Characterization of Radar Signals Using Neural Networks

THESIS

Daniel R. Zahirniak
Capt

AFIT/GE/ENG/90D-69

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/90D-69

Characterization of Radar Signals Using Neural Networks

THESIS

Daniel R. Zahirniak
Capt

AFIT/GE/ENG/90D-69

Approved for public release; distribution unlimited

AFIT/GE/ENG/90D-69

Characterization of Radar Signals Using Neural Networks

THESIS

Presented to the Faculty of the

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science Electrical Engineering

Daniel R. Zahirniak, B.S.E.E

Capt

Dec, 1990

Preface

The purpose of this study was to determine if artificial networks could be used to characterize radar signals from sampled feature data of their electromagnetic signals. Hyperplane Classifiers, trained via backpropagation to optimize the Mean Square Error, Cross Entropy and Classification Figure of Merit objective functions, were first analyzed and tested. Kernel Classifiers, using radial basis functions as the kernel functions, were then analyzed mathematically and tested under various training algorithms. Finally, the Probability Neural Networks were analyzed and tested for comparison with the Hyperplane Classifier and Kernel Classifier networks.

The amount of work accomplished could not have been done without the aid of my thesis committee, Dr. M. Kabrisky, Dr. B. Sutter, and Dr. V. Pyatti. I am especially indebted to my thesis advisor Maj S. Rogers for his direction and support during this project. Also, I thank Capt Eddy and Anthony Schooler for their help with the SUN Workstations and LATEX. Finally, I would like to thank my wife Annette for the sacrifices she made in order that this milestone in our lives could be completed.

Daniel R. Zahirniak

Table of Contents

List of Figures

Figure Page

List of Tables

Abstract

Recent work concerning artificial neural networks has focused on decreasing network training times. Kernel Classifier networks, using radial basis functions (RBFs) as the kernel function, can be trained quickly with little performance degradation. Short training times are critical for systems which must adapt to changing environments.

The function of Kernel Classifier networks is based on the principle that multivariate functions can be approximated via linear combinations of RBFs. RBFs can also perform probability density estimations, making classifications approximating a Baye's optimal descriminant.

Methods used to set the RBF centers included matching the training data, Kohonen Training, K-Means Clustering and placement at averages of data clusters of the same class.

Test results indicate the performance of these networks was equal to that of Hyperplane Classifier networks trained, via backpropagation, to optimize the Mean Square Error, Cross Entropy, and Classification Figure of Merit objective functions. However, the RBF networks trained much faster. The RBF networks also outperformed the Probability Neural Networks,(PNN) indicating the weights in the output layer offset the choice of non-optimal spreads.

This ability to train quickly while obtaining high classification accuracies make RBF Kernel Classifier networks an attractive option for systems which must adapt quickly to changing environments.

Characterization of Radar Signals Using Neural Networks

I. Problem Description

1.1 Introduction

Due to the increasing proliferation of hostile radar systems, the current radar warning devices installed in many Air Force aircraft may have trouble meeting their real-time data processing requirements in the near future. Since artificial neural networks are designed to process data in a distributed manner, they will able to process data much quicker than current computer systems when the parallel distributed processing hardware becomes available . Thus, artificial neural networks may provide the key to solving the real-time data processing requirements of future radar warning devices. This thesis will characterize several types of artificial neural networks and determine if any are suited to accurately characterize radar systems. This problem description will begin by reviewing the background of these radar warning devices. The exact problem to be solved will then be described. This description will be followed by a summary of the knowledge currently available concerning artificial neural networks and a brief outline of the scope of the thesis. The standards and approach taken to solve the problem will then be discussed. This chapter will conclude with a brief overview of the remaining chapters.

1.2 Background

The main task required of a radar warning system is to analyze the electromagnetic environment, determine if this environment contains a radar signal, classify the signal as being generated from either a hostile or friendly radar system, and identify the exact type of radar system transmitting this signal or classify the signal as being from an unknown emitter. This analysis should be accomplished even though the radar system has the ability to change the signal's electromagnetic characteristics (19:42). Accomplishing this task requires the radar warning system to analyze the environment quickly and accurately, classifying signal data according to known parameters. Even with the use of high speed computers, this task is computationally intense and can take several seconds to properly classify the radar signal. As the electromagnetic spectrum becomes increasingly crowded through the proliferation of hostile radar systems, it will take longer for the current radar warning devices to accomplish their mission. However, aircraft are required to operate in real-time, responding to changes in this electromagnetic environment in a matter of milliseconds. This has

led to research in the use of artificial neural networks as a possible method of characterizing a radar system from its transmitted signal.

1.3 Problem

This thesis will characterize several different types of artificial neural networks and determine which would be able to classify radar systems from data concerning their electromagnetic signals. An artificial neural network can be thought of as a massively parallel, interconnected system of simple computing elements, called nodes, which can accomplish certain pattern classification tasks quickly, in a way motivated by a biological nervous system. Since neural networks have been shown to be equivalant to a Bayes' optimal discriminant (21) and capable of performing arbitrary complex transformations (5), it seems logical to assume these networks will be able to characterize radar systems.

1.4 Current Knowledge

Basically, artificial neural networks may be categorized as either Probabilistic Classifiers, Exemplar Classifiers, Hyperplane Classifiers, or Kernel Classifiers (11:47-63). A Probabilistic Classifier neural network seeks to classify patterns by using probability distributions to maximize the probabilities associated with a classification. As such, these networks require an assumption of the probability distribution of the input data (9:1-7). An Exemplar Classifier neural network classifies unknown feature data based on a nearest-neighbor calculation with the training data. That is, the closer an unknown data point is to a known data point in the feature space, the stronger the probability that the two features represent the same object (8:167-170). A Hyperplane Classifier neural network forms decision regions by using hyperplanes to partition the feature space into regions of interest. This partitioning allows the network to make classifications of similar data (19:48-63). A Kernel Classifier neural network uses overlapping kernel function nodes to create complex decision regions over the feature space. These decision regions will determine the classification of each pattern as similar patterns will be identified within the decision regions (11:49).

Usually, these artificial neural networks are developed using unsupervised training, supervised training, or a combination of supervised and unsupervised training. In unsupervised training, the feature data from the environment are input to the network. The nodes in the network are then allowed to arrange their parameters, or cluster, in positions reflecting the distribution of the data (22:151-193). In supervised training, the feature data, in the form of a pattern vector, is presented to the network along with the desired output pattern for that particular input pattern vector. The

difference, or error, between the network output and the desired output is then calculated and used to adjust the network parameters in such a way that the error is minimized. A combination of unsupervised and supervised training can also be used to develop an artificial neural network. In this type of training, the network is first trained using unsupervised training to allow the network parameters to be distributed according to the feature data. After stabilization, the network is then trained, in a supervised fashion, to produce the correct classification for a given input pattern.

1.5 Scope

The final product of this thesis will be a characterization of the Hyperplane and Kernel Classifier neural networks and a determination if they can accurately characterize radar signals. The Probabilistic Classifier will be briefly studied for comparison purposes. The Exemplar Classifier will not be studied as the computational time required to make a classification would exceed the real-time data processing requirements of the aircraft. The Hyperplane Classifier networks will consist of a double hidden layer network, as shown in figure 1.1. This network will be trained,

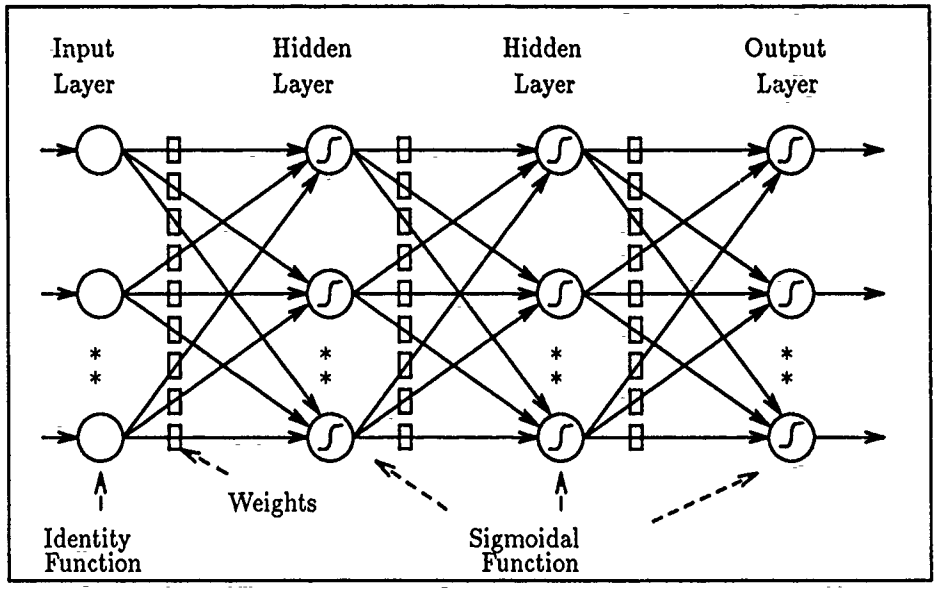

Figure 1.1. Double Hidden Layer Hyperplane Classifier Network

via backpropagation, to optimize the Mean Squared Error, (MSE), the Cross Entropy (CE) and the Classification Figure of Merit (CFM) objective functions (28). The transfer function for each network node will be sigmoidal. The Kernel Classifier will be a single hidden layer network, as shown in figure 1.2, developed using the combined training method with radial basis functions as

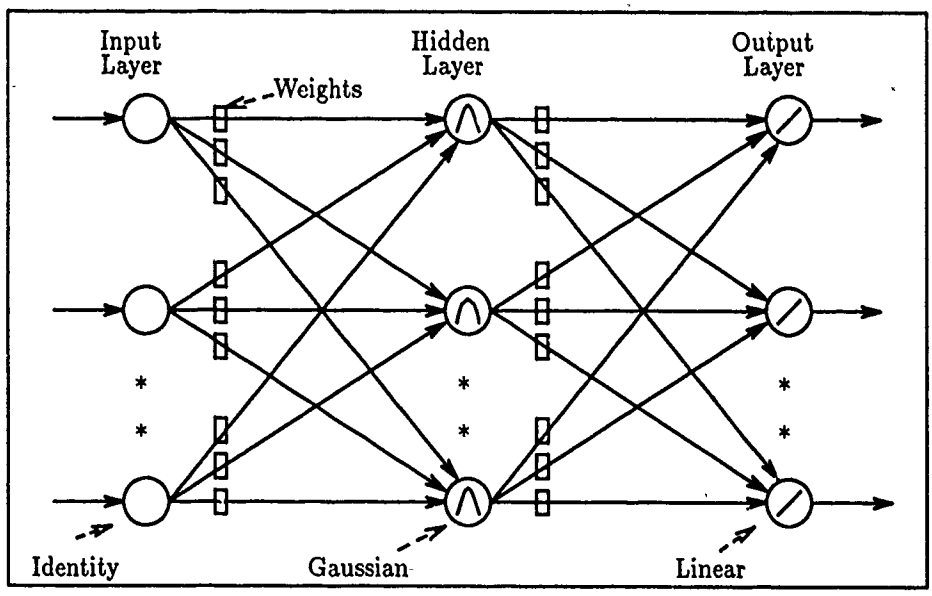

Figure 1.2. Single Hidden Layer Kernel Classifier Network

kernel functions. The transfer function for each of the kernel nodes in the hidden layer will be gaussian. The transfer function for the nodes in the output layer will be linear.

Several methods will be used to determine the weights linking the network's hidden layer nodes to the input layer nodes. The first method will set the weights at values equal to the features of the training set patterns. The second method will set the weights via the Kohonen training algorithm. The third method will set the weights via a K-means clustering algorithm. The fourth method will set the weights at the average of clusters within the pattern classes. The weights linking the kernel nodes in the hidden layer with the nodes in the output layer will be established by a global minimization of the MSE function.

1.6 Standards

The performance criteria for each network will be the classification accuracy and the amount of time it takes to train the network. Classification accuracy is the more important of the two performance criteria since near perfect classification accuracy is a mission requirement. The amount of time it takes to train the network is important since it is highly likely that the threats in the environment can change from mission to mission. Short training times can be crucial to building networks to adapt to this changing environment. The accuracy of each trained network will be calculated by applying test data to the network, and allowing the network to make a classification.

An error will result when the network's classification does not match the known classification. The accuracy will then be calculated as the ratio of the number of correct classifications to the number of input patterns.

1.7 Approach

As part of this thesis, a software environment will be developed on the SUN graphic workstations. This software will be written in ANSI C and designed according to an object-oriented approach. The software will allow the user to select the number of layers for the neural network and to select the training rule for each layer. In this manner, different combinations of training rules can be combined and their overall performance evaluated. Furthermore, this software will allow the user to select either the sigmoidal, gaussian or linear transfer functions for each node in the network. This will allow construction of many different types of networks even though their topology may be the same. Since the main task of this thesis is in pattern recognition, this software will implement only strict feed forward networks. However, the software will be designed to allow future expansion to recurrent and lateral inhibition networks.

1.8 Chapter Outlines

The following is a brief discussion of the information to be found in each of the chapters of the thesis.

1.8.1 Chapter 2 This chapter will provide a review pattern recognition in general followed by a brief synopsis of biological and artificial neural networks. This chapter will then conclude with a history of the development of artificial neural networks, a description of the training methods used to implement these networks and a discussion of the different classifications of artificial neural networks.

1.8.2 Chapter 3 This chapter will provide a mathematical analysis of the algorithms to be used to train the neural networks implemented as part of this thesis. In particular the backpropagation learning algorithms for the MSE, CE and CFM objective functions will be derived for the perceptron-based Hyperplane Classifier networks. The algorithms developed for the radial basis function Kernel Classifier networks will concentrate on first setting the weights of the hidden layer nodes prior to establishing the weights in the output layer nodes via a global minimization of the MSE objective function.

1.8.3 Chapter 4 This chapter will provide a detailed analysis of the software developed for this thesis. This will include a discussion of the software structure and a mapping of the algorithms developed in Chapter 3.

1.8.4 Chapter 5 This chapter will discuss the testing and results for each of the networks implemented with the software described in Chapter 4. This will include an overview of the classification problem, a discussion of the data used to train and test the network and an analysis of the results.

1.8.5 Chapter 6 This chapter will provide conclusions based on the results detailed in Chapter 5 and include recommendations for areas of future study.

1.9 Summary

This thesis will characterize the performance of several artificial neural networks and determine if the networks can be trained to accurately classify radar systems from data concerning their electromagnetic signals. For this thesis, Hyperplane Classifier, Kernel Classifier, and Probabilistic Classifier networks will be developed and tested to analyze their performance. The results of this thesis will provide a determination of the feasibility of using artificial neural networks as the basis for the Air Force's next generation of radar warning devices.

II. Literature Review

2.1 Introduction

Current computer systems may not be able to process data fast enough to meet the future real-time performance requirements of many military weapons systems. Since artificial neural networks are designed to process data in a distributed manner, they may provide the key to solving these data processing requirements. This literature review begins with the background associated with the military's strict data processing requirements followed by a brief review of the concepts of pattern recognition. Research showing how the brain may use biological neural networks to process information is then examined. After describing artificial neural networks in general, this review will cover some of the important milestones in the development of artificial neural networks. Finally, this literature review will describe the current methods used to train these networks and discuss how artificial neural networks are now being categorized according to their methods of classifying data.

2.2 Background

A task required of many military weapon systems is to analyze the environment and determine, in a matter of milliseconds, whether a target of interest is present. Solving this problem of pattern recognition usually requires the weapon system to combine the data from a multitude of sensors, segment the data into areas of interest, extract the important features, and classify these features according to known threat patterns (20:1-12). Currently, these pattern recognition tasks are accomplished using the traditional Von Neumann computers. These computers consist of a Central Processing Unit which performs complex sequential computations, one at a time, under control of a system clock (6:2). Even with the use of high speed computers, this task of pattern recognition is usually so computationally intense that it may take hours to properly classify the pattern (26:7). These time frames are unacceptable for weapon systems operating in real time. The only object capable of performing this type of analysis in real-time is the human brain. Thus, a computing architecture, based upon the way the brain is assumed to function, may be able to solve these pattern recognition problems in the time frames required (26:7). These computing architectures, commonly known as artificial neural networks, perform computations in a manner significantly different than traditional computers. For instance, an artificial neural network processing unit may do only one type of simple calculation, such as producing a single output from a simple transformation of its inputs. However, since there may be thousands of these simple processing units, each interconnected to many others, extremely complex computations can be accomplished,

in parallel, via the network as a whole. This parallel processing of data allows the artificial neural network to process data extremely quickly and may provide the key to rapidly classifying patterns from their distinct environmental features.

2.3 Pattern Recognition

Recognition of patterns is a basic characteristic of many living organisms, including human beings (27:5). The fact that a human being is a very sophisticated information processing system is primarily due to the fact that human beings possess a superior pattern recognition capability (27:5). That is, even though our senses are constantly flooded with an overwhelming variety of patterns from the environment, we still have the ability to determine what information is most important and react accordingly. It is this characteristic of discriminating unknown patterns between populations that a pattern recognition system seeks to emulate.

2.4 Pattern Recognition Systems

Basically, all pattern recognition systems seek to categorize the input data into identifiable classes via the extraction of significant features from a background of irrelevant detail (27:6). Thus, the tasks of a pattern recognition system are to sense the environment, provide data concerning patterns of possible interest, extract relevant features from this data, and classify the pattern as a member of one of the groups under consideration. These tasks are shown in the block diagram of figure 2.1.

2.4.1 Sensing The first task of a pattern recognition system is to represent the pattern under study as a group, or vector, of measurements. This process, known as sensing, attempts to describe the characteristics of the pattern under study and represent the pertinent information available about the pattern (27:9). For example, if the task was to recognize, or classify a radar signal, the measurements of the pulse repetition interval, scan rate and operating frequency might be taken.

2.4.2 Feature Selection The second task of a pattern recognition system is to take the measured data obtained under sensing and extract the intraset and interset features, or attributes, which will enable the system to perform classification. This step is perhaps the most critical as good features make for good classification systems (19:47). However, at this time, there is no strict set of rules available to determine which features actually characterize a class. 'Too few or poorly

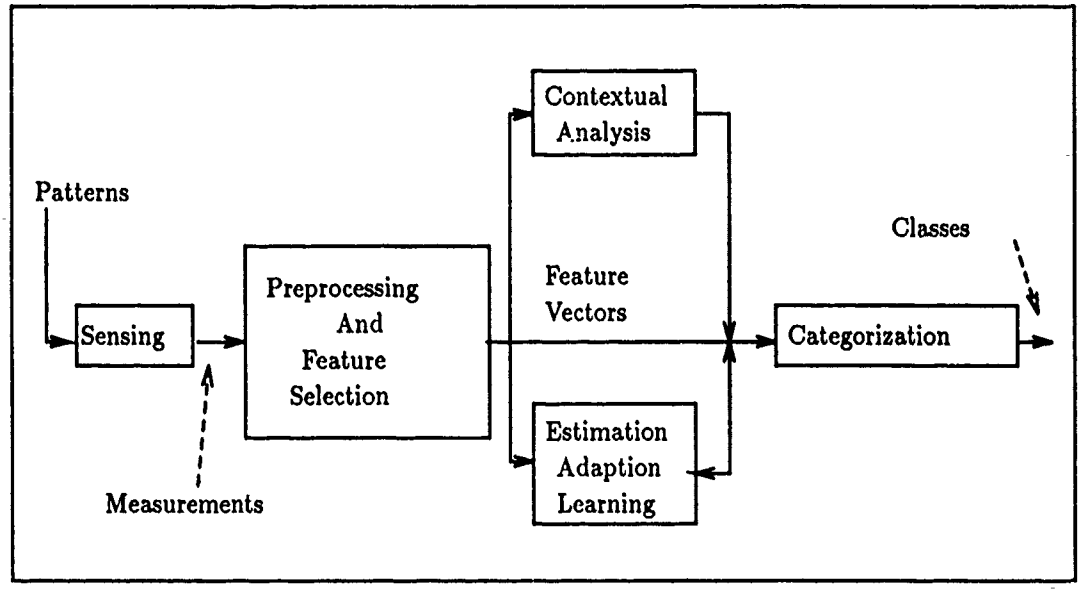

Figure 2.1. Block Diagram of a Pattern Recognition System (27:16)

chosen features will not allow the system to characterize the input patterns sufficiently to allow categorization (27:7).

2.4.3 Categorization The third task of a pattern recognition system is to classify, or categorize, the input pattern as belonging to one of a set of possible classes. In this step, the features extracted from the unknown pattern are analyzed and used to decide from which class the unknown pattern is mostly likely to be a member. This categorization is usually based upon some decision function such as Bayes' optimal discriminant.

Currently, most of these pattern recognition tasks are accomplished via a variety of classical statistical methods such as template matching, frequency histogram associations, and probability density estimations. These methods can require much contextual analysis of the data prior to classification. However, artificial neural networks can be constructed as pattern recognition systems, adapting their internal parameters according to a set or predefined rules, without the need for intensive human analysis of the data. To understand how this process may occur requires an understanding of biological neural networks.

2.5 *Biological Neural Networks*

Biologically speaking, a neuron, as shown in figure 2.2, is a nerve cell which is used to process, store, retrieve and manipulate information received from the environment (26:9). This neuron is

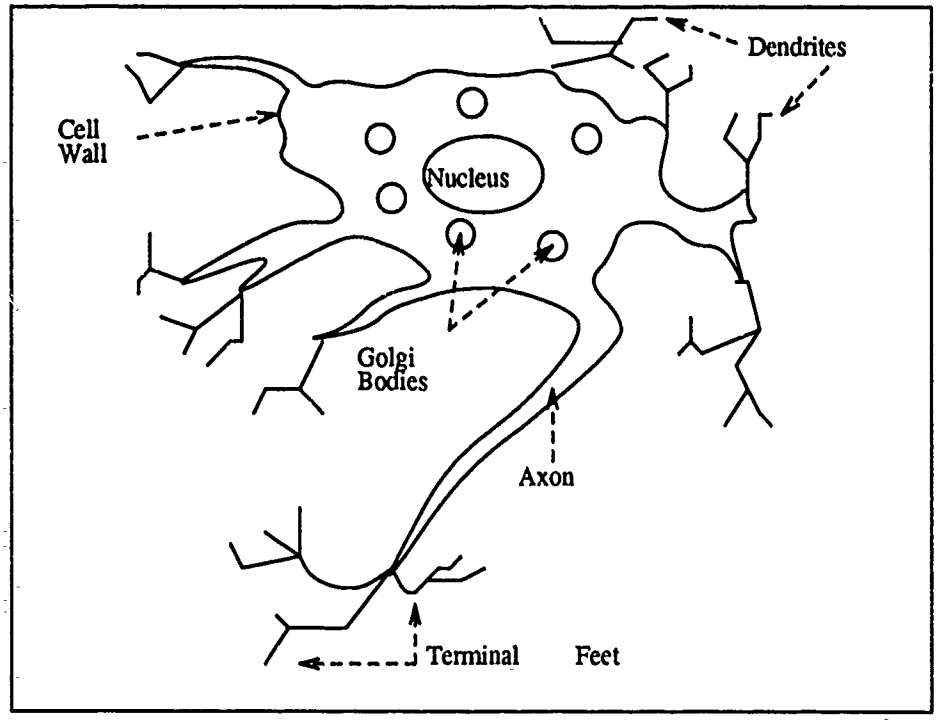

Figure 2.2. The Neuron (19:19)

a cell that has been modified to become a simple processing element whose primary function is to receive, process and transmit electrochemical signals across the brain's neural pathways (29:12) (19:17). The main modifications are the addition of dendrite and axon appendages to the cell body. The dendrites act as the input communication channels while the axon acts as the output channel. Each neuron is connected to the axon of many other neurons via its dendrites. These dendrite extensions allow a neuron to receive chemical neurotransmissions from other neurons at junctions called synapses (29:12). When the neuron receives these signals, it will become excited if the combined input signals exceed a threshold. When excited, the neuron will transmit an electrical signal along its axon, sending the signal to each attached neuron. The attached neurons may or may not become excited, depending on the strength of the connection between the neurons. Thus, it is in the synapse that the information is stored in the form of synaptic weights. Since there are between 10^{10} - 10^{11} neurons in the human brain and an estimated 10^{15} interconnections between these neurons, a vast amount of information can be stored and quickly processed (29:12).

2.6 *Artificial Neural Networks*

As a pattern recognition device, artificial neural networks attempt to assign an unknown pattern from the environment into one of a set of selected classes by emulating this structure of the neurons in the human brain (11:47). These networks can be thought of as massively parallel, interconnected networks of simple computing elements, called nodes, which seek to interact with the real world in way similar to a biological nervous system (10:251). As shown in figure 2.3, each node in the network performs a simple transformation of inputs from other nodes in the network, or from the environment, to produce a single output signal. This transformation can be via a

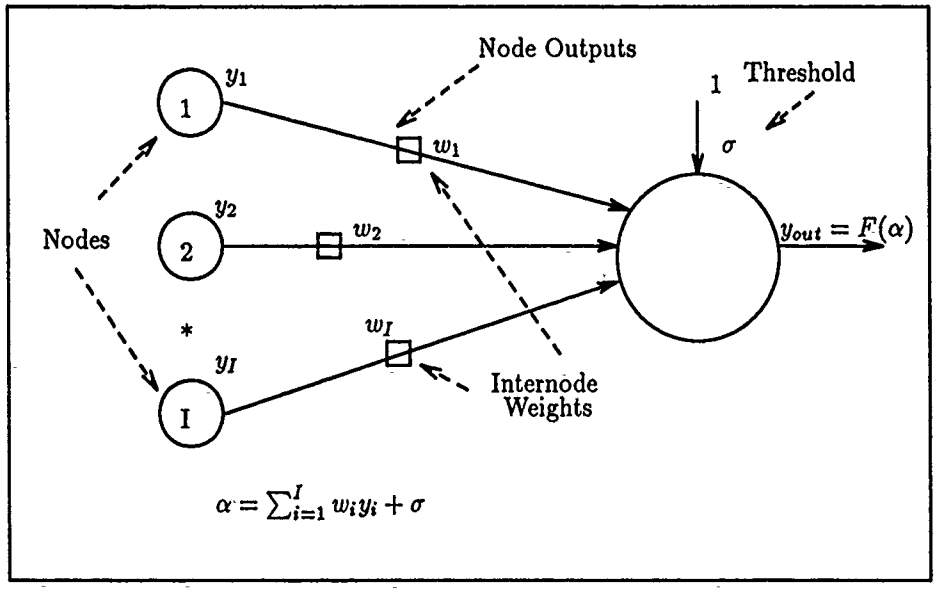

Figure 2.3. Artificial Neural Network Node (19:48)

linear function or a nonlinear function such as sigmoidal, gaussian, or threshold function. The signal output from this transformation is then fed to other nodes or interpreted as the output of the network.

The connections, or weights between the nodes, function in a manner similar to the axon-dendrite synaptic connection of biological neurons. That is, each weight has a "strength" associated with it which serves to either amplify or inhibit the signals transmitted between nodes along these connections. Typically, an artificial neural network will consist of one or more layers of nodes as shown in figure 2.4. The first layer of nodes serve to simply pass the input features, via weighted interconnections, to feature detector nodes in the second layer. These feature detector nodes will usually respond to certain features of the input data. Their responses are then passed, via another

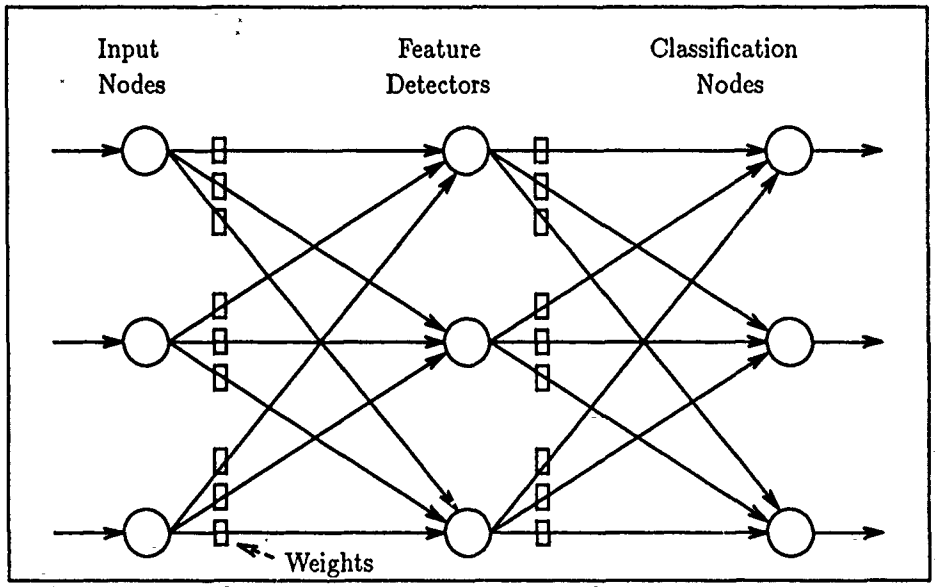

Figure 2.4. Multilayer Artificial Neural Network (6:75)

set of weighted interconnections, to the output layer which performs the classification task based on the outputs of the feature detectors. Through the proper interconnection of nodes and weights, and through the use of an appropriate transfer function, artificial neural networks have been developed which can accurately classify patterns ranging from phonemes to tanks (23:461-466) (20:1-7).

2.7 Historical Review

Biological neural networks have been studied for years. Ever since the 18th century, when Galvani investigated the connection between electricity and the frog's central nervous system, man has been seeking to unlock the secrets of the brain's computing power (19:5). From Santiago Cajal's discovery of the dense interconnection of neurons in the cortex to the first estimation of a neuron's transfer function via experiments with the Limulus' photoreceptors, man has been seeking a method of modeling the function of the brain in the form of artificial neural networks (19:6-31).

The first major milestone in the development of artificial neural networks came in 1943 from McCulloch and Pitts. They showed how neural-like networks, using a simple two state logical decision element which modeled the first order characteristics of a neuron, could compute a Boolean function (19:9). Since Turing later showed that any computable function could be computed with Boolean Logic, the basis for the development of computing machines, based on the principle of

using a dense connection of simple neural like elements, was established. However, McCulloch and Pitts did not show how a network made of these elements could be made to "learn" (22:152).

This problem of learning was addressed by Donald Hebb in 1949. Basically, Hebb proposed that the strength or weight between two neurons be increased whenever both the presynaptic and postsynaptic units were active simultaneously. These ideas remained untested due to the lack of technology capable of implementing these theories. (22:152-153).

M. Minsky and D. Edmonds were the first to actively implement Hebb's ideas in the form of a learning/computational machine developed in 1951. This machine was composed of tubes, motors and electrical clutches. The machine's memory was stored in the positions of control knobs by which the machine adjusted itself (22:153).

This milestone was followed by Rosenblatt's introduction of the perceptron in 1957. Basically, the perceptron is a single unit which produces an output only when the weighted sum of its inputs exceeds some preset threshold. The function of the perceptron was modeled on the first order characteristics of neurons. Using Hebb's ideas as a basis for developing his learning algorithms, Rosenblatt proved that the perceptron could learn anything it could represent (29:29). Rosenblatt also helped pioneer the simulation of the perceptron using digital computers and developed a set of rules that would allow the perceptron to learn (19:13). In reference to pattern recognition, Rosenblatt showed that a two layer perceptron could carry out any of the 2^{2^N} possible classifications of N binary inputs using 2^N perceptrons (22:158).

In 1959, Bernard Widrow invented the adaptive linear neuron (ADALINE) which, in a manner similar to Rosenblatt's perceptron, would output a signal only when the weighted sum of its inputs exceeded a preset threshold. The weight parameters of the ADALINE were adjusted over time using equations based on Hebb's original ideas (6:31). Widrow implemented ADALINE based systems that could predict the weather and balance a broom on a moving platform (19:12).

These developments led to an increased level of activity in the field of artificial neural networks until 1969. It was in this year that Minsky and Papert proved that the single layer perceptron could not classify patterns with features in disjoint regions in the feature space. An example of such a pattern is shown in figure 2.5. At this time, no method of updating the weights for any nodes except those weights attached to the output layer nodes for a multilayer perceptron had been established (6:29). Thus, all perceptron based networks at this time were limited to a single input layer and output layer with only one set of adjustable weights. Minsky's and Pappert's proof helped stifle neural network research until the middle 1970's.

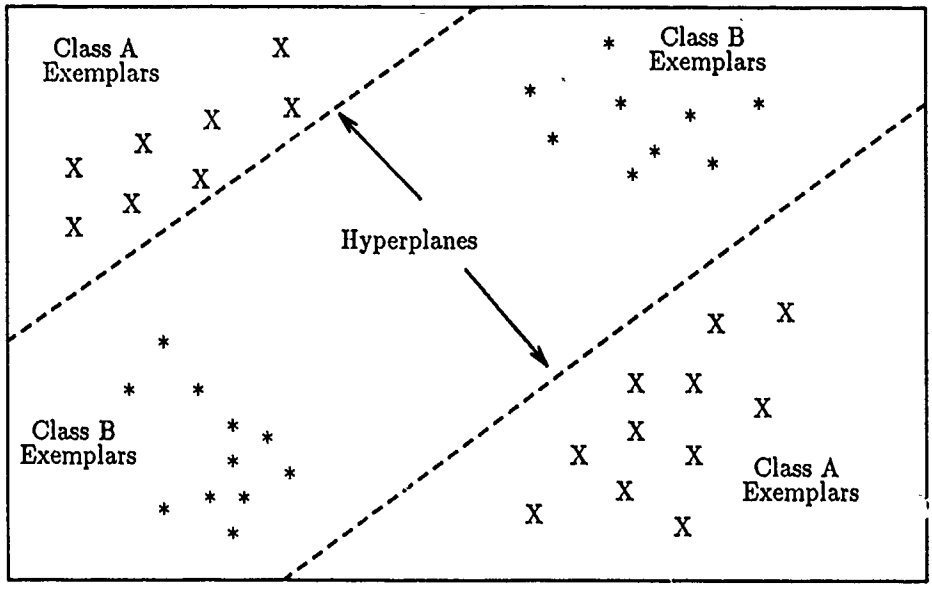

Figure 2.5. Disjoint Regions for Similar Classes (6:30)

Interest in artificial neural networks was rekindled in 1974. It was at this time that Paul Werbos introduced a method of updating the weights in the hidden layers of a multilayer perceptron network. These equations allow the multilayer perceptron to overcome the disjoint region problem suffered by the single layer perceptron network.

However, little work continued to be done in the field until 1982 when John Hopfield developed an artificial neural network capable of providing associative memory and solving optimization problems (6:37). This development led to increased activity in the field of artificial neural networks and the development of many different types of training rules and architectures.

2.8 Network Training

Artificial neural networks are not programmed, as are traditional computers, and there is little need for the development of application specific algorithms to perform the classification task. Artificial neural networks learn by example (6:11). In pattern recognition tasks, artificial neural networks are usually trained to produce a desired output whenever a known input is applied. This training is accomplished by applying input patterns to the network and allowing the network nodes to adjust their parameters in a predetermined fashion (29:22). That is, the artificial neural network is presented with data which characterize such patterns as images, speech signals and radar signals.

These networks are then allowed to adjust their internal parameters, such as weights, to allow the network to discover the distinguishing features needed to perform a classification task (6:13).

Basically, there are three primary methods of developing or training artificial neural networks. These methods consist of unsupervised training, supervised training, or combined training, a combination of supervised and unsupervised training.

2.8.1 Unsupervised Training In unsupervised training, the feature data from the environment are fed to the network. The interconnection weights between the nodes in the network are then arranged, or clustered, into positions reflecting the distribution of the training data (22:151-193). After training is completed, application of an input from a given class will produce a specific output. However, there is no way, before training, to predetermine the mapping from input to output. A Kohonen Self-Organizing feature map network, trained in this manner, has proven feasible for classifying speech patterns (1:1-7).

2.8.2 Supervised Training In supervised training, the feature data is presented to the network, along with the desired output pattern for that particular input pattern. The difference between the network output and the desired output, or error, is then calculated and used to adjust the network parameters, such as the weights linking the nodes, so that this error is minimized. This process is repeated continuously until the network is able to produce the mapping from the input pattern to the desired output pattern. A multi-layer perceptron network, developed with supervised training at the Air Force Institute of Technology, has proven capable of classifying tactical targets such as trucks, tanks, and jeeps (20).

2.8.3 Combined Training A combination of unsupervised and supervised training can also be used to develop an artificial neural network. In this type of network, the first layer is usually trained through unsupervised training, allowing the network parameters to be distributed according to the feature data. After stabilization of the first layer, the remaining nodes are then trained in a supervised fashion, to produce a desired signal from knowledge of the distribution of the network parameters in the first layer. A network trained in this manner has been studied by the Royal Naval Engineering College as a possible method of classifying radar signals (2:1-4).

2.9 Network Categorization

According to Lippmann, neural networks may be categorized as either a Probabilistic Classifier, an Exemplar Classifier, a Hyperplane Classifier, or a Kernel Classifier, depending upon the

method the network uses to perform classification (11:47-63).

