic decreasing functions of $r$ for $r$ positive. Common choices in the literature for $\varphi(r)$ include the following

1. $\varphi(r) = \frac{1}{(r^2 + \alpha^2)^{\frac{\gamma}{2}}} \quad$ for $\alpha > 0$, and $r > 0$
2. $\varphi(r) = \exp\left(\frac{-r^2}{2\sigma^2}\right) \quad$ for $\sigma > 0$, and $r > 0$

(5-70)

For the pattern recognition problem we are trying to find a function $F(\mathbf{x})$ which separates the pattern space into regions for deciding each of the classes. In this way $F(\mathbf{x})$ is a mapping from the sample space to the integers $1, 2, ..., K$ which represent the $K$ classes.
Rather than use the integers we usually select a set of target values for each of the classes similar to those given in section 5.6 for the feedforward neural network approach. If \( x_1, x_2, \ldots, x_N \) represent a total of \( N \) training samples from \( K \) classes and \( t_1, t_2, \ldots, t_N \), their respective target values, we can define the sum of the squared error performance measure \( \tilde{E} \) for the approximation as

\[
\tilde{E} = \sum_{j=1}^{M} \left( \sum_{x_j} (F(x_1, x_2, \ldots, x_n) \& \alpha_j \& \beta_j \& \phi(z_j x_j)) \right)^2
\]  

(5-71)

The basic problem then is to select the \( \alpha, \{\beta_j : j = 1, 2, \ldots, M\} \), and \( \{z_j : j = 1, 2, \ldots, M\} \) such that we approximate this mapping function by minimizing the performance measure \( \tilde{E} \).

Finding the best approximation is usually broken up into two parts. First the \( M \) and \( \{z_j : j = 1, 2, \ldots, M\} \) could be determined by using some form of clustering algorithm on the training data. Next with these fixed \( \{z_j : j = 1, 2, \ldots, M\} \) any minimization procedure could be used to find the best \( \alpha, \{\beta_j : j = 1, 2, \ldots, M\} \). Thus the training of the network does not use the backpropagation algorithm.
5.14 CAPABILITIES OF NEURAL ELEMENTS AND NETWORKS

The design of neural networks to solve specific problems using a set of training samples appears on the surface to be straightforward. We simply “turn on” one the algorithms presented and wait for it to converge; however there are a number of difficulties that are sometimes encountered and overlooked.

5.14.1 Possible Problems in Neural Net Designs

Perhaps the most common problem is that of failure to converge in the allotted number of iterations. Failure to converge can be the result of selecting a structure that may not be sufficient to approximate the decision function. For example the choice of a single layer of neurons in most cases is not sufficient. Even if the number of layers are selected properly the number of neurons in the layers may not be enough or be appropriately distributed.

Also the algorithm may not converge simply because we did not select a sufficient number for the maximum number of iterations, that is, we just did not wait long enough.

In some cases due to improper normalization or the selection of the training parameters the algorithm produces a lock up situation. By lock up we mean that the neural net weights and nodal outputs reach values that will not allow changes or produce almost infinitesimal changes. Such situations result when we obtain saturation of the nonlinearities. Reduction of training parameters $\eta$ can usually give the algorithm “room” to operate successfully and result in convergence. Other times if the momentum parameter is too large the weights may wildly fluctuate again resulting eventually in a lock up situation.

For the two class and small sample problems there is a very slight possibility that the algorithm can produce oscillations thus never resulting in convergence. These are limit cycles resulting from the iterative nonlinear weight update algorithm and there occurrence is limited to constructive examples with prescribed initial conditions.

A very important problem area is that of successful training and performance on training samples but poor performance when implemented on non training samples. In other words the design does not have the ability to generalize. This problem can result for many different reasons. The first major reason is that representative training samples have not been selected properly during the training phase of the design. That is the training samples did not exhibit all of the data variability. Another major problem is that the training phase was carried on with too strict of a tolerance parameter which resulted in a design that may only recognize the training set. It is also possible that the performance of even the best design has an inherent performance measure that might be unacceptable to you, thus the design obtained using the neural net can never result in satisfactory performance.