2.9.1 Probabilistic Classifiers A Probabilistic Classifier neural network seeks to classify patterns by using probability distributions to maximize the probabilities associated with a classification as shown in figure 2.6. As such, these networks require enough training data to allow an assump-

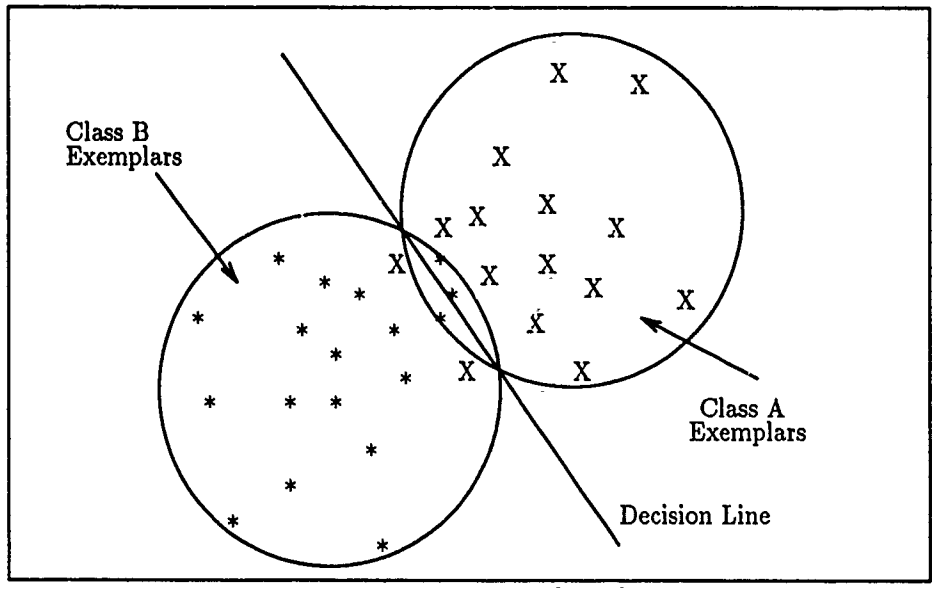

Figure 2.6. Probabilistic Network Decision Regions (11:49)

tion of the probability distributions of the patterns to be made. Either unsupervised or supervised training is then used to train the network. These networks perform best when the assumed distributions are accurate models of the test data. An example of this type of classifier is the Bayes' Classifier or the Probabilistic Neural Network (9:1-7) (24).

2.9.2 Exemplar Classifiers An Exemplar Classifier neural network classifies unknown feature data based on a nearest-neighbor calculation with the training data fixed in the feature space as shown in figure 2.7. These nearest-neighbor calculations allow an estimation of the conditional probability density functions for each class (8:166-169). That is, the closer an unknown pattern is to a known pattern, the stronger the probability that the two patterns represent the same class (10:3). An example of this type of network is the K-Nearest-Neighbor classifier. Networks of this type can be trained rather quickly through either supervised or unsupervised methods, but can require large amounts of memory and long computational times for classification (11:49).

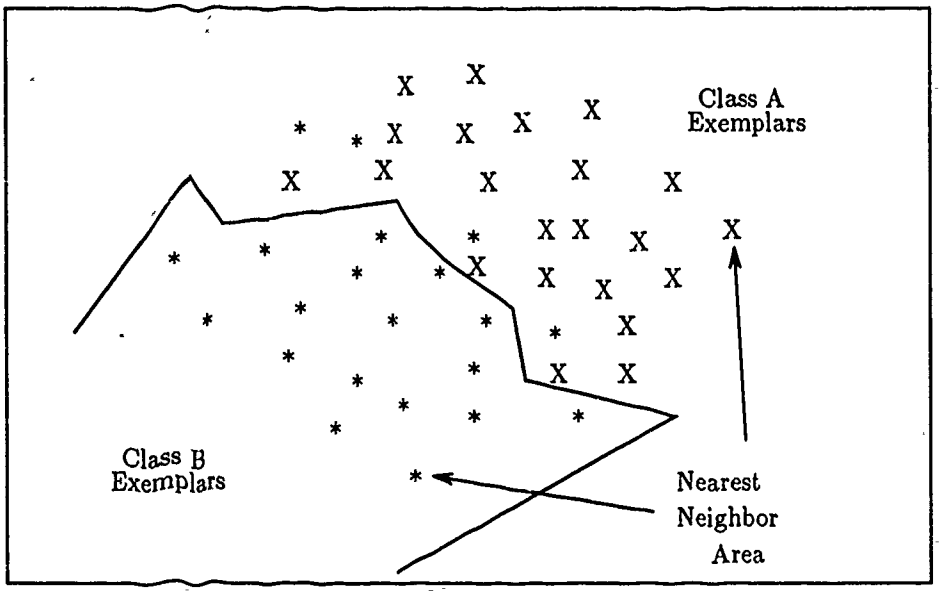

Figure 2.7. Exemplar Network Decision Regions (11:49)

2.9.3 Hyperplane Classifiers A Hyperplane Classifier neural network forms decision regions by using hyperplanes to partition the feature space into the regions of interest as shown in figure 2.8. Perhaps one of the more studied networks of this type is the multilayer perceptron network (19:44-63). The function of this network is based on the property that any multivariate function can be approximated by a finite superposition of sigmoidal functions (5:303-313). This property can be implemented with a single hidden layer neural network, where each node within the network uses a sigmoidal function to calculate its output from the sum of the product of its inputs and their associated weights. This network is usually trained under the supervised training method, using a technique called backpropagation, to minimize the error between a given input and a desired output (19:104-114). However, the amount of time required to train these networks can take hours (23:466) (14:4). Once trained, these networks usually provide high accuracy for pattern classification while requiring relatively short computational times (11:49).

2.9.4 Kernel Classifiers A Kernel Classifier neural network uses overlapping kernel functions to create complex receptive-field decision regions over the feature space as shown in figure 2.9. One of the more recent types of Kernel function classifiers is the radial basis function classifier. The function of this network is based on the property that any multivariate function can be reasonably approximated using a linear combination of radial basis functions centered on the data points, or a subset of the data points (17:143-167) (15:1-20) (16:978-980). This translates

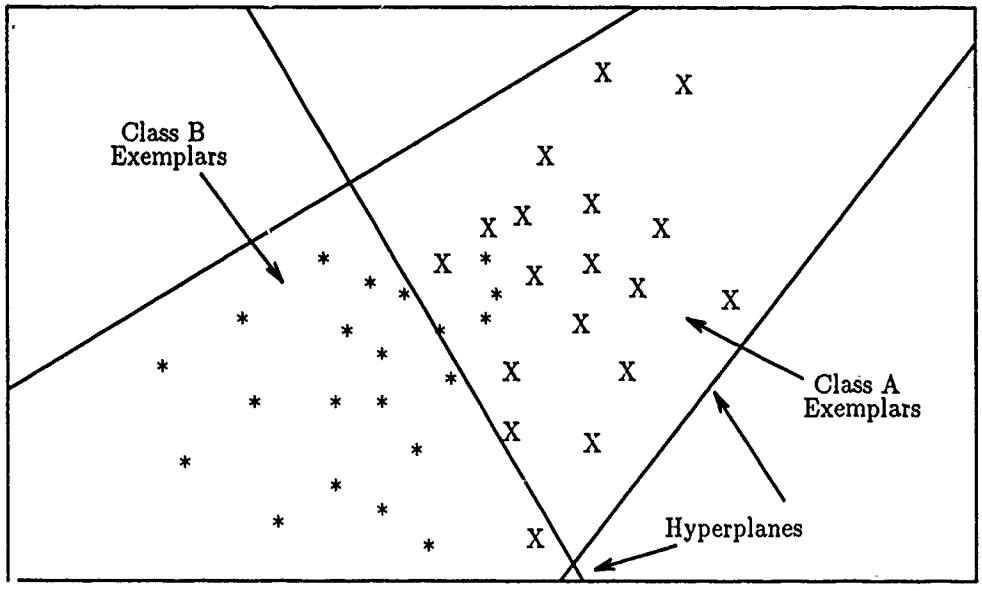

Figure 2.8. Hyperplane Network Decision Regions (11:49)

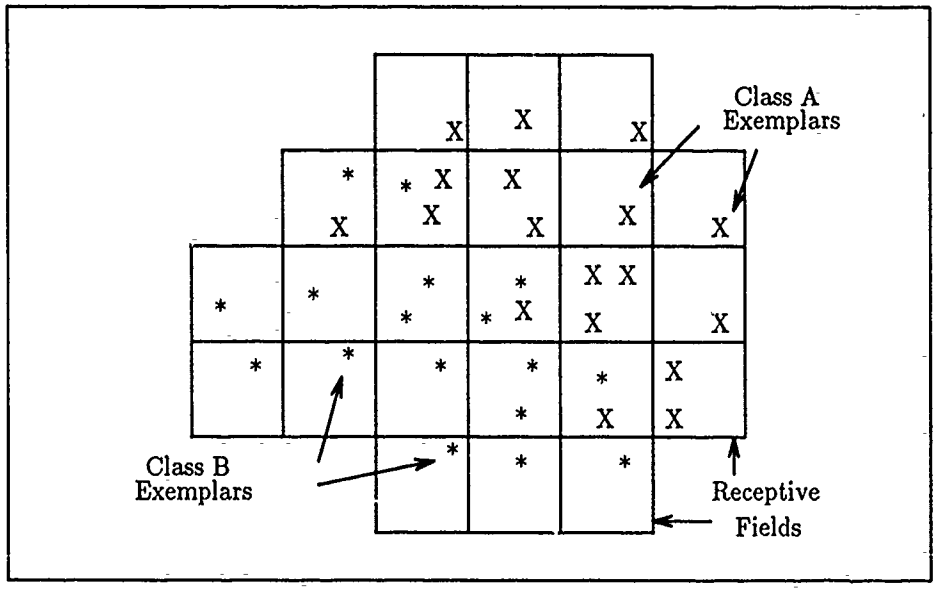

Figure 2.9. Kernel Classifier Network Decision Regions (11:49)

into the establishment of a single hidden layer network, with the nodes in the hidden layer using radial basis functions to transform their inputs to outputs. These networks have been successfully trained to classify phoneme data for speech processing (23:461-466) (18:437-439) (14:1-14). The most appealing characteristic of the radial basis function Kernel Classifier networks is the almost instantaneous training times involved with setting the network parameters (23:461-466) (18:432-439). These networks can also be made to adapt to new data by adding additional nodes as required (12:3). Generally, Kernel Classifiers can be trained relatively quickly through either supervised or unsupervised methods and have intermediate memory and computational requirements (11:49).

2.10 Summary

Artificial neural networks, may provide military weapon systems with the ability to accurately characterize their operating environment in real time. These networks seek to emulate the function of the brain; using a dense connection of simple computational elements called nodes to perform pattern classification. These networks can be trained in either a supervised or unsupervised fashion, or both, and can be categorized by the method used to classify the data.

III. Mathematical Analysis

3.1 Introduction

The main function of a pattern recognition system is to make a decision as to which class an unknown pattern belongs (27:39). This decision is usually based upon the application of decision functions which segment the feature space. Hyperplane Classifiers use linear decision functions, in the form of hyperplanes, to partition the feature space while Kernel Classifiers use higher order decision functions, in the form of hyperspheres or hyperellipsoids, to partition the feature space. This chapter begins with a discussion of Hyperplane Classifiers, including an analysis of the objective functions used to implement these classifiers as neural networks and their parameter update equations. After discussing Kernel Classifiers in general, the relationship between pattern recognition, functional interpolation and probability estimation are examined. This chapter concludes with the development of the training algorithms used to implement Kernel Classifiers as neural networks.

3.2 Hyperplane Classifiers

3.2.1 Decision Functions Consider the two-dimensional exemplars representing two pattern classes shown in figure 3.1. As can be seen, the two patterns can be separated by a line drawn in

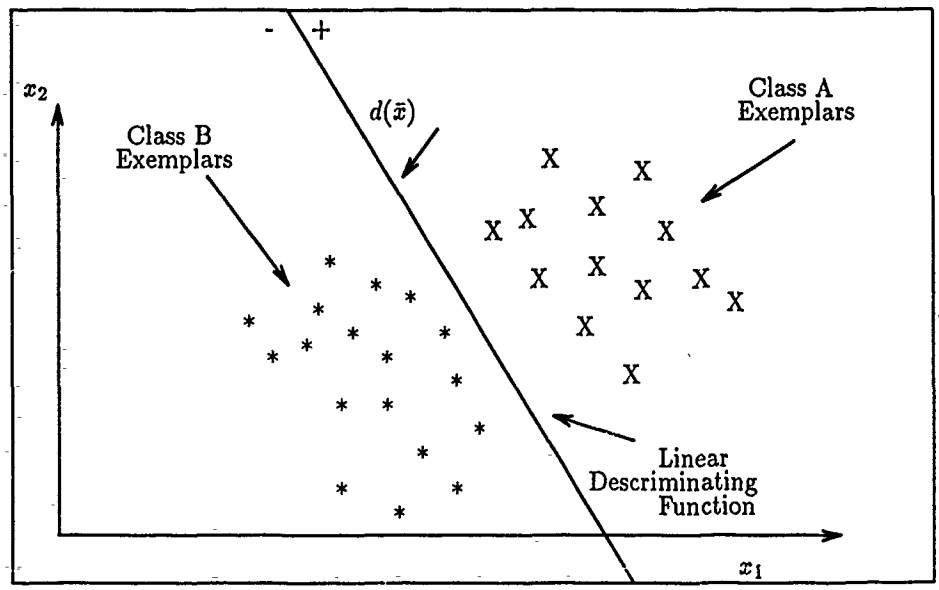

Figure 3.1. Linear Decision Function(27:40)

the feature space. The general equation for this line is

$$d(\bar{x}) = w_1 x_1 + w_2 x_2 + \sigma \tag{3.1}$$

Here

$d(\bar{x})$ is the linear decision function

$\bar{w}$ is a vector containing the weights or scaling coefficients

$\bar{x}$ is the pattern vector containing the feature values

σ is an offset or threshold

From this figure, $d(\bar{x})$ can be positioned such that any pattern vector, $\bar{x}$, belonging to class A will yield a positive quantity when the features are substituted into $d(\bar{x})$ while any pattern belonging to class B will yield a negative quantity (27:39). Thus, $d(\bar{x})$ can be considered a linear decision function since, given an unknown pattern $\bar{x}$, $d(\bar{x})$ will be positive for class A and negative for class B. When the feature space has K dimensions, the general equation for the linear decision function is of the form

$$d(\bar{x}) = w_1 x_1 + w_2 x_2 + \ldots w_K x_K + \sigma \tag{3.2}$$

The main problem associated with the linear decision function is to find a set of weights associated with the decision function which allows the feature space to be partitioned in a manner which separates the classes (27:48).

3.2.2 Network Implementation The characteristics of the linear decision functions can be modeled as a neural-type element by assigning, to the neural element, the hyper-sigmoidal transfer function

$$y(\bar{x}) = [1 + e^{-\phi(\bar{x})}]^{-1} \tag{3.3}$$

where

$$\phi(\bar{x}) = \sum_{k=1}^{K} x_k w_k + \sigma \tag{3.4}$$

This nonlinear function, as shown in figure 3.2, is a nondecreasing function in which the output for $\phi(\bar{x}) < 0$ is less than 1/2 while for $\phi(\bar{x}) > 0$ the output is greater than 1/2. This model

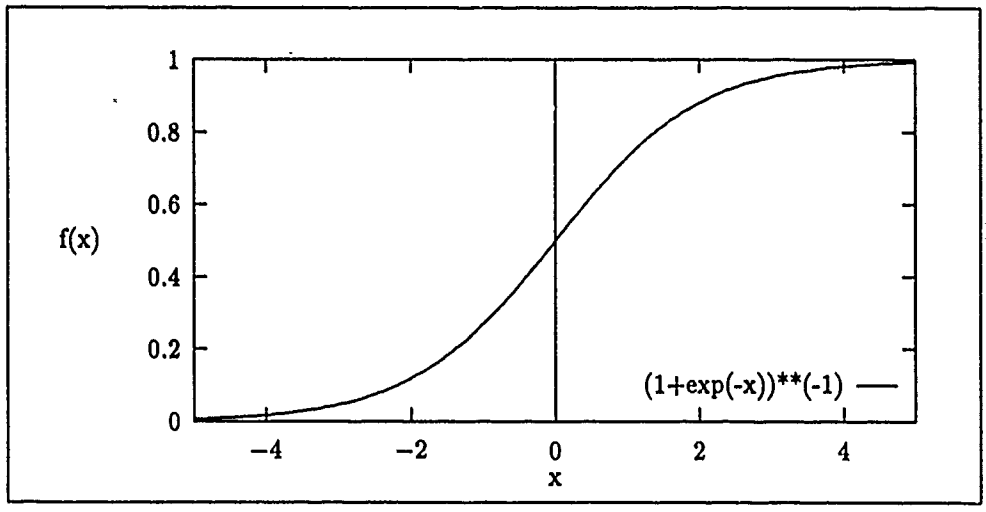

Figure 3.2. Sigmoidal Transfer Function

is based on the Rosenblatt's perceptron introduced in 1957. Again, as with the linear decision function described above, the main problem associated with the perceptron is to set the weights, w_n, and offset, σ, such that the feature space is partitioned to allow proper classification (19:50). These parameters can be established by performing a gradient descent using a method known as backpropagation.

3.2.3 Network Training The training method most commonly used with this perceptron-based Hyperplane Classifier is the method of backpropagation. In this method, the network is presented a pattern vector and allowed to produce its own output. This output is then compared with the desired output using some predefined objective function. If there is no major difference, then no learning takes place. Otherwise the weights and offsets for each node are changed in a manner which optimizes the classification objective function (22:322). This optimization is performed via an incremental gradient descent on the surface of the weight space whose height at any point is equal to a measure of the performance of the classification objective function (22:322). The three main types of classification objective functions are Mean Square Error (MSE), Cross Entropy (CE) and Classification Figure of Merit (CFM) (28:217).

3.2.3.1 Mean Square Error (MSE) Objective Function The Mean Square Error (MSE) objective function seeks to minimize the mean squared error between the network's actual output and the desired output for each classification node in the output layer (28:217). Suppose a particular pattern recognition problem had N classes and the network was developed such that each node in

the output layer represented only one of the N classes. Let d_n be the desired output for the n^{th} node for a given input pattern. The MSE then is defined as

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)^2 \tag{3.5}$$

Usually during training, d_n is taken to be 1 for the node responsible for a class and the 0 for the rest of the output nodes. This objective function was the first to be implemented in the study of artificial neural networks and is the most widely used of the three objective functions. Any network trained using this objective function will make a classification based on the Bayes' optimal descriminant (21). However, there are certain properties of this function which don't permit accurate classifications in all cases (3). As shown in Appendix A, there are certain areas in the feature space in which the mean square error is higher for a correct classification than for an incorrect classification (28:218). This implies there are certain areas of the feature space in which backpropagation according to the MSE function may fail to separate the classes correctly.

A network implemented using the MSE objective function will have its parameters set to minimize the MSE. Thus, the general update equation for the network's parameters will have the following form:

$$w_j^+ = w_j^- - \eta(\frac{\partial MSE}{\partial w_j}) \tag{3.6}$$

Here, η is a constant which controls the update rate. The incremental update equations for each of the parameters of the network shown in figure 3.3 are derived in Appendix B and summarized below. The update equation for a weight linking node M in layer 2 to a node N in layer 3, w_{MN}, is

$$w_{MN}^+ = w_{MN}^- - \eta(y_N - d_N)y_N(1 - y_N)y_M \tag{3.7}$$

while the update rule for the offset of the node N, σ_N, in layer 3 is

$$\sigma_N^+ = \sigma_N^- - \eta(y_N - d_N)y_N(1 - y_N) \tag{3.8}$$

The update equation for a weight linking node L in layer 1 to node M in layer 2, w_{LM}, is

$$w_{LM}^+ = w_{LM}^- - \eta\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)w_{Mn}y_M(1 - y_M)y_L \tag{3.9}$$

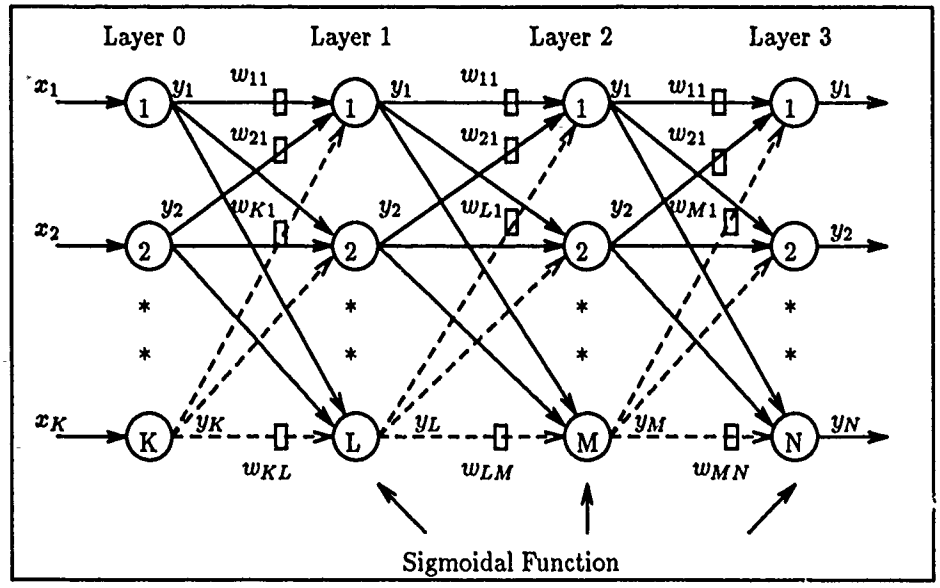

Figure 3.3. Two Hidden Layer Sigmoidal Network Topology

while the update equation for the offset of node M in layer 2, σ_M, is

$$\sigma_M^+ = \sigma_M^- - \eta \sum_{n=1}^{N} (y_n - d_n) y_n (1 - y_n) w_{Mn} y_M (1 - y_M) \tag{3.10}$$

The update equation for a weight linking node K in layer 0 to node L in layer 1, w_{KL}, is

$$w_{KL}^+ = w_{KL}^- - \eta \sum_{n=1}^{N} (y_n - d_n) y_n (1 - y_n) [\sum_{m=1}^{M} w_{mn} y_m (1 - y_m) w_{Lm} y_L (1 - y_L) y_K] \tag{3.11}$$

while the update equation for the offset of node L in layer 1, σ_L, is

$$\sigma_L^+ = \sigma_L^- - \eta \sum_{n=1}^{N} (y_n - d_n) y_n (1 - y_n) [\sum_{m=1}^{M} w_{mn} y_m (1 - y_m) w_{Lm} y_L (1 - y_L)] \tag{3.12}$$

3.2.3.2 Cross Entropy (CE) Function The Cross Entropy (CE) Function considers the actual value of an output node as the probability that the ideal binary output state of the node is a 1. The CE function seeks to minimize the difference between the actual output , y_n, and the ideal output, d_n, by minimizing the cross entropy between the actual and desired probability density functions driving the output nodes (28:217). That is

$$CE = -\frac{1}{N}\sum_{n=1}^{N}[d_n \log(y_n) + (1 - d_n)\log(1 - y_n)] \tag{3.13}$$

Again during training, d_n is usually set to 1 for the node responsible for the correct class and to 0 for the rest of the output nodes. As shown in Appendix A, the CE function is also characterized by some areas in the feature space in which the CE is greater for an correct classification than for an incorrect classification (28:218). This implies there are certain areas of the feature space in which backpropagation according to the CE function may fail to separate the classes correctly.

A network implemented to use the CE function as the classification objective function will have its parameters set to minimize the CE. Thus, the general update equation for the network's parameters will have the following form:

$$w_j^+ = w_j^- - \eta(\frac{\partial CE}{\partial w_j}) \tag{3.14}$$

Here, η is a constant which controls the learning rate. The incremental update equations for each of the parameters of the network shown in figure 3.3 are derived in Appendix B and summarized below. The update equation for a weight linking node M in layer 2 to a node N in layer 3, w_{MN}, is

$$w_{MN}^+ = w_{MN}^- + \eta(d_N - y_N)y_M \tag{3.15}$$

while the update rule for the offset of the node N, σ_N, in layer 3 is

$$\sigma_N^+ = \sigma_N^- + \eta(d_N - y_N) \tag{3.16}$$

The the update equation for a weight linking node L in layer 1 to node M in layer 2, w_{LM}, is

$$w_{LM}^+ = w_{LM}^- + \eta\sum_{n=1}^{N}(d_n - y_n)w_{Mn}y_M(1 - y_M)y_L \tag{3.17}$$

while the update equation for the offset of node M in layer 2, σ_M, is

$$\sigma_M^+ = \sigma_M^- + \eta\sum_{n=1}^{N}(d_n - y_n)w_{Mn}y_M(1 - y_M) \tag{3.18}$$

The update equation for a weight linking node K in layer 0 to node L in layer 1, w_{KL}, is

$$w_{KL}^{+} = w_{KL}^{-} + \eta \sum_{n=1}^{N} (d_n - y_n)[\sum_{m=1}^{M} w_{mn} y_m (1 - y_m) w_{Lm} y_L (1 - y_L) y_K] \tag{3.19}$$

while the update equation for the offset of node L in layer 1, σ_L, is

$$\sigma_L^{+} = \sigma_L^{-} + \eta \sum_{n=1}^{N} (d_n - y_n)[\sum_{m=1}^{M} w_{mn} y_m (1 - y_m) w_{Lm} y_L (1 - y_L)] \tag{3.20}$$

3.2.3.3 Classification Figure of Merit (CFM) Objective Function The Classification Figure of Merit (CFM) objective function was introduced by Waibel to minimize the classification errors due to the characteristics of the MSE and CE objective functions (28). This CFM function does not consider the notion of an ideal output during training. This function is merely concerned with forcing the correct node to be the maximum output node for the correct input features. The CFM objective function first compares the activation level of each of the output nodes to the output node which should have the highest activation. The CFM then applies a sigmoidal function to differences between the activation levels for each of the nodes as follows:

$$CFM = \frac{1}{N-1} \sum_{n=1_{n \neq c}}^{N} \alpha[1 + e^{(-\beta \delta_n + \zeta)}]^{-1} \tag{3.21}$$

where $\delta_n = y_c - y_n$

y_c = response of the correct node

y_n = response of the incorrect node

N = total number of output nodes or classes.

α = sigmoid scaling parameter.

β = sigmoid discontinuity parameter.

ζ = sigmoid lateral shift parameter.

The application of the sigmoidal function keeps the network from trying to produce ideal values as the CFM function yields decreasingly marginal updates for increasingly ideal output patterns. Also, in order to keep the network from attempting to learn extreme statistical outliers of a class, the CFM seeks to apply decreasing marginal penalties for increasingly bad misclassifications (28). As shown in Appendix A, these characteristics may allow the CFM objective function to reduce the areas within the feature space in which the CFM is higher for an incorrect response than for a correct response.

A network implemented using the CFM objective function will have its parameters set such that the CFM objective function is maximized. Thus, the general form for the update equation of any parameter w_j of the network will be

$$w_j^+ = w_j^- + \eta(\frac{\partial CFM}{\partial w_j}) \tag{3.22}$$

Again, η is a constant which controls the learning rate. The incremental update equations for the parameters of the network, shown in figure 3.3, are derived in Appendix B and summarized below. The update equation for a weight linking node M in layer 2 to an incorrect node N in layer 3, w_{MN}, is

$$w_{MN}^+ = w_{MN}^- - \eta z_N(1-z_N)y_N(1-y_N)y_M \tag{3.23}$$

while the update equation for the offset of the incorrect node N, σ_N, is

$$\sigma_N^+ = \sigma_N^- - \eta z_N(1-z_N)y_N(1-y_N) \tag{3.24}$$

The update rule for the weight linking node M in layer 2 to the correct node C in layer 3, w_{MC}, is

$$w_{MC}^+ = w_{MC}^- + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)y_C(1-y_C)y_M \tag{3.25}$$

while the update rule for the offset of the correct node C, σ_C, in layer 3 is

$$\sigma_C^+ = \sigma_C^- + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)y_C(1-y_C) \tag{3.26}$$

The the update equation for a weight linking node L in layer 1 to node M in layer 2, w_{LM}, is

$$w_{LM}^+ = w_{LM}^- + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)[y_c(1-y_c)w_{Mc} - y_n(1-y_n)w_{Mn}]y_M(1-y_M)y_L \tag{3.27}$$

while the update equation for the offset of node M in layer 2, σ_M, is

$$\sigma_M^+ = \sigma_M^- + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)[y_c(1-y_c)w_{Mc} - y_n(1-y_n)w_{Mn}]y_M(1-y_M) \tag{3.28}$$

The update equation for a weight linking node K in layer 0 to node L in layer 1, w_{KL}, is

$$\begin{aligned} w_{KL}^+ &= w_{KL}^- + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)[y_c(1-y_c)\sum_{m=1}^{M} w_{mc} \\ &\quad - y_n(1-y_n)\sum_{m=1}^{M} w_{mn}]y_m(1-y_m)w_{Lm}y_L(1-y_L)y_K \end{aligned} \tag{3.29}$$

while the update equation for the offset of node L in layer 1, σ_L, is

$$\begin{aligned} \sigma_L^+ &= \sigma_L^- + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)[y_c(1-y_c)\sum_{m=1}^{M} w_{mc} \\ &\quad - y_n(1-y_n)\sum_{m=1}^{M} w_{mn}]y_m(1-y_m)w_{Lm}y_L(1-y_L) \end{aligned} \tag{3.30}$$

3.3 Kernel Classifiers

3.3.1 Decision Functions Consider the two-dimensional exemplars representing the two classes of patterns shown in figure 3.4. As can be seen, the two patterns can be separated by placing variable diameter circles around the data points corresponding to each class. The placement of these circles corresponds to a partitioning of the feature space into receptive fields with each circle responding to a pattern only when the features falls within its radius. The general equation for a decision function for this type of pattern recognition system is given by

$$d(\bar{x}) = \sum_{j=1}^{K} w_{jj}x_j^2 - \sum_{i=1}^{K} w_i x_i + w_{K+1} \tag{3.31}$$

for a K-dimensional feature vector (27:50). Again, the weights, w_{jj}, w_i, and w_{K+1} represent the coefficients of the decision function. This decision function can be written in matrix form as

$$d(\bar{x}) = \bar{x}A\bar{x}^T - \bar{x}B + c \tag{3.32}$$

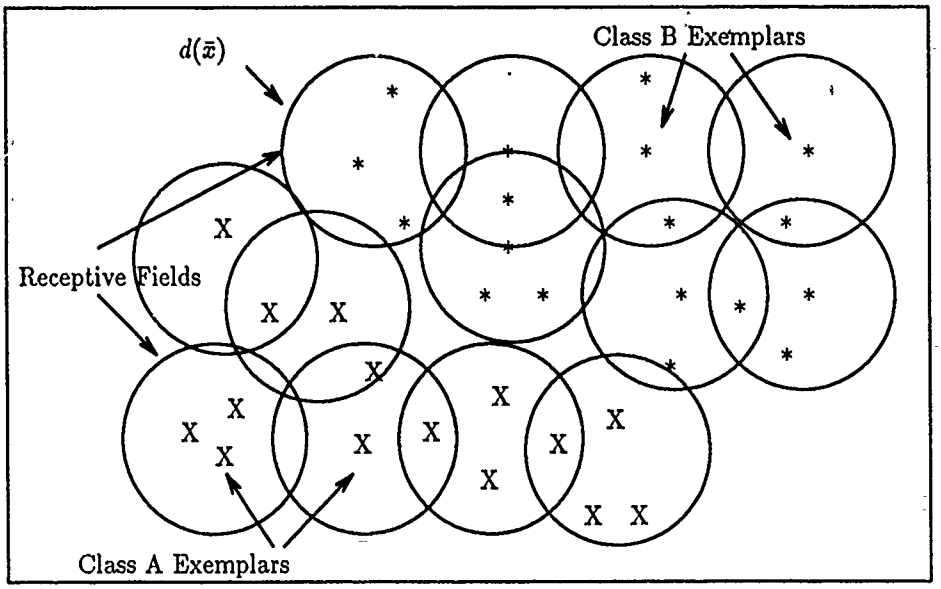

Figure 3.4. Circular Decision Functions

Where $\bar{x}^T$ = the transpose of $\bar{x}$, a K-dimensional vector containing the input features.

$$\bar{x} = [x_1, x_2, \ldots, x_K]$$

Also A is a K by K diagonal matrix containing the coefficients of the squared input features.

$$A = \begin{vmatrix} w_{11} & 0 & \ldots & 0 \\ 0 & w_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \ldots & w_{KK} \end{vmatrix}$$

B^T is a k-dimensional vector containing the weights or coefficients for linear input feature terms.

$$B^T = [w_1, w_2, \ldots, w_K]$$

Finally, C is a constant.

The coefficients of the A matrix determine the shape of the decision boundaries. If A is the identity matrix, the decision functions become hyperspheres. When A is positive definite, the decision functions become hyperellipsoids and when A is positive semidefinite the decision functions become a hyperellipsoid cylinder (27:52). Again, the main problem associated with these high order decision functions is to find a set of coefficients, or weights, associated with the decision function which allows the feature space to be partitioned in a manner which separates the classes (27:48).

3.3.2 Network Implementation The characteristics of the hyperspherical decision functions can be modeled as a neural type element by assigning, to the neural element, the gaussian transfer function

$$y(\bar{x}) = e^{-\phi(x)} \tag{3.33}$$

where

$$\phi(\bar{x}) = \sum_{k=1}^{K} \frac{(x_k - w_k)^2}{2\sigma_k^2} \tag{3.34}$$

Here

$x_k = k^{th}$ dimension of the input pattern vector $\bar{x}$

$w_k = k^{th}$ dimension of the weight vector $\bar{w}$

σ_k = spread or threshold in the k^{th} direction

The only task left is to determine how to architect the network to partition the feature space with these neural elements and perform a task of pattern recognition. This architecture can be derived by applying the theory of approximating multivariate functions using Radial Basis Functions (RBFs).

3.3.3 Functional Approximation According to Powell (17), the real multivariable interpolation problem is, given P different points $(\bar{x}_p; p = 1, 2, \ldots, P)$ in a K-dimensional space, and P real numbers $(d_p; p = 1, 2, \ldots, P)$, determine a function, $f(\bar{x})$ from $\Re^K$ into $\Re$ that satisfies the interpolation conditions

$$f(\bar{x}_p) = d_p \tag{3.35}$$

for $(p = 1, 2, \ldots, P)$. This function, $f(\bar{x})$, can be decomposed into a linear combination of radial basis functions.

$$f(\bar{x}) = \sum_{p=1}^{P} \lambda_p \phi(\| \bar{x} - \bar{w}_p \|) \tag{3.36}$$

where

$\bar{x} \in$ of $\Re^K$ and $p = 1, 2, \ldots, P$

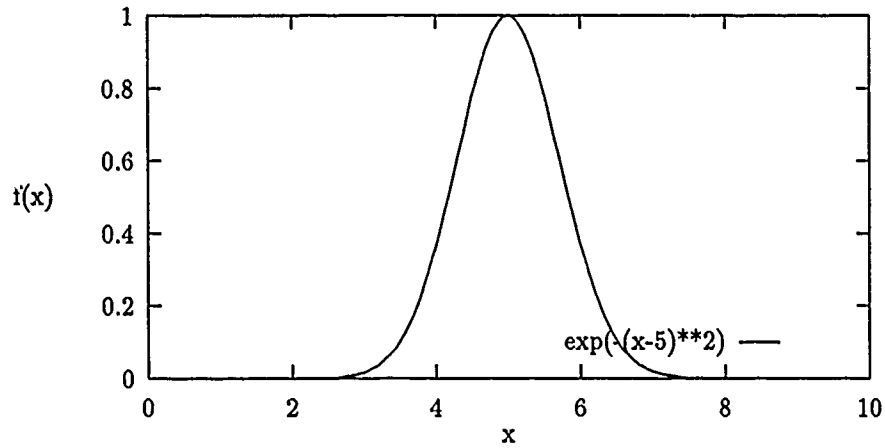

Figure 3.5. One Dimensional Radial Basis Function

λ_p = scaling coefficient

The $\| \ldots \|$ is usually taken to be the Euclidean Norm while the $\bar{w}_p$ are the centers of the basis functions (4:2). A radial basis function is a function, such as the gaussian shown in figure 3.5, which is symmetric in all radial directions and approaches zero as the distance from the center increases. Figure 3.6 shows the reconstruction of a periodic square wave from a linear combination of these gaussian radial basis functions.