Many practical problems require classification into a large number of pattern classes using high dimension pattern vectors. For example the design of a neural network to recognize typewritten letters from the English alphabet of 26 letters. In problems like this the feature selection is critical. Also breaking the total design into smaller problems can result in more efficient designs. For example one neural net might be trained to distinguish between the four sub classes

$$A_1 = \{g, p, q, y, j\} \quad A_2 = \{b, d, f, h, k, l, t\}$$
$$A_3 = \{w, m ,n, z, x, s, e, a\} \quad A_4 = \{c, o, u, v, r, i\}$$
and then neural nets trained to discriminate between individual letters within each of those classes. Thus the final design would be composed of five neural nets connected as shown in Figure 5-46. This structure is an example of what is called a decision tree structure. The choice of the number of branches and the number in each branch offers many different ways of decomposing the original problem.

![Figure 5-46 Decision tree structure with neural net branches.](image)

### 5.14.2 Capabilities of Neural Networks

There are many theorems in modern mathematics that deal with the universal approximation of functions. Since the decision rule is a function these theorems provide guidance for the existence of structures for universal approximators. The theorems give conditions in general of the structures, parameters, and functions that can be used in approximating functions. The two layer feedforward neural network structure can be thought of as a special case of these structures.

Thus we are led to the important result that with the proper number and type of nonlinearities and the appropriate number of nodes, feedforward neural networks can approximate any function. In this way there is a promise of solving every pattern recognition problem using the “proper” neural network. It may not be the most efficient or the simplest but it will solve the problem.

Neural networks have been successfully used for the purpose of identifying systems since the input and output of the system are governed by a functional relationship that is learned using training samples and their outputs as desired or targets. The technique is particularly useful in identifying nonlinear systems.

Control systems respond to input functions to give a control that produces a desirable output or result thus neural networks can be trained to produce the desired control.
5.15 SUMMARY
PROBLEMS

5.1 Use McCollough-Pitts Neuronal Models only to implement the logical function

\[ f(x_1, x_2, x_3) = \overline{x_1} \overline{x_2} \overline{x_3} \]  

Where \( \overline{\cdot} \) means complement.

5.2 Using the simple two input McMollough-Pitts Neuron shown below
(a) Give all values of \( T \), the threshold, such that the model will implement a logical "or" function. The inputs \( x_1 \) and \( x_2 \in \{0,1\} \).
(b) Repeat for logical "and" function.

5.3 Give the logic function that the following neural network generates. The inputs \( x_1, x_2, \) and \( x_3 \) are either 0 or 1 as well as the outputs.

5.4 For the neural network shown and using a logistic nonlinear activation function for each node as

\[ g(\text{net}) = \frac{1}{1 + e^{-\lambda \text{net}}} \]

(a) For \( \lambda = 1 \), give a two dimensional representation of the output for \( x_1 \) and \( x_2 \) as continuous inputs over the domains \( x_1 < 5 \) and \( x_2 < 5 \) (You can use a contour plot, two dimensional gray level, or two dimensional surface).
(b) Repeat for \( \lambda = 10 \) and comment on the differences of your results from (a) and (b).
5.5 (a) Derive the delta training rule for the hyperbolic activation function for a single neural element. 
(b) Express your answer in terms of $f(\text{net})$ the output of the neural element if possible.