This functional approximation can be implemented with a neural network architecture if one considers the task of pattern recognition as a functional mapping from the set of data points to the output of the network. Suppose a set of P exemplars characterize the pattern recognition problem. That is, sampling of the environment has led to P data points for which the desired classification of each data point is known. Further, suppose that each exemplar is a K-dimensional vector.

$$\bar{x} = [x_1, x_2, \ldots, x_K] \tag{3.37}$$

Thus the set of P, K-dimensional exemplars characterize the pattern recognition problem. Now, suppose that this set of exemplars can be classified into M distinct classes. This classification can be thought of as a mapping from a K-dimensional feature space, where the exemplars reside, into an M-dimensional space where the classification takes place. Let the desired classification vector of a given exemplar, say $\bar{x}_p$, be labeled as $\bar{d}_m$. If the classification problem is considered as a mapping problem, a function needs to be determined that produces the following result for each of the P

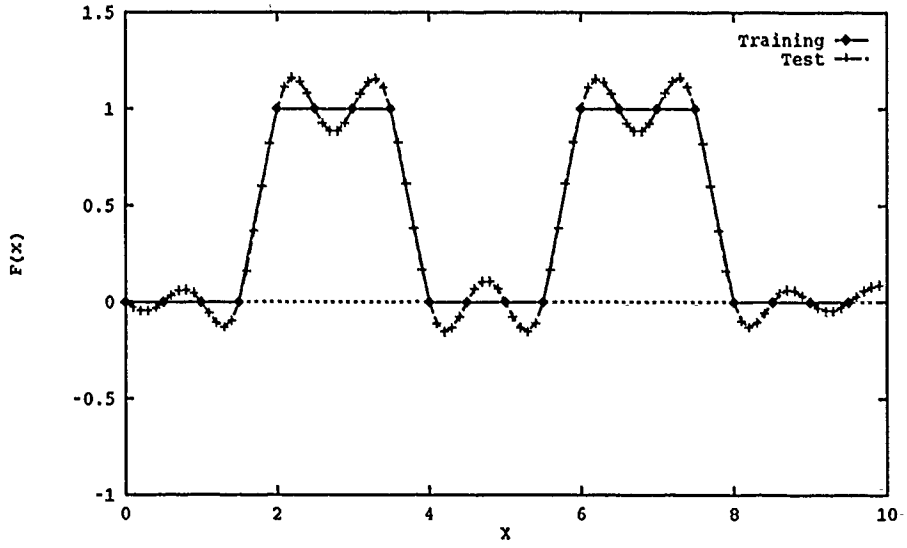

Figure 3.6. Square Wave Reconstructed Via Radial Basis Functions

exemplar vectors, $\bar{x}_p$:

$$f(\bar{x}_p) = \bar{d}_m \tag{3.38}$$

This function, once found, will produce the desired mapping from the exemplar points to their classifications. Applying the theory of approximating multivariate functions with a set of radial basis functions, this function-finding problem can be modeled as a real multivariable interpolation problem. Using the gaussian function as the radial basis function, the approximation can then be written as

$$f(\bar{x}_p) = \sum_{p=1}^{P} \lambda_p e^{-[\sum_{k=1}^{K} \frac{(x_{pk}-w_{pk})^2}{2\sigma_{pk}^2}]} \tag{3.39}$$

This approximation can be implemented as a neural network architecture as shown in figure 3.7 where the λ_p's are implemented as weights, w_{lm}, linking the nodes in the hidden layer, layer 1, to the nodes in the output layer, layer 2. Here, the nodes in the hidden layer have the gaussian radial basis function as their transfer function. The nodes in the output layer compute an linear

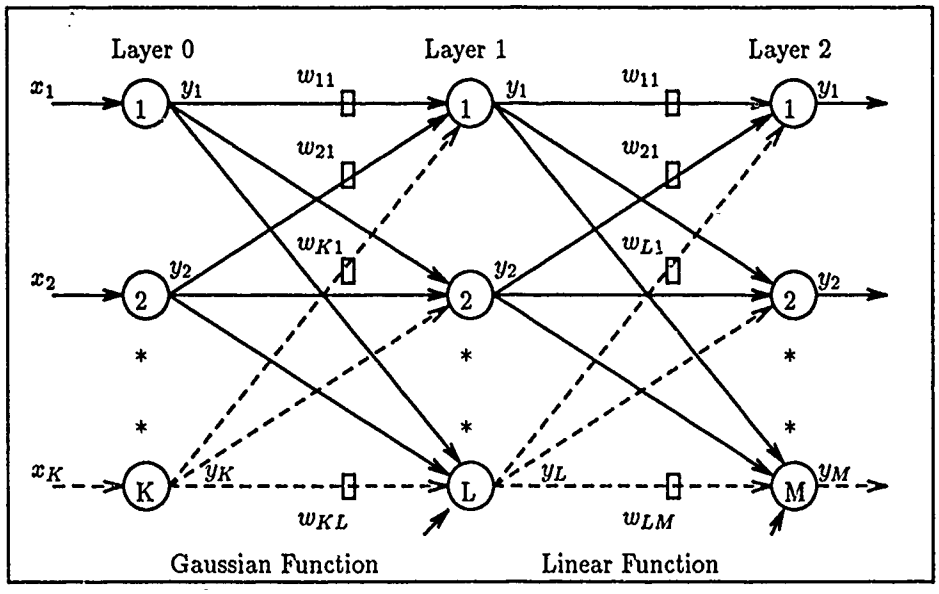

Figure 3.7. RBF Neural Network Topology

combination of outputs from the nodes in the hidden layer. Thus, the overall mapping function, for a single input pattern, takes the form of

$$y_m = \sum_{l=1}^{L} w_{lm} y_l \tag{3.40}$$

where

$$y_l = e^{-[\sum_{k=1}^{K} \frac{(x_k - w_{kl})^2}{2\sigma_{kl}^2}]} \tag{3.41}$$

Once this network is established, the outputs can then be considered as a mapping from the input space to the output space. This mapping can also be considered a probability density estimation, via the technique of Parzen Windows, of the input pattern given a particular output.

3.3.4 Density Estimation The task of pattern recognition is often considered as a problem of assigning an unknown sample, say $\bar{x}$, to one of J classes. Bayes' rule for this classification problem, called Bayes' optimal discriminant rule, assigns $\bar{x}$ to class i if

$$P(\bar{x}/G_i)P(G_i) > P(\bar{x}/G_j)P(G_j) \tag{3.42}$$

for all $i \neq j$ In order to implement this rule, the underlying probability density functions (pdf's) for $P(\bar{x}/G_j)$ and $P(G_j)$ for all J classes must be known. One method of estimating these density functions is to group the elements of a given class into a histogram. The problem with this method is, if the rectangular cells, or bins, into which the data is grouped are too small, the estimate may not be smooth. If the rectangular cells are too big, the fine details of the distribution may be lost. These smoothness problems can be overcome by using the method of estimating density functions through Parzen Windows or kernel estimators (8:162).

As shown in figure 3.8, the Parzen Window estimate of the density function, $P(\bar{x}/G_j)$, solves

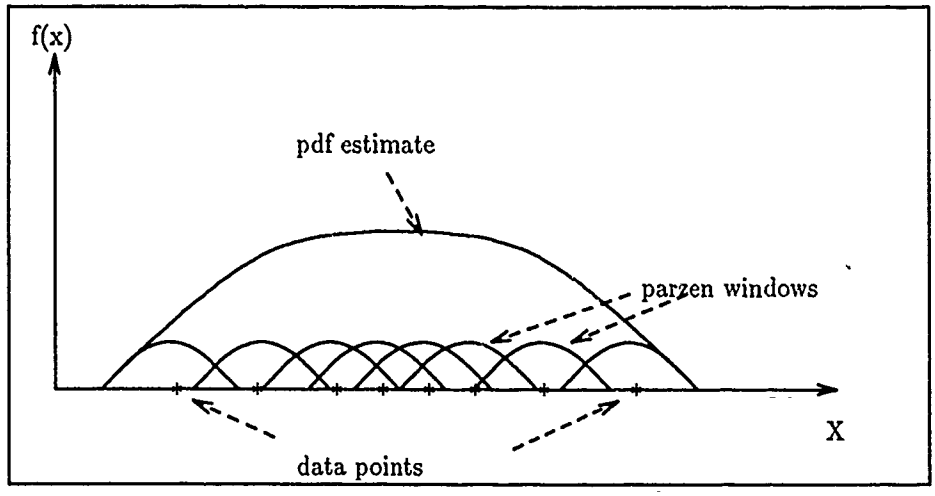

Figure 3.8. Parzen Window PDF Estimation (8:164)

this smoothness problem by assuming that each value of the data, occurring in the sample set, also raises the probability of any value occurring close to that value of the data. By centering a kernel function at each data point, the final value of the estimate can be obtained by summing together all the contributions from each value of the sample data (8:162). That is the Parzen Window estimate has the form

$$P(\bar{x}/G_J) = \frac{1}{N_J} \sum_{j=1}^{N_J} [\frac{1}{h^K} \phi(\frac{\bar{x} - \bar{x}_j}{h})] \tag{3.43}$$

where

K = the number of dimensions

N_J = the number of data points in class J

h = a function of N_J referred to as the window width.

$\phi(\bar{x})$ = the kernel function.

To implement this estimation, an appropriate kernel function and window width must be selected. As shown in Appendix C, the gaussian radial basis function of

$$\phi(\|\bar{x} - \bar{x}_n\|) = (2\pi\sigma^2)^{-\frac{K}{2}} e^{-[\sum_k^K \frac{(x_k - x_{kn})^2}{2\sigma^2}]} \tag{3.44}$$

can be used as the kernel function for a Parzen Window estimation. In this case, the equation for the density becomes

$$P(\bar{x}/G_J) = \frac{1}{N_J} \sum_{j=1}^{N_J} (2\pi\sigma_J^2)^{-\frac{K}{2}} e^{-[\sum_{k=1}^K \frac{(x_k - x_{kj})^2}{2\sigma_J^2}]} \tag{3.45}$$

For a two class problem, with the ratio of the number of sample points in each class to the total number of sample points reflecting the apriori probabilities, the classification rule is to assign $\bar{x}$ to group I if

$$\frac{N_I}{N} P(\bar{x}/G_I) > \frac{N_J}{N} P(\bar{x}/G_J) \tag{3.46}$$

Substituting for the estimated densities provides the classification rule

$$\frac{N_I}{N} \frac{1}{N_I} \sum_{i=1}^{N_I} (2\pi\sigma_I^2)^{-\frac{K}{2}} e^{-[\sum_{k=1}^K \frac{(x_k - x_{ki})^2}{2\sigma_I^2}]} > \frac{N_J}{N} \frac{1}{N_J} \sum_{j=1}^{N_J} (2\pi\sigma_J^2)^{-\frac{K}{2}} e^{-[\sum_{k=1}^K \frac{(x_k - x_{kj})^2}{2\sigma_J^2}]} \tag{3.47}$$

This reduces to

$$\sum_{i=1}^{N_I} (2\pi\sigma_I^2)^{-\frac{K}{2}} e^{-[\sum_{k=1}^K \frac{(x_k - x_{ki})^2}{2\sigma_I^2}]} > \sum_{j=1}^{N_J} (2\pi\sigma_J^2)^{-\frac{K}{2}} e^{-[\sum_{k=1}^K \frac{(x_k - x_{kj})^2}{2\sigma_J^2}]} \tag{3.48}$$

If the spreads, or window widths, are the same for each class, $\sigma_I = \sigma_J$, then equation 3.48 becomes a sum of gaussian radial basis functions.

$$\sum_{i=1}^{N_I} e^{-\sum_{k=1}^K \frac{(x_k - x_{ki})^2}{2\sigma_I^2}} > \sum_{j=1}^{N_J} e^{-\sum_{k=1}^K \frac{(x_k - x_{kj})^2}{2\sigma_J^2}} \tag{3.49}$$

These equations have been implemented by Specht as Probability Neural Networks (PNNs) with the topology shown in figure 3.9 (24). In this network, the hidden layer weights, or centers

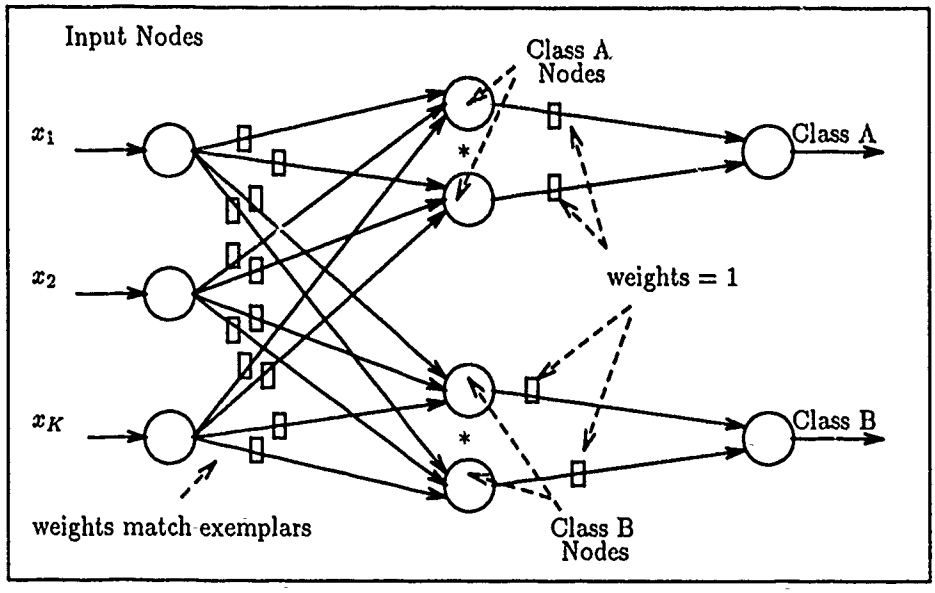

Figure 3.9. Probabilistic Neural Network Topology (24:528)

of the gaussian radial basis functions, are set to match the features of each of the training vectors. The classification nodes in the output layer are connected only to the nodes in the hidden layer which belong to their class. These output layer classification nodes implement equation 3.49 by forming the simple sum of the outputs from the hidden layer nodes in their class. In this type of network, the σ's are usually chosen on a trial and error basis.

The only difference between the PNN network and the RBF Kernel Classifier network is that the classification nodes in the output layer of an RBF network are connected, via weighted interconnections, to all the nodes in the hidden layer. Since it has been shown (21) that any neural network that has its parameters set to minimize the MSE objective function will operate as a Baye's optimum discriminant, if the weights,w, and sigmas ,σ's, for the RBF network are established via minimization of the MSE objective function, the RBF network should approximate the Bayes' optimal discriminant function without the trial and error approach to setting the σ's as in the PNN. Thus, the performance of PNN network shown in figure 3.9 should approach the performance of the RBF network shown in figure 3.7 once an optimal choice of σ is made.

3.3.5 Network Supervised Training As with the Hyperplane Classifier network, all the parameters for the Kernel Classifier network can be established by minimizing the MSE objective function, incrementally, via backpropagation. Again, the MSE is defined as

$$MSE = \frac{1}{M}\sum_{m=1}^{M}(y_m - d_m)^2 \tag{3.50}$$

The general form of the update equation for a network parameter w_j then becomes

$$w_j^+ = w_j^- - \eta\frac{\partial MSE}{\partial w_j} \tag{3.51}$$

Here, η is a constant which controls the learning rate. These update equations for a network with the topology shown in figure 3.7 are derived in Appendix D. For a weight linking node L in Layer 1 to node M in the Layer 2 w_{LM}, the update equation is

$$w_{LM}^+ = w_{LM}^- - \eta(y_M - d_M)y_L \tag{3.52}$$

For a weight linking node K in Layer 0 to node L in layer 1, w_{KL}, the update equation is

$$w_{KL}^+ = w_{KL}^- - \eta\sum_{m=1}^{M}(y_m - d_m)w_{Lm}y_L\frac{(x_K - w_{KL})}{\sigma_{KL}^2} \tag{3.53}$$

while for the spread of node L in layer 1 in the direction of node K in layer 0, σ_{KL} the update equation is

$$\sigma_{KL}^+ = \sigma_{KL}^- - \eta\sum_{m=1}^{M}(y_m - d_m)w_{Lm}y_L\frac{(x_K - w_{KL})^2}{\sigma_{KL}^3} \tag{3.54}$$

3.3.6 Network Combined Training In this type of training, the hidden layer, layer 1, weights (radial basis function centers), the hidden layer, layer 1, spreads (radial basis function sigmas), and the output layer, layer 2, weights are set separately. That is, the hidden layer weights can be set by any of the following rules:

1. Nodes at the Data Points
2. Kohonen Training
3. K-Means Clustering
4. Center at Class-Cluster Averages

The hidden layer spreads can be set by any of the following rules

1. Set Sigmas at a constant.
2. Set Sigmas at P-Neighbor Averages
3. Scale Sigmas by Class Interference.

The weights linking the output layer to the hidden layer nodes can be set by one of the following rules:

1. Incremental MSE Minimization
2. Global MSE Minimization
3. PNN Implementation

3.3.6.1 Nodes at the Data Points In this training algorithm, the hidden layer weights, or centers, of the radial basis functions are set to match the features of each of the training vectors. Suppose there are P pattern vectors where each vector is of dimension K. The vector for the p^{th} pattern vector can be written as

$$\bar{x}_p = [x_{p1}, x_{p2}, \ldots x_{pK}] \tag{3.55}$$

Setting the weights to match the exemplars will then allow P hidden layer nodes to be created where the output of the l^{th} node, due to the p^{th} input pattern will be defined from

$$y_{pl} = e^{-[\sum_{k=1}^{K} \frac{(x_{pk}-w_{kl})^2}{2\sigma_{kl}^2}]} \tag{3.56}$$

Since the weights match the exemplar features, the weight vector for the l^{th} radial basis function will be exactly the same as the feature vector for the l^{th} exemplar, $\bar{w}_l = \bar{x}_l$. This allows the output, due to the p^{th} input pattern, for the l^{th} radial basis function to be written as

$$y_l = e^{-[\sum_{k=1}^{K} \frac{(x_{pk}-x_{kl})^2}{2\sigma_{kl}^2}]} \tag{3.57}$$

There are several advantages of establishing the weights, or centers, of the radial basis nodes, in this manner. First, this allows the direct application of the theory of approximating multivariate functions with radial basis functions. Second, this method allows a direct implementation of the Parzen Window probability density estimation for the training data. Third, the computational time

for setting the layer 1 weights is negligible. Finally, each radial basis function node is guaranteed to represent a particular class of data at its maximum output.

There are some disadvantages of making the weights, or centers, of the radial basis function nodes match the exemplar features. First, a large number of nodes could be required to effectively partition the whole feature space. Second, the computational time to required to establish the weights in the output layer will increase significantly as the number of nodes increases. Finally, the weights, of centers of the radial basis functions, could include the noise associated with the input patterns.

3.3.6.2 Kohonen Training The Kohonen Training Algorithm is a clustering algorithm which seeks to learn the underlying probability density function of the data (19:64). Using this algorithm to set the weights, or centers, of the radial basis functions should allow a radial basis function node to respond strongly to similar inputs. Basically, Kohonen Training calls for the establishment of a rectangular grid of nodes as shown in figure 3.10. The weights for these nodes

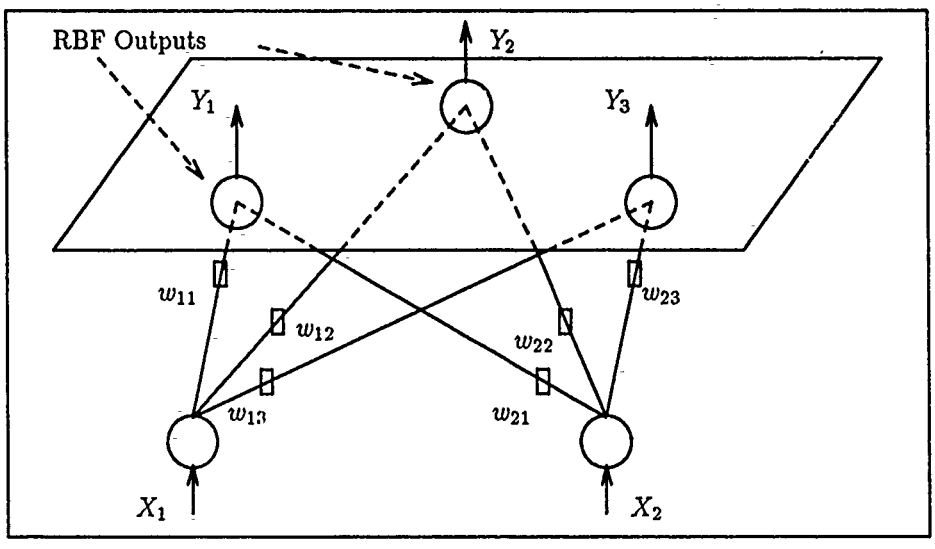

Figure 3.10. Rectangular Grid of Kohonen Nodes (19:65)

are adapted by applying a training vector to the layer and computing the Euclidean distance between the weights for each of the nodes and the input vector (19:65-68).

$$d_l = \sum_{k=1}^{K} (x_{pk} - w_{kl})^2 \tag{3.58}$$

Notice, this distance measurement is the same as that of the numerator of the gaussian transfer function of each of the nodes in the hidden layer. After this distance calculation is made, the node whose weights are nearest to the features of the input patterns are updated, along with nodes in the vicinity, or neighborhood of this nearest node, according to the equation

$$w_{kl}^{+} = w_{kl} + \alpha(t)(x_k - w_{kl}) \tag{3.59}$$

This update equation serves to move the weights of each of the updated nodes toward the input pattern in a method which represents the vectorial difference between the weight vector and the input feature vector (19:67). This algorithm of presenting an input pattern, finding the node with the most similar weights and updating that node and its neighbors, is repeated over a specified number of iterations. Once the Kohonen layer has been trained, each node will represent clusters, or pockets, of pattern vectors.

The main advantage of training the weights in this manner is that the number of nodes in the layer will not depend explicitly on the number of exemplars. That is, there can be far less nodes than exemplars. Also, each node will represent more than one exemplar as the weights are trained to represent clusters of data.

The main disadvantage of training the weights in this manner is the amount of time it takes to train. At this time, there is no formal criteria for determining when the weights have all been adapted to represent the underlying distribution of the data. Also, the weights are adapted in a manner which does not reflect classification of the data. It is possible for a node in the Kohonen layer to respond strongly for several different classes. This could serve to hinder the training of the weights in the output layer. Finally, the number of nodes in the Kohonen layer is arbitrary. At this time, there is no formal method of predetermining the number of nodes necessary to provide the optimum performance.

3.3.6.3 K-Means Clustering The K-Means Clustering Algorithm is a method of training the weights, or centers, of the radial basis function nodes such that the distance from all points in a cluster to the cluster center is minimized (27:94). In this procedure, the number of radial basis function nodes in the hidden layer is preset to a number K. The weights for each of these nodes are initialized to match the features of the first K pattern vectors. That is $\bar{w}_l = \bar{x}_l$ for all $l \leq K$. All training vectors are then presented to the network. Each vector, $\bar{x}_p$, is assigned a cluster, denoted by S_j, by $\bar{x}_p \in S_j$ if $\|\bar{x}_p - \bar{w}_j\| < \|\bar{x}_p - \bar{w}_i\|$ for all $i = 1, 2, \ldots, K$ and $i \neq j$. Here, the norm is taken to be the Euclidean distance. Notice the norm is the same as the numerator in the gaussian transfer

function for the radial basis function nodes. This means each new pattern vector is associated with the node whose center is the closest in a Euclidean measure. Once all patterns have been assigned a cluster, the new cluster center weights are computed as the average of the features of the pattern vectors assigned to the cluster. That is

$$\bar{w}_l^+ = \frac{1}{N_c} \sum_{n_c=1}^{N_c} \bar{x}_{n_c} \tag{3.60}$$

Here, N_c is the number of pattern vectors assigned to the cluster and $\bar{x}_{n_c}$ is a pattern vector assigned to that cluster. Since this procedure adapts the weights, it must be repeated until the weights stabilize, or no longer adapt. This occurs when $\bar{w}_l(n+1) = \bar{w}_l(n)$ for all cluster centers.

The main advantage of training the weights, or centers of the radial basis function in this manner is that the number of nodes does not depend on the number of exemplars. This means there can be many more exemplars than nodes, with each node's weights centered at the average of the pattern vector features associated with the cluster. Furthermore, each radial basis function node will now be able to represent pattern vectors with similar features.

The main disadvantages are that the number of radial basis function nodes, which is determined by K, the number of clusters, is arbitrary and each node is now allowed to respond strongly to pattern vectors of different classes. Also, the performance of the algorithm is dependent on the number of clusters, initial location of the clusters and the properties of the data (27:95). Finally, there is no guarantee that the algorithm will converge.

3.3.6.4 Center at Class-Cluster Averages In this algorithm the weights, or centers of the radial basis function nodes are allowed to adapt themselves, in an iterative process, to the centers of clusters of pattern vectors of the same class. Furthermore, this algorithm is adaptive in the sense that the number of nodes does not need to be preselected. The distribution of the data will determine the required number of radial basis function nodes. In this algorithm a cluster radius, R, is first preset and the network begins with one node whose weights match the first pattern vector. This node is also set to respond to the class of the first pattern vector. A new pattern vector, $\bar{x}_p$, is then applied to the network. The cluster assignment rule is $\bar{x} \in S_j$ if $\|\bar{x} - \bar{w}_j\| < \|\bar{x} - \bar{w}_i\| < R$ and the class of $\bar{x}$ is the same as S_j. If this relation isn't true then a new node is added such that the weights and class of the new node match that of the new input pattern. If this relation is true, then the weights of the cluster center are adapted to the new average by

$$\bar{w}_j(t+1) = \bar{w}_j(t) + \frac{\bar{x}_{N+1} - \bar{w}(t)_j}{N+1} \tag{3.61}$$

This process of presenting a new pattern vector to the network, checking to see if this new pattern vector can be associated with an existing cluster and adjusting the cluster center or adding a new cluster center will continue until all exemplars are tested. Since it is possible for exemplars to become "uncovered" during the update of cluster centers, the algorithm is repeated until no new nodes are added.

The main advantages of this algorithm are that the number of radial basis function nodes does not need to be selected beforehand and each node will respond strongly to only one class.

The main disadvantage of this algorithm is that the association radius, or vigilance parameter R, must be selected arbitrarily.

3.3.6.5 Set Sigmas at Constant In this algorithm, the sigma, or spread, for each radial basis function node is preset to a constant, C. Under this condition, the output for the l^{th} radial basis function node due to the p^{th} pattern vector $\bar{x}_p$ becomes

$$y_{pl} = e^{-[\frac{1}{2C^2}\sum_{k=1}^{K}(x_{pk}-w_{kl})^2]} \tag{3.62}$$

If the weight vectors had been previously set to match the features of the exemplar vectors, the network then calculates a Parzen Window estimate of the probability distributions of the data.

The main disadvantage of training the sigmas in this manner is that, since the constant is preset and not changed, there is no way of determining if the entire feature space is partitioned.

3.3.6.6 Set Sigmas at P-Neighbor Averages In this algorithm, the sigmas for each radial basis function node are allowed to vary according to a distance metric between their weights, or centers, and the weights or centers of their P nearest neighbors. That is, after the weights, or centers, of each radial basis function node is set, the Euclidean distance between the center of each radial basis function node and its neighbors are calculated. For example, the distance between radial basis function node i and radial basis function node j is

$$d_{ij} = \sum_{k=1}^{K}(w_{kj} - w_{ki})^2 \tag{3.63}$$

From these calculations, the P radial basis functions having the smallest distance, d_{ip}, are then used to set the sigma, or spread, (σ_i), for the i^{th} radial basis function by the following equation:

$$\sigma_i = \sqrt{\frac{1}{P}\sum_{p=1}^{P} d_{ip}^2} \tag{3.64}$$

This equation makes σ_i equal to the root mean square of the sum of distances between the center of i^{th} radial basis function and its P nearest neighbors (13:137).

The main advantages of setting the σ's in this manner are that each σ can be different for each node and, since each σ is a function of the separation between node centers, the feature space will usually be completely partitioned.

The main disadvantage of setting the σ's in this manner is that P must be determined beforehand. If P is too small, the σ's will be small and the feature space will not be covered adequately. If P is too large, the σ's will allow too much overlap between patterns of different classes. This could result in a node responding too strongly to more than one class.

3.3.6.7 Scale Sigmas by Class Interference In this algorithm, the σ's are adjusted, from a preset constant, to prevent the radial basis function nodes from responding too strongly from pattern vectors of different classes. In order for this algorithm to work, each radial basis function node must be assigned the responsibility for responding to only one class for the training data. This can be done by setting the weights using the Nodes at Data Points or Center at Class-Cluster Averages Algorithms previously discussed. After the weights are set, this algorithm then presents an exemplar pattern $\bar{x}_p$ and calculates the output for each radial basis function node by

$$y_{pl} = e^{-\frac{1}{2\sigma^2}\sum_{k=1}^{K}(x_{pk}-w_{kl})^2} \tag{3.65}$$

If the output for that node is above some preset threshold, T, and the node is not assigned to respond to the same class as that of the pattern vector, $\bar{x}_p$, then that node's σ is scaled by a constant until the output is less than T. That is if $y_{pl} > T$ then

$$\sigma_l^+ = (1 - C)\sigma_l^- \tag{3.66}$$

This process is then repeated for each pattern vector in the training set.

The main advantage of setting the σ's in this manner is that each node is guaranteed to respond to one and only one class for the training data within some threshold T. Another advantage is that each node can now have a separate σ.

The main disadvantage of setting the σ's in this manner is that if the pattern vectors are very close together, via the Euclidean distance measurement, then each node will lose its ability to generalize.

3.3.6.8 Incremental MSE minimization As shown in appendix D, the update equation for the weight linking node L in layer 1 to node M in layer 2, w_{LM} is

$$w_{LM}^{+} = w_{LM}^{-} - \eta(y_M - d_M)y_L \tag{3.67}$$

These parameters can be updated through backpropagation even though the other network parameters have been preset. However, this process can still take many iterations to converge.

3.3.6.9 Global MSE minimization As shown in appendix D, if the weights and sigmas (spreads) of the first layer have been established, the weights linking nodes layer 1 to nodes in layer 2 can be established by a global minimization of the MSE function over all training patterns (23). When trained in this manner, the update equation for a weight linking node B in layer 1 to node D in layer 2 is

$$W = (M^T)^{-1}Y^T S \tag{3.68}$$

Here, W is an L by M matrix containing the weights linking the nodes in the hidden layer to an nodes in the output layer. That is

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1M} \\ w_{21} & w_{22} & \cdots & w_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ w_{L1} & w_{L2} & \cdots & w_{LM} \end{bmatrix} \tag{3.69}$$

where w_{LM} is the weight linking the L^{th} node in the hidden layer, layer 1, to the M^{th} node in the output layer, layer 2. The M matrix is an L by L matrix containing the summation, over all patterns, of the product of each radial basis function output, for a given input pattern and the B^{th} radial basis function output for that pattern. That is

$$M = \begin{bmatrix} M_{11} & M_{12} & \dots & M_{1L} \\ M_{21} & M_{22} & \dots & M_{2L} \\ \vdots & \vdots & \vdots & \vdots \\ M_{L1} & M_{L2} & \dots & M_{LL} \end{bmatrix} \tag{3.70}$$

where $M_{lB} = \sum_{p=1}^{P} y_{pl} y_{pB}$. Also, Y is a P by L matrix containing the outputs for each of the L radial basis functions for all P patterns. That is,

$$Y = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1L} \\ y_{21} & y_{22} & \cdots & y_{2L} \\ \vdots & \vdots & \vdots & \vdots \\ y_{P1} & y_{P2} & \cdots & y_{PL} \end{bmatrix} \tag{3.71}$$

Finally S is a P by M matrix containing the desired outputs for each of the M output nodes for all P patterns. That is,

$$S = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1M} \\ d_{21} & d_{22} & \dots & d_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ d_{P1} & d_{P2} & \dots & d_{PM} \end{bmatrix} \tag{3.72}$$

This method only works for matrices that do not become singular or near-singular, which can happen if the exemplar data points used to center the radial basis functions contain redundant information. If they do, the Singular Valued Decomposition of the matrix may be used. Conversely, the σs of the offending nodes may be adjusted to eliminate the redundancy.

3.3.6.10 Probability Neural Network (PNN) As shown in Chapter 3, after establishing the parameters for the nodes in the hidden layer, layer 1, a PNN can be constructed by connecting each output layer node to the hidden layer nodes representing the output layer node's class. In this network, the weights connecting the hidden layer nodes in layer 1 to the output layer nodes in layer 2 are set to 1.

3.4 Summary

This chapter discussed the general operation of Hyperplane and Kernel Classifier neural networks. The objective functions used to implement the Hyperplane Classifier networks were then analyzed, followed by the development of the equations necessary to implement these classifiers as neural networks. The relationship between pattern recognition, functional interpolation and probability density estimation were then presented as implementable properties of Kernel Classifier networks. This chapter concluded with the development of equations implementing these classifiers as neural networks.

IV. Software Description

4.1 Introduction

The software to be described in this chapter was designed according to an object-oriented approach. This chapter begins with a description of the data structures implemented for the software and concludes with a brief discussion of the software modules. An in-depth description of these items, along with the mapping of the training algorithms developed in chapter 3, into software functions, is given in Appendix F.

4.2 Approach

Artificial neural networks are composed of nodes. Each node has associated with it certain parameters such as a weight vector, an offset, a transfer function, and a class to which the node responds. The main difference between different types of networks is the way in which the nodes are connected to one another and the method of setting the network parameters. Therefore, in order to maximize the types of networks which could be configured, the only entity implemented as an object was the node. Each node was then assigned the following attributes:

(a) weights - w_j

(b) sigmas - σ_j

(b) connections

(d) transfer-function

(e) class

The operations that can be performed on each node are the following:

(a) assign transfer function

(b) calculate node output

(c) initialize node weights and sigmas

(d) assign a node to a class

(e) assign a node to be connected to another node

(f) update (train) weights and sigmas

With these attributes and operations a variety of networks can by formulated. This thesis implemented just the feedforward type of network architecture. However, this object oriented

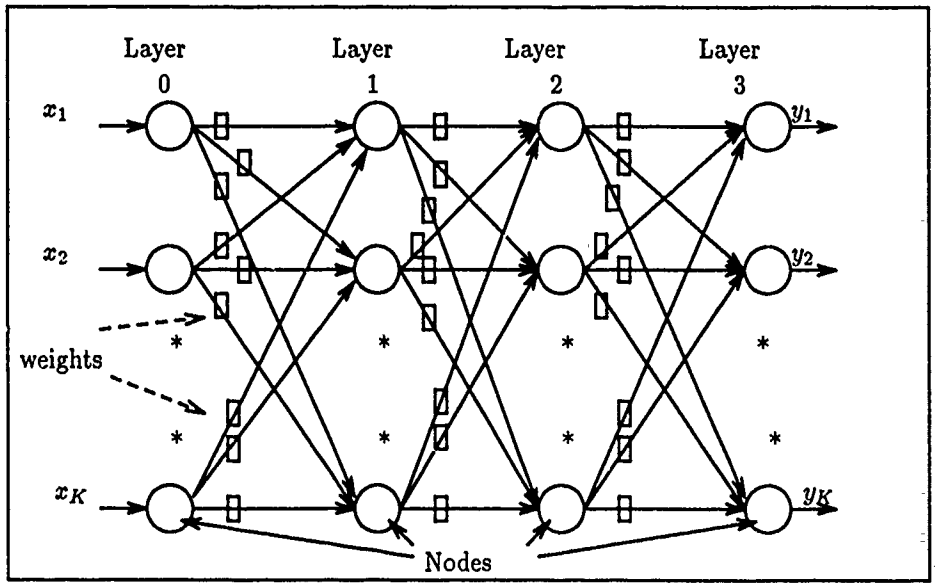

Figure 4.1. Feed Forward Network (19:56)

design approach will allow future enhancements to more complicated networks such as recurrent networks and higher-order networks.

4.3 Networks

The only type of network implemented at this time is the feed forward network. A feed forward network, as shown in figure 4.1, is a network in which each node is assigned to a particular layer and receives inputs only from the nodes in the previous layer. Pattern vectors are input to the network via the nodes in layer 0. These nodes have the identity transfer function and serve to propagate the features from the input pattern, across the internodal weights and thresholds, to nodes in layer 1. The layer 1 nodes will transform these inputs into internal representations, using their assigned transfer functions, and transmit these representations, via the internodal weights and thresholds, to the nodes in layer 2. This process of transforming the data and propagating the new representation will continue through each layer of the network. The outputs for the last layer in the network will be used to determine the classification of the input pattern.

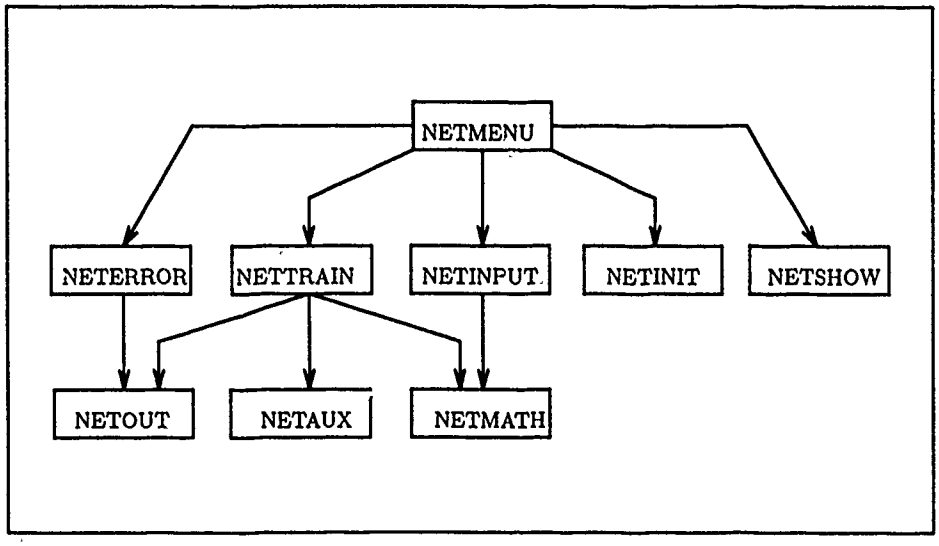

Figure 4.2. Software Structure Chart

4.4 Structure

The software implemented in this thesis consists of the nine modules shown by the structure chart in figure 4.2.

4.4.1 NETMENU This module is the overall controlling module of the network. It provides the user interface to the software via the SUN terminal and keyboard and calls the appropriate modules to execute the users decisions.

4.4.2 NETERROR This module contains the functions necessary to determine the network's classification of a data vector and the error performance of the network.