5.6 The neural element on the right is used as a pattern recognizer. Let 
\[ w_1, w_2, w_3 \& \text{ net} = f(\text{net}) \& \text{ net}^2. \]

(a) Using the following form for the decision rule
   \[ \text{If } y > 1 \text{ decide class } C_1 \text{ otherwise decide class } C_2. \]
   Find the decision boundaries in the original pattern space and identify the regions that go with each class.
(b) What kind of regions could be generated if $f(\text{net})$ were polynomial in form. (give specific examples)

5.7 An artificial neural element has the non-monotonic activation function given below

\[ f(\text{net}) = \text{net} (\text{net-1})(\text{net-2}). \]

It is to be used as a pattern recognizer using the following decision rule
\[ \text{IF } y > 0 \text{ decide class } C_1 \]
\[ \text{otherwise } \text{ decide class } C_2. \]

Find the decision boundaries in the original continuous pattern space $(x_1, x_2) \in \mathbb{R}^2$ and identify the regions in that space that go with class $C_1$.

5.8 For the neural net shown where $x_1$ and $x_2$ are continuous variables. Illustrate the three regions in the pattern space where the output $y$ is -1, 0, and 1 respectively.
Using the logistic unipolar activation function $f(\text{net})$ given by

$$f(\text{net}) = \frac{1}{1 + \exp(-\lambda \text{net})}$$

(a) Determine the scale factors $\text{SF}_1$ and $\text{SF}_0$ for the weight update equation using the delta algorithm.
(b) Plot the scale factors for $f(\text{net})$ equals zero to one.
(c) Comment on the shape of your curves when the output is other than the desired value.

5.10 Synthesize a Neural Network (3 layers) using unit step activation functions that will give a 1 output for all pattern vectors in the shaded area and give a "0" output for all pattern vectors outside the shaded area.
5.12 Synthesize a neural network using signum activation functions for which the output will be a 1 if \((x_1, x_2) \in R_1\) (including boundary) and -1 otherwise.

5.13 Synthesize a two layer neural network using the unit step function that will give a 1 output for all pattern vectors in the shaded area (including boundary) and a 0 output for all pattern vectors outside the shaded area.

5.14 Illustrate a two layer neural network that will realize a pattern recognizer that has the decision regions shown. Be precise specifying all nonlinearities and weights.

5.15 Synthesize a neural network including coefficients and nonlinearities that would discriminate all patterns within the unit cube shown from those patterns outside the unit cube.

5.16 A three class problem is to be solved using a neural network with logistic activation functions.  
(a) Give two possible structures for the neural network that might solve the problem.  
(b) Give the desired values or vectors for the outputs for each of the cases.
A single layer continuous perceptron shown below is to be used to separate two pattern classes using the sets $P_1$ and $P_2$ respectively. The target values for the output $y$ are 0.8 for $x \in P_1$ and 0.2 for $x \in P_2$. Use

$$f(\text{net}) = \frac{1}{1 + \exp(-\text{net})}$$

and the initial weight vector

$$w(0) = (0.5, 0.5, 0.2)^T.$$

(a) Calculate the next weight vector $w(1)$ if the first training sample $x(1) = (1, 1)^T$ is from $P_1$ using an $\eta = 0.5$.
(b) Repeat if this $x(1)$ happened to be from $P_2$.
(c) After the 10th pattern the weight vector is found to be $w(10) = (0.2, 0.3, 0.1)^T$, calculate the next weight vector $w(11)$ if $x(11) = (1, 0)^T$ where $x(11)$ is from $P_1$.

The data set XORN.DAT is to be used to find a neural network to distinguish between the two classes. Use the LCL Neural Simulator for the design.

(a) Run the simulator and record the number of iterations required for the design to be obtained for the following structure.
   - 2 layers, 3 nodes, first layer, 1 node output layer
   - bipolar activation function, train by sample
   - learning $\eta = 0.7$, momentum $\alpha = 0.8$
   - error tolerance = 0.01, Ng-Widrow on.
   - Max iterations 8000
(b) Illustrate your design showing weights, etc.
(c) For $\eta = 0.7$ and $\alpha = 0.3$ and the same structure above run the simulator 10 times and record the number of iterations required. If convergence did not occur use another value and compute the average number of iterations required.
(d) The following is a small project to explore the problem of finding the optimum $\eta$ and $\alpha$ in the sense of minimizing the average number of iterations for convergence of an acceptable design with respect to the error tolerance given. Make a grid of values for $\eta = 0$ to 1 and $\alpha = 0$ to 1 in increments of 0.2. Then repeat (c) with different initial starting values and plot as a two dimensional the average number of iterations required for convergence. Locate the minimum of this plot thus finding the “optimum parameters for this particular problem. Every problem will have different values for parameters that lead to a “fast” solution; however, the optimum values are more a curiosity as any design that converges provides a feasible design which satisfies the tolerance given and it is thus not necessary to find or use the optimum parameters.
and the training patterns are as follows.

Class 1:

\[
\begin{array}{ccc}
1 & 1 & 1 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{array}, \quad
\begin{array}{ccc}
0 & 0 & 0 \\
1 & 1 & 1 \\
0 & 0 & 0 \\
\end{array}, \quad
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

"rows"

Class 2:

\[
\begin{array}{ccc}
1 & 0 & 0 \\
1 & 0 & 0 \\
1 & 0 & 0 \\
\end{array}, \quad
\begin{array}{ccc}
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
\end{array}, \quad
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 0 & 1 \\
0 & 0 & 1 \\
\end{array}
\]

"columns"

(a) Use the LCL Neural Simulator to train a neural network to perform class separation using training samples above and illustrate your design.

(b) Use a logical expression to represent a solution of the same classification problem using the binary training patterns.

(c) Compare the structures determined in (a) and (b) above.

(d) Tell what you would do differently for (a) and (b) if the training samples were no longer binary but took on values on the interval \([0,1]\).

(e) Use the neural network in (a) to classify the following patterns and comment on your results.

\[
\begin{array}{ccc}
1 & 1 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\end{array}, \quad
\begin{array}{ccc}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{array}, \quad
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{array}, \quad
\begin{array}{ccc}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{array}, \quad
\begin{array}{ccc}
1 & 1 & 1 \\
0 & 0 & 1 \\
0 & 0 & 1 \\
\end{array}
\]

5.20 Use the LCLNET Simulator to design a neural network that uses a functional link net for the XORN.DAT file.

Use link as \([y_1, y_2, y_3, y_4]=[x_1, x_2, x_1x_2, 1]\).

(a) Train a ONE LAYER net with a single output mode using, bipolar activation function, train by sample, learning \(\eta=0.7\), momentum = 0.3, error tolerance = 0.001, ng-windrow on, Max iterations = 5000.

(b) Run 10 trials for design and compute average number of iterations for design to converge.
(c) Compare these results with those of Problems 5.18. Discuss clearly the basis for your comparison.

5.21 A given neural element has an

\[ f(\text{net}) = \frac{1}{m \sqrt{2\pi}} \exp \left( \frac{-z^2}{2} \right) dz \]

(a) Derive the delta training weight update equation for that \( f(\text{net}) \).
(b) Is \( f(\text{net}) \) a bipolar or unipolar activation function?
(c) Is \( f(\text{net}) \) a discrete or continuous activation function?

5.22 Find the delta rules for weight updates for the following activation functions.

(a) \( f(\text{net}) = \frac{1}{m \alpha \sqrt{\pi}} \exp \left( \frac{-\alpha x^2}{2} \right) dx \)

(b) \( f(\text{net}) = \int_{-\infty}^{\infty} x \exp(\alpha x) dx \)

5.23 The activation function for all neurons of a neural net is shown on the right.
(a) Is this a unipolar or bipolar activation function?
(b) If the standard backpropagation algorithm is used to train these neural elements of the net, find rule \#1 for updating the output layer neurons weights.
(c) Also give rule \#2 for updating the weights in the hidden layers.