4.4.3 NETTRAIN This module contains the functions necessary to establish the network weights via the following training procedures:

a) Nodes at the Data Points

b) Center at Class-Cluster Averages

c) K-means Clustering

d) Train via Kohonen

e) Global MSE Minization

f) Backpropagation for MSE, CE and CFM algorithms.

g) Probability Neural Networks

Each of these functions accomplishes its training routine by executing the specialized functions contained in NETAUX. NETTRAIN also contains the functions necessary to establish the σ's for the network nodes via the following training procedures:

a) Scale Sigma by Class Interference

b) Set Sigma According to P-Neighbor Distance

c) Set Sigma to a Constant

4.4.4 NETINPUT This module contains the functions necessary to load the training and test data patterns. This data may be loaded from separate training and test files or from a single file. This loading of training and test patterns may also be accomplished either in the sequence listed in the data files or in a random manner.

4.4.5 NETINIT This module contains the functions which allocate memory for the nodes and data records, correct node weights and connections, and initialize the node weights, sigmas, transfer functions and network connections.

4.4.6 NETSHOW This module contains the output functions necessary to display and file the performance and parameters of the network.

4.4.7 NETOUT This module contains the functions necessary to compute the outputs for each node in the network, the outputs for each layer of a feedforward network, and the output for the entire network due to a given input pattern.

4.4.8 NETAUX This module contains the training subfunctions called by NETTRAIN. Appendix F contains a detailed description of each function in this module.

4.4.9 NETMATH This module contains the mathematical functions used by the various training algorithms within the module NETTRAIN.

4.5 Implementation

The software code developed for this thesis and implemented under these modules is listed in Appendix G.

4.6 Summary

This chapter provided a brief overview of the software developed for this thesis. After developing the data structures implemented in the softwared, the sectioning of the software into modules was outlined.

V. Data Analysis

5.1 Introduction

Using the software developed in Chapter 4, two pattern classification problems will be analyzed. The first problem deals with the classification of coded digital communication signals while the second problem deals with classification of radar platforms. This chapter begins by discussing the data used to train and test the neural networks developed to classify coded digital communication signals. After detailing the methods used to train various networks to solve this classification problem, the results of the training and testing are then presented. This chapter concludes with a discussion of the data used to train and test the neural networks developed to classify radar systems and an analysis of training and test results.

5.2 Communication Signal Characterization

5.2.1 Data Description In this two-class problem, an acousto-optic correlation system was used to capture correlation signatures of spread spectrum signals for both a direct sequence and a linear-stepped frequency hopped signal. Over 200 pattern vectors were first formed by sampling known waveforms at 1000 data points. After averaging consecutive data point pairs, thereby reducing the number of data points to 500, the peak of the signals were identified and 25 points on each side of the peak were extracted. These 50 data points, now representing a 50 dimensional feature vector, were normalized to values between -1 and 1 by dividing each dimension by the magnitude of the largest component. One hundred feature vectors for each class now represented the signature for the direct sequence and the linear stepped frequency hopped encoding schemes. Feature vectors representing the direct sequence signatures were then assigned to class 1 and feature vectors representing the linear stepped frequency-hopped signatures were assigned to class 2. The final data set contained 101 pattern vectors for each class; with each pattern vector having 50 dimensions.

5.2.2 Testing This data was processed using both Hyperplane Classifiers and Kernel Classifier networks. The parameters for each of these networks were set using the algorithms developed in Chapter 3. The training data for each of the classes was randomly selected for each network run. For each test, 51 feature vectors from each class were used to train the network and a different 50 feature vectors from each class were used to test the network. For each network, there were 50 nodes in layer 0, one node for each of the dimensions of the input data. The number of nodes in the final, or output, layer of the network was set at two with each node assigned to represent

one of the two classes. The number of hidden layers and the number of nodes in the hidden layers were allowed to vary according to the parameters of the network. For this classification problem, the network was allowed to make a classification based on which node in the output layer had the higher output.

5.2.3 Hyperplane Classifiers The Hyperplane Classifier networks developed for this classification problem were based on the topology shown in figure 3.3 with each network consisting of two hidden layers. The 50 nodes in the input layer, layer 0, had the identity transfer function. The number of nodes in the first hidden layer, layer 1, was set at 18, the number of nodes in the second hidden layer, layer 2, was set at ten, and the number of nodes in the output layer, layer 3, was set at two. The nodes in each of these layers were assigned the sigmoidal transfer function. The parameters for each of the nodes were trained via backpropagation according to either the MSE, CE or CFM objective functions discussed in Chapter 3.

5.2.3.1 MSE Objective Function To characterize the performance of Hyperplane Classifiers trained using this algorithm, ten different sets of training data were applied to the network and the performance of the network was measured for each set as shown Table E.1. Here, a correct response for the training data is defined to occur when the output for the correct classification node was greater than .9 and the output for the incorrect classification node was less than .1. As shown in figure 5.1, the average performance of the network converged rapidly until about 15000 -

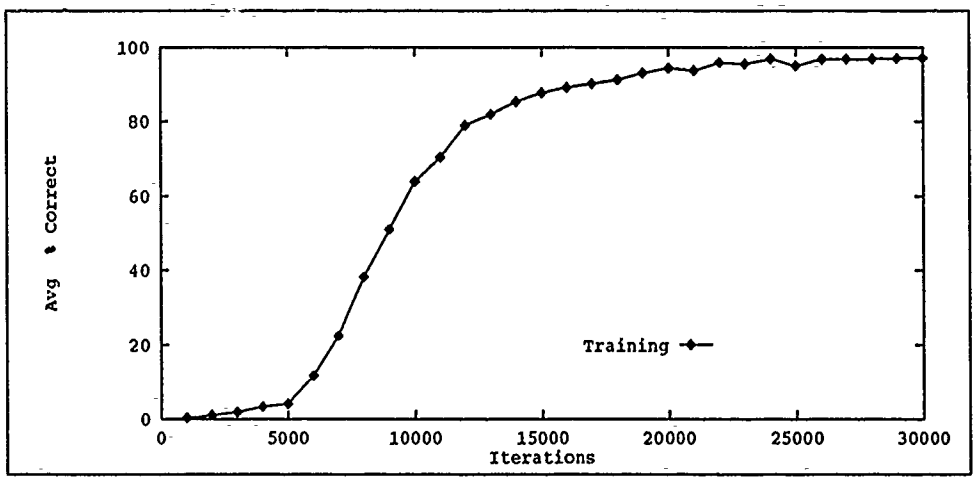

Figure 5.1. Performance vs Training Iterations for MSE Algorithm

20000 iterations. At this point, the categorization performance of the network reached 90 percent. From 20000 - 30000 iterations, the categorization performance slowly increased to the final average

of 97.16 percent. The robustness of the network, for both the training and the test data, was calculated as shown in Table 5.1.

Table 5.1. Robustness Measure for MSE Training

	% Correct	
	Training	Test
Avg	97.16	79.7
Std	4.51	5.92

5.2.3.2 CE Objective Function To characterize the performance of Hyperplane Classifiers trained using this algorithm, ten different sets of training data were applied to the network and the performance of the network was measured for each set as shown in Table E.2. Here, a correct response for the training data is defined to occur when the output for the correct classification node was greater than .9 and the output for the incorrect classification node was less than .1. As shown in figure 5.2, the average performance of the network converged rapidly until about 5000 - 10000

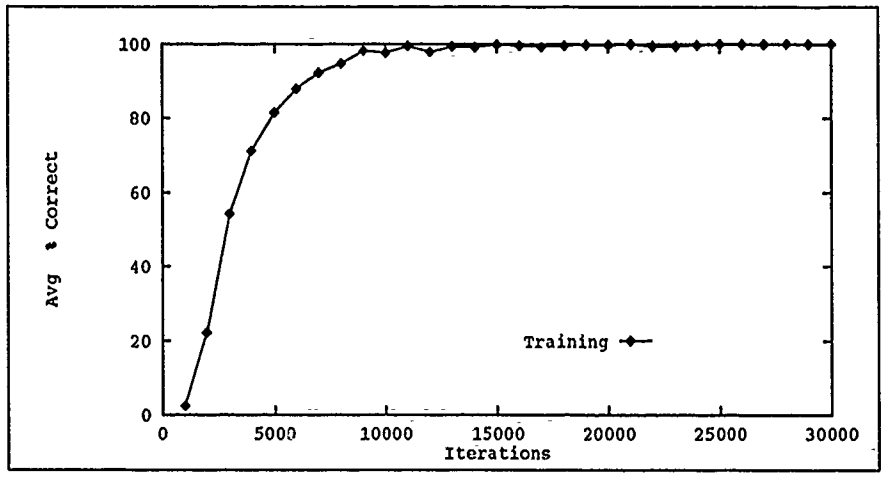

Figure 5.2. Performance vs Training Iterations for CE Algorithm

iterations. At this point, the performance of the network remained relatively stable. From 10000 - 15000 iterations, the categorization performance slowly increased to the final average of 100.00 percent. The robustness of the network was calculated as shown in Table 5.2.

As compared to the networks trained to minimize the MSE objective function, the networks trained to minimize the CE objective function performed at about the same level of categorization accuracy but converged in about half the iterations as the MSE objective function. This is due to

Table 5.2. Robustness Measure for CE Training

	% Correct	
	Training	Test
Avg	100.0	81.2
Std	0.0	4.62

the lack of the term $y_n(1 - y_n)$ in the CE update equations. Since the maximum value for this term is 1/4, the parameters for the MSE network are adapted much more slowly than that of the CE network.

5.2.3.3 CFM Objective Function To characterize the performance of Hyperplane Classifiers trained using this algorithm,, ten different sets of training data were applied to the network and the performance of the network measured for each set as shown in Table E.3. As shown in figure 5.3, the average performance of the network converged rapidly until about 30000 - 35000

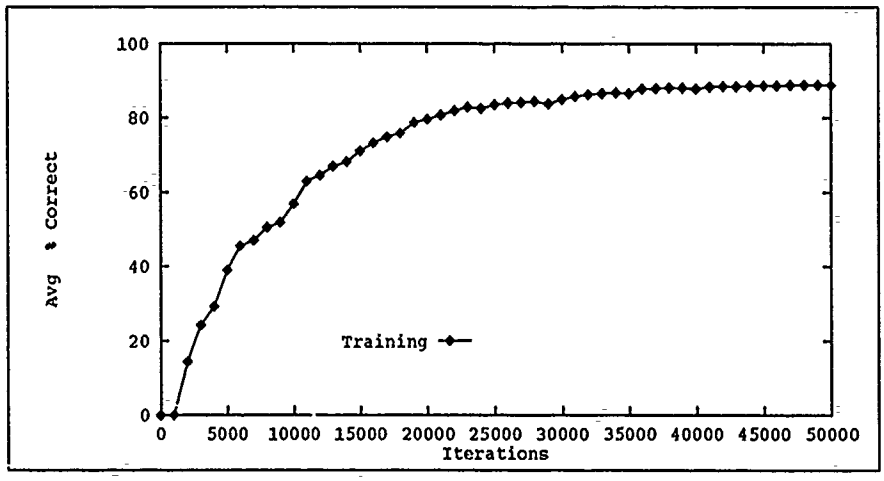

Figure 5.3. Performance vs Training Iterations for CFM Algorithm

iterations. At this point, the performance of the network remained relatively stable. From 35000 - 50000 iterations, the categorization performance slowly increased to the final average of 88.83 %. The robustness of the network was calculated as shown in Table 5.3. As expected, the categorization performance for the training data was less than that for either the MSE or CE objective functions. However, the performance for the test data proved lower than that of either the MSE and CE objective functions for this data. If the two tests having the lowest categorization performance on the test data are removed, the categorization performance of the CFM for the test data rises to

Table 5.3. Robustness Measure for CFM Training

	% Correct	
	Training	Test
Avg	88.83	73.20
Std	5.37	7.93

76.63 % which nearly matches that of the MSE. This seems to indicate the pattern space does not contain pockets of data which would cause the MSE and CE algorithms to have larger classification errors for a correct response than for an incorrect response.

5.2.4 Kernel Classifiers The networks developed to categorize this data are based on the topology shown in figure 3.7. The 50 nodes in the input layer, layer 0, had the identity transfer function and two nodes in the output layer, layer 2, had the linear transfer function. The number of nodes in the hidden layer, layer 1, was a function of the algorithm used to train the layer 1 weights. These nodes were assigned the gaussian transfer function.

5.2.4.1 Nodes at the Data Points In this test, the performance of the Kernel Classifier network, using the interpolation theory of applying the nodes at the data points, was measured. The weights, or centers, for the nodes in layer 1 were set using the Nodes at Data Points algorithm and the sigmas were set using the Scale Sigmas by Class Interference algorithm. The weights linking the nodes in the output layer to the nodes in the first layer were trained via global minimization of the MSE objective function. The performance of the network was then analyzed as the number of nodes in the hidden layer was allowed to vary from 10 to 50. The complete data are shown in Tables E.4 and E.5. A plot of this performance is shown in figure 5.4. As expected, as the number of RBF nodes in the hidden layer, layer 1, increased, the classification performance of the network increased for both the training and the test data. The maximum performance occurred when the number of nodes matched the number of exemplars at 102.

The overall performance, or robustness, of the network when layer 1 contained 102 nodes was found to be as shown in Table 5.4. As shown by this table, setting the weights of the layer 1 nodes at the training data points allows the network to 'memorize' the training data while still performing relatively well on the test data. However, this performance is at the expense of many layer 1 nodes.

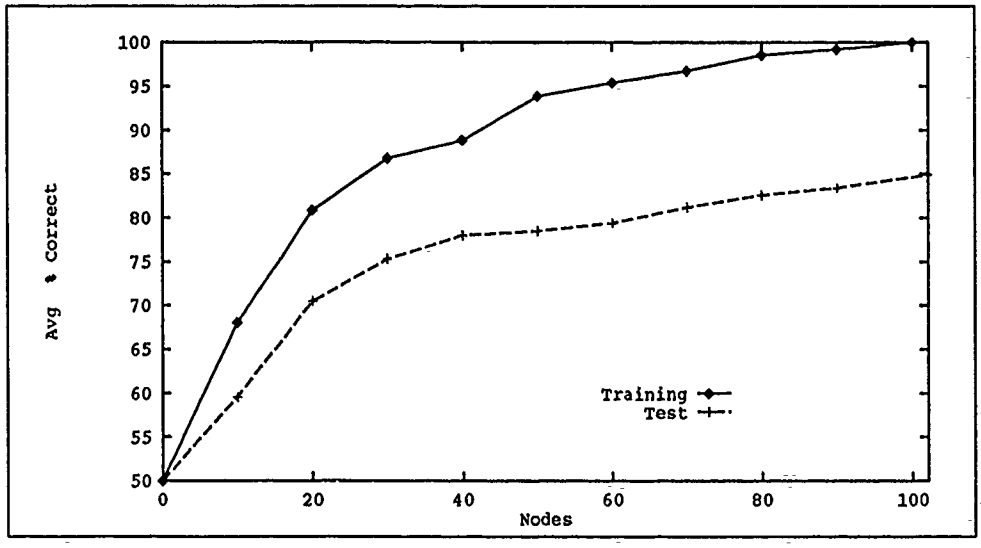

Figure 5.4. Performance vs Nodes for Nodes at Data Points

Table 5.4. Robustness Measure for Nodes At Data Points

	% Correct	
	Training	Test
Avg	100.00	84.90
Std	0.00	3.94

5.2.4.2 Kohonen Training In this test, the ability of a fixed number of Kohonen layer nodes to distribute themselves to cover the pattern space was measured. The weights, or centers, for the nodes in the hidden layer, layer 1, were trained using the Kohonen Training algorithm and the sigmas set using the Set Sigmas at P-Neighbor Averages algorithm. The weights linking the nodes in the output layer to the nodes in the first layer were trained via global minimization of the MSE.

With P arbitrarily held at six, the performance of the network was analyzed as the number of nodes in the Kohonen layer was increased from 16 to 100. The complete data are shown in Tables E.6 and E.7. As shown in figure 5.5 as the number of nodes increased, the ability of the network to

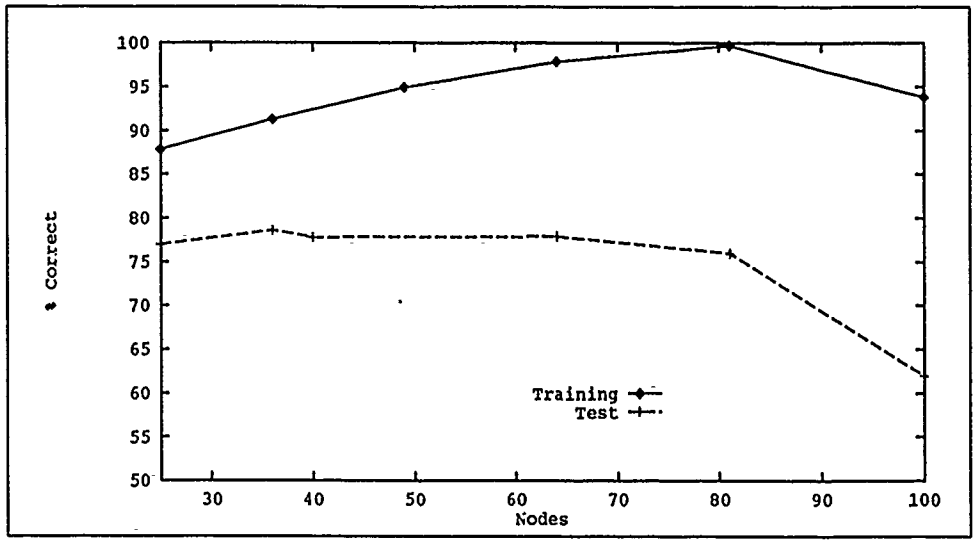

Figure 5.5. Performance vs Nodes for Kohonen Training with Six P Neighbors

categorize the training data increased. However, the ability of the network to categorize the test data decreased. This is due to the fact that as the number of nodes is increases, the distance to the six nearest neighbors decreases. Thus, the network loses its ability to generalize. The decrease in the training performance for 100 nodes was due to two of the networks converging to a performance of less than 85%. If these two tests are removed from the performance calculations, the average performance rises to 97.06%.

The performance of the network was then analyzed by allowing P to be equal to the square root of the number of nodes in the Kohonen layer. The complete data are shown in Tables E.8 and E.9. As shown in figure 5.6 allowing P to increase as the number of nodes increased had little effect on the performance of the network. Again, the decrease in the training performance for 100

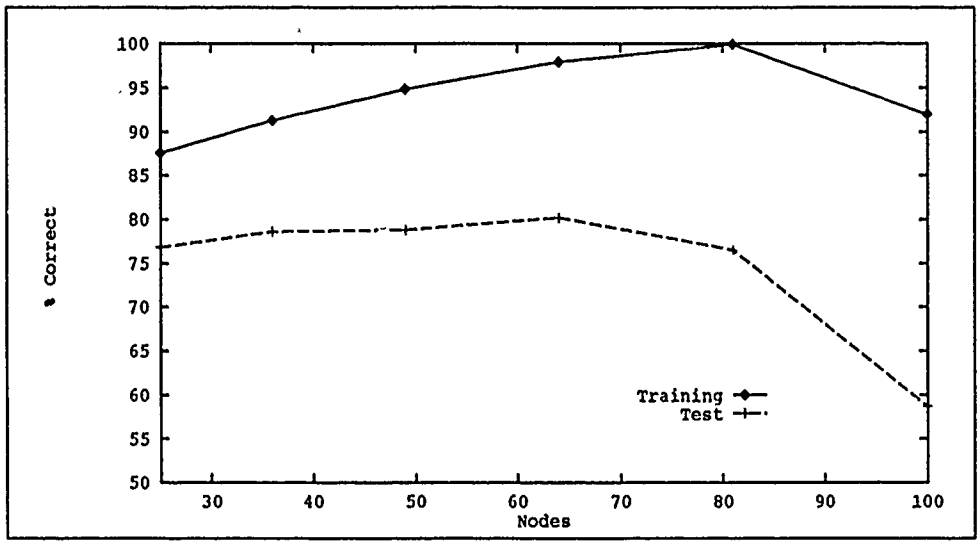

Figure 5.6. Performance vs Nodes for Kohonen Training with Variable P Neighbors

nodes was due to two of the networks converging to a performance of less than 85%. If these two tests are removed from the performance calculations, the average performance rises to 96.79%.

The maximum robustness of the network occurred when the number of Kohonen nodes was 64 and the number of P-Neighbors used to determine the RBF spreads was eight. This data is shown in Table 5.5. Comparing these results to that of the networks trained via the Nodes at Data

Table 5.5. Robustness Measure for Kohonen Training

	% Correct	
	Training	Test
Avg	97.94	80.20
Std	1.42	3.84

Points algorithm shows that the maximum accuracy for the test data was about 5% less for the Kohonen Training algorithm. Also, when 100 nodes were used to train the network, the training performance became unpredictable. This is probably due to the fact that the Kohonen Training algorithm adapts the layer 1 weights to the data independent of the class of the data. Thus, it is highly likely that certain layer 1 nodes actually represent more than one class of the data.

5.2.4.3 K-Means Cluster In this test, the ability of the network to distribute a set number of nodes to cover the pattern space, using the K-Means algorithm, was studied. The

number of nodes in the hidden layer, layer 1, was set to K, the number of clusters. The weights, or centers, for the nodes in layer 1 were trained using the K-Means Clustering algorithm and the sigmas set using the Set Sigma at P Neighbor Averages algorithm. The weights linking the nodes in the output layer to the nodes in the first layer were trained via global minimization of the MSE. The performance of the network was first analyzed by setting the P-Neighbors to 6 and letting the number of nodes in the hidden layer vary. The complete data are shown in Tables E.10 and E.11. As shown in figure 5.7 the performance of the network increased until as the number of nodes

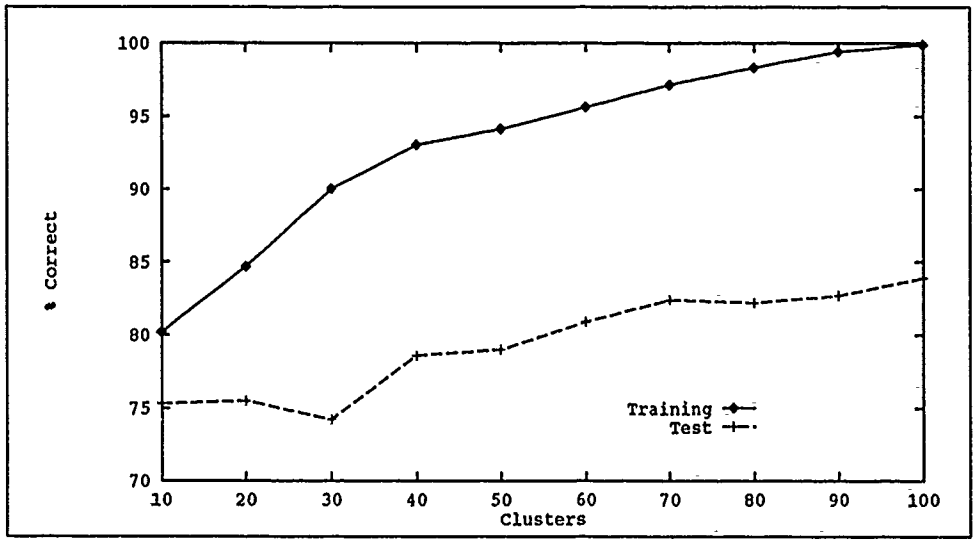

Figure 5.7. Performance vs Nodes for K-Means Clustering with Six P-Neighbors

in the hidden layer increased. When number of nodes reached the range of 60 to 70 nodes, the performance of the network, over the test data, leveled out, indicating the pattern space was fully covered.

The performance of the network was then analyzed by setting the number of nodes in the hidden layer, layer 1, to 60 and varying the number of P-Neighbors from 1 to 30. The complete data are shown in Tables E.12 and E.13. As shown in figure 5.8 the categorization performance of the network was relatively constant until the number of P-Neighbors was eight. At that point, the performance of the network, over the test data, decreased slightly as P was increased. The robustness of the network with 60 nodes and P set at 6 is shown in Table 5.6.

Comparing the performance of this training algorithm to the Kohonen Training algorithm shows the performance of two algorithms was roughly equivalent. However, the amount of time

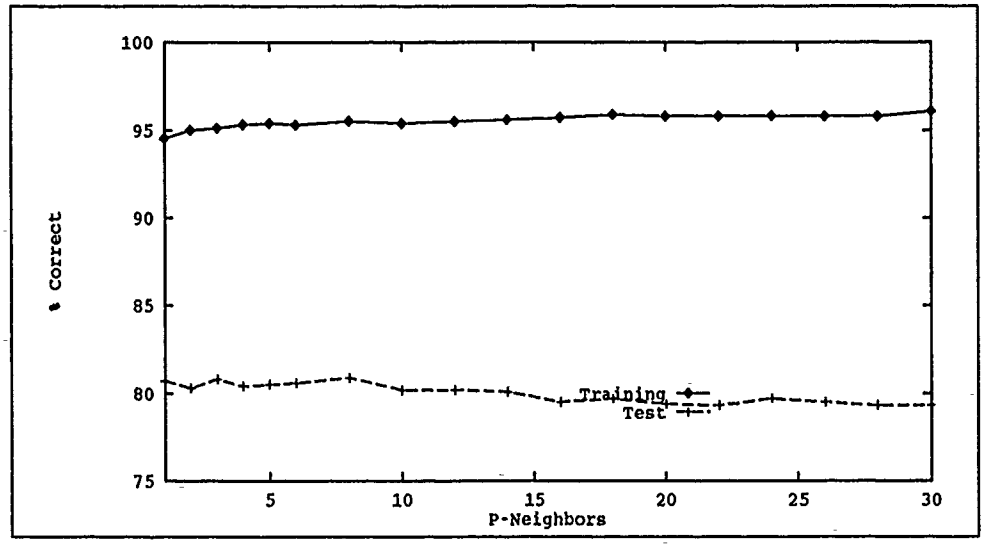

Figure 5.8. Performance vs P-Neighbors for K-Means Clustering with Sixty Clusters

Table 5.6. Robustness Measure for K-Means Clustering

	% Correct	
	Training	Test
Avg	95.59	80.90
Std	1.71	3.36

required to train via the K-Means Clustering algorithm was less than 30 minutes while the amount of time required to train via the Kohonen Training algorithm exceeded 120 minutes.

5.2.4.4 Center at Class-Cluster Averages In this test, the ability of the network to add the required number of nodes to cover the input data space, using the Center at Class-Cluster Averages algorithm, was measured. The weights, or centers, for the nodes in the first layer were trained using the Center at Class-Cluster Averages algorithm and the sigmas set using the Scale Sigmas by Class Interference algorithm. The weights linking the nodes in the output layer to the nodes in the first layer were trained via global minimization of the MSE. The performance of the network was analyzed by varying the average threshold (vigilance) of the nodes in the hidden layer. The complete data are shown in Tables E.14 and E.15. As shown in figure 5.9,

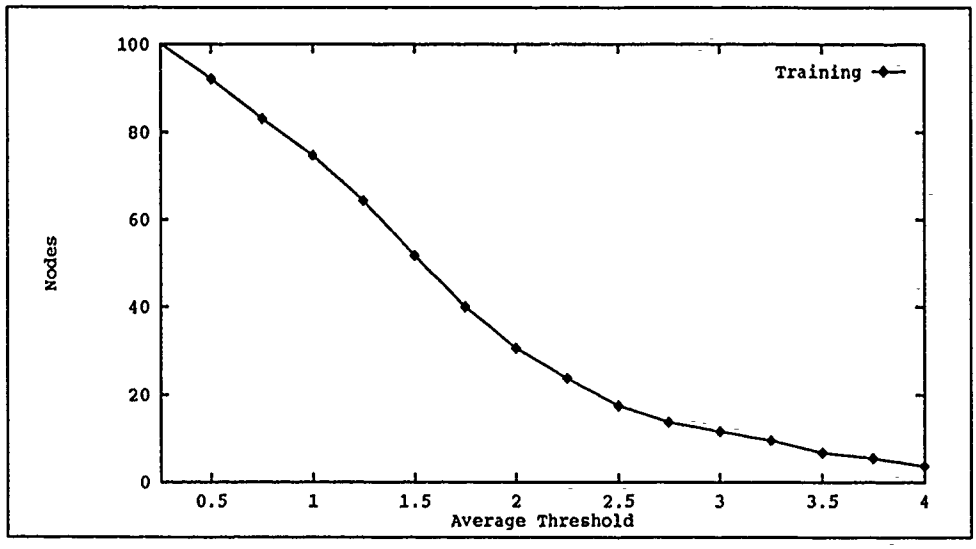

Figure 5.9. Nodes vs Average Threshold for Center at Class Averages

as the average threshold increased, the number of nodes required to cover the pattern space of the training data decreased. However, as shown in figure 5.10, as the number of nodes decreased, the categorization performance of the network decreased. By comparing both figures, it can be seen that the categorization performance of the network remains fairly constant until the average threshold increased to a value of 1.5. At this point, 55 to 60 nodes adequately cover the pattern space. As the average threshold increased between 1.5 and 2.5, the number of nodes continued to decrease dramatically while the classification performance decreased slowly. As the average threshold increased past 2.5, the performance of the network deteriorates rapidly. The robustness of networks, trained in this manner with an average threshold of 2.0, is shown in Table 5.7.

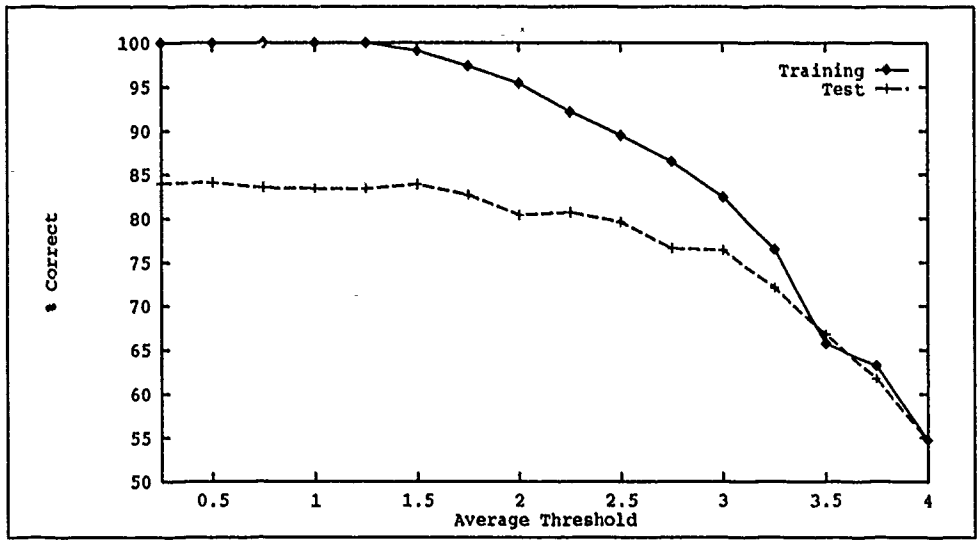

Figure 5.10. Performance vs Average Threshold for Center at Class Averages

Table 5.7. Robustness Measure of Center at Class Averages

	% Correct	
	Training	Test
Avg	95.39	80.40
Std	1.96	3.17

It is interesting to compare the results of this training algorithm to the results obtained from the K-Means algorithm. Both algorithms are clustering algorithms which set the weights of their clusters equal to the averages of the features of pockets of data. It can be seen that the robustness measure for the Center at Class-Cluster Averages algorithm, which used an average of 24 nodes in the hidden layer, layer 1, was roughly equivalent to that of the K-Means algorithm which used 60 nodes. This shows that centering the nodes at pockets of patterns according to class may be better than centering the nodes at pockets of data without regard to class.

5.2.4.5 PNN Training In this test, the performance of the Probabilistic Neural Network (PNN), developed by Specht, versus the RBF Kernel Classifier was analyzed. The number of nodes in the hidden layer was set equal to the number of training points, 102. The weights for the layer 1 nodes were set using the Nodes at Data Points algorithm. For the PNN, the output layer, layer 2, nodes were only connected to the hidden layer nodes representing their class. The weights connecting these hidden layer nodes to the respective output layer nodes were set to one.

For the Radial Basis Function (RBF) network, the weights linking the nodes in the output layer to the nodes in the first layer were trained via global minimization of the MSE. The sigmas for each network were then allowed to vary from .5 to 3.0. The categorization performance of the network for both the training and test data were then documented as shown in Tables E.17, E.18, E.19, and E.20 and plotted as shown in figures 5.11 and 5.12.

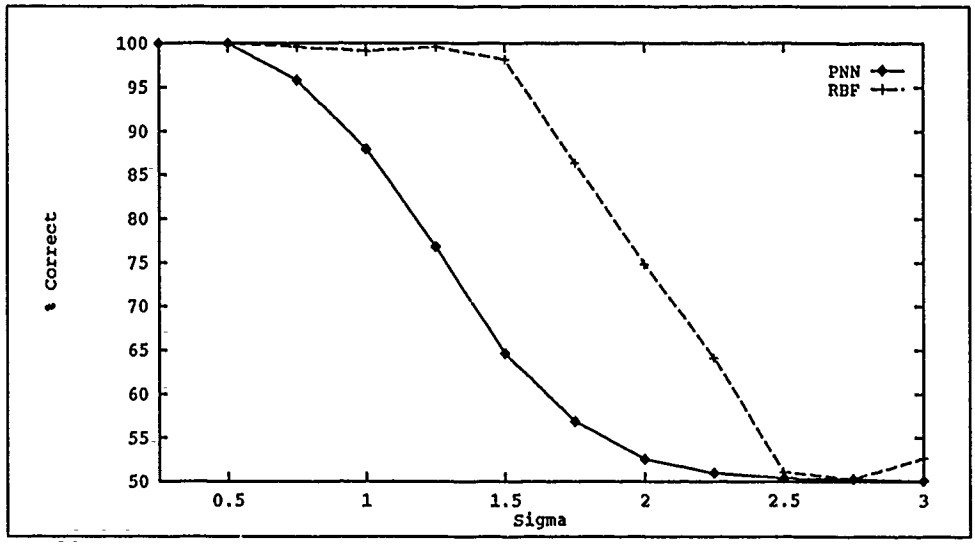

Figure 5.11. PNN vs RBF Performance for Training Data

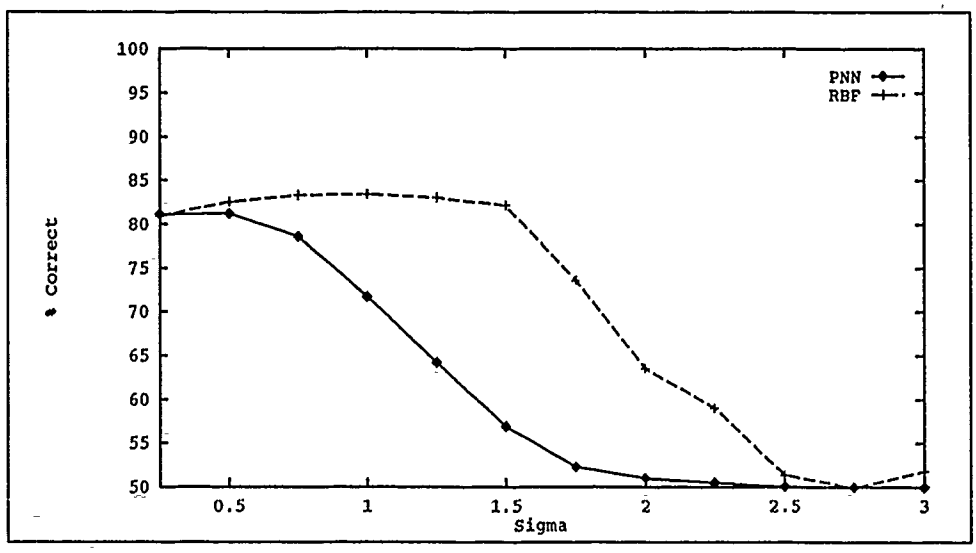

Figure 5.12. PNN vs RBF Performance for Test Data

As the sigmas, or RBF spreads, for both networks were increased to .5, the performances were relatively the same. However, as the sigma increased from .5 to 1.5, the performance of the PNN decreased while the RBF Network remained relatively constant. This indicates the weights in the RBF Network are serving to offset the choice of a bad sigma for the PNN. As sigma increased from 1.75 to 3.0, the performance of the RBF network began to deteriorate rapidly. At a sigma of 3.0, the both networks performed at the same level. The best performance for the PNN Network, as shown in Table 5.8, occurred when sigma was set to .5. On the other hand, the best performance

Table 5.8. Robustness Measure of PNN Network

	% Correct	
	Training	Test
Avg	100.00	82.50
Std	0.00	81.20

for the RBF Network, as shown in Table 5.9, occurred when sigma was set to .75. This shows

Table 5.9. Robustness Measure of RBF Network

	% Correct	
	Training	Test
Avg	99.61	83.30
Std	0.78	5.40

the increase in the sigma, or receptive field spread, allowed the RBF Network to generalize a little better than the PNN Network.

5.2.5 Summary A comparison of the performance of the Hyperplane and Kernel Classifier networks is shown in Tables 5.10 and 5.11.

These tables show both types of networks performed equally well. The Hyperplane Classifier networks used less nodes then the Kernel Classifiers but took longer to train. Similar observations

Table 5.10. Hyperplane Classifier Network Robustness Summary

Output Layer Objective Function	Convergence Iterations	Avg % Correct		
		Training	Test	Total
MSE	30,000	97.16	79.70	88.52
CE	28,000	100.00	81.20	90.69
CFM	50,000	88.83	73.20	81.09

Table 5.11. Kernel Classifier Network Robustness Summary

Layer 1		Avg % Correct		
Training Method	Number of Nodes	Training	Test	Total
Node at Data Points	102	100.00	84.90	93.45
Kohonen Training	64	97.94	80.20	90.05
K-Means Clustering	60	95.59	80.90	89.20
Center at Class Avgs	24	95.39	80.40	88.85
PNN Network	102	100.00	81.20	90.69
RBF Network	102	99.61	83.30	91.54

have been made by Moody (13). The performance of the different Kernel Classifier networks depended on the number of RBF nodes allocated to layer 1. The best performance occurred when an RBF node was placed at each of the 102 training vectors. However, only slightly degraded performance occurred when a lesser number of nodes was allowed to adapt, via the Kohonen Training, K-Means Clustering, and Center at Class-Cluster Averages algorithms, to reflect the data. Finally, the performance of the PNN was inferior to that of the RBF based Kernel Classifier network. This shows the weights linking the layer 1 nodes to the layer 2 nodes in the Kernel Classifier may serve to correct for non-optimal choices of the spreads of the RBFs.

5.3 Radar System Characterization

5.3.1 Introduction This section addresses the development and testing of a neural network capable of categorizing a radar platform from the characteristics of its electromagnetic signal.

5.3.2 Data Description The data deemed necessary to perform classification of radar platforms are the radio frequency of the electromagnetic signal, the pulse repetition interval of the pulsed waveform, the stagger level of the pulse repetition interval, the width of the transmitted pulse, the scan type used by the radar, and the circular period of the scan.

5.3.2.1 Radio Frequency The Radio Frequency (RF), as shown in figure 5.13, of a radar platform's transmitted signal represents the frequency, in hertz, at which the carrier waveform is transmitted. This feature is not limited to a specific frequency for each platform but can vary within a given a range, or band, of frequencies over which the carrier is transmitted. Also, many radar platforms transmit their carrier in several different RF bands, depending on the mode in which the platform is operating.

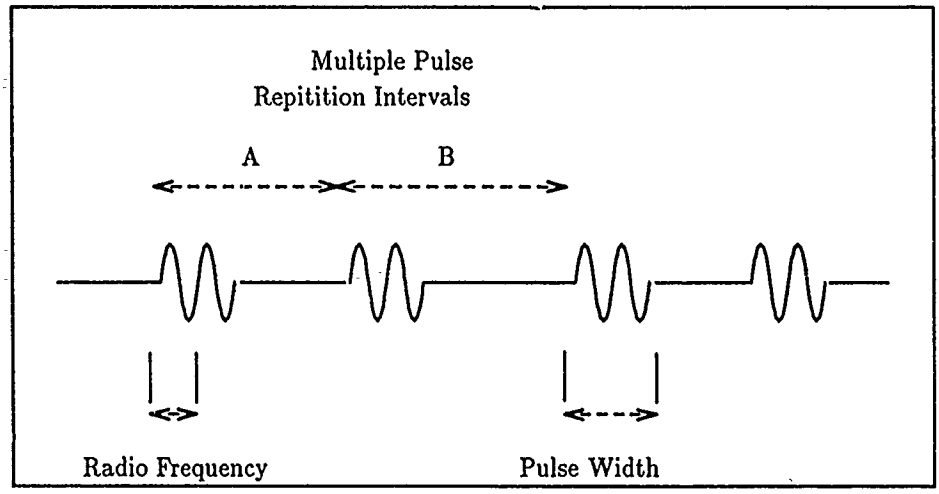

Figure 5.13. Radar Signal

5.3.2.2 Pulse Repetition Interval The Pulse Repetition Interval (PRI), as shown in figure 5.13, of a radar platform's transmitted signal is the interval of time, in microseconds, between consecutive transmitted pulses. This feature is not limited to a specific time for each platform but can vary within a given range of time. Also, many radar platforms change their PRI operating bands depending on the mode in which the platform is operating.

5.3.2.3 Stagger Level The use of more than one pulse repetition frequency, as shown in figure 5.13, by a radar platform is known as its Stagger Level. This change in repetition frequency of the transmitted pulses allows the radar to overcome the blind speeds inherent in the detection of moving targets.

5.3.2.4 Pulse Width The Pulse Width (PW), as shown in figure 5.13, of a radar platform's transmitted signal is the time duration, in microseconds, of the transmitted pulses. This characteristic determines the radar's ability to resolve closely spaced targets within its range. This feature is not limited to a specific unit of time for each platform but can vary within a given range of times. Furthermore, many radar platforms can change the range of the PW of their transmitted signals.

5.3.2.5 Scan Type The Scan Type is the method the radar platform uses, such as a conical scan, to direct its antenna beam at a target. This study concentrated o circular scan platforms with different types of scans.

5.3.2.6 Circular Scan Period The Circular Scan Period is the length of time, in milliseconds, the radar platform takes to repeat one frame of its search. This feature is not limited to a specific unit of time for each platform but can vary within a given range of times. Also, many radar platforms can change the length of their Circular Scan Period depending on the mode in which the platform is operating.

5.3.3 Data Processing

5.3.3.1 Data Normalization The data used to develop the network does not represent actual radar platform features but sets the ranges allocated to each platform in the form of a range vector. Because the comparative ranges for each of the patterns were unequal, the data was normalized by dividing each feature of the data by one-half the maximum value allowed for that feature. This limits the range for each of the feature components to a number between 0 and 2. Also, in this problem, several of the ten radar platforms had features which overlapped in the feature space.

5.3.3.2 Data Generation For each platform the training and test vectors were generated randomly. This was accomplished by taking each range vector within a class and generating a set number of random pattern vectors, according to a uniform distribution, using the range vector as a template. For the RF, PRI, PW and Scan Period features, the random number generated was forced to reside inside the ranges allocated to each platform. For the Scan Type and the Stagger Level, the features were the same as the template vector.

5.3.4 Network Development For this problem, Kernel Classifier networks were developed to categorize ten radar platforms. Since the networks developed using the Center at Class-Cluster Averages algorithm performed comparatively well, for a smaller number of nodes, for the communications signal categorization problem, these Kernel Classifier networks were also developed using this algorithm. For these networks, a correct classification resulted when the output of the correct classification node exceeded a classification threshold.

5.3.4.1 RBF Network This network was constructed according to the topology shown in figure 3.7. The input layer, layer 0, contained six nodes while the nodes in the output layer, layer 2, contained ten nodes, one node representing each class. The network used the Center at Class-Cluster Averages algorithm to determine the number of nodes in the hidden layer, layer 1, and their corresponding weights. These nodes were assigned the gaussian transfer function while the nodes in the output layer were assigned the linear transfer function. The sigmas for the hidden layer

nodes were established using the Scale Sigmas by Class Interference algorithm while the weights linking the output layer nodes with the hidden layer nodes set via global minimization of the MSE. Three hundred pattern vectors, 30 from each platform, were then used to train the network while 1990 pattern vectors, about 200 from each platform, were used to test the network. Measurements were then taken of the performance of the network as the classification threshold varied. This data is shown in Table E.21. and plotted in figure 5.14. From this figure, it can be seen that

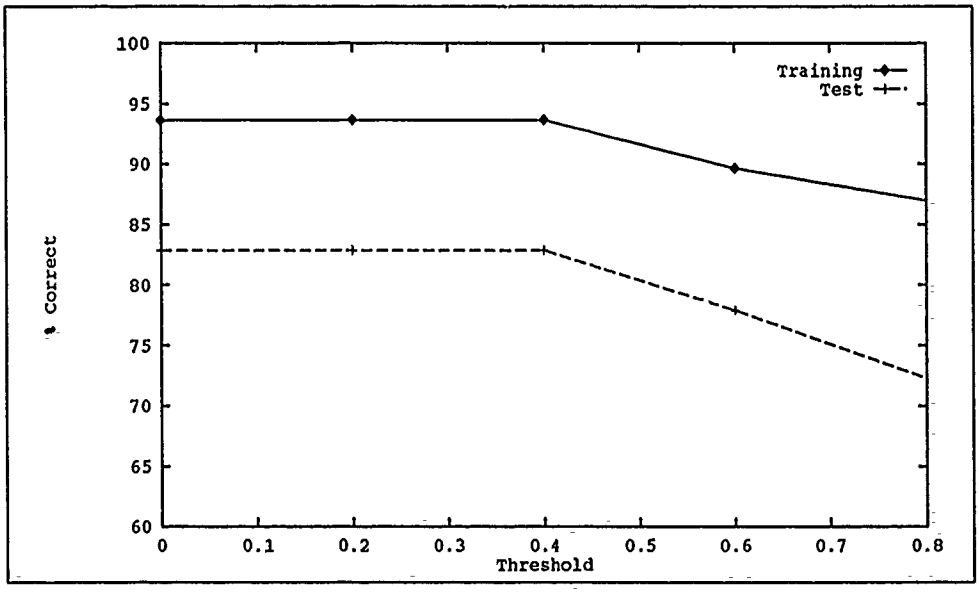

Figure 5.14. Performance Radial Basis Function Network for Radar Data

the categorization performance of the network decreased as the class threshold increased until a classification threshold of .4 was reached. At this point the categorization performance leveled out at 93.67 % for the training data and 82.88 % for the test data. This indicates that if the output of a node was greater than .4, there is an 82.88% chance that a correct classification was made. If the output of a node was greater than .8, there is a 72.29% chance that a correct classification was made. The inability of the network to train at 100 % was due to the overlap of the parameters in the feature space from different platforms.

5.3.4.2 Arbitrator Network This network was constructed according to the topology shown in figure 5.15. In this system, Network A was trained to categorized platforms 1-5 while Network B was trained to categorized platforms 6-10. Network C was trained to categorize all ten platforms by arbitrating between Network A and Network B.

For Networks A and B, the input layer, layer 0, contained six nodes while the nodes in

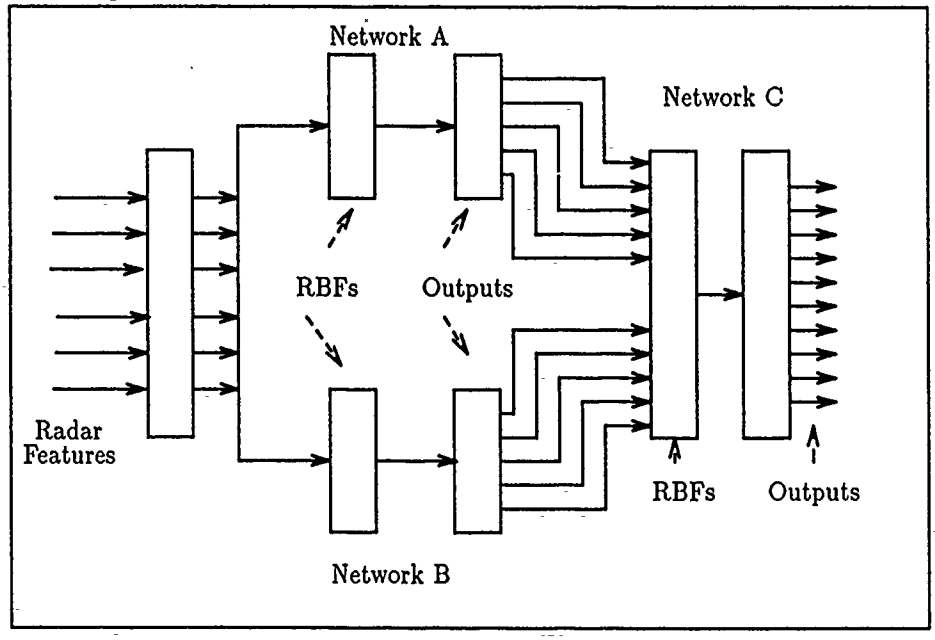

Figure 5.15. Radar Data Arbitration Network

their output layer, layer 2, contained five nodes, one node representing each class. These networks used the Center at Class-Cluster Averages algorithm to determine the number of nodes in the hidden layer, layer 1, and their corresponding weights. These nodes were assigned the gaussian transfer function while the nodes in the output layer were assigned the linear transfer function. For Network A, the sigmas for the hidden layer nodes were established using the Scale Sigmas by Class Interference algorithm while, for Network B, the sigmas were set using the P-Neighbors algorithm. For both networks, the weights linking the output layer nodes with the hidden layer nodes were set via global minimization of the MSE. Three hundred pattern vectors, 60 from each platform, were then used to train each network independently. Network A was then tested with 1000 pattern vectors from classes one through five, group A, while Network B was tested with 990 pattern vectors from classes six through ten, group B. Measurements were taken of the performance each network for as the classification threshold varied. This data is shown in Tables E.22 and E.23 and plotted in figures 5.16 and 5.17.

After the parameters for Network A and B were set, Network C was established to arbitrate the outputs between Networks A and B. This network had ten nodes in its input layer, one for each of the ten outputs from the Networks A and B, and ten nodes in its output layer; one for each radar platform. The Center as Class-Cluster Averages algorithm was used to determine the

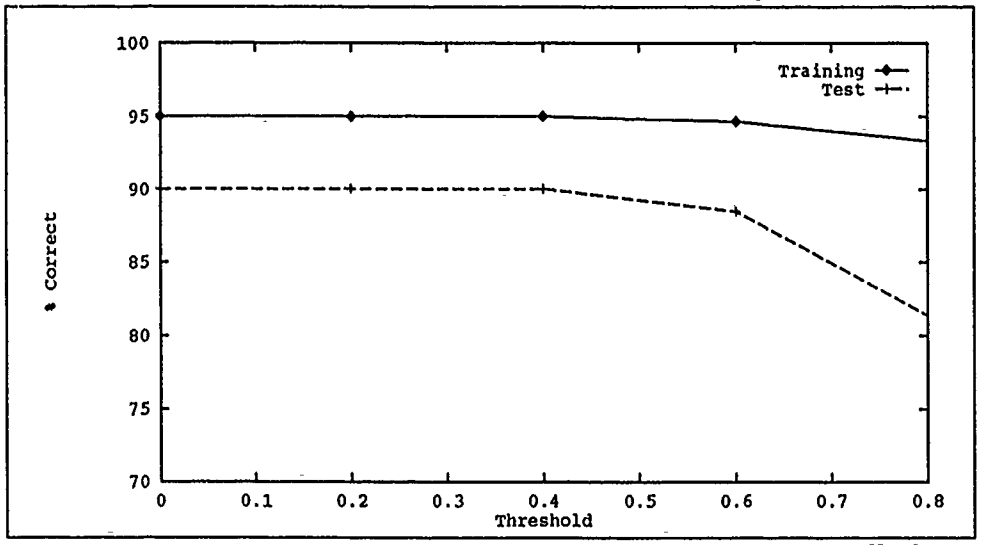

Figure 5.16. Performance of Network A for Group A Radar Data

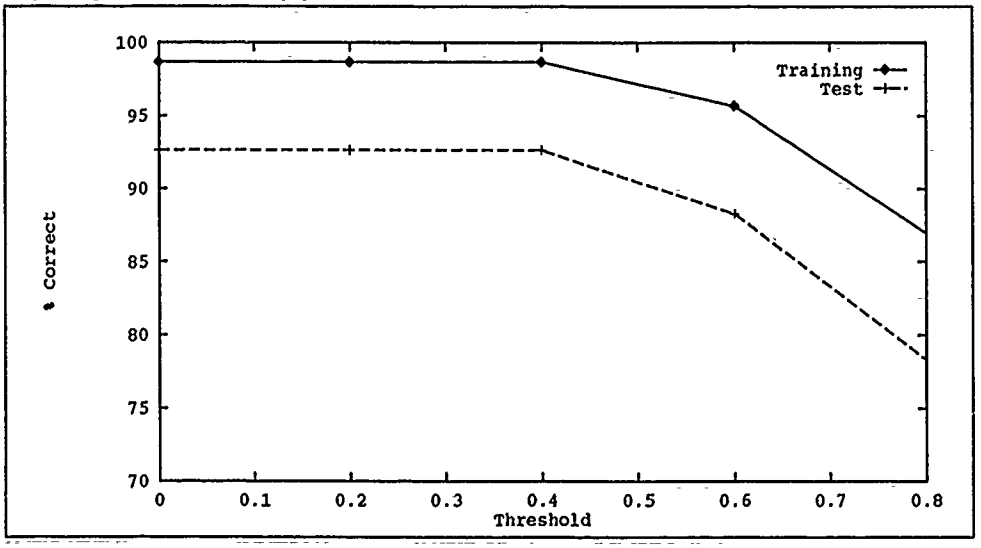

Figure 5.17. Performance of Network B for Group B Radar Data

number of nodes in its hidden layer, layer 1, and their corresponding weights. These nodes were assigned the gaussian transfer function while the nodes in the output layer were assigned the linear transfer function. The sigmas for the hidden layer nodes were established using the Scale Sigmas by Class Interference algorithm while the weights linking the output layer nodes with the hidden layer nodes set via global minimization of the MSE. Three hundred pattern vectors, 30 from each platform, were then used to train the network while 1990 pattern vectors, about 200 from each platform, were used to test the network. Measurements were then taken of the performance of the network as the classification threshold varied. This data is shown in Table E.24 and plotted in figure 5.18.

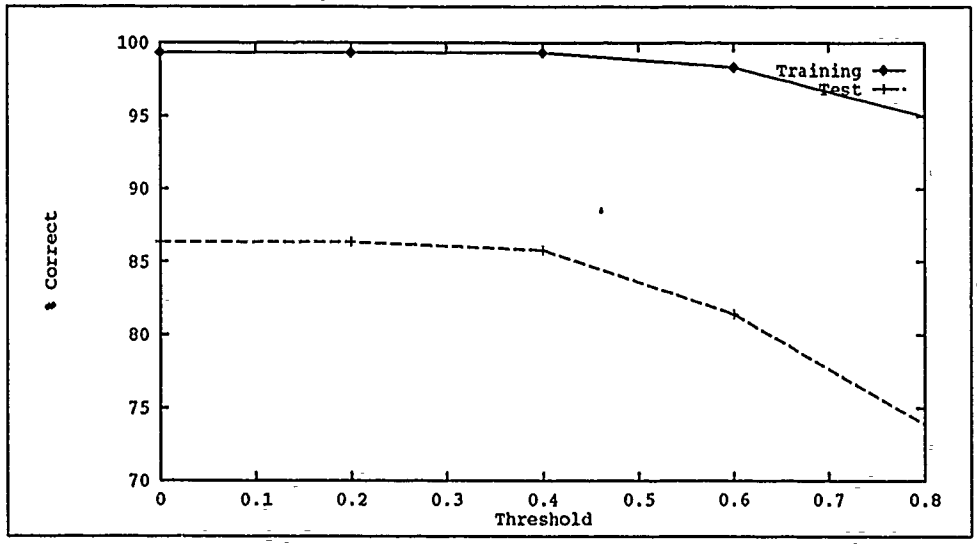

Figure 5.18. Performance of Arbitrator Network for Group A and B Radar Data

From these figures, it can be seen that the categorization performance of the network increased as the class threshold decreased until the class threshold reached .4 . At this point the classification accuracy of the total network was 99.33% for the training data 86.35% for the test data. This indicates that if the output of a node was greater than .2, there is an 86.35% chance that a correct classification was made. If the output of a node was greater than .8, there is a 73.90% chance that a correct classification was made. The inability of the network to train at 100 % was due to the overlap of the parameters in the feature space from different platforms.

5.3.4.3 Summary A summary of the performance of the networks trained to classify the radar data is shown in Table 5.12. A comparison of the performance of the RBF network and the Arbitrator network shows the Arbitrator network's performance was about 2% better than that

Table 5.12. Radar Categorization Summary

Classification Threshold	RBF Network % Correct		Arbitrator Network % Correct	
	Training	Test	Training	Test
.8	87.00	72.29	95.00	73.90
.6	89.67	77.91	98.33	81.43
.4	93.67	82.88	99.33	85.79
.2	93.67	82.88	99.33	86.35
.0	93.67	82.88	99.33	86.35

of the RBF network. This is probably due to the ability to the use of more training vectors to train networks A and B in the Arbitrator scheme.

5.4 Summary

This chapter began by discussing the data used to train and test neural networks to classify coded digital communication signals. After describing the methods used to train various networks to solve this classification problem, the results of the training and testing were then presented. This chapter concluded with a discussion of the data used to train and test neural networks to classify radar systems and an analysis of these training and test results.

VI. Conclusions/Recommendations

6.1 Introduction

The purpose of this thesis was to characterize the Hyperplane and Kernel Classifier types of neural networks and determine either type could be used to accurately characterize radar signal. After drawing some conclusions on the performance of each type of network, based the test results discussed in Chapter 5, this chapter will recommend areas of future study.

6.2 Conclusions

6.2.1 Hyperplane Classifier In this thesis, Hyperplane Classifier networks were constructed using the MSE, CE and CFM objective functions. For the communications problem, each network contained 18 nodes in the first hidden layer, ten nodes in the second hidden layer and two nodes in the output layer. Under these conditions, the network trained by minimizing the CE performed slightly better, in the area of classification, than the networks trained via MSE and CFM objective functions. The average CE performance on the training data was 100% while on the test data the performance dropped to 81.2%. This compares favorably with the MSE performances of 97.16% and 79.7% and the CFM performances of 88.83% and 73.20% for the training and test data respectively. Furthermore, the CE algorithm converged about twice as fast as the MSE algorithm and seven times as fast as the CFM algorithm. As shown in figures 5.1, 5.2, and 5.3, the CE algorithm reached a 90% accuracy in about 7000 iterations while the MSE and CFM algorithms reached this level in about 19000 and 50000 iterations respectively. This decrease in convergence time is due to the lack of a $y_n(1-y_n)$ term in the CE update equations.

6.2.2 Kernel Classifier For this thesis, several different Kernel Classifier networks were developed to solve the communications signal categorization problem discussed in Chapter 5. Based on the performance results detailed in Chapter 5, two Kernel Classifier types of networks were then constructed to categorize radar systems.

6.2.2.1 Communications Data The performance of networks trained using the Nodes at Data Points was found to be a critical function of the number of nodes used in the hidden layer. As the number of nodes increased from 0 to 102, the classification performance of the network increased accordingly; peaking at 100% for the training data and 84.90% for the test data for 102 nodes. However, for less than 60 nodes, the performance of the network deteriorated rapidly.

The performance of the networks trained using the Kohonen Training algorithm were found to be a function of the both the number of nodes allocated to the hidden layer and the number of

P nearest neighbors used in the calculation of the spreads of the RBFs. As the number of nodes increased from 0 to 81 the classification performance of the network increased, peaking at 100% for the training data but only around 76% for the test data. This is due to the fact that the Kohonen Training algorithm allocates more nodes to areas in the feature space where there are pockets of data. Increasing the total number of nodes in the hidden layer increases the amount of nodes which can be allocated to each pocket. However, this increase in the number of nodes within these pockets serves to decrease the distance to the P nearest neighbors. Thus, the spreads for the RBFs become small and the network loses its ability to generalize past the training data. As shown in figure 5.6, this problem can be overcome by using less nodes in the Kohonen layer. For this data, the optimum number of nodes was 64. Though the performance over the training data was only 97.94%, the performance over the test data improved to 80.20%.

The performance of the networks trained using the K-means algorithm were also found to be a function of the number of nodes allocated to the hidden layer. As the number of nodes increased from 0 to 100, the classification performance of the network generally increased; peaking at 100% for the training data and 83.90% for the test data. The performance remained above 80% for both the test and training data until less than 60 nodes were allocated to layer 1. At this point the classification performance of the network decreased rapidly. This is probably due to the grouping of dissimilar classes into the same clusters. The performance of networks trained using the K-means algorithm were found to be somewhat invariant to the number of P nearest neighbors, used in the calculation of the spreads of the RBFs.

The performance of networks trained using the Center at Class-Cluster Averages algorithm was also found to be a function of the number of nodes used in the hidden layer. As the number of nodes increased from 0 to 100, the classification performance of the network increased accordingly, peaking at 100% for the training data and 84% for the test data. This indicates there was redundant data in the training set. Also, networks trained in this manner performed at a performance level of above 80%, for both the training and test data, as long as the number of nodes remained above 20. This indicates the data in this problem may be grouped, by class, into small pockets of data.

The performance of the networks trained using the PNN algorithm were found to be a function of the spreads assigned to the RBF nodes. As the spreads decreased, from 3 to .25, the classification performance of the network increased; peaking at 100% for the training data and 81.29% for the test data. The performance of an RBF network, with the weights for the hidden layer nodes trained in the same manner at those for the PNN, was found to be less of a function of the spread assigned to the RBF nodes. As shown in figures 5.11 and 5.12, even as the spreads increased to 1.5, the performance of the RBF network remained relatively constant for both the training and test data.

This is due to the weights in the output layer compensating for the poor choice of the spread in the hidden layer. However, as the spread of the RBFs increased past 1.5, the performance of the RBF network deteriorated to that of the PNN.

6.2.2.2 Radar Data For this problem, ten different radar platforms had to be categorized from data concerning their electromagnetic signals.

The performance of the standard RBF network developed to solve this problem was a function of the classification threshold required to assign a correct decision for the network. For a classification threshold of .8, indicating the output for the correct node was above .8, the performance of the network was 87% for the training data and 72.29% for the test data. As the classification threshold was decreased to .4, the performance of the network increased to 93.67 % and 82.88 % for the training and test data respectively. This performance then remained constant as the threshold decreased. The inability of the network to train at the 100% level indicated the overlap in the feature space between patterns of different classes. The 10% difference between the performance of the network for training and test data followed the general performance degradation found in the communications problem.

The performance of the Arbitrator network developed to solve this problem was also a function of the classification threshold required to assign a correct decision for the network. For a classification threshold of .8, indicating the output for the correct node was above .8, the performance of the network was 95% for the training data and 73.90% for the test data. As the classification threshold was decreased to .2, the performance of the network increased to 99.33 % and 86.35 % for the training and test data respectively. This performance then remained constant as the threshold decreased. Again, the failure of the network to train a 100 % indicates the overlap in the features space between patterns of different classes.

In general, the performance of the Arbitrator network was between 1.6% to 3.5% better than that of the standard RBF network. This is due to the establishment of the two subnetworks to distinguish between smaller groups of platforms, the use of more training data to train these networks, and the of ability of the Arbitrator network to separate the overlap between the groups. However, this increase in performance for the Arbitrator network required almost three times as many nodes as that of the standard RBF network. Furthermore, since the two subnetworks were trained separately, the total training time for the Arbitrator network was almost three times that of standard RBF network of 45 minutes.

6.3 *Recommendations*

From the results found in this thesis, better algorithms need to be developed to set the spreads of the radial basis function nodes for the Kernel Classifier RBF networks. In order to optimize the performance of the networks, a trail and error basis was used to select the algorithm, and algorithm parameters, to set the spreads. A single, adaptive, algorithm to accomplish this same task could improve the performance of these networks. Also, it may be beneficial to study accuracies of Kernel Classifier networks developed using the backpropagation algorithms to set the weights connecting nodes in layer 1 to nodes in layer 2. In this this thesis, these weights were set via a global minimization of the MSE using a matrix inversion algorithm. Since this algorithm used Gaussian Elimination of the rows to obtain the inverse, it is possible that small roundoff errors could have accumulated in such a manner as to prevent a true global MSE minimization from being obtained. Another area of research, which may hold promise, is the use of the RBF networks to reconstruct functions from past samples. These networks could be useful in signal processing applications in which a reconstruction of an unknown signal is needed in a timely manner.

6.4 *Summary*

This chapter presented some conclusions, based on the test results discussed in Chapter 5, on the performance of the Hyperplane and Kernel Classifier neural networks. For the data processed in this theses, it was found that the Kernel Classifier networks could perform at the same level as that of the Hyperplane Classifiers and be developed in a much shorter time period. However, these Kernel Classifier networks usually required more nodes. Some areas of future research may include the development of algorithms to automatically establish the spreads of the RBFs in the Kernel Classifiers and the application of Kernel Classifer networks to the problem of function reconstruction based on sampled values.

Appendix A. *Objective Function Analysis*

A.1 Introduction

In this appendix, the classification properties of the Mean Square Error, Cross Entropy and Classification Figure of Merit objective functions will be analyzed.

A.2 Mean Square Error (MSE) Function

This section will show how the mean square error for proper network classification can be greater than the mean square error for an improper classification.

Suppose the network was required to make a classification of an unknown pattern into one of N classes. That is, the neural network was developed such that each output node represents only one of the N classes. Let d_n be the desired output for the n^{th} node for a given input pattern. The MSE, for a given input pattern, is then defined as

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)^2 \tag{A.1}$$

Usually during training, d_n is taken to be 1 for the node responsible for a class and 0 for the rest of the output nodes. Specifically, assume that, after training, a test pattern is applied and the correct node on the output layer has the highest activation. In this case the maximum MSE is

$$MSE_{max} \approx \frac{N-1}{N} \tag{A.2}$$

This occurs when the desired node produces a 1 and all of the other nodes produce activation values very close to 1. That is

$$MSE_{max} = \frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)^2 = \frac{1}{N}[(1-1)^2 + \sum_{n=1}^{N-1}(1-0)^2] \tag{A.3}$$

This reduces to

$$MSE_{max} = \frac{1}{N}\sum_{n=1}^{N-1}(1)^2 \tag{A.4}$$

Simplifying,

$$MSE_{max} = \frac{N-1}{N} \tag{A.5}$$

Next, assume that, after training, a test pattern is applied and an incorrect classification is made. That is, a node representing an incorrect classification in the output layer has the highest activation. In this case the minimum MSE is

$$MSE_{min} = \frac{1}{2N} \tag{A.6}$$

This occurs when the output for the correct node is approximately .5 and the output for a single incorrect node is approximately .5. The outputs for the rest of the nodes are zero. That is

$$MSE_{min} = \frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)^2 \tag{A.7}$$

Substituting the output values gives

$$MSE_{min} = \frac{1}{N}[(1-.5)^2 + (0-.5)^2] \tag{A.8}$$

which simplifies to

$$MSE_{min} = \frac{1}{2N} \tag{A.9}$$

Thus, under certain conditions, the MSE for a correct response, in which the correct output node is the highest, can be greater than the MSE for an incorrect response. As the number of classes becomes very large, then MSE_{max} for a correct response approaches 1 while MSE_{min} for an incorrect response approaches 0. Therefore, the ratio of MSE_{max} to MSE_{min} is

$$lim_{N\to\infty}\frac{MSE_{max}}{MSE_{min}} = \frac{1}{0} = \infty \tag{A.10}$$

A.3 Cross Entropy (CE) Function

This section will show how the cross entropy for proper network classification can be greater than the cross entropy for an improper classification.

Suppose the network was required to make a classification of an unknown pattern into one of N classes. That is, the neural network was developed such that each output node represents only one of the N classes. Let d_n be the desired output for the n^{th} node for a given input pattern. The CE is then defined as

$$CE = -\frac{1}{N}\sum_{n=1}^{N}[d_n \log(y_n) + (1 - d_n)\log(1 - y_n)] \tag{A.11}$$

Here during training, d_n is usually set to 1 for the node responsible for a class and to 0 for the rest of the output nodes. Now, assume that, after training, a test pattern is applied and the correct node on the output layer has the highest activation. In this case the maximum CE is

$$CE_{max} \approx \infty \tag{A.12}$$

This occurs when the correct classification node produces a 1 and at least one of the other nodes produces an activation value very close to 1. That is

$$CE = -\frac{1}{N}\sum_{n=1}^{N}[d_n \log(y_n) + (1 - d_n)\log(1 - y_n)] \tag{A.13}$$

Substituting the values for the outputs

$$CE_{max} = -\frac{1}{N}[1 \log(1) + (1)\log(1 - 1)] \tag{A.14}$$

or

$$CE_{max} = \infty \tag{A.15}$$

Next, assume that, after training, a test pattern is applied and an incorrect node on the output layer has the highest activation. In this case the minimum CE is

$$CE_{min} = \frac{2\log(2)}{N} \tag{A.16}$$

This occurs when the output for the correct node is approximately .5 and the output for a single incorrect node is approximately .5. The outputs for the rest of the nodes are zero. That is

$$CE = -\frac{1}{N}\sum_{n=1}^{N}[d_n \log(y_n) + (1 - d_n)\log(1 - y_n)] \tag{A.17}$$

Substituting the assumed values for the outputs

$$CE_{min} = -\frac{1}{N}[1 \log(.5) + (1 - 0)\log(1 - .5)] \tag{A.18}$$

which simplifies to

$$CE_{min} = \frac{2\log(2)}{N} \tag{A.19}$$

Thus, under certain conditions, the CE for a correct response, in which the correct output node is the highest, can be greater than the CE for an incorrect response. The ratio of CE_{max} for a correct response to CE_{min} for an incorrect response is

$$lim_{N\to\infty}\frac{CE_{max}}{CE_{min}} = \infty \tag{A.20}$$

A.4 Classification Figure of Merit (CFM) Function

This section will show how the CFM objective function seeks to alleviate the classification error associated with the MSE and CE objective functions.

Suppose the network was required to make a classification of an unknown pattern into one of N classes. That is, the neural network was developed such that each output node represents only one of the N classes. The CFM objective function is then defined as follows:

$$CFM = \frac{1}{N-1}\sum_{n=1_{n\neq c}}^{N}\frac{\alpha}{1 + e^{(-\beta\delta_n + \zeta)}} \tag{A.21}$$

where $\delta_n = y_c - y_n$

y_c = response of the correct node

y_n = response of the incorrect node

N = total number of output nodes or classes.

α = sigmoid scaling parameter.

β = sigmoid discontinuity parameter.

ζ = sigmoid lateral shift parameter.

Specifically, assume that, after training, a test pattern is applied to the network and the correct node on the output layer has the highest activation. In this case the minimum CFM is

$$CFM_{min} \approx CFM_n(0) \tag{A.22}$$

where

$$CFM_n(\delta_n) = \frac{\alpha}{1 + e^{(-\beta\delta_n + \zeta)}} \tag{A.23}$$

This occurs when the correct output node produces a 1 and all of the other output nodes produce activation values very close to 1. That is

$$CFM = \frac{1}{N-1} \sum_{n=1_{n \neq c}}^{N} \frac{\alpha}{1 + e^{(-\beta\delta_n + \zeta)}} \tag{A.24}$$

but here $\delta_n = y_c - y_n \approx 0$ and therefore, CFM_{min} simplifies to

$$CFM_{min} = \frac{1}{N-1} \sum_{n=1_{n \neq c}}^{N} \frac{\alpha}{1 + e^{[-\beta(0) + \zeta]}} = \frac{1}{N-1} \sum_{n=1_{n \neq c}}^{N} CFM(0) \tag{A.25}$$

which can be written as

$$CFM_{min} \approx \frac{1}{N-1}(N-1)CFM(0) \tag{A.26}$$

This simplifies to

$$CFM_{min} = CFM(0) \tag{A.27}$$

Next, assume that, after training, a test pattern is applied and an incorrect node on the output layer has the highest activation. In this case the maximum CFM is

$$CFM_{max} = \frac{1}{N-1}[(N-2)CFM_n(1) + CFM_n(0)] \tag{A.28}$$

This occurs when the output for the correct node is approximately 1 and the output for one of the remaining nodes is approximately 1 while the rest are zero. For the nodes which have output very near zero

$$CFM(\delta_n) = \frac{1}{1+e^{(-\beta y_c+\zeta)}} = \frac{1}{1+e^{(-\beta+\zeta)}} = CFM(1) \tag{A.29}$$

The equation for the CFM can now be written as

$$CFM = \frac{1}{N-1}[\sum_{n=1_{n\neq c}}^{N-1} \frac{\alpha}{1+e^{(-\beta+\zeta)}} + \frac{\alpha}{1+e^{[-\beta(0)+\zeta]}}] \tag{A.30}$$

This can be written as

$$CFM_{max} = \frac{1}{N-1}[\sum_{n=1_{n\neq c}}^{N-1} CFM(1) + CFM(0)] \tag{A.31}$$

Expanding via the summation

$$CFM_{max} = \frac{1}{N-1}[(N-2)CFM(1) + CFM(0)] \tag{A.32}$$

Now, taking the limit as N approaches infinity

$$CFM_{max} \approx CFM_n(1) \tag{A.33}$$

For large β, then $CFM_{max} \approx 1$. Thus, the ratio of CFM_{max} for a correct response to CFM_{min} for an incorrect response, when $\zeta = 0$ is

$$\frac{CFM_{max}}{CFM_{min}} = 2 \tag{A.34}$$

Comparing this to the CE and MSE functions, we find that the CFM function has less region in the feature space where an incorrect classification will be made.

Appendix B. *Hyperplane Classifier Parameter Update Equations*

B.1 Introduction

In this appendix, the update equations for each of the objective functions will be derived. These equations will be based on a network with the topology defined in figure 3.3.

B.2 Identities

In this section, the identities needed to establish the update algorithms will be derived. Consider a feedforword artificial neural network as shown in figure 3 with the following paramenters:

Layer 0, input layer, has K possible nodes

Layer 1, first hidden layer, has L possible nodes

Layer 2, the second hidden layer has M possible nodes.

Layer 3,the output layer, has N possible nodes.

Let the weights between layers be defined as follows:

w_{kl} = weight linking node k in layer 0 to node l in layer 1.

w_{lm} = weight linking node l in layer 1 to node m in layer 2.

w_{mn} = weight linking node m in layer 2 to node n in layer 3.

Let the offsets for the nodes in each layer be defined as follows:

σ_l = offset for node l in layer 1.

σ_m = offset for node m in layer 2.

σ_n = offset for node n in layer 3.

Let the transfer function for the nodes in each layer be as follows:

Layer 0 - $y_k = x_k$ Outputs are same as input features.