5.24 Repeat 5.23 for the \( f(\text{net}) = \frac{e^{\text{net} \delta} e^{\text{net} \sigma_{\text{net}}}}{e^{\text{net} \sigma_{\text{net}}}} \).

5.25 A basic neural element is shown. It is to be used to separate two pattern classes \( C_1 \) and \( C_2 \) using sets of training samples \( P_1 \) and \( P_2 \) respectively.

\[
P_1 = \begin{cases} x_a \begin{bmatrix} 0 \\ 0 \end{bmatrix}, & x_b \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ \end{cases} \\
P_2 = \begin{cases} x_c \begin{bmatrix} 1 \\ 0 \end{bmatrix}, & x_d \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ \end{cases} 
\]
Assume \( f(\text{net}) = \tanh(\text{net}/2), \ \eta = 0.5, \) and the desired outputs of 0.9 if \( x \in \mathcal{O}_1 \) and -0.9 if \( x \in \mathcal{O}_2 \).

(a) Use the sequential (local) delta training algorithm and compute \( w(2) \) and \( w(3) \) using \( w(1) = [0.2, -0.3, 0.1]^T \) and \( x(1) = x_a, \ x(2) = x_c \).

(b) Using \( x_a \), find all weight vectors that will make the local error \( \frac{1}{2}(d(x_a) - y)^2 \) equal to zero and give one example with numbers.

(c) Repeat (a) using the batch (global) delta training algorithm.

5.26 After several iterations, the weights for a neural net for a two pattern class problem, using \( f(\text{net}) = \frac{1}{1 + \exp(-\text{net})} \) are as shown below. Assume target values of 0.9 if \( x \in \mathcal{O}_1 \) and 0.1 if \( x \in \mathcal{O}_2 \) and a training parameter \( \eta = 1/3 \). (Any relation to a neural net living or dead is purely coincidental)

(a) Determine \( y \) for the training sample \( x_5 = [0, 0]^T, \ x_5 \in \mathcal{O}_1 \).

(b) Determine the new weight \( w_{11}^{(3)}(6) \) using the back propagation algorithm.

(c) Determine the new weight \( w_{11}^{(2)}(6) \) using the back propagation algorithm.

(d) Determine the new weight \( w_{11}^{(1)}(6) \) using the back propagation algorithm.

5.27 The artificial neural net below is being trained using the backpropagation training algorithm and the weights are a result of the 7th training iteration. The new training sample is \( x(7) = [0.5, 0.8]^T \) with a desired output \( y \) as 0.8. Find the following next iteration weights in terms of the training parameter \( \eta \)

\( (a) \ w_{12}^{(2)}(8) \ and \ (b) \ w_{23}^{(1)}(8). \)
5.28 For the neural network shown below

Assume all the activation function for the neurons are \( f(\text{net}) = \tanh(\text{net}/2) \) and let the desired values for each of the classes be given by

\[
\begin{bmatrix}
  d_1(x) & d_2(x)
\end{bmatrix}^T = \begin{bmatrix}
  0.9 & 0.9
\end{bmatrix} \quad \text{if } x \in C_1
\]

\[
\begin{bmatrix}
  0.9 & 0.9
\end{bmatrix} \quad \text{if } x \in C_2
\]

(a) Calculate \( y_1^{(2)} \) and \( y_2^{(2)} \) for input vector \( x(1) = [0 \ 1 \ 1]^T \) which is a member of Class \( C_1 \).

(b) If \( \eta = 0.5 \) find the new weight \( w_{11}^{(2)} \) which would result from using \( x(1) \) as the first training sample in the back propagation algorithm and values given on network as initial conditions on weights.

(c) If \( \eta = 0.5 \) find the new weight \( w_{12}^{(2)} \) which would result from using \( x(1) \) as the first training sample in the back propagation algorithm and values given on network as initial conditions on weights.

(d) Which class would the neural net and decision rule classify the point (pattern \( x_t \)) which gives \( [y_1^{(2)}, y_2^{(2)}] = [0.6, 0.4] \) using the minimum distance to given desired values for classification rule.
5.29 Suppose a neural net has been trained to distinguish between three classes using a single unipolar output neuron with logistic activation function. Using target values of the output of this neuron as follows

\[ C_1 \ 0.2, \ C_2 \ 0.6, \ C_3 \ 0.8. \]

(a) Illustrate in the output space a reasonable assignment of thresholds for discriminating between the classes.

(b) Using your assignments classify a pattern that gives an output of \( y = 0.45 \).

5.30 A five class pattern recognition problem is to be solved using a neural net (using tanh nonlinearities) with only one output node, \( y \), followed by a decision rule that would be used for classification.

(a) Give the desired values for the output of the neural net for training for each of the classes.

(b) Describe the decision rule needed for classification using \( y \).

5.31 Consider the 4 class problem where patterns are 4x4 images. Let \( f_1(m,n), f_2(m,n), f_3(m,n), \) and \( f_4(m,n) \), given below, be pattern vectors from \( C_1, C_2, C_3, \) and \( C_4 \) respectively.

\[
\begin{align*}
f_1(m,n)' &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, & f_2(m,n)' &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, & f_3(m,n)' &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, & f_4(m,n)' &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}
\end{align*}
\]

Define the classes \( C_i \) as \( C_i = \{ \text{all translates in both horizontal and vertical directions of } f_i(m,n) \text{ that stay within 4x4 grid (i.e. no circular translates)} \} \)

(a) Train a neural network to distinguish between the patterns operating first on the patterns with the Magnitude of the DFT to obtain features. Be clear on parameter selection and show your design.

(b) Classify the following patterns using your network in (a).

\[
\begin{align*}
x_1' &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, & x_2' &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, & x_3' &= \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, & x_4' &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}
\end{align*}
\]

(c) Discuss the ability of your net to classify the distorted patterns above. Tell how you might modify your training set to include binary errors.

(d) With the net designed in (a) classify the following patterns.
and discuss the ability of your net to classify the "noisy" patterns above. Suggest how you could change your training set to adjust for this type of error.

(e) An obvious extension of the above problem is that of having pattern classes composed of not only translates but include rotations as well as scaled versions. How would you train a network to do that job?

5.32 Suppose that a neural network has been trained to distinguish between three classes using three output neurons which have the following target values

\[ 0.9, 0.1, 0.1 \quad 6 \quad C_1, \quad 0.1, 0.9, 0.1 \quad 6 \quad C_2, \quad 0.1, 0.1, 0.9 \quad 6 \quad C_3 \]

(a) If a pattern \( \mathbf{x} \) to be classified results in the output vector \( \mathbf{y}' = [0.8, 0.2, 0.3] \), which class do we assign it to? (b) Classify \( \mathbf{y}' = [0.4, 0.3, 0.3] \). (c) Classify \( \mathbf{y}' = [0.4, 0.4, 0.1] \).

(d) Could we solve the same problem using just a single output node? If so tell how, if not tell why you can't.

5.33 In your homework you designed a number of artificial neural networks to solve text book type problems. What would be some of your concerns for solving problems in the "real world". Identify several problem areas and comment on the importance of feature selection.

5.34 Assume that patterns from two classes are four dimensional vectors

\[ \mathbf{x}' = [x_1, x_2, x_3, x_4]^T \]

and that the decision boundary for the classes is a hyperplane (in extended space) representing an hyperellipsoid in the original pattern space. Illustrate a functional link net that could be used to learn the boundary between the classes. What is the dimension of the resulting extended pattern space as a result of the functional link.

5.35 One form of an approximation theorem approximates \( F(x_1, x_2, \ldots, x_n) \) as follows

\[
F(x_1, x_2, \ldots, x_n) = \frac{1}{M} \sum_{i=1}^{M} \alpha_i \varphi \left( \sum_{j=1}^{n} \beta_{ij} x_j \right) \varphi_{i,\alpha(i)}
\]

where \( \varphi(x) \) is an unspecified function bounded by 0 and 1 and monotonic increasing function on \((-4, 4)\).