Layer 1 - $y_l = [1 + e^{-(\sum_{k=1}^{K} w_{kl} y_k + \sigma_l)}]^{-1}$

Layer 2 - $y_m = [1 + e^{-(\sum_{l=1}^{L} w_{lm} y_l + \sigma_m)}]^{-1}$

Layer 3 - $y_n = [1 + e^{-(\sum_{m=1}^{M} w_{mn} y_m + \sigma_n)}]^{-1}$

Now, looking at the node outputs for each of the layers, the following identities need to be established:

$$\frac{\partial y_N}{\partial w_{MN}}, \frac{\partial y_N}{\partial w_{LM}}, \frac{\partial y_N}{\partial w_{KL}}, \frac{\partial y_M}{\partial w_{LM}}, \frac{\partial y_M}{\partial w_{KL}}, \frac{\partial y_L}{\partial w_{KL}}$$

Starting with y_N, the output for a node in layer 3

$$\frac{\partial y_N}{\partial w_{MN}} = \frac{\partial}{\partial w_{MN}}[1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-1} \tag{B.1}$$

which expands to

$$\frac{\partial y_N}{\partial w_{MN}} = -[1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-2} \frac{\partial}{\partial w_{MN}}[e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}] \tag{B.2}$$

which simplifies to

$$\frac{\partial y_N}{\partial w_{MN}} = -[1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-2} [-e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}] \frac{\partial}{\partial w_{MN}}(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N) \tag{B.3}$$

This equation can be written as

$$\frac{\partial y_N}{\partial w_{MN}} = [1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-1} [1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-1} [e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}] y_M \tag{B.4}$$

which finally simplifies to

$$\frac{\partial y_N}{\partial w_{MN}} = y_N (1 - y_N) y_M \tag{B.5}$$

Similarly, using the same arguments as above

$$\frac{\partial y_M}{\partial w_{LM}} = y_M (1 - y_M) y_L \tag{B.6}$$

and

$$\frac{\partial y_L}{\partial w_{KL}} = y_L (1 - y_L) y_K \tag{B.7}$$

Now let's find $\frac{\partial y_N}{\partial w_{LM}}$.

$$\frac{\partial y_N}{\partial w_{LM}} = \frac{\partial}{\partial w_{LM}}(1 + e^{-\phi_{mN}})^{-1} \tag{B.8}$$

where

$$\phi_{mN} = \sum_{m=1}^{M} w_{mN} y_m + \sigma_N \tag{B.9}$$

therefore

$$\frac{\partial y_N}{\partial w_{LM}} = \frac{\partial}{\partial w_{LM}} (1 + e^{-\phi_{mN}})^{-1} \tag{B.10}$$

which expands to

$$\frac{\partial y_N}{\partial w_{LM}} = -(1 + e^{-\phi_{mN}})^{-2} \frac{\partial}{\partial w_{LM}} (e^{-\phi_{mN}}) \tag{B.11}$$

this, in turn, simplifies to

$$\frac{\partial y_N}{\partial w_{LM}} = (1 + e^{-\phi_{mN}})^{-2} (e^{-\phi_{mN}}) \frac{\partial}{\partial w_{LM}} (\phi_{mN}) \tag{B.12}$$

which can be written as

$$\frac{\partial y_N}{\partial w_{LM}} = y_N (1 - y_N) \frac{\partial}{\partial w_{LM}} (\sum_{m=1}^{M} w_{mN} y_m + \sigma_N) \tag{B.13}$$

which simplifies to

$$\frac{\partial y_N}{\partial w_{LM}} = y_N (1 - y_N) w_{MN} \frac{\partial y_M}{\partial w_{LM}} \tag{B.14}$$

Substituting equation B.6 for $\frac{\partial y_M}{\partial w_{LM}}$ gives

$$\frac{\partial y_N}{\partial w_{LM}} = y_N (1 - y_N) w_{MN} y_M (1 - y_M) y_L \tag{B.15}$$

Similarly

$$\frac{\partial y_M}{\partial w_{KL}} = y_M (1 - y_M) w_{LM} y_L (1 - y_L) y_K \tag{B.16}$$

Finally, let's find $\frac{\partial y_N}{\partial w_{KL}}$.

$$\frac{\partial y_N}{\partial w_{KL}} = \frac{\partial}{\partial w_{KL}} (1 + e^{-\phi_{Nm}})^{-1} \tag{B.17}$$

where

$$\phi_{Nm} = \sum_{m=1}^{M} w_{mN} y_m + \sigma_N \tag{B.18}$$

therefore

$$\frac{\partial y_N}{\partial w_{KL}} = \frac{\partial}{\partial w_{KL}} (1 + e^{-\phi_{mN}})^{-1} \tag{B.19}$$

which expands to

$$\frac{\partial y_N}{\partial w_{KL}} = -(1 + e^{-\phi_{mN}})^{-2} \frac{\partial}{\partial w_{KL}} (e^{-\phi_{mN}}) \tag{B.20}$$

which further expands to

$$\frac{\partial y_N}{\partial w_{KL}} = (1 + e^{-\phi_{mN}})^{-2} (e^{-\phi_{mN}}) \frac{\partial}{\partial w_{KL}} (\sum_{m=1}^{M} w_{mN} y_m + \sigma_N) \tag{B.21}$$

this simplifies to

$$\frac{\partial y_N}{\partial w_{KL}} = y_N (1 - y_N) \sum_{m=1}^{M} w_{mN} \frac{\partial y_m}{\partial w_{KL}} \tag{B.22}$$

Now substituting from equation B.16 for $\frac{\partial y_m}{\partial w_{KL}}$ provides

$$\frac{\partial y_N}{\partial w_{KL}} = y_N (1 - y_N) \sum_{m=1}^{M} w_{mN} y_m (1 - y_m) w_{Lm} y_L (1 - y_L) y_K \tag{B.23}$$

Finally, let's find $\frac{\partial y_N}{\partial \sigma_N}$, $\frac{\partial y_M}{\partial \sigma_M}$ and $\frac{\partial y_L}{\partial \sigma_L}$. Starting with y_N, the output for a node in layer 3

$$\frac{\partial y_N}{\partial \sigma_N} = \frac{\partial}{\partial \sigma_N} [1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-1} \tag{B.24}$$

which expands to

$$\frac{\partial y_N}{\partial \sigma_N} = -[1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-2} \frac{\partial}{\partial \sigma_N} e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)} \tag{B.25}$$

which simplifies to

$$\frac{\partial y_N}{\partial \sigma_N} = -[1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-2}[-e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}] \frac{\partial}{\partial \sigma_N}(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N) \quad \text{(B.26)}$$

This equation can be written as

$$\frac{\partial y_N}{\partial \sigma_N} = [1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-1}[1 + e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}]^{-1}[e^{-(\sum_{m=1}^{M} w_{mN} y_m + \sigma_N)}] \quad \text{(B.27)}$$

which simplifies to

$$\frac{\partial y_N}{\partial \sigma_N} = y_N(1 - y_N) \quad \text{(B.28)}$$

Similarly, using the same arguments as above

$$\frac{\partial y_M}{\partial \sigma_M} = y_M(1 - y_M) \quad \text{(B.29)}$$

and

$$\frac{\partial y_L}{\partial \sigma_L} = y_L(1 - y_L) \quad \text{(B.30)}$$

Now let's find $\frac{\partial y_N}{\partial \sigma_M}$.

$$\frac{\partial y_N}{\partial \sigma_M} = \frac{\partial}{\partial \sigma_M}(1 + e^{-\phi_{mN}})^{-1} \quad \text{(B.31)}$$

where

$$\phi_{mN} = \sum_{m=1}^{M} w_{mN} y_m + \sigma_N \quad \text{(B.32)}$$

therefore

$$\frac{\partial y_N}{\partial \sigma_M} = \frac{\partial}{\partial \sigma_M}(1 + e^{-\phi_{mN}})^{-1} \quad \text{(B.33)}$$

which expands to

$$\frac{\partial y_N}{\partial \sigma_M} = -(1+e^{-\phi_{mN}})^{-2}\frac{\partial}{\partial \sigma_M}(e^{-\phi_{mN}}) \tag{B.34}$$

this, in turn, simplifies to

$$\frac{\partial y_N}{\partial \sigma_M} = (1+e^{-\phi_{mN}})^{-2}(e^{-\phi_{mN}})\frac{\partial}{\partial \sigma_M}(\phi_{mN}) \tag{B.35}$$

which can be written as

$$\frac{\partial y_N}{\partial \sigma_M} = y_N(1-y_N)\frac{\partial}{\partial \sigma_M}(\sum_{m=1}^{M} w_{mN}y_m + \sigma_N) \tag{B.36}$$

which simplifies to

$$\frac{\partial y_N}{\partial \sigma_M} = y_N(1-y_N)w_{MN}\frac{\partial y_M}{\partial \sigma_M} \tag{B.37}$$

Substituting from equation B.29 for $\frac{\partial y_M}{\partial \sigma_M}$ provides

$$\frac{\partial y_N}{\partial \sigma_M} = y_N(1-y_N)w_{MN}y_M(1-y_M) \tag{B.38}$$

Similarly

$$\frac{\partial y_M}{\partial \sigma_L} = y_M(1-y_M)w_{LM}y_L(1-y_L) \tag{B.39}$$

Finally, lets find $\frac{\partial y_N}{\partial \sigma_L}$.

$$\frac{\partial y_N}{\partial \sigma_L} = \frac{\partial}{\partial \sigma_L}(1+e^{-\phi_{Nm}})^{-1} \tag{B.40}$$

where

$$\phi_{Nm} = \sum_{m=1}^{M} w_{mN}y_m + \sigma_N \tag{B.41}$$

therefore

$$\frac{\partial y_N}{\partial \sigma_L} = \frac{\partial}{\partial \sigma_L}(1+e^{-\phi_{mN}})^{-1} \tag{B.42}$$

which expands to

$$\frac{\partial y_N}{\partial \sigma_L} = -(1+e^{-\phi_{mN}})^{-2}\frac{\partial}{\partial \sigma_L}(e^{-\phi_{mN}}) \tag{B.43}$$

which further expands to

$$\frac{\partial y_N}{\partial \sigma_L} = (1+e^{-\phi_{mN}})^{-2}(e^{-\phi_{mN}})\frac{\partial}{\partial \sigma_L}(\sum_{m=1}^{M} w_{mN}y_m + \sigma_N) \tag{B.44}$$

this simplifies to

$$\frac{\partial y_N}{\partial \sigma_L} = y_N(1-y_N)(\sum_{m=1}^{M} w_{mN}\frac{\partial y_m}{\partial \sigma_L}) \tag{B.45}$$

and thus, substituting equation B.39 for $\frac{\partial y_m}{\partial \sigma_L}$ provides

$$\frac{\partial y_N}{\partial \sigma_L} = y_N(1-y_N)\sum_{m=1}^{M} w_{mN}y_m(1-y_m)w_{Lm}y_L(1-y_L) \tag{B.46}$$

With these identities, the incremental update equations for networks implemented using the MSE, CE and CFM objective functions can be found.

B.3 Mean Square Error (MSE)

The incremental update rules for the network parameters can be found by minimizing the MSE function with respect to the network parameters for an instantaneous pattern. The MSE function is defined as

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)^2 \tag{B.47}$$

The update equations for the parameters will be

$$w_{MN}^{+} = w_{MN}^{-} - \eta\frac{\partial MSE}{\partial w_{MN}} \tag{B.48}$$

and

$$w_{LM}^{+} = w_{LM}^{-} - \eta \frac{\partial MSE}{\partial w_{LM}} \tag{B.49}$$

and

$$w_{KL}^{+} = w_{KL}^{-} - \eta \frac{\partial MSE}{\partial w_{KL}} \tag{B.50}$$

and

$$\sigma_{N}^{+} = \sigma_{N}^{-} - \eta \frac{\partial MSE}{\partial \sigma_{N}} \tag{B.51}$$

and

$$\sigma_{M}^{+} = \sigma_{M}^{-} - \eta \frac{\partial MSE}{\partial \sigma_{M}} \tag{B.52}$$

and

$$\sigma_{L}^{+} = \sigma_{L}^{-} - \eta \frac{\partial MSE}{\partial \sigma_{L}} \tag{B.53}$$

Here η is constant which determines how much each parameter is updated for a given iteration. Taking the derivative with respect to the output layer weights

$$\frac{\partial MSE}{\partial w_{MN}} = \frac{\partial}{\partial w_{MN}} [\frac{1}{N} \sum_{n=1}^{N} (y_n - d_n)]^2 \tag{B.54}$$

This simplifies to

$$\frac{\partial MSE}{\partial w_{MN}} = \frac{2}{N} (y_N - d_N) \frac{\partial y_N}{\partial w_{MN}} \tag{B.55}$$

Substituting B.5 for $\frac{\partial y_N}{\partial w_{MN}}$ provides

$$\frac{\partial MSE}{\partial w_{MN}} = \frac{2}{N} (y_N - d_N) y_N (1 - y_N) y_M \tag{B.56}$$

Now let's find $\frac{\partial MSE}{\partial w_{LM}}$.

$$\frac{\partial MSE}{\partial w_{LM}} = \frac{\partial}{\partial w_{LM}}[\frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)]^2 \tag{B.57}$$

Simplifying

$$\frac{\partial MSE}{\partial w_{LM}} = \frac{2}{N}\sum_{n=1}^{N}(y_n - d_n)\frac{\partial y_n}{\partial w_{LM}} \tag{B.58}$$

Substituting equation B.15 for $\frac{\partial y_n}{\partial w_{LM}}$ provides

$$\frac{\partial MSE}{\partial w_{LM}} = \frac{2}{N}\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)w_{Mn}y_M(1 - y_M)y_L \tag{B.59}$$

Finally, let's find $\frac{\partial MSE}{\partial w_{KL}}$.

$$\frac{\partial MSE}{\partial w_{KL}} = \frac{\partial}{\partial w_{KL}}[\frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)]^2 \tag{B.60}$$

This simplifies to

$$\frac{\partial MSE}{\partial w_{KL}} = \frac{2}{N}\sum_{n=1}^{N}(y_n - d_n)\frac{\partial y_n}{\partial w_{KL}} \tag{B.61}$$

Substituting B.23 for $\frac{\partial y_n}{\partial w_{KL}}$ provides

$$\frac{\partial MSE}{\partial w_{KL}} = \frac{2}{N}\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)[\sum_{m=1}^{M} w_{mn}(y_m(1 - y_m)w_{Lm}y_L(1 - y_L)y_K] \tag{B.62}$$

Now lets find the $\frac{\partial MSE}{\partial \sigma_N}$.

$$\frac{\partial MSE}{\partial \sigma_N} = \frac{\partial}{\partial \sigma_N}[\frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)]^2 \tag{B.63}$$

This can be written as

$$\frac{\partial MSE}{\partial \sigma_N} = \frac{2}{N}(y_N - d_N)\frac{\partial y_N}{\partial \sigma_N} \tag{B.64}$$

Substituting B.28 for $\frac{\partial y_N}{\partial \sigma_N}$ provides

$$\frac{\partial MSE}{\partial \sigma_N} = \frac{2}{N}(y_N - d_N)y_N(1 - y_N) \tag{B.65}$$

Now lets find the $\frac{\partial MSE}{\partial \sigma_M}$.

$$\frac{\partial MSE}{\partial \sigma_M} = \frac{\partial}{\partial \sigma_M}[\frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)]^2 \tag{B.66}$$

This can be written as

$$\frac{\partial MSE}{\partial \sigma_M} = \frac{2}{N}\sum_{n=1}^{N}(y_N - d_N)\frac{\partial y_N}{\partial \sigma_M} \tag{B.67}$$

Substituting B.38 for $\frac{\partial y_N}{\partial \sigma_M}$ provides

$$\frac{\partial MSE}{\partial \sigma_M} = \frac{2}{N}\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)w_{Mn}y_M(1 - y_M) \tag{B.68}$$

Now lets find the $\frac{\partial MSE}{\partial \sigma_L}$.

$$\frac{\partial MSE}{\partial \sigma_L} = \frac{\partial}{\partial \sigma_L}[\frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)]^2 \tag{B.69}$$

This can be written as

$$\frac{\partial MSE}{\partial \sigma_L} = \frac{2}{N}\sum_{n=1}^{N}(y_n - d_n)\frac{\partial y_n}{\partial \sigma_L} \tag{B.70}$$

Substituting for B.46 for $\frac{\partial y_n}{\partial \sigma_L}$ provides

$$\frac{\partial MSE}{\partial \sigma_L} = \frac{2}{N}\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)[\sum_{m=1}^{M} w_{mn}y_m(1 - y_m)w_{Lm}y_L(1 - y_L)] \tag{B.71}$$

Therefore, substituting the appropriate derivatives into the update equations for each parameter provides the learning rules for a network implemented using the MSE objective function. By

defining $\eta = 2C/N$, where C is a constant, the update equations for the parameters will be as follows:

$$w_{MN}^{+} = w_{MN}^{-} - \eta(y_N - d_N)y_N(1 - y_N)y_M \tag{B.72}$$

and

$$w_{LM}^{+} = w_{LM}^{-} - \eta\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)w_{Mn}y_M(1 - y_M)y_L \tag{B.73}$$

and

$$w_{KL}^{+} = w_{KL}^{-} - \eta\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)[\sum_{m=1}^{M} w_{mn}y_m(1 - y_m)w_{Lm}y_L(1 - y_L)y_K] \tag{B.74}$$

and

$$\sigma_N^{+} = \sigma_N^{-} - \eta(y_N - d_N)y_N(1 - y_N) \tag{B.75}$$

and

$$\sigma_M^{+} = \sigma_M^{-} - \eta\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)w_{Mn}y_M(1 - y_M) \tag{B.76}$$

and

$$\sigma_L^{+} = \sigma_L^{-} - \eta\sum_{n=1}^{N}(y_n - d_n)y_n(1 - y_n)[\sum_{m=1}^{M} w_{mn}y_m(1 - y_m)w_{Lm}y_L(1 - y_L)] \tag{B.77}$$

B.4 Cross Entropy (CE)

The incremental update rules for the network parameters can be found by minimizing the CE function with respect to the network parameters. The CE function is defined as

$$CE = -\frac{1}{N}\sum_{n=1}^{N}[d_n log(y_n) + (1 - d_n)log(1 - y_n)] \tag{B.78}$$

Letting η be a constant which controls the learning rate, the update equations for the parameters will be as follows:

$$w_{MN}^{+} = w_{MN}^{-} - \eta \frac{\partial CE}{\partial w_{MN}} \tag{B.79}$$

and

$$w_{LM}^{+} = w_{LM}^{-} - \eta \frac{\partial CE}{\partial w_{LM}} \tag{B.80}$$

and

$$w_{KL}^{+} = w_{KL}^{-} - \eta \frac{\partial CE}{\partial w_{KL}} \tag{B.81}$$

and

$$\sigma_{N}^{+} = \sigma_{N}^{-} - \eta \frac{\partial CE}{\partial \sigma_{N}} \tag{B.82}$$

and

$$\sigma_{M}^{+} = \sigma_{M}^{-} - \eta \frac{\partial CE}{\partial \sigma_{M}} \tag{B.83}$$

and

$$\sigma_{L}^{+} = \sigma_{L}^{-} - \eta \frac{\partial CE}{\partial \sigma_{L}} \tag{B.84}$$

First let's find $\frac{\partial CE}{\partial w_{MN}}$.

$$\frac{\partial CE}{\partial w_{MN}} = -\frac{1}{N} \frac{\partial}{\partial w_{MN}} \sum_{n=1}^{N} [d_n \log(y_n) + (1 - d_n) \log(1 - y_n)] \tag{B.85}$$

The derivative of the *log* function is

$$D_x[\log_a(u)] = \frac{1}{u \ln(a)} D_x u \tag{B.86}$$

Thus

$$\frac{\partial CE}{\partial w_{MN}} = -\frac{1}{N}[d_N \frac{\partial}{\partial w_{MN}} \log(y_N) + (1 - d_N) \frac{\partial}{\partial w_{MN}} \log(1 - y_n)] \tag{B.87}$$

Taking the derivative provides

$$\frac{\partial CE}{\partial w_{MN}} = -\frac{1}{N}[\frac{d_N}{y_N \ln(10)} \frac{\partial y_N}{\partial w_{MN}} + \frac{(1 - d_N)}{(1 - y_N) \ln(10)} \frac{\partial (1 - y_N)}{\partial w_{MN}}] \tag{B.88}$$

This can be written as

$$\frac{\partial CE}{\partial w_{MN}} = -\frac{1}{N \ln(10)}[\frac{d_N}{y_N} - \frac{(1 - d_N)}{(1 - y_N)}] \frac{\partial y_N}{\partial w_{MN}} \tag{B.89}$$

This simplifies to

$$\frac{\partial CE}{\partial w_{MN}} = -\frac{1}{N \ln(10)}[\frac{(d_N - y_N d_N - y_N + y_N d_N)}{y_N (1 - y_N)}] \frac{\partial y_N}{\partial w_{MN}} \tag{B.90}$$

Which in turn simplifies to

$$\frac{\partial CE}{\partial w_{MN}} = -\frac{1}{N \ln(10)}[\frac{(d_N - y_N)}{y_N (1 - y_N)}] \frac{\partial y_N}{\partial w_{MN}} \tag{B.91}$$

Substituting equation B.5 for $\frac{\partial y_N}{\partial w_{MN}}$ gives

$$\frac{\partial CE}{\partial w_{MN}} = -\frac{1}{N \ln(10)}[\frac{(d_N - y_N)}{y_N (1 - y_N)}] y_N (1 - y_N) y_M \tag{B.92}$$

This in turn simplifies to

$$\frac{\partial CE}{\partial w_{MN}} = -\frac{(d_N - y_N) y_M}{N \ln(10)} \tag{B.93}$$

Similarly

$$\frac{\partial CE}{\partial \sigma_N} = -\frac{1}{N \ln(10)}[\frac{(d_N - y_N)}{y_N (1 - y_N)}] \frac{\partial y_N}{\partial \sigma_N} \tag{B.94}$$

Substituting equation B.28 for $\frac{\partial y_N}{\partial \sigma_N}$ provides

$$\frac{\partial CE}{\partial \sigma_N} = -\frac{1}{N \ln(10)}[\frac{(d_N - y_N)}{y_N (1 - y_N)}][y_N (1 - y_N)] \tag{B.95}$$

which simplifies to

$$\frac{\partial CE}{\partial \sigma_N} = -\frac{(d_N - y_N)}{N \ln(10)} \tag{B.96}$$

Now let's find $\frac{\partial CE}{\partial w_{LM}}$.

$$\frac{\partial CE}{\partial w_{LM}} = -\frac{1}{N}\frac{\partial}{\partial w_{LM}}\sum_{n=1}^{N}[d_n \log(y_n) + (1 - d_n)\log(1 - y_n)] \tag{B.97}$$

Thus

$$\frac{\partial CE}{\partial w_{LM}} = -\frac{1}{N}\sum_{n=1}^{N}[d_n \frac{\partial}{\partial w_{LM}}\log(y_n) + (1 - d_n)\frac{\partial}{\partial w_{LM}}\log(1 - y_n)] \tag{B.98}$$

Taking the derivative provides

$$\frac{\partial CE}{\partial w_{LM}} = -\frac{1}{N}\sum_{n=1}^{N}[\frac{d_n}{y_n \ln(10)}\frac{\partial y_N}{\partial w_{LM}} + \frac{(1 - d_n)}{(1 - y_n)\ln(10)}\frac{\partial(1 - y_N)}{\partial w_{LM}}] \tag{B.99}$$

This can be written as

$$\frac{\partial CE}{\partial w_{LM}} = -\frac{1}{N \ln(10)}\sum_{n=1}^{N}[\frac{d_n}{y_n} - \frac{(1 - d_n)}{(1 - y_n)}]\frac{\partial y_n}{\partial w_{LM}} \tag{B.100}$$

This simplifies to

$$\frac{\partial CE}{\partial w_{LM}} = -\frac{1}{N \ln(10)}\sum_{n=1}^{N}[\frac{(d_n - y_n d_n - y_n + y_n d_n)}{y_n(1 - y_n)}]\frac{\partial y_n}{\partial w_{LM}} \tag{B.101}$$

Which in turn simplifies to

$$\frac{\partial CE}{\partial w_{LM}} = -\frac{1}{N \ln(10)}\sum_{n=1}^{N}[\frac{(d_n - y_n)}{y_n(1 - y_n)}]\frac{\partial y_n}{\partial w_{LM}} \tag{B.102}$$

Substituting equation B.15 for $\frac{\partial y_n}{\partial w_{LM}}$ provides

$$\frac{\partial CE}{\partial w_{LM}} = -\frac{1}{N \ln(10)}\sum_{n=1}^{N}[\frac{(d_n - y_n)}{y_n(1 - y_n)}][y_n(1 - y_n)w_{Mn}y_M(1 - y_M)y_L] \tag{B.103}$$

This can be written as

$$\frac{\partial CE}{\partial w_{LM}} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}(d_n - y_n)w_{Mn}y_M(1-y_M)y_L \tag{B.104}$$

Similarly

$$\frac{\partial CE}{\partial \sigma_M} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}[\frac{(d_n - y_n)}{y_n(1-y_n)}]\frac{\partial y_n}{\partial \sigma_M} \tag{B.105}$$

Substituting equation B.38 for $\frac{\partial y_n}{\partial \sigma_M}$ provides

$$\frac{\partial CE}{\partial \sigma_M} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}[\frac{(d_n - y_n)}{y_n(1-y_n)}][y_n(1-y_n)w_{Mn}y_M(1-y_M)] \tag{B.106}$$

This simplifies to

$$\frac{\partial CE}{\partial \sigma_M} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}(d_n - y_n)w_{Mn}y_M(1-y_M) \tag{B.107}$$

Now let's find $\frac{\partial CE}{\partial w_{KL}}$.

$$\frac{\partial CE}{\partial w_{KL}} = -\frac{1}{N}\frac{\partial}{\partial w_{KL}}\sum_{n=1}^{N}[d_n\log(y_n) + (1-d_n)\log(1-y_n)] \tag{B.108}$$

Thus

$$\frac{\partial CE}{\partial w_{KL}} = -\frac{1}{N}\sum_{n=1}^{N}[d_n\frac{\partial}{\partial w_{KL}}\log(y_n) + (1-d_n)\frac{\partial}{\partial w_{KL}}\log(1-y_n)] \tag{B.109}$$

Taking the derivative provides

$$\frac{\partial CE}{\partial w_{KL}} = -\frac{1}{N}\sum_{n=1}^{N}[\frac{d_n}{y_n\ln(10)}\frac{\partial y_n}{\partial w_{KL}} + \frac{(1-d_n)}{(1-y_n)\ln(10)}\frac{\partial(1-y_n)}{\partial w_{KL}}] \tag{B.110}$$

This can be written as

$$\frac{\partial CE}{\partial w_{KL}} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}[\frac{d_n}{y_n} - \frac{(1-d_n)}{(1-y_n)}][\frac{\partial y_n}{\partial w_{KL}}] \tag{B.111}$$

This simplifies to

$$\frac{\partial CE}{\partial w_{KL}} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}[\frac{(d_n - y_n d_n - y_n + y_n d_n)}{y_n(1-y_n)}]\frac{\partial y_n}{\partial w_{KL}} \tag{B.112}$$

Which in turn simplifies to

$$\frac{\partial CE}{\partial w_{KL}} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}[\frac{(d_n - y_n)}{y_n(1-y_n)}]\frac{\partial y_n}{\partial w_{KL}} \tag{B.113}$$

Substituting B.23 for $\frac{\partial y_n}{\partial w_{KL}}$ gives

$$\frac{\partial CE}{\partial w_{KL}} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}\frac{(d_n - y_n)}{y_n(1-y_n)}y_n(1-y_n)[\sum_{m=1}^{M} w_{mn}y_m(1-y_m)w_{Lm}y_L(1-y_L)y_K] \tag{B.114}$$

This in turn simplifies to

$$\frac{\partial CE}{\partial w_{KL}} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}(d_n - y_n)[\sum_{m=1}^{M} w_{mn}y_m(1-y_m)w_{Lm}y_L(1-y_L)y_K] \tag{B.115}$$

Similarly

$$\frac{\partial CE}{\partial \sigma_L} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}\frac{(d_n - y_n)}{y_n(1-y_n)}\frac{\partial y_n}{\partial \sigma_L} \tag{B.116}$$

Substituting equation B.46 for $\frac{\partial y_n}{\partial \sigma_L}$ provides

$$\frac{\partial CE}{\partial \sigma_L} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}\frac{(d_n - y_n)}{y_n(1-y_n)}y_n(1-y_n)[\sum_{m=1}^{M} w_{mn}y_m(1-y_m)w_{Lm}y_L(1-y_L)] \tag{B.117}$$

which simplifies to

$$\frac{\partial CE}{\partial \sigma_L} = -\frac{1}{N\ln(10)}\sum_{n=1}^{N}(d_n - y_n)[\sum_{m=1}^{M} w_{mn}y_m(1-y_m)w_{Lm}y_L(1-y_L)] \tag{B.118}$$

Letting $\eta = C/[N ln(10)]$ where C is a constant, the ι late equations for the network parameters will be

$$w_{MN}^{+} = w_{MN}^{-} + \eta[(d_N - y_N)y_M] \tag{B.119}$$

and

$$w_{LM}^{+} = w_{LM}^{-} + \eta \sum_{n=1}^{N}[(d_n - y_n)w_{Mn}y_M(1 - y_M)y_L] \tag{B.120}$$

and

$$w_{KL}^{+} = w_{KL}^{-} + \eta \sum_{n=1}^{N}(d_n - y_n)[\sum_{m=1}^{M} w_{mn}y_m(1 - y_m)w_{Lm}y_L(1 - y_L)y_K] \tag{B.121}$$

and

$$\sigma_N^{+} = \sigma_N^{-} + \eta(d_N - y_N) \tag{B.122}$$

and

$$\sigma_M^{+} = \sigma_M^{-} + \eta \sum_{n=1}^{N}(d_n - y_n)w_{Mn}y_M(1 - y_M) \tag{B.123}$$

and

$$\sigma_L^{+} = \sigma_L^{-} + \eta \sum_{n=1}^{N}(d_n - y_n)[\sum_{m=1}^{M} w_{mn}(y_m(1 - y_m)w_{Lm}y_L(1 - y_L)] \tag{B.124}$$

B.5 Classification Figure of Merit (CFM)

The update rules for the network parameters can be found by maximizing the CFM function with respect to the network parameters. The classification CFM objective function is defined as

$$CFM = \frac{1}{N-1} \sum_{n=1_{n \neq c}}^{N} \alpha[(1 + e^{-\beta y_c + \beta y_n + \zeta})]^{-1} \tag{B.125}$$

where

y_n = output of an incorrect classification node

y_c = output of the correct classification node

α = a sigmoid scaling parameter

β = a sigmoid discontinuity parameter

ζ = a sigmoid lateral shift parameter

Furthermore, let

$$z_n = \frac{1}{1 + e^{(-\beta y_c + \beta y_n + \zeta)}} \tag{B.126}$$

Here z_n is a function of both the correct and incorrect node outputs. Therefore, the CFM objective function can be written as

$$CFM = \frac{1}{N-1} \sum_{n=1_{n \neq c}}^{N} \alpha z_n(y_n, y_c) \tag{B.127}$$

Letting η be a constant which controls the learning rate, the incremental update equations for the parameters can be found from the following equations:

$$w_{MN}^{+} = w_{MN}^{-} + \eta \frac{\partial CFM}{\partial w_{MN}} \tag{B.128}$$

and

$$w_{LM}^{+} = w_{LM}^{-} + \eta \frac{\partial CFM}{\partial w_{LM}} \tag{B.129}$$

and

$$w_{KL}^{+} = w_{KL}^{-} + \eta \frac{\partial CFM}{\partial w_{KL}} \tag{B.130}$$

and

$$\sigma_{N}^{+} = \sigma_{N}^{-} + \eta \frac{\partial CFM}{\partial \sigma_{N}} \tag{B.131}$$

and

$$\sigma_M^+ = \sigma_M^- + \eta \frac{\partial CFM}{\partial \sigma_M} \tag{B.132}$$

and

$$\sigma_L^+ = \sigma_L^- + \eta \frac{\partial CFM}{\partial \sigma_L} \tag{B.133}$$

Now, taking the following derivatives

$$\frac{\partial z_n}{\partial y_n} = \frac{\partial}{\partial y_n}[1 + e^{(-\beta y_c + \beta y_n + \zeta)}]^{-1} \tag{B.134}$$

Simplifying

$$\frac{\partial z_n}{\partial y_n} = -[1 + e^{(-\beta y_c + \beta y_n + \zeta)}]^{-2} \frac{\partial}{\partial y_n}[e^{(-\beta y_c + \beta y_n + \zeta)}] \tag{B.135}$$

This can be written as

$$\frac{\partial z_n}{\partial y_n} = -\beta e^{(-\beta y_c + \beta y_n + \zeta)}[1 + e^{-(\beta y_c + \beta y_n + \zeta)}]^{-2} \tag{B.136}$$

which simplifies to

$$\frac{\partial z_n}{\partial y_n} = -\beta z_n (1 - z_n) \tag{B.137}$$

Similarly

$$\frac{\partial z_n}{\partial y_c} = \frac{\partial}{\partial y_c}[1 + e^{(-\beta y_c + \beta y_n + \zeta)}]^{-1} \tag{B.138}$$

Simplifying

$$\frac{\partial z_n}{\partial y_c} = -[1 + e^{(-\beta y_c + \beta y_n + \zeta)}]^{-2} \frac{\partial}{\partial y_c}[e^{(-\beta y_c + \beta y_n + \zeta)}] \tag{B.139}$$

This can be written as

$$\frac{\partial z_n}{\partial y_c} = \beta e^{(-\beta y_c + \beta y_n + \zeta)}[1 + e^{(-\beta y_c + \beta y_n + \zeta)}]^{-2} \tag{B.140}$$

Simplifying again

$$\frac{\partial z_n}{\partial y_z} = \beta z_n(1 - z_n) \tag{B.141}$$

First, let's find the update rules for a weight linking node M in layer 2 to an incorrect classification node N in layer 3, w_{MN}.

$$\frac{\partial CFM}{\partial w_{MN}} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} \frac{\partial z_n}{\partial w_{MN}} \tag{B.142}$$

But $z_N = f(y_c, y_n)$ therefore,

$$\frac{\partial CFM}{\partial w_{MN}} = \frac{\alpha}{N-1} \frac{\partial z_N}{\partial y_N} \frac{\partial y_N}{\partial w_{MN}} \tag{B.143}$$

Substituting equation B.5 for $\frac{\partial y_N}{\partial w_{MN}}$ and equation B.137 for $\frac{\partial z_N}{\partial y_N}$ provides

$$\frac{\partial CFM}{\partial w_{MN}} = -(\frac{\alpha\beta}{N-1}) z_N(1 - z_N) y_N(1 - y_N) y_M \tag{B.144}$$

Similarly

$$\frac{\partial CFM}{\partial \sigma_N} = \frac{\alpha}{N-1} \frac{\partial z_N}{\partial y_N} \frac{\partial y_N}{\partial \sigma_N} \tag{B.145}$$

Substituting equations B.28 for $\frac{\partial y_N}{\partial \sigma_N}$ and equation B.137 for $\frac{\partial z_N}{\partial y_N}$ provides, for an incorrect classification node,

$$\frac{\partial CFM}{\partial \sigma_N} = -(\frac{\alpha\beta}{N-1}) z_N(1 - z_N) y_N(1 - y_N) \tag{B.146}$$

For the correct classification node, the $\frac{\partial CFM}{\partial w_{MC}}$ needs to be found to maximize the CFM objective function, where

$$\frac{\partial CFM}{\partial w_{MC}} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} \frac{\partial z_n}{\partial w_{MC}} \tag{B.147}$$

This can be simplified to

$$\frac{\partial CFM}{\partial w_{MC}} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} \frac{\partial z_n}{\partial y_C} \frac{\partial y_C}{\partial w_{MC}} \tag{B.148}$$

Substituting equation B.5 for $\frac{\partial y_C}{\partial w_{MC}}$ and equation B.141 for $\frac{\partial z_n}{\partial y_C}$ provides

$$\frac{\partial CFM}{\partial w_{MC}} = \frac{\alpha\beta}{N-1} \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)y_C(1-y_C)y_M \tag{B.149}$$

Similarly

$$\frac{\partial CFM}{\partial \sigma_C} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} \frac{\partial z_n}{\partial y_C}\frac{\partial y_C}{\partial \sigma_C} \tag{B.150}$$

Substituting equation B.28 for $\frac{\partial y_C}{\partial \sigma_C}$ and equation B.141 for $\frac{\partial z_n}{\partial y_C}$ provides

$$\frac{\partial CFM}{\partial \sigma_C} = \frac{\alpha\beta}{N-1} \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)y_C(1-y_C) \tag{B.151}$$

Now let's find the $\frac{\partial CFM}{\partial w_{LM}}$.

$$\frac{\partial CFM}{\partial w_{LM}} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} \frac{\partial z_n}{\partial w_{LM}} \tag{B.152}$$

This can be written as

$$\frac{\partial CFM}{\partial w_{LM}} = (\frac{\alpha}{N-1}) \sum_{n=1_{n\neq c}}^{N} (\frac{\partial z_n}{\partial y_n}\frac{\partial y_n}{\partial w_{LM}} + \frac{\partial z_n}{\partial y_c}\frac{\partial y_c}{\partial w_{LM}}) \tag{B.153}$$