(a) Illustrate \( F(...) \) in neural net structure showing all nonlinearities, weights, etc.

(b) How many layers and how many nodes per layer are specified?
(c) Give the simplified formulas for using the backpropagation algorithm for updating all necessary parameters.
(d) Give reasonable upper and lower bounds for $F(.)$ in terms of parameters given.

5.36 The "new" activation function in the LCLNET Simulator is given by

$$f(\text{net}) = \frac{1}{\exp(2.336007 \times \text{net}) \times 1/2.236007} \times \frac{1.236007}{2}$$

Suggested target values might be 1.618033989 and -0.618033988.
(a) Find the delta rule for the following activation function
(b) Find the scale factors for this activation function and the given target values and comment on shapes. Why might these be desirable scale factors.
(c) This choice of nonlinearity tries to avoid the lock up situation. Find the scale factors and comment on why this might be true.

5.37 In this problem you are asked to design a pattern recognizer for segments of speech containing only the vowels "a", "e", and "i". Data was collected from a given student using a sound blaster board for recording. The vowels were voiced in succession but with a null space in between to eliminate vocal connectiveness between utterances. An example of the total waveform is shown in Figure P5.37.

Data sets vowelt2.dat and vowelt3.dat each contain 21 "snipits", short time segments, of the vowels seven from each one at different times through out the vowel segments.
(a) Use the snipits in vowelt2.dat as pattern vectors to train an artificial neural net to distinguish between the three vowels. Call the classes 1, 2, and 3 for vowels a, e, and i respectively. Illustrate your final design, showing weight vectors and the results of using your design to classify the training set.
(b) Using the design from part (a), classify the patterns given in the testing data set vowelt3.dat.
(c) The testing data vowelt3.dat has the correct classification given, so determine the confusion matrix using the 21 pattern vectors of that data set. The confusion matrix gives an estimate of the effectiveness of your design to extrapolate outside the training set. Did your results surprise you? Explain.
(d) fvowelt2.dat and fvowelt3.dat are the magnitudes of some of the components of the Fourier transforms of vowelt2.dat and vowelt3.dat. Use this transformed data to repeat (a), (b) and (c). Compare your design with that of using the samples directly (Part (a)).
(e) Comment on the importance of feature extraction to the pattern recognition problem.
References


1994.


[22] Minsky and Pappert

[23] Kolomogorov

[24] Cybenko

[25] Sobajic
5 / PATTERN RECOGNITION

USING NEURAL NETWORKS .......................... 5-1
5.0 INTRODUCTION .................................... 5-1
5.1 HISTORICAL PERSPECTIVE ........................ 5-2
5.2 A REAL NEURON .................................. 5-3
  5.2.1 Neuronal Anatomy ............................ 5-3
  5.2.2 Neuron Communication ......................... 5-5
5.3 MCCOLLOCH-PITTS NEURONAL MODEL ............ 5-7
5.4 ARTIFICIAL NEURAL ELEMENT ..................... 5-9
  5.4.1 Logistic function ............................. 5-11
  5.4.2 Hyperbolic activation function ................. 5-12
  5.4.3 Unit step activation function .................. 5-13
  5.4.4 Signum function activation function .......... 5-13
  5.4.5 Linear activation function ..................... 5-13
  5.4.6 Non monotonic activation Functions .......... 5-13
5.5 GENERAL CONCEPT OF TRAINING A SYSTEM ....... 5-14
  5.5.1 Problem Formulation .......................... 5-14
  5.5.2 System To Be Trained ......................... 5-14
  5.5.3 Training Samples .............................. 5-14
  5.5.4 Performance Measure .......................... 5-15
  5.5.5 Training Algorithm ............................ 5-15
5.6 TRAINING AN ARTIFICIAL NEURAL ELEMENT ...... 5-16
  5.6.1 Signum function activation training algorithm 5-16
  5.6.2 The Delta training algorithm ................. 5-17
  5.6.3 Continuous Perceptron Training Algorithm ... 5-19
  5.6.4 Specific Delta Training Algorithms ............ 5-19
    Logistic activation function training algorithm 5-19
    Hyperbolic Tangent Activation Function Training Algorithm 5-20
    Linear Activation Function training algorithm .... 5-20
  5.6.5 Scale Factors in Training ...................... 5-21
5.7 MULTILAYER NEURAL NETWORKS ................... 5-22
  5.7.1 Feedforward structures for pattern recognition 5-23
    Single node output ................................ 5-24
    Multi-node output ................................ 5-24
    Multi-node coded output ............................ 5-24
  5.7.2 Analysis of multilayer neural networks ........ 5-24
  5.7.3 Synthesis of multilayer feedforward neural networks 5-26
5.8 TRAINING A MULTILAYER FEEDFORWARD NEURAL NETWORK 
    .................................................. 5-29
  5.8.1 Rule #1 (layer L) ............................. 5-31
  5.8.2 Rule #2 (layer L-1) ............................ 5-32
  5.8.3 Specific Rules for Various Nonlinearities ...... 5-33
    Logistic activation function ....................... 5-33
    Hyperbolic activation function ................... 5-33
5.9 BACKPROPAGATION TRAINING ALGORITHMS .......... 5-34
  5.9.1 Backpropagation Algorithm ..................... 5-35
5.9.2 Backpropagation with Momentum Algorithm ............... 5-37

5.10 BACKPROPAGATION ALGORITHM PARAMETER SELECTION ... 5-38

5.10.1 Selection of Initial Conditions for Weights ................ 5-39
5.10.2 Selection of Training Patterns .......................... 5-39
5.10.3 Number of Training Samples Required ..................... 5-39
5.10.4 Sample Ordering for Training ............................ 5-39
5.10.5 Normalization of Training Samples ....................... 5-40
5.10.6 Selection of Training Procedure .......................... 5-40
5.10.7 Choice of Training Parameter ................................ 5-40
5.10.8 Choice of Node Nonlinearities ........................... 5-41
5.10.9 Choice of Momentum Factor ................................ 5-41
5.10.10 Number of Layers and Nodes ............................. 5-41
5.10.11 Selection of Error Tolerance ............................. 5-41
5.10.12 Choice of Maximum Number of Iterations ................ 5-42
5.10.13 Selection of Target Values ............................... 5-42
  (a) Single output - K classes. .................................. 5-43
  (b) Multiple output - K classes uncoded. ....................... 5-44
  (c) Multiple output - K classes coded. ....................... 5-45

5.11 FUNCTIONAL LINK FEEDFORWARD NETWORKS ................. 5-46

5.11.1 Functional Link Structure ................................ 5-46
5.11.2 Solutions for the Parity 2 Problem ....................... 5-48
  Neural Network Implementation ............................... 5-49
  Functional Link Implementation .............................. 5-49

5.12 PERFORMANCE FOR NEURAL NETWORK DESIGNS ............... 5-50

5.12.1 Confusion Matrix ......................................... 5-51
5.12.2 Probabilities of Error ...................................... 5-52
5.12.3 Cost per Decision ......................................... 5-53

5.13 RADIAL BASIS FUNCTION NEURAL NETWORKS ............... 5-53

5.14 CAPABILITIES OF NEURAL ELEMENTS AND NETWORKS ....... 5-55

5.14.1 Possible Problems in Neural Net Designs ................ 5-55
5.14.2 Capabilities of Neural Networks .......................... 5-56

5.15 SUMMARY ................................................... 5-57

PROBLEMS .......................................................... 5-58

References .......................................................... 5-71