Substituting equation B.15 for $\frac{\partial y_n}{\partial w_{LM}}$ and $\frac{\partial y_c}{\partial w_{LM}}$ and equations B.137 and B.141 for $\frac{\partial z_n}{\partial y_n}$ and $\frac{\partial z_n}{\partial y_c}$ gives

$$\frac{\partial CFM}{\partial w_{LM}} = \frac{\alpha\beta}{N-1} \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)[y_c(1-y_c)w_{Mc} - y_n(1-y_n)w_{Mn}]y_M(1-y_M)y_L \tag{B.154}$$

Similarly

$$\frac{\partial CFM}{\partial \sigma_M} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} (\frac{\partial z_n}{\partial y_n}\frac{\partial y_n}{\partial \sigma_M} + \frac{\partial z_n}{\partial y_c}\frac{\partial y_c}{\partial \sigma_M}) \tag{B.155}$$

Substituting B.38 for $\frac{\partial y_c}{\partial \sigma_M}$ and $\frac{\partial y_n}{\partial \sigma_M}$ and equations B.137 and B.141 for $\frac{\partial z_n}{\partial y_n}$ and $\frac{\partial z_n}{\partial y_c}$ gives

$$\frac{\partial CFM}{\partial \sigma_M} = \frac{\alpha\beta}{N-1} \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)[y_c(1-y_c)w_{Mc} - y_n(1-y_n)w_{Mn}]y_M(1-y_M) \qquad \text{(B.156)}$$

Now let's find the $\frac{\partial CFM}{\partial w_{KL}}$.

$$\frac{\partial CFM}{\partial w_{KL}} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} \frac{\partial z_n}{\partial w_{KL}} \qquad \text{(B.157)}$$

This can be written as

$$\frac{\partial CFM}{\partial w_{KL}} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} \left(\frac{\partial z_n}{\partial y_n}\frac{\partial y_n}{\partial w_{KL}} + \frac{\partial z_n}{\partial y_c}\frac{\partial y_c}{\partial w_{KL}}\right) \qquad \text{(B.158)}$$

Substituting equation B.23 for $\frac{\partial y_n}{\partial w_{KL}}$ and $\frac{\partial y_c}{\partial w_{KL}}$ and equations B.137 and B.141 for $\frac{\partial z_n}{\partial y_n}$ and $\frac{\partial z_n}{\partial y_c}$ gives

$$\begin{aligned}\frac{\partial CFM}{\partial w_{KL}} = \frac{\alpha\beta}{N-1} \sum_{n=1_{n\neq c}}^{N} & z_n(1-z_n)[y_c(1-y_c)\sum_{m=1}^{M} w_{mc} \\ & - y_n(1-y_n)\sum_{m=1}^{M} w_{mn}]y_m(1-y_m)w_{Lm}y_L(1-y_L)y_K \end{aligned} \qquad \text{(B.159)}$$

Similarly

$$\frac{\partial CFM}{\partial \sigma_L} = \frac{\alpha}{N-1} \sum_{n=1_{n\neq c}}^{N} \frac{\partial z_n}{\partial y_n}\frac{\partial y_n}{\partial \sigma_L} + \frac{\partial z_n}{\partial y_c}\frac{\partial y_c}{\partial \sigma_L} \qquad \text{(B.160)}$$

Substituting B.46 for $\frac{\partial y_n}{\partial \sigma_L}$ and $\frac{\partial y_c}{\partial \sigma_L}$ and equations B.137 and B.141 for $\frac{\partial z_n}{\partial y_n}$ and $\frac{\partial z_n}{\partial y_c}$ gives

$$\begin{aligned}\frac{\partial CFM}{\partial \sigma_L} = \frac{\alpha\beta}{N-1} \sum_{n=1_{n\neq c}}^{N} & z_n(1-z_n)[y_c(1-y_c)\sum_{m=1}^{M} w_{mc} \\ & - y_n(1-y_n)\sum_{m=1}^{M} w_{mn}]y_m(1-y_m)w_{Lm}y_L(1-y_L) \end{aligned} \qquad \text{(B.161)}$$

Thus, if $\eta = C\alpha\beta/(N-1)$ where C is a constant, the update equation for a weight linking node M to an incorrect classification node N, w_{MN} is

$$w_{MN}^{+} = w_{MN}^{-} - \eta z_N(1 - z_N)y_N(1 - y_N)y_M \tag{B.162}$$

and the update equation for the offset of an incorrect classification node N, σ_N, is

$$\sigma_N^{+} = \sigma_N^{-} - \eta z_N(1 - z_N)y_N(1 - y_N) \tag{B.163}$$

The update rule for the weight linking node M in layer 2 to the correct node C in layer 3 is

$$w_{MC}^{+} = w_{MC}^{-} + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1 - z_n)y_C(1 - y_C)y_M \tag{B.164}$$

The update rule for the offset of the correct node C in layer 3 is

$$\sigma_C^{+} = \sigma_C^{-} + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1 - z_n)y_C(1 - y_C) \tag{B.165}$$

The update equation for a weight linking node L in layer 1 to node M in layer 2, w_{LM} as

$$w_{LM}^{+} = w_{LM}^{-} + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1 - z_n)[y_c(1 - y_c)w_{Mc} - y_n(1 - y_n)w_{Mn}]y_M(1 - y_M)y_L \tag{B.166}$$

while the update equation for the offset of node M in layer 2, σ_M is

$$\sigma_M^{+} = \sigma_M^{-} + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1 - z_n)[y_c(1 - y_c)w_{Mc} - y_n(1 - y_n)w_{Mn}]y_M(1 - y_M) \tag{B.167}$$

The update equation for a weight linking node K in layer 0 to node L in layer 1, w_{KL} is

$$\begin{aligned} w_{KL}^{+} \;=\; & w_{KL}^{-} + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1 - z_n)[y_c(1 - y_c) \sum_{m=1}^{M} w_{mc} \\ & - y_n(1 - y_n) \sum_{m=1}^{M} w_{mn}]y_m(1 - y_m)w_{Lm}y_L(1 - y_L)y_K \end{aligned} \tag{B.168}$$

while the update equation for the offset of node L in layer 1, σ_L as

$$\sigma_L^+ = \sigma_L^- + \eta \sum_{n=1_{n\neq c}}^{N} z_n(1-z_n)[y_c(1-y_c)\sum_{m=1}^{M} w_{mc} - y_n(1-y_n)\sum_{m=1}^{M} w_{mn}]y_m(1-y_m)w_{Lm}y_L(1-y_L) \quad \text{(B.169)}$$

Appendix C. *Parzen Window/Radial Basis Function Relationship*

C.1 Introduction

This appendix will discuss the relationship between the Radial Basis Function and Parzen Window approach to the estimation of a probability density function from analysis of a set of data points.

C.1.1 Density Estimation The Parzen Window estimate of a conditional probability density function, $P(\bar{x}/G_j)$, provides a smooth estimate of the density function from a set of sample data points by assuming that each value of the data occurring in the sample set also raises the probability of any value occurring close to that value of the data. By centering a kernel function at each of the data points, the final value of the estimate can be obtained by summing together all the contributions from each value of the sample data.(8:162) That is the Parzen Window estimate has the form

$$P(\bar{x}/G_J) = \frac{1}{N_J} \sum_{j=1}^{N_J} \frac{1}{h^K(N_J)} \phi[\frac{\bar{x} - \bar{x}_j}{h(N_J)}] \tag{C.1}$$

where

K = the number of dimensions

N_J = the number of data points in class J

$h(N_J)$ = a function of N_J commonly referred to as the window width.

$\phi(\bar{x})$ = the kernel estimate function.

Thus, to form a Parzen Window estimate of the density function from a set of known points, a kernel function and a window width have to be chosen. While the window width can be somewhat arbitrary, the kernel function must fulfill several conditions.

C.1.2 Conditions For the Parzen window estimate to work, the following conditions must hold true (7:174):

$$\int_{-\infty}^{\infty} \phi(u)du = 1 \tag{C.2}$$

$$\int_{-\infty}^{\infty} |\phi(u)|du < \infty \tag{C.3}$$

$$Sup|\phi(u)| < \infty \tag{C.4}$$

$$\lim_{u\to\infty} |u\phi(u)| = 0 \tag{C.5}$$

$$\lim_{N\to\infty} h^K(N) = 0 \tag{C.6}$$

$$\lim_{N\to\infty} Nh^K(N) = \infty \tag{C.7}$$

$$\lim_{N\to\infty} Nh^K(N)^2 = \infty \tag{C.8}$$

If all these conditions are met, the Parzen estimate is asymptotically unbiased and consistent at all the continuous points of $P(\bar{x}/G_I)$ (7:173-175).

C.2 Kernel Selection

One type of kernel function which satisfies these conditions is the gaussian radial basis function

$$\phi(\|\bar{x} - \bar{x}_n\|) = [2\pi\sigma^2 h(N)]^{-\frac{K}{2}} e^{-[\sum_{k=1}^{K} \frac{(x_k - x_{kn})^2}{2\sigma^2 h(N)}]} \tag{C.9}$$

where

σ = Constant defining the window width

$h(N) = N^{\frac{-.01}{k}}$ = a function of N

N = Number of data points

K = Number of dimensions.

C.3 Proofs

The proofs that the gaussian radial basis function meets the requirements for the Parzen Window are shown below. In these proofs, the following substitutions have been made to simplify the algebra. Let

$$a = \frac{1}{2\sigma^2 h(N)} \tag{C.10}$$

and

$$u_k^2 = (\bar{x}_k - \bar{x}_{kn})^2 = (\bar{x}_{kn} - \bar{x}_k)^2 \tag{C.11}$$

and thus

$$\phi(\|\bar{x} - \bar{x}_n\|) = (\frac{\pi}{a})^{-\frac{K}{2}} e^{-[\sum_{k=1}^{K} a(u_k)^2]} \tag{C.12}$$

The first condition the kernel function must meet is that

$$\int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} \phi(u) du_1 \dots du_K = 1 \tag{C.13}$$

substituting the equation for the kernel provides

$$\int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} \phi(u) du_1 \dots du_K = (\frac{\pi}{a})^{-\frac{K}{2}} \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} e^{-[\sum_{k=1}^{K} a(u_k)^2]} du_1 \dots du_K \tag{C.14}$$

which can be written as

$$\int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} \phi(u) du_1 \dots du_K = (\frac{\pi}{a})^{-\frac{K}{2}} [2 \int_0^{\infty} e^{-a(u_1)^2} du_1] \dots [2 \int_0^{\infty} e^{-a(u_K)^2} du_K] \tag{C.15}$$

now from (25:640)

$$\int_0^{\infty} e^{-a(u)^2} du = \frac{1}{2}\sqrt{\frac{\pi}{a}} \tag{C.16}$$

substituting into equation C.15 provides

$$2^K (\frac{\pi}{a})^{-\frac{K}{2}} (\frac{1}{2}\sqrt{\frac{\pi}{a}})^K = 1 \tag{C.17}$$

Therefore the equation C.2 is satisfied. The second equation that must be met is that

$$\int_{-\infty}^{\infty} |\phi(u)| du < \infty \tag{C.18}$$

substituting the equation for the kernel function provides

$$(\frac{\pi}{a})^{-\frac{K}{2}} \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} |e^{-[\sum_{k=1}^{K} a(u_k)^2]}| du_1 \ldots du_K < \infty \tag{C.19}$$

Since equation C.2 is satisfied, then so is equation C.3. Equation C.4 describes the third condition that must be met. For this condition, the supremum, or least upper bound must exist. That is

$$Sup|\phi(u)| < \infty \tag{C.20}$$

Substituting for $|\phi(u)|$ reveals

$$Sup|\phi(u)| = Sup|(\frac{\pi}{a})^{-\frac{K}{2}} e^{-[\sum_{k}^{K} a(u_k)^2]}| \tag{C.21}$$

The maximum value for $|\phi(u)|$ is unity. Therefore

$$Sup|\phi(u)| = 1 < \infty \tag{C.22}$$

Thus, equation C.4 is satisfied. Equation C.5 describes the fourth condition that must be met. For this condition

$$\lim_{u \to \infty} |u\phi(u)| = 0 \tag{C.23}$$

Substituting the equation for the kernel, and suppressing the constant, $(\pi/a)^{-K/2}$, provides, for the K^{th} dimension

$$\lim_{u \to \infty} |u_k \phi(u_k)| = \lim_{u \to \infty} |u_k e^{-a(u_k)^2}| \tag{C.24}$$

now

$$\lim_{u_k \to \infty} |u_k e^{-a(u_k)^2}| = \lim_{u_k \to \infty} |\frac{u_k}{e^{a(u_k)^2}}| \tag{C.25}$$

using L'Hopital's rule

$$\lim_{u_k \to \infty} \frac{u_k}{e^{a(u_k)^2}} = \lim_{u_k \to \infty} \frac{1}{2au_k e^{a(u_k)^2}} = 0 \tag{C.26}$$

Thus, this condition is satisfied. Equation C.6 describes the fifth condition that must be met. Here,

$$\lim_{N \to \infty} h^K(N) = 0 \tag{C.27}$$

must be satisfied. Substituting the equation for $h(N)$ provides

$$\lim_{N \to \infty} h^K(N) = \lim_{N \to \infty} [N^{\frac{-.01}{K}}]^K \tag{C.28}$$

Simplifying the equation shows

$$\lim_{N \to \infty} h^K(N) = \lim_{N \to \infty} N^{-.01} = 0 \tag{C.29}$$

Thus, this condition is satisfied. Equation C.7 describes the sixth condition that must be met. For this condition,

$$\lim_{N \to \infty} Nh^K(N) = \infty \tag{C.30}$$

must be met. Substituting the equation for $h(N)$ provides

$$\lim_{N \to \infty} Nh^K(N) = \lim_{N \to \infty} N(N^{\frac{-.01}{K}})^K \tag{C.31}$$

This can be written as

$$\lim_{N \to \infty} Nh^K(N) = \lim_{N \to \infty} N^{.99} = \infty \tag{C.32}$$

Thus, this condition is met. Equation C.8 describes the seventh condition that must be met. For this condition,

$$\lim_{N \to \infty} Nh(N)^2 = \infty \tag{C.33}$$

Substituting the equation for $h(N)$ shows

$$\lim_{N\to\infty} Nh(N)^2 = \lim_{N\to\infty} N[(N^{\frac{-.01}{K}})^2]^K \tag{C.34}$$

This can be written as

$$\lim_{N\to\infty} Nh(N)^2 = \lim_{N\to\infty} N(N^{-.02}) \tag{C.35}$$

which can be simplified to

$$\lim_{N\to\infty} Nh(N)^2 = \lim_{N\to\infty} N^{.98} = \infty \tag{C.36}$$

Thus, this condition is fulfilled.

By meeting these conditions, the guassian radial basis function can be used to estimate a conditional probability distribution from a set of sampled points. In this estimate,

$$P(\bar{x}/G_J) = \frac{1}{N_J}\sum_{j=1}^{N_J}[\frac{1}{h^K(N)}][2\pi\sigma_j^2 h(N)]^{-\frac{K}{2}} e^{-[\sum_{k=1}^{K}\frac{(x_k - x_{kn})^2}{2\sigma_j^2 h(N)}]} \tag{C.37}$$

now, since $h(N) = N^{\frac{-.01}{k}} \approx 1$ even for large values of N, the equation becomes

$$P(\bar{x}/G_J) = \frac{1}{N_J}\sum_{j=1}^{N_J}(2\pi\sigma_j^2)^{-\frac{K}{2}} e^{-[\sum_{k=1}^{K}\frac{(x_k - x_{kn})^2}{2\sigma_j^2}]} \tag{C.38}$$

Appendix D. *Kernel Classifier Network Training Algorithms*

D.1 Introduction

In this section, the training algorithms for a radial basis function network will be derived.

Consider the feedforward artificial neural network as shown in figure 3.7 with the following parameters:.

Layer 0 - input layer with K possible nodes

Layer 1 - hidden layer with L possible nodes

Layer 2 - output layer with M possible nodes

Let the weights between layers be defined as follows:

w_{kl} -weight linking node k in layer 0 to node l in layer 1.

w_{lm} -weight linking node l in layer 1 to node m in layer 2.

Let the transfer function for the nodes in each layer be as follows:

Layer 0 - $y_k = x_k$ = identity function

Layer 1 - $y_l = e^{-[\sum_{k=1}^{K} \frac{(x_k - w_{kl})^2}{2\sigma^2}]}$ = gaussian radial basis function

Layer 2 - $y_m = \sum_{l=1}^{L} w_{lm} y_l$ = linear function

Thus the parameters of weights $\bar{w}_L$ and $\bar{w}_M$, and the spreads σ_L will characterize the network.

D.2 Incremental MSE Minimization

One method of determining these network parameters is to use the method of incremental backpropagation according to the MSE objective function defined here as

$$MSE = \frac{1}{M} \sum_{m=1}^{M} (y_m - d_m)^2 \tag{D.1}$$

A network developed using the MSE objective function seeks to have a minimum error over all patterns with respect to the network parameters. The incremental update equations for the network parameters are defined as

$$w_{LM}^{+} = w_{LM}^{-} - \eta \frac{\partial MSE}{\partial w_{LM}} \tag{D.2}$$

and

$$w_{KL}^{+} = w_{KL}^{-} - \eta \frac{\partial MSE}{\partial w_{KL}} \tag{D.3}$$

and

$$\sigma_{KL}^{+} = \sigma_{KL}^{-} - \eta \frac{\partial MSE}{\partial \sigma_{KL}} \tag{D.4}$$

Here η is a constant which controls the rate of update. Let's first minimize the error with respect to a specific weight, say w_{LM}, linking node L in the hidden layer to node M in the output layer.

$$\frac{\partial MSE}{\partial w_{LM}} = \frac{\partial}{\partial w_{LM}} [\frac{1}{M} \sum_{m=1}^{M} (y_m - d_m)^2] \tag{D.5}$$

or

$$\frac{\partial MSE}{\partial w_{LM}} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial}{\partial w_{LM}} (y_m - d_m)^2 \tag{D.6}$$

therefore

$$\frac{\partial MSE}{\partial w_{LM}} = \frac{2}{M} (y_M - d_M) \frac{\partial y_M}{\partial w_{LM}} \tag{D.7}$$

but

$$y_M = \sum_{l=1}^{L} w_{lM} y_l \tag{D.8}$$

thus

$$\frac{\partial y_M}{\partial w_{LM}} = y_L \tag{D.9}$$

therefore

$$\frac{\partial MSE}{\partial w_{LM}} = \frac{2}{M} (y_M - d_M) y_L \tag{D.10}$$

If $\eta = 2C/M$ where C is a constant, the training rule for this weight is

$$w_{LM}^{+} = w_{LM}^{-} - \eta(y_M - d_M)y_L \tag{D.11}$$

Now, let's minimize the error with respect to the weight, or center, of radial basis function L in layer 1 providing the link to node K in layer 0, w_{KL}.

$$\frac{\partial MSE}{\partial w_{KL}} = \frac{\partial}{\partial w_{KL}}[\frac{1}{M}\sum_{m=1}^{M}(y_m - d_m)^2] \tag{D.12}$$

or

$$\frac{\partial MSE}{\partial w_{KL}} = \frac{1}{M}\sum_{m=1}^{M}\frac{\partial}{\partial w_{KL}}(y_m - d_m)^2 \tag{D.13}$$

thus

$$\frac{\partial MSE}{\partial w_{KL}} = \frac{2}{M}\sum_{m=1}^{M}(y_m - d_m)\frac{\partial y_m}{\partial w_{KL}} \tag{D.14}$$

but

$$y_m = \sum_{l=1}^{L} w_{lm} y_l \tag{D.15}$$

therefore

$$\frac{\partial y_m}{\partial w_{KL}} = \frac{\partial}{\partial w_{KL}}\sum_{l=1}^{L} w_{lm} y_l \tag{D.16}$$

or

$$\frac{\partial y_m}{\partial w_{KL}} = w_{Lm}\frac{\partial y_L}{\partial w_{KL}} \tag{D.17}$$

but

$$y_L = e^{-[\sum_{k=1}^{K}\frac{(x_k - w_{kL})^2}{2\sigma_{kL}^2}]} \tag{D.18}$$

or

$$y_L = e^{-[\sum_{k=1}^{K-1} \frac{(x_k - w_{kL})^2}{2\sigma_{kL}^2}]} e^{-[\frac{(x_K - w_{KL})^2}{2\sigma_{KL}^2}]} \tag{D.19}$$

therefore

$$\frac{\partial y_L}{\partial w_{KL}} = e^{-[\sum_{k=1}^{K-1} \frac{(x_k - w_{kL})^2}{2\sigma_{kL}^2}]} \frac{\partial}{\partial w_{KL}} e^{-[\frac{(x_K - w_{KL})^2}{2\sigma_{KL}^2}]} \tag{D.20}$$

Simplifying

$$\frac{\partial y_L}{\partial w_{KL}} = e^{-[\sum_{k=1}^{K} \frac{(x_k - w_{kL})^2}{2\sigma_{kL}^2}]} \frac{\partial}{\partial w_{KL}} [-\frac{(x_K - w_{KL})^2}{2\sigma_{KL}^2}] \tag{D.21}$$

or

$$\frac{\partial y_L}{\partial w_{KL}} = y_L \frac{(x_K - w_{KL})}{\sigma_{KL}^2} \tag{D.22}$$

Combining equations D.14, D.17, and D.22 gives

$$\frac{\partial MSE}{\partial w_{KL}} = \frac{2}{M} \sum_{m=1}^{M} (y_m - d_m) w_{Lm} y_L \frac{(x_K - w_{KL})}{\sigma_{KL}^2} \tag{D.23}$$

If $\eta = 2C/M$ where C is a constant, the update equation for the centers of the radial basis functions, or weights linking the input layer, layer 0, nodes to the hidden layer, layer 1, nodes, can be written as

$$w_{KL}^{+} = w_{KL}^{-} - \eta \sum_{m=1}^{M} (y_m - d_m) w_{Lm} y_L \frac{(x_K - w_{KL})}{\sigma_{KL}^2} \tag{D.24}$$

Now let's minimize the error with respect to the spread of the L^{th} radial basis function in the direction of the K^{th} node in the input layer, layer 0, σ_{KL}.

$$\frac{\partial MSE}{\partial \sigma_{KL}} = \frac{\partial}{\partial \sigma_{KL}} [\frac{1}{M} \sum_{m=1}^{M} (y_m - d_m)^2] \tag{D.25}$$

or

$$\frac{\partial MSE}{\partial \sigma_{KL}} = \frac{1}{M} \sum_{m=1}^{M} \frac{\partial}{\partial \sigma_{KL}} (y_m - d_m)^2 \tag{D.26}$$

or

$$\frac{\partial MSE}{\partial \sigma_{KL}} = \frac{2}{M}\sum_{m=1}^{M}(y_m - d_m)\frac{\partial y_m}{\partial \sigma_{KL}} \tag{D.27}$$

but

$$y_m = \sum_{l=1}^{L} w_{lm} y_l \tag{D.28}$$

therefore

$$\frac{\partial y_m}{\partial \sigma_{KL}} = \sum_{l=1}^{L} w_{lm}\frac{\partial y_l}{\partial \sigma_{KL}} \tag{D.29}$$

This expands to

$$\frac{\partial y_m}{\partial \sigma_{KL}} = w_{Lm}\frac{\partial y_L}{\partial \sigma_{KL}} \tag{D.30}$$

but

$$y_L = e^{-[\sum_{k=1}^{K}\frac{(x_k - w_{kL})^2}{2\sigma_{kL}^2}]} \tag{D.31}$$

or

$$y_L = e^{-[\sum_{k=1}^{K-1}\frac{(x_k - w_{kL})^2}{2\sigma_{kL}^2}]} e^{-[\frac{(x_K - w_{KL})^2}{2\sigma_{KL}^2}]} \tag{D.32}$$

therefore

$$\frac{\partial y_L}{\partial \sigma_{KL}} = e^{-[\sum_{k=1}^{K-1}\frac{(x_k - w_{kL})^2}{2\sigma_{kL}^2}]}\frac{\partial}{\partial \sigma_{KL}} e^{-[\frac{(x_K - w_{KL})^2}{2\sigma_{KL}^2}]} \tag{D.33}$$

Simplifying

$$\frac{\partial y_L}{\partial \sigma_{KL}} = e^{-[\sum_{k=1}^{K}\frac{(x_k - w_{kL})^2}{2\sigma_{kL}^2}]}\frac{\partial}{\partial \sigma_{KL}}[-\frac{(x_K - w_{KL})^2}{2\sigma_{KL}^2}] \tag{D.34}$$

or

$$\frac{\partial y_L}{\partial \sigma_{KL}} = y_L \frac{(x_K - w_{KL})^2}{\sigma_{KL}^3} \tag{D.35}$$

Combining D.27, D.30, and D.35 gives

$$\frac{\partial MSE}{\partial \sigma_{KL}} = \frac{2}{M} \sum_{m=1}^{M} (y_m - d_m) w_{Lm} y_L \frac{(x_K - w_{KL})^2}{\sigma_{KL}^3} \tag{D.36}$$

Letting $\eta = 2C/M$ where C is a constant, the incremental training rule can now be defined as

$$\sigma_{KL}^{+} = \sigma_{KL}^{-} - \eta \sum_{m=1}^{M} (y_m - d_m) w_{Lm} y_L \frac{(x_K - w_{KL})^2}{\sigma_{KL}^3} \tag{D.37}$$

D.3 Incremental Average Update

In this section, the update rule for keeping the centers, or weights, of the radial basis functions at the average of the patterns within their assigned class will be derived. Let $\bar{w}_l(t)$ be the previous average of the N pattern vectors in the cluster at time t. Then

$$\bar{w}_l(t) = \frac{1}{N} \sum_{n=1}^{N} \bar{x}_n \tag{D.38}$$

Similarly

$$\bar{w}_l(t+1) = \frac{1}{N+1} \sum_{n=1}^{N+1} \bar{x}_n \tag{D.39}$$

This can be written as

$$\bar{w}_l(t+1) = \frac{1}{N+1} (\sum_{n=1}^{N} \bar{x}_n + \bar{x}_{N+1}) \tag{D.40}$$

Which in turn can be written as

$$\bar{w}_l(t+1) = \frac{1}{N+1} (\frac{N \sum_{n=1}^{N} \bar{x}_n}{N} + \bar{x}_{N+1}) \tag{D.41}$$

This reduces to

$$\bar{w}_l(t+1) = \frac{1}{N+1}[N\bar{w}_l(t) + \bar{x}_{N+1}] \tag{D.42}$$

Further simplifying

$$\bar{w}_l(t+1) = (1 - \frac{1}{N+1})\bar{w}_l(t) + \frac{\bar{x}_{N+1}}{N+1} \tag{D.43}$$

This reduces to

$$\bar{w}_l(t+1) = \bar{w}_l(t) + \frac{\bar{x}_{N+1} - \bar{w}(t)_{kl}}{N+1} \tag{D.44}$$

D.4 Global MSE Minimization

In this section, the technique of globally minimizing the MSE objective function through the use of a matrix inversion will be established (23). Define the total error due to all input training patterns, P, as

$$MSE = \frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M}(y_{pm} - d_{pm})^2 \tag{D.45}$$

where d_{pm} is the desired value for the m^{th} output node due to the p^{th} pattern and y_{pm} is the actual value for the m^{th} output node due to the p^{th} pattern. This error must be minimized with respect to the weights connecting a particular node, say node B, in the hidden layer, to a particular node, say node D, in the output layer. That is, the error can be minimized by setting

$$\frac{\partial MSE}{\partial w_{BD}} = 0 \tag{D.46}$$

but

$$\frac{\partial MSE}{\partial w_{BD}} = \frac{\partial}{\partial w_{BD}}[\frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M}(y_{pm} - d_{pm})^2] \tag{D.47}$$

or

$$\frac{\partial MSE}{\partial w_{BD}} = \sum_{p=1}^{P}(y_{pD} - d_{pD})\frac{\partial y_{pD}}{\partial w_{BD}} \tag{D.48}$$

but

$$y_{pD} = \sum_{l=1}^{L} w_{lD} y_{pl} \tag{D.49}$$

therefore

$$\frac{\partial y_{pD}}{\partial w_{BD}} = \frac{\partial}{\partial w_{BD}} (\sum_{l=1}^{L} w_{lD} y_{pl}) \tag{D.50}$$

or

$$\frac{\partial y_{pD}}{\partial w_{BD}} = y_{pB} \tag{D.51}$$

thus

$$\frac{\partial MSE}{\partial w_{BD}} = \sum_{p=1}^{P} (y_{pD} - d_{pD}) y_{pB} \tag{D.52}$$

Now, substituting for y_{pD}, provides

$$\frac{\partial MSE}{\partial w_{BD}} = \sum_{p=1}^{P} (\sum_{l=1}^{L} w_{lD} y_{pl} - d_{pD}) y_{pB} \tag{D.53}$$

or

$$\frac{\partial MSE}{\partial w_{BD}} = \sum_{p=1}^{P} \sum_{l=1}^{L} w_{lD} y_{pl} y_{pB} - \sum_{p=1}^{P} d_{pD} y_{pB} \tag{D.54}$$

Setting the equation to zero to minimize the error gives

$$\sum_{p=1}^{P} \sum_{l=1}^{L} w_{lD} y_{pl} y_{pB} = \sum_{p=1}^{P} d_{pD} y_{pB} \tag{D.55}$$

Now, let's define a new variable

$$M_{lB} = \sum_{p=1}^{P} y_{pl} y_{pB} \tag{D.56}$$

Here, y_{pl} is the output of the l^{th} radial basis function node, where $1 \leq l \leq L$, due to the p^{th} pattern and y_{pB} is the output of the B^{th} radial basis function due to the p^{th} pattern. This allows the following equation to be written:

$$\sum_{l=1}^{L} w_{lD} M_{lB} = \sum_{p=1}^{P} d_{pD} y_{pB} \tag{D.57}$$

Now, the MSE can be minimized by ensuring the $\partial MSE/\partial w_{lm} = 0$ for all weights. Define a weight matrix W as an L by M matrix as

$$W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1M} \\ w_{21} & w_{22} & \dots & w_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ w_{L1} & w_{L2} & \dots & w_{LM} \end{bmatrix} \tag{D.58}$$

Minimizing the MSE is the same as making $\partial MSE/\partial W = 0$ for all weights. That is, the error is minimized by setting the $\partial MSE/\partial w_{lm} = 0$ for each weight in the weight matrix. But, from above, for a given weight w_{BD}, the $\partial MSE/\partial w_{BD} = 0$ when

$$\sum_{l=1}^{L} w_{lD} M_{lB} = \sum_{p=1}^{P} d_{pD} y_{pB} \tag{D.59}$$

This equation states that the derivative of the error with respect to a weight linking the B^{th} node radial basis function in the hidden layer to the D^{th} node in the output layer is minimized when the sum of all the weights in the hidden layer, multiplied by the summation of the product of the radial basis function outputs and the B^{th} radial basis function output, summed over all patterns, is equal summation of the product of the desired output of the D^{th} output node and the B^{th} radial basis function summed over all patterns. For example, when $B = 1, D = 1$, $\partial MSE/\partial w_{11} = 0$ implies that

$$\sum_{l=1}^{L} w_{l1} M_{l1} = \sum_{p=1}^{P} d_{p1} y_{p1} \tag{D.60}$$

which expands to

$$w_{11} M_{11} + w_{21} M_{21} + \ldots + w_{L1} M_{L1} = d_{11} y_{11} + d_{21} y_{21} + \ldots + d_{P1} y_{P1} \tag{D.61}$$

Also, when $B = 1, D = 2$, then $\partial MSE/\partial w_{12} = 0$ implies that

$$\sum_{l=1}^{L} w_{l2} M_{l1} = \sum_{p=1}^{P} d_{p2} y_{p1} \tag{D.62}$$

which expands to

$$w_{12}M_{11} + w_{22}M_{21} + \ldots + w_{L2}M_{L1} = d_{12}y_{11} + d_{22}y_{21} + \ldots + d_{P2}y_{P1} \tag{D.63}$$

Finally, when $B = 1, D = 2$, then $\partial MSE/\partial w_{21} = 0$ implies that

$$\sum_{l=1}^{L} w_{l1} M_{l2} = \sum_{p=1}^{P} d_{p1} y_{p2} \tag{D.64}$$

which expands to

$$w_{11}M_{12} + w_{21}M_{22} + \ldots + w_{L1}M_{L2} = d_{11}y_{12} + d_{21}y_{22} + \ldots + d_{P1}y_{P2} \tag{D.65}$$

and so forth for each of the weights connection nodes in the hidden layer to nodes in the output layer. This gives a set of L*M equations with L*M unknown weights which can be written as

$$M^T W = Y^T S \tag{D.66}$$

Here M is an L by L matrix containing the summation, over all patterns, of the product of each radial basis function output, for a given input pattern and the B^{th} radial basis function output for that pattern. That is

$$M = \begin{bmatrix} M_{11} & M_{12} & \ldots & M_{1L} \\ M_{21} & M_{22} & \ldots & M_{2L} \\ \vdots & \vdots & \vdots & \vdots \\ M_{L1} & M_{L2} & \ldots & M_{LL} \end{bmatrix} \tag{D.67}$$

where $M_{lB} = \sum_{p=1}^{P} y_{pl} y_{pB}$

Also, W is an L by M matrix containing the weights linking the nodes in the hidden layer to an nodes in the output layer. That is

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1M} \\ w_{21} & w_{22} & \cdots & w_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ w_{L1} & w_{L2} & \cdots & w_{LM} \end{bmatrix} \tag{D.68}$$

where w_{BD} is the weight linking the B^{th} node in the hidden layer to the D^{th} node in the output layer. Here, Y is a P by L matrix containing the outputs for each of the L radial basis functions for all P patterns. That is

$$Y = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1L} \\ y_{21} & y_{22} & \cdots & y_{2L} \\ \vdots & \vdots & \vdots & \vdots \\ y_{P1} & y_{P2} & \cdots & y_{PL} \end{bmatrix} \tag{D.69}$$

Finally S is a P by M matrix containing the desired outputs for each of the M output nodes for all P patterns. That is

$$S = \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1M} \\ d_{21} & d_{22} & \cdots & d_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ d_{P1} & d_{P2} & \cdots & d_{PM} \end{bmatrix} \tag{D.70}$$

Thus, the weights which minimize the MSE can be found by using the following equation:

$$W = (M^T)^{-1} Y^T S \tag{D.71}$$

Now, the optimized weight w^*_{BD} can be found as:

$$(M^T)^{-1} = N = \begin{bmatrix} n_{11} & n_{12} & \cdots & n_{1L} \\ n_{21} & n_{22} & \cdots & n_{2L} \\ \vdots & \vdots & \vdots & \vdots \\ n_{L1} & n_{L2} & \cdots & n_{LL} \end{bmatrix} \tag{D.72}$$

or

$$w^*_{BD} = N(Y^T S)_{BD} \tag{D.73}$$

Now

$$Y^T S = \begin{bmatrix} \sum_{p=1}^{P} y_{p1} d_{p1} & \sum_{p=1}^{P} y_{p1} d_{p2} & \cdots & \sum_{p=1}^{P} y_{p1} d_{pM} \\ \sum_{p=1}^{P} y_{p2} d_{p1} & \sum_{p=1}^{P} y_{p2} d_{p2} & \cdots & \sum_{p=1}^{P} y_{p2} d_{pM} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{p=1}^{P} y_{pL} d_{p1} & \sum_{p=1}^{P} y_{pL} d_{p2} & \cdots & \sum_{p=1}^{P} y_{pL} d_{pM} \end{bmatrix} \tag{D.74}$$

therefore:

$$w^*_{BD} = \sum_{l=1}^{L} (\sum_{p=1}^{P} y_{pl} d_{pD}) N_{Bl} \tag{D.75}$$

This method only works for matrices that do not become singular or near-singular, which can happen if the exemplar data points used to center the radial basis functions contain redundant information. If they do, the Singular Valued Decomposition of the matrix may be used. Conversely, the σs of the offending nodes may be adjusted to eliminate the redundancy.

Appendix E. *Tables for Data Analysis*

E.1 Introduction

This appendix contains the data obtained from the neural network testing, discussed in Chapter 4, for the communications data and the radar signal data. This appendix begins with the compilation of the data for the Kernel Classifier and Hyperplane Classifier networks implemented to categorize the communications data. This is followed by a compilation of the data for the Kernel Classifiers used to categorize the radar data.

E.2 Communications Signal Characterization

This data consisted of 202, 50-dimensional pattern vectors. These pattern vectors represented either a direct sequence or a linear-stepped frequency-hopped digital communications signal. Both Hyperplane and Kernel Classifier networks were developed to categorize this data.

E.2.1 Hyperplane Classifiers The topology for the sigmoidal-based Hyperplane Classifier networks implemented for this problem is shown in figure 3.3. The network parameters, being the weights and the offsets, were set using the backpropagation algorithms developed in Chapter 3.

E.2.1.1 MSE Algorithm Table E.1 shows the categorization performance for ten Hyperplane Classifier networks whose parameters were trained using the incremental backpropagation algorithm according to the MSE objective function. The data seed matched the run number and 51 training vectors were loaded for each class. The weight seed and sigma seed were both zero and the record seed was one. The network had 50 nodes in layer 0, 18 nodes in layer 1, ten nodes in layer 2 and two nodes in layer 3. Each of the nodes was assigned the sigmoidal transfer function.

E.2.1.2 CE Algorithm Table E.2 shows the categorization performance for ten networks whose parameters were trained using the incremental backpropagation algorithm according to the CE objective function. The data seed matched the run number and 51 training vectors were loaded for each class. The weight seed and sigma seed were both zero and the record seed was one. The network had 50 nodes in layer 0, 18 nodes in layer 1, ten nodes in layer 2 and two nodes in the layer 3. Each node was assigned the sigmoidal transfer function.

E.2.1.3 CFM Algorithm Table E.3 shows the categorization performance for ten Hyperplane Classifier networks whose parameters were trained using the incremental backpropagation algorithm according to the CFM objective function. The data seed matched the run number and

Table E.1. MSE Network Performance

Iterations	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Run 10 %crct	Avg %crct	Std %crct
1000	0.00	0.00	3.92	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.39	1.18
2000	4.90	0.00	5.88	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.08	2.17
3000	2.94	1.96	7.84	0.00	0.98	0.00	2.90	0.00	0.00	2.90	1.93	2.32
4000	7.84	0.00	7.84	0.00	5.88	1.96	4.90	2.90	0.00	2.00	3.33	2.95
5000	9.80	4.90	7.84	5.88	3.92	0.00	5.90	1.00	2.00	0.00	4.12	3.18
6000	8.82	18.63	9.80	23.53	5.88	15.67	6.90	4.90	23.50	0.00	11.76	7.71
7000	12.75	39.22	19.61	43.14	7.84	23.53	19.60	6.90	44.10	7.80	22.45	14.00
8000	19.61	50.98	35.30	66.67	24.51	50.98	30.40	10.80	49.00	44.10	38.24	16.25
9000	37.25	63.73	52.94	72.55	33.30	58.80	47.10	28.40	62.70	53.00	50.98	13.64
10000	55.88	63.73	58.82	81.37	60.78	76.47	57.80	45.00	73.50	63.70	63.91	10.45
11000	60.78	66.67	73.53	84.31	64.71	71.57	69.60	55.90	84.30	72.50	70.39	8.67
12000	70.59	80.39	78.43	84.31	75.49	80.39	80.40	71.60	88.20	80.40	79.02	5.09
13000	83.33	83.33	77.45	73.53	81.37	90.20	87.30	75.50	85.20	82.40	81.96	4.94
14000	85.29	77.45	76.47	82.35	85.29	93.14	90.20	82.40	95.10	86.30	85.40	5.82
15000	84.31	84.31	86.27	86.27	80.39	100.00	92.21	86.20	94.10	83.30	87.74	5.61
16000	83.33	89.22	88.24	89.22	73.53	100.00	94.10	91.20	94.10	89.20	89.21	6.71
17000	86.27	86.27	87.25	84.31	83.33	100.00	96.10	92.20	93.10	93.10	90.19	5.23
18000	89.22	88.24	89.22	90.20	85.33	100.00	99.00	93.10	93.10	85.30	91.27	5.21
19000	90.20	90.20	84.31	93.14	87.25	100.00	97.10	96.10	98.00	94.10	93.04	4.75
20000	91.18	94.12	90.20	93.14	87.25	100.00	99.00	95.10	96.10	98.00	94.41	3.87
21000	85.29	97.06	78.43	100.00	85.29	100.00	97.10	97.10	97.10	100.00	93.74	7.35
22000	93.14	96.08	91.18	100.00	90.20	100.00	99.00	95.10	94.10	100.00	95.88	3.55
23000	100.00	97.06	87.25	100.00	80.39	100.00	99.00	98.00	93.10	100.00	95.48	6.36
24000	100.00	99.02	88.23	100.00	86.27	100.00	99.00	97.10	99.00	100.00	96.86	4.90
25000	100.00	99.02	81.37	100.00	82.35	100.00	100.00	97.10	90.20	100.00	95.00	7.17
26000	100.00	99.02	91.18	100.00	82.35	100.00	100.00	98.00	97.10	100.00	96.77	5.46
27000	100.00	99.02	97.06	100.00	78.43	100.00	100.00	96.10	98.00	100.00	96.86	6.29
28000	100.00	97.06	97.06	100.00	82.35	100.00	100.00	96.20	96.10	100.00	96.88	5.11
29000	100.00	99.02	94.12	100.00	86.30	100.00	100.00	98.00	92.20	100.00	96.96	4.43
30000	100.00	100.00	94.12	100.00	86.30	100.00	100.00	99.00	92.20	100.00	97.16	4.51
Test % crct	86.00	76.00	66.00	86.00	79.00	78.00	86.00	77.00	79.00	84.00	79.70	5.92

Table E.2. CE Network Performance

Iteration	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Run 10 %crct	Average %crct	Std %crct
1000	0.00	6.86	0.00	2.94	0.00	0.00	5.88	8.82	0.00	0.00	2.45	3.29
2000	19.61	25.49	31.37	20.59	18.63	16.67	27.45	11.76	40.20	9.80	22.16	8.73
3000	52.94	54.90	63.73	44.12	76.47	39.22	54.90	52.94	47.06	55.88	54.22	9.85
4000	95.10	73.53	46.08	83.33	90.20	40.20	78.43	74.51	60.78	69.61	71.18	16.89
5000	95.10	91.18	57.84	96.08	91.18	57.84	89.22	80.39	75.49	81.37	81.57	13.42
6000	95.10	84.31	91.18	90.20	97.06	67.65	90.20	79.41	89.22	95.10	87.94	8.40
7000	99.02	88.24	91.18	92.16	96.08	79.41	92.16	100.00	84.31	100.00	92.25	6.52
8000	99.02	97.06	97.06	89.22	93.14	79.41	99.02	100.00	93.14	100.00	94.71	6.11
9000	98.04	96.08	100.00	97.06	100.00	94.12	100.00	100.00	96.08	100.00	98.14	2.08
10000	99.02	99.02	100.00	96.08	100.00	99.02	100.00	100.00	83.33	100.00	97.65	4.91
11000	100.00	100.00	100.00	98.04	100.00	100.00	100.00	100.00	96.08	100.00	99.41	1.26
12000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	78.41	100.00	97.84	6.48
13000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	92.16	100.00	99.22	2.35
14000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	91.18	100.00	99.12	2.65
15000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	98.04	100.00	99.80	0.59
16000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	95.10	100.00	99.51	1.47
17000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	92.16	100.00	99.22	2.35
18000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	95.10	100.00	99.51	1.47
19000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	98.04	100.00	99.80	0.59
20000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	97.06	100.00	99.71	0.88
21000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	99.02	100.00	99.90	0.29
22000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	94.12	100.00	99.41	1.76
23000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	94.12	100.00	99.41	1.76
24000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	97.06	100.00	99.71	0.88
25000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	99.02	100.00	99.90	0.29
26000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	99.02	100.00	99.90	0.29
27000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	99.02	100.00	99.90	0.29
28000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
29000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
30000	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
Test % Crct	77.00	71.00	85.00	80.00	83.00	88.00	83.00	78.00	85.00	82.00	81.20	4.64

Table E.3. CFM Network Performance

Iterations	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Run 10 %crct	Avg %crct	Std %crct
1000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2000	6.86	37.25	0.00	38.24	33.33	0.00	12.75	15.69	0.00	0.00	14.41	15.30
3000	14.71	45.10	19.61	44.12	43.14	0.00	40.20	16.67	1.96	16.67	24.22	16.59
4000	11.76	49.02	17.65	49.02	47.06	21.57	37.25	34.31	17.65	6.86	29.22	15.24
5000	31.37	50.98	33.33	52.94	48.04	41.18	36.27	36.27	31.37	28.43	39.02	8.36
6000	54.90	50.98	43.14	53.92	49.02	48.04	44.12	37.25	37.25	37.25	45.59	6.48
7000	61.76	48.04	43.14	50.00	51.96	49.02	39.22	39.22	49.02	38.24	46.96	6.88
8000	68.63	46.08	49.02	36.27	52.94	50.98	44.12	48.04	62.75	46.08	50.49	8.81
9000	74.51	29.41	61.76	46.08	44.12	50.00	45.10	54.91	66.67	46.08	51.66	12.33
10000	78.43	36.27	71.57	41.18	54.90	53.92	50.98	55.88	74.51	50.00	56.76	13.26
11000	79.41	52.94	81.37	52.94	58.82	55.88	60.78	52.94	77.45	56.86	62.94	11.09
12000	81.31	59.80	77.45	52.94	58.82	54.90	65.69	53.92	78.43	58.82	64.51	10.86
13000	83.33	62.75	78.43	54.90	63.72	55.88	68.63	62.75	81.37	57.84	66.96	10.04
14000	83.33	63.73	83.33	50.00	65.69	55.88	71.59	70.59	82.35	54.90	68.14	11.67
15000	85.29	64.71	87.25	50.98	64.71	56.86	76.47	76.47	87.25	60.78	71.08	12.56
16000	83.33	71.57	88.24	58.82	67.65	5C.86	76.47	79.41	85.29	64.71	73.24	10.51
17000	84.31	70.59	89.22	61.76	69.61	57.84	80.39	83.33	87.25	64.71	74.90	10.78
18000	85.29	76.47	89.22	65.69	67.65	57.84	80.39	83.33	87.25	65.69	75.88	10.36
19000	85.29	90.20	90.20	72.55	71.57	56.86	86.27	82.35	88.24	63.73	78.73	11.22
20000	86.27	79.41	91.18	74.51	71.57	57.84	87.25	83.33	88.24	76.47	79.61	9.48
21000	87.25	81.37	91.18	75.49	72.55	58.82	87.25	86.27	89.22	77.45	80.69	9.39
22000	87.25	83.33	91.18	79.41	72.55	58.82	88.24	89.22	89.22	80.39	81.96	9.45
23000	87.25	84.31	90.02	81.37	74.51	58.82	88.24	90.20	89.22	85.29	82.92	9.23
24000	87.25	86.27	91.18	87.25	73.53	49.02	88.24	91.18	89.22	81.37	82.45	12.21
25000	87.25	85.29	91.18	90.20	73.53	53.92	88.24	91.18	89.22	85.29	83.53	11.01
26000	87.25	85.29	91.18	89.22	74.51	52.94	88.24	91.18	89.22	90.20	83.92	11.32
27000	87.25	86.27	91.18	89.22	74.51	54.90	88.24	91.18	89.22	90.20	84.22	10.80
28000	87.25	83.33	97.06	91.18	74.51	51.96	88.24	91.18	89.22	91.17	84.51	12.23
29000	87.25	83.33	91.18	89.22	74.51	52.94	88.24	91.18	89.22	90.20	83.73	11.31
30000	87.25	86.27	91.18	92.16	74.51	55.88	88.24	91.18	89.22	93.14	84.90	10.89
31000	87.25	86.27	91.18	92.16	74.51	64.71	88.24	91.18	89.22	93.14	85.79	8.63
32000	87.25	87.25	91.18	93.14	74.51	68.63	88.24	91.18	89.22	93.14	86.37	7.79
33000	87.25	87.25	91.18	92.16	74.51	72.55	88.24	91.18	89.22	92.16	86.57	6.77
34000	87.25	87.25	91.18	92.16	74.51	73.53	88.24	91.18	89.22	93.14	86.77	6.65
35000	87.25	89.22	91.18	94.12	74.51	78.43	88.24	91.18	89.22	92.16	86.57	6.77
36000	87.25	90.20	91.18	92.16	74.51	81.37	88.24	91.18	89.22	93.14	87.85	5.45
37000	87.25	89.22	91.18	93.14	74.51	82.35	88.24	91.18	89.22	93.14	87.94	5.39
38000	87.25	90.20	91.18	93.14	74.51	83.33	88.24	91.18	89.22	93.14	88.14	5.33
39000	87.25	90.20	89.22	94.12	74.51	83.33	88.24	91.18	89.22	93.14	88.04	5.35
40000	87.25	91.18	90.20	94.12	74.51	79.41	88.24	91.18	89.22	93.14	87.85	5.89
41000	87.25	90.20	91.18	94.12	74.51	85.29	88.24	91.18	89.22	93.14	88.43	5.28
42000	87.25	91.18	91.18	94.12	74.51	85.29	88.24	91.18	89.22	93.14	88.53	5.32
43000	87.25	90.20	91.18	94.12	74.51	85.29	88.24	91.18	89.22	93.14	88.43	5.28
44000	87.25	91.18	91.18	94.12	74.51	86.27	88.24	91.18	89.22	93.14	88.63	5.27
45000	87.25	92.16	91.18	94.12	74.51	86.27	88.24	91.18	89.22	93.14	88.73	5.32
46000	87.25	92.16	91.18	94.12	74.51	86.27	88.24	91.18	89.22	93.14	88.73	5.32
47000	87.25	92.16	92.16	94.12	74.51	86.27	88.24	91.18	89.22	93.14	88.83	5.37
48000	87.25	92.16	92.16	94.12	74.51	86.27	88.24	91.18	89.22	93.14	88.83	5.37
49000	87.25	92.16	92.16	94.12	74.51	86.27	88.24	91.18	89.22	93.14	88.83	5.37
50000	87.25	92.16	92.16	94.12	74.51	86.27	88 24	91.18	89.22	93.14	88.83	5.37
Test % crct	57.00	74 00	77.00	74 00	62.00	72 00	71.00	82 00	80 00	83 00	73 20	7.93

51 training vectors were loaded for each class. The weight seed and sigma seed were both zero and the record seed was one. The network had 50 nodes in layer 0, 18 nodes in layer 1, ten nodes in layer 2 and two nodes in layer 3. Each node was assigned the sigmoidal transfer function.

E.2.2 Kernel Classifiers The topology for the radial basis function Kernel Classifier networks implemented for this problem is shown in figure 3.7 The network parameters, being the weights and the spreads, were set using the algorithms developed in Chapter 3.

E.2.2.1 Nodes at Data Points Tables E.4 and E.5 show the training and test categorization performance for a Kernel Classifier network with a variable number of hidden layer, layer 1, nodes. The weights for these nodes were established using the Nodes at Data Points algorithm.

Table E.4. Nodes at Data Points Training Performance vs Nodes

Nodes	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
102	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
90	100.00	99.02	99.02	99.02	99.02	100.00	100.00	100.00	98.04	98.04	99.22	0.73
80	99.02	99.02	97.06	97.06	99.02	99.02	99.02	100.00	98.04	98.04	98.53	0.90
70	94.12	97.06	96.08	96.08	95.10	98.04	97.06	100.00	97.06	97.06	96.77	1.52
60	95.10	97.06	97.06	95.10	93.14	94.12	91.17	99.02	96.08	96.08	95.39	2.11
50	93.14	97.06	95.10	95.10	93.14	92.16	89.22	97.06	96.08	90.20	93.83	2.59
40	87.25	97.06	91.18	88.24	88.24	89.22	83.33	82.16	91.18	90.20	88.81	3.98
30	84.31	94.12	91.18	84.31	89.22	85.29	75.49	92.16	88.24	83.33	86.77	5.14
20	79.41	89.22	85.29	73.53	84.31	74.51	71.46	88.24	83.33	79.41	80.87	5.90
10	65.59	73.53	65.73	68.63	75.49	60.78	62.75	78.43	72.55	58.82	68.03	6.36
0	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	0.00

Table E.5. Nodes at Data Points Test Performance vs Nodes

Nodes	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
102	85.00	81.00	77.00	87.00	83.00	92.00	89.00	85.00	86.00	84.00	84.90	3.94
90	84.00	82.00	77.00	78.00	84.00	92.00	85.00	85.00	84.00	83.00	83.40	3.90
80	83.00	76.00	74.00	80.00	82.00	92.00	84.00	85.00	84.00	86.00	82.60	4.84
70	82.00	75.00	77.00	82.00	79.00	86.00	83.00	84.00	85.00	79.00	81.20	3.40
60	84.00	73.00	75.00	78.00	78.00	87.00	75.00	83.00	80.00	81.00	79.40	4.22
50	87.00	71.00	75.00	79.00	79.00	85.00	75.00	80.00	79.00	75.00	78.50	4.59
40	78.00	75.00	73.00	81.00	75.00	87.00	76.00	76.00	83.00	76.00	78.00	4.12
30	78.00	71.00	69.00	73.00	71.00	90.00	63.00	79.00	85.00	74.00	75.30	7.52
20	75.00	71.00	68.00	66.00	65.00	74.00	60.00	74.00	80.00	72.00	70.50	5.52
10	60.00	62.00	59.00	58.00	60.00	64.00	51.00	59.00	59.00	64.00	59.60	3.50
0	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	0.00

For the software parameters, the output threshold was set to one, the sigma threshold was set to four. The training rule for the sigma was the Scale Sigma According to Class Interference with interference threshold of .4. The data seed was zero and 51 training vectors were loaded for each class. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 is shown in the table and the number of nodes in layer 2 was two. The transfer function for the nodes in layer 1 was gaussian while the transfer function for the nodes in layer 2 was linear. The weights linking layer 1 nodes to layer 2 were set via global minimization of the MSE.

E.2.2.2 Kohonen Training Tables E.6 and E.7 show the training and test categorization performance of Kernel Classifier networks with the layer 1 weights trained using the Kohonen Training algorithm with the RBF spreads set using the P-Nearest Neighbor algorithm and P held constant at six.

Table E.6. Kohonen Training Performance vs Nodes with Six P-Neighbors

Nodes	P	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	6	100.00	82.35	100.00	100.00	96.08	79.41	96.08	91.18	100.00	93.14	93.82	7.14
81	6	99.02	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	98.04	99.71	0.63
64	6	96.08	99.02	96.08	97.06	99.02	98.04	99.02	99.02	100.00	95.10	97.84	1.57
49	6	95.10	94.12	94.20	95.10	96.08	96.08	93.14	94.12	96.08	95.10	94.91	0.95
36	6	91.18	94.12	88.24	92.20	87.25	94.12	93.14	91.18	89.22	92.16	91.28	2.26
25	6	88.24	84.31	90.20	88.24	89.22	88.24	86.29	87.25	88.24	88.24	87.85	1.53
16	6	77.45	88.24	85.30	82.35	84.31	83.33	80.39	83.33	86.27	86.27	83.72	2.98

Table E.7. Kohonen Test Performance vs Nodes with Six P-Neighbors

Nodes	P	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	6	63.00	59.00	56.00	75.00	62.00	54.00	64.00	64.00	72.00	51.00	62.00	7.13
81	6	74.00	65.00	76.00	79.00	82.00	85.00	66.00	79.00	80.00	74.00	76.00	6.16
64	6	75.00	77.00	79.00	75.00	80.00	77.00	69.00	86.00	80.00	81.00	77.90	4.28
49	6	80.00	71.00	76.00	79.00	73.00	86.00	77.00	76.00	82.00	78.00	77.80	4.09
36	6	76.00	74.00	81.00	75.00	75.00	82.00	83.00	80.00	80.00	80.00	78.60	3.10
25	6	80.00	69.00	82.00	75.00	78.00	77.00	81.00	72.00	80.00	76.00	77.00	3.92
16	6	79.00	71.00	76.00	71.00	76.00	77.00	79.00	69.00	78.00	71.00	74.70	3.61

For the software parameters, the data seed matched the run number and 51 training vectors were loaded from each class. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 is shown in the table and the number of nodes in layer 2 was two. The transfer function for the nodes in layer 1 was gaussian while the transfer function for the nodes in layer 2 was linear. The weights linking layer 1 nodes to the layer 2 nodes were set using the global MSE minimization algorithm.

Tables E.8 and E.9 show the categorization performance of Kernel Classifier networks with the layer 1 weights trained using the Kohonen Training algorithm. Here the spreads of the RBFs were set using the P-Nearest Neighbors algorithm with P allowed to vary as the square root of the number of Kohonen nodes.

Table E.8. Kohonen Training Performance vs Nodes with Variable P-Neighbors

Nodes	P	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	10	100.00	99.02	61.76	97.06	94.12	100.00	100.00	83.33	94.12	90.02	91.94	11.27
81	9	99.02	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	99.90	0.29
64	8	96.08	99.02	96.08	98.04	99.02	97.06	99.04	99.02	100.00	96.08	97.94	1.42
49	7	95.10	95.10	93.14	95.10	96.08	97.06	91.18	94.12	96.08	95.10	94.81	1.58
36	6	91.18	94.12	88.24	92.16	87.25	94.12	93.14	91.18	89.22	92.16	91.28	2.26
25	5	88.24	83.33	89.22	86.27	89.22	88.24	86.29	88.24	88.24	88.24	87.55	1.70
16	4	77.45	87.25	84.31	81.37	84.31	81.37	80.39	83.33	88.24	87.25	83.53	3.28

Table E.9. Kohonen Test Performance vs Nodes with Variable P-Neighbors

Nodes	P	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	10	75.00	66.00	60.00	56.00	56.00	61.00	58.00	56.00	46.00	53.00	58.70	7.36
81	9	80.00	65.00	73.00	78.00	77.00	72.00	84.00	77.00	82.00	77.00	76.50	5.16
64	8	79.00	76.00	77.00	77.00	79.00	79.00	83.00	90.00	81.00	81.00	80.20	3.84
49	7	80.00	72.00	79.00	80.00	74.00	87.00	79.00	77.00	82.00	77.00	78.70	3.95
36	6	76.00	74.00	81.00	75.00	75.00	82.00	83.00	80.00	80.00	80.00	78.60	3.10
25	5	80.00	68.00	80.00	75.00	78.00	77.00	81.00	72.00	81.00	76.00	76.80	4.02
16	4	78.00	71.00	77.00	72.00	76.00	78.00	79.00	69.00	76.00	73.00	74.90	3.24

For the software parameters, the data seed matched the run number and 51 training vectors were loaded from each class. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 is shown in the table and the number of nodes in layer 2 was two. The transfer function for the nodes in layer 1 was gaussian while the transfer function for the nodes in layer 2 was linear. The weights linking layer 1 nodes to the layer 2 nodes were set using the global MSE minimization algorithm.

E.2.2.3 K-Means Clustering Tables E.10 and E.11 show the categorization performance of a Kernel Classifier with the hidden layer, layer 1, weights trained via the K-Means Clustering Algorithm as the number of nodes increased.

Table E.10. K-Means Training Performance vs Nodes with Six P-Neighbors

Nodes	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	100.00	100.00	100.00	100.00	99.02	100.00	100.00	100.00	100.00	100.00	99.90	0.29
90	100.00	99.02	100.00	100.00	98.04	98.04	100.00	100.00	99.02	100.00	99.41	0.78
80	97.06	99.02	95.10	98.04	97.06	98.04	100.00	100.00	97.06	99.02	98.33	1.52
70	94.12	99.02	92.16	96.08	95.10	98.04	100.00	100.00	96.08	98.04	97.16	2.30
60	92.16	98.04	92.16	95.10	95.10	97.06	96.08	97.06	97.06	94.12	95.59	1.71
50	94.12	98.04	94.18	95.10	94.12	94.12	93.14	95.10	94.12	91.18	94.13	1.75
40	91.18	96.08	88.24	93.14	94.12	94.12	94.12	93.14	93.14	90.20	93.04	2.12
30	87.25	92.16	87.25	89.22	92.16	90.20	89.22	91.20	88.24	89.22	90.01	1.57
20	86.27	84.31	84.31	86.27	88.24	84.31	76.47	88.24	84.31	83.33	84.70	3.23
10	70.59	77.45	77.45	84.31	80.39	79.41	77.45	84.31	72.55	82.35	80.19	3.96

Table E.11. K-Means Test Performance vs Nodes with Six P-Neighbors

Nodes	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	82.00	79.00	74.00	91.00	84.00	86.00	91.00	87.00	78.00	87.00	83.90	5.34
90	86.00	79.00	73.00	89.00	82.00	85.00	90.00	85.00	78.00	84.00	82.70	4.86
80	80.00	76.00	72.00	86.00	81.00	84.00	87.00	83.00	81.00	86.00	82.20	4.64
70	78.00	80.00	79.00	83.00	78.00	86.00	85.00	85.00	84.00	84.00	82.40	2.73
60	75.00	83.00	75.00	83.00	78.00	80.00	85.00	85.00	84.00	78.00	80.90	3.36
50	79.00	79.00	73.00	82.00	78.00	80.00	80.00	80.00	87.00	76.00	79.00	3.71
40	79.00	74.00	70.00	80.00	79.00	82.00	82.00	80.00	86.00	74.00	78.60	4.45
30	83.00	71.00	72.00	79.00	79.00	81.00	54.00	77.00	78.00	72.00	74.20	7.52
20	79.00	70.00	71.00	77.00	72.00	80.00	78.00	73.00	76.00	75.00	75.50	3.93
10	77.00	70.00	78.00	72.00	73.00	74.00	79.00	75.00	80.00	73.00	75.30	3.29

For the software parameters, the RBF sigmas were set via the P-Nearest Neighbor algorithm with P held at six. The data seed matched the run number and the data was loaded by classes. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 is shown in the table

Table E.9. Kohonen Test Performance vs Nodes with Variable P-Neighbors

Nodes	P	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	10	75.00	66.00	60.00	56.00	56.00	61.00	56.00	56.00	46.00	53.00	58.70	7.36
81	9	80.00	65.00	73.00	78.00	77.00	72.00	64.00	77.00	82.00	77.00	76.50	5.16
64	8	79.00	76.00	77.00	77.00	79.00	79.00	83.00	90.00	81.00	81.00	80.20	3.84
49	7	80.00	72.00	79.00	80.00	74.00	87.00	79.00	77.00	82.00	77.00	78.70	3.95
36	6	76.00	74.00	81.00	75.00	73.00	82.00	83.00	80.00	80.00	80.00	78.60	3.10
25	5	80.00	68.00	80.00	75.00	78.00	77.00	81.00	72.00	81.00	76.00	76.80	4.02
16	4	78.00	71.00	77.00	72.00	76.00	78.00	79.00	69.00	76.00	73.00	74.90	3.24

For the software parameters, the data seed matched the run number and 51 training vectors were loaded from each class. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 is shown in the table and the number of nodes in layer 2 was two. The transfer function for the nodes in layer 1 was gaussian while the transfer function for the nodes in layer 2 was linear. The weights linking layer 1 nodes to the layer 2 nodes were set using the global MSE minimization algorithm.

E.2.2.3 K-Means Clustering Tables E.10 and E.11 show the categorization performance of a Kernel Classifier with the hidden layer, layer 1, weights trained via the K-Means Clustering Algorithm as the number of nodes increased.

Table E.10. K-Means Training Performance vs Nodes with Six P-Neighbors

Nodes	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	100.00	100.00	100.00	100.00	99.02	100.00	100.00	100.00	100.00	100.00	99.90	0.29
90	100.00	99.02	100.00	100.00	98.04	98.04	100.00	100.00	99.02	100.00	99.41	0.78
80	97.06	99.02	95.10	98.04	97.06	98.04	100.00	100.00	97.06	99.02	98.33	1.52
70	94.12	99.02	92.16	96.08	95.10	98.04	100.00	100.00	96.08	98.04	97.16	2.30
60	92.16	98.04	92.16	95.10	95.10	97.06	96.08	97.06	97.06	94.12	95.59	1.71
50	94.12	98.04	94.18	95.10	94.12	94.12	93.14	95.10	94.12	91.18	94.13	1.75
40	91.18	96.08	88.24	93.14	94.12	94.12	94.12	93.14	93.14	90.20	93.04	2.12
30	87.25	92.16	87.25	89.22	92.16	90.20	89.22	91.20	88.24	89.22	90.01	1.57
20	86.27	84.31	84.31	86.27	88.24	84.31	76.47	88.24	84.31	83.33	84.70	3.23
10	70.59	77.45	77.45	84.31	80.39	79.41	77.45	84.31	72.55	82.35	80.19	3.96

Table E.11. K-Means Test Performance vs Nodes with Six P-Neighbors

Nodes	Run 0 %crct	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Avg %crct	Std %crct
100	82.00	79.00	74.00	91.00	84.00	86.00	91.00	87.00	78.00	87.00	83.90	5.34
90	86.00	79.00	73.00	89.00	82.00	85.00	90.00	85.00	78.00	84.00	82.70	4.86
80	80.00	76.00	72.00	86.00	81.00	84.00	87.00	83.00	81.00	86.00	82.20	4.64
70	78.00	80.00	79.00	83.00	78.00	86.00	85.00	85.00	84.00	84.00	82.40	2.73
60	75.00	83.00	75.00	83.00	78.00	80.00	85.00	85.00	84.00	78.00	80.90	3.36
50	79.00	79.00	73.00	82.00	78.00	80.00	80.00	80.00	87.00	76.00	79.00	3.71
40	79.00	74.00	70.00	80.00	79.00	82.00	82.00	80.00	86.00	74.00	78.60	4.45
30	83.00	71.00	72.00	79.00	79.00	81.00	84.00	77.00	78.00	72.00	74.20	7.52
20	79.00	70.00	71.00	77.00	72.00	80.00	78.00	73.00	76.00	75.00	75.50	3.93
10	77.00	70.00	78.00	72.00	73.00	74.00	79.00	75.00	80.00	73.00	75.30	3.29

For the software parameters, the RBF sigmas were set via the P-Nearest Neighbor algorithm with P held at six. The data seed matched the run number and the data was loaded by classes. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 is shown in the table

E.2.2.4 Center at Class-Cluster Averages Tables E.14 and E.15 show the categorization performance for a Kernel Classifier network whose hidden layer weights were trained using the Center at Class-Cluster Averages algorithm. Table E.16 shows the number of layer 1 nodes created to cover the input data space for each run.

Table E.14. Center at Class Averages Training Performance vs Avg Threshold

Avg Thrshld	Run 0 %cret	Run 1 %cret	Run 2 %cret	Run 3 %cret	Run 4 %cret	Run 5 %cret	Run 6 %cret	Run 7 %cret	Run 8 %cret	Run 9 %cret	Avg %cret	Std %cret
0.25	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
0.50	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
0.75	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
1.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
1.25	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
1.50	99.02	99.02	98.04	98.04	98.04	100.00	99.02	100.00	100.00	100.00	99.12	0.81
1.75	98.04	96.08	96.08	96.08	97.06	96.08	97.06	98.04	99.02	100.00	97.35	1.32
2.00	96.08	95.10	94.12	94.12	93.14	95.10	93.14	96.08	97.06	100.00	95.39	1.96
2.25	91.18	91.18	89.22	89.22	94.18	92.16	93.14	95.10	96.08	90.20	92.17	2.28
2.50	90.02	92.16	84.30	84.30	87.25	93.14	94.12	91.20	90.20	89.22	89.49	3.11
2.75	92.16	91.18	76.47	76.47	87.35	92.16	84.31	82.35	84.31	88.24	86.48	6.09
3.00	86.27	87.25	74.51	74.51	84.31	87.25	83.33	77.45	85.29	83.33	82.45	4.77
3.25	87.25	87.25	62.75	62.75	80.39	75.50	69.61	66.67	77.45	81.37	76.47	8.91
3.50	86.27	70.58	60.78	60.78	75 49	[illegible]13	69.61	56.86	50.98	62.75	65.78	9.56
3.75	86.27	68.63	58.82	58.82	61.76	56.56	59.80	56.86	51.96	62.75	63.23	9.22
4.00	55.88	61.76	50.00	50.00	55.88	56.86	50.00	56.86	51.96	58.82	54.80	3.91

Table E.15. Center at Class Averages vs Avg Threshold

Avg Thrshld	Run 0 %cret	Run 1 %cret	Run 2 %cret	Run 3 %cret	Run 4 %cret	Run 5 %cret	Run 6 %cret	Run 7 %cret	Run 8 %cret	Run 9 %cret	Avg %cret	Std %cret
0.25	83.00	81.00	77.00	87.00	82.00	87.00	88.00	85.00	85.00	85.00	84.00	3.16
0.50	83.00	82.00	77.00	87.00	82.00	87.00	89.00	85.00	85.00	84.00	84.10	3.21
0.75	81.00	81.00	77.00	85.00	85.00	86.00	89.00	85.00	82.00	84.00	83.50	3.17
1.00	82.00	81.00	77.00	84.00	83.00	86.00	88.00	84.00	85.00	84.00	83.40	2.84
1.25	82.00	80.00	77.00	85.00	84.00	86.00	86.00	85.00	83.00	86.00	83.40	2.84
1.50	83.00	81.00	78.00	86.00	85.00	90.00	85.00	84.00	83.00	84.00	83.90	2.98
1.75	83.00	79.00	80.00	80.00	82.00	90.00	84.00	85.00	82.00	82.00	82.70	3.00
2.00	84.00	78.00	83.00	77.00	75.00	81.00	85.00	83.00	78.00	80.00	80.40	3.17
2.25	83.00	74.00	79.00	79.00	80.00	86.00	88.00	82.00	79.00	80.00	80.70	3.29
2.50	82.00	76.00	80.00	74.00	78.00	86.00	85.00	78.00	76.00	78.00	79.60	4.27
2.75	80.00	71.00	78.00	76.00	77.00	82.00	85.00	71.00	68.00	78.00	76.60	5.02
3.00	78.00	72.00	78.00	68.00	83.00	78.00	82.00	69.00	81.00	72.00	76.40	5.57
3.25	77.00	72.00	67.00	57.00	73.00	74.00	71.00	64.00	65.00	70.00	72.10	8.35
3.50	75.00	73.00	82.00	69.00	59.00	71.00	71.00	57.00	57.00	54.00	66.80	8.91
3.75	75.00	69.00	60.00	69.00	54.00	60.00	59.00	55.00	51.00	54.00	61.80	8.08
4.00	57.00	58.00	59.00	50.00	52.00	59.00	50.00	55.00	51.00	56.00	54.70	3.47

Table E.16. Nodes Generated for Center at Class Averages vs Avg Threshold

Average Threshold	Run 0 Nodes	Run 1 Nodes	Run 2 Nodes	Run 3 Nodes	Run 4 Nodes	Run 5 Nodes	Run 6 Nodes	Run 7 Nodes	Run 8 Nodes	Run 9 Nodes	Avg Nodes	Std Nodes
0.25	102.00	100.00	101.00	102.00	99.00	100.00	99.00	100.00	100.00	100.00	100.30	1.00
0.50	95.00	89.00	93.00	96.00	92.00	95.00	90.00	91.00	91.00	89.00	92.10	2.43
0.75	86.00	77.00	86.00	86.00	84.00	85.00	83.00	81.00	81.00	81.00	83.00	2.83
1.00	77.00	72.00	78.00	75.00	76.00	78.00	75.00	69.00	71.00	75.00	74.60	2.87
1.25	65.00	64.00	68.00	66.00	62.00	68.00	66.00	61.00	57.00	66.00	64.30	3.26
1.50	54.00	45.00	54.00	50.00	50.00	53.00	54.00	51.00	52.00	55.00	51.80	2.82
1.75	42.00	36.00	41.00	40.00	41.00	41.00	39.00	37.00	40.00	43.00	40.00	2.05
2.00	32.00	28.00	32.00	28.00	28.00	32.00	32.00	30.00	29.00	35.00	30.60	2.24
2.25	24.00	21.00	24.00	20.00	24.00	23.00	27.00	24.00	22.00	23.00	23.70	3.07
2.50	19.00	19.00	15.00	15.00	18.00	16.00	20.00	15.00	14.00	16.00	17.50	3.56
2.75	14.00	15.00	14.00	11.00	14.00	13.00	15.00	12.00	11.00	13.00	13.70	2.45
3.00	12.00	12.00	12.00	8.00	13.00	13.00	12.00	10.00	10.00	11.00	11.60	1.85
3.25	11.00	12.00	10.00	5.00	12.00	9.00	8.00	8.00	8.00	8.00	9.50	2.20
3.50	9.00	9.00	9.00	5.00	7.00	7.00	8.00	4.00	4.00	6.00	6.80	1.89
3.75	8.00	8.00	7.00	3.00	4.00	5.00	6.00	3.00	3.00	6.00	5.50	2.06
4.00	3.00	5.00	6.00	2.00	3.00	4.00	3.00	3.00	3.00	5.00	3.70	1.19

For the software parameters, the average threshold was allowed to vary, the data seed matched the run number, the sigma threshold was set at four and the training rule for the RBF sigma was

the Scale Sigma According to Class Interference with interference threshold of .4. The data seed matched the run number while 51 training vectors were loaded for each class. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 is shown in the table and the number of nodes in layer 2 was two. The transfer function for the nodes in layer 1 was gaussian while the transfer function for the nodes in layer 2 was linear. The weights linking layer 1 nodes to layer 2 nodes were set using the global MSE minimization algorithm.

E.2.2.5 PNN/RBF Comparison Tables E.17 and E.18 show the categorization performance of a PNN as the sigma varied. The weights in the hidden layer were trained using the Nodes at Data Points algorithm.

Table E.17. PNN Training Performance vs Sigma

Sigma	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Run 10 %crct	Avg %crct	Std %crct
0.25	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
0.50	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
0.75	94.12	97.06	97.06	96.08	96.08	95.10	97.06	94.12	95.10	96.08	95.79	1.08
1.00	88.24	85.29	90.20	91.18	89.22	91.18	87.25	87.25	86.27	83.33	87.94	2.45
1.25	80.39	76.47	75.49	77.45	78.43	79.42	77.45	72.55	77.45	72.55	76.77	2.48
1.50	67.65	62.75	68.63	62.75	69.61	60.78	60.78	60.78	70.59	61.76	64.61	3.81
1.75	56.86	55.88	57.89	58.82	60.78	54.91	51.96	53.92	59.80	57.84	56.87	2.59
2.00	50.98	50.00	53.92	51.96	56.86	52.94	50.98	50.00	54.10	53.92	52.57	2.08
2.25	50.98	50.00	50.98	50.98	54.91	50.00	50.00	50.00	51.96	50.00	50.98	1.46
2.50	50.00	50.00	50.00	50.98	52.94	50.00	50.00	50.00	50.00	50.00	50.39	0.90
2.75	50.00	50.00	50.00	50.00	52.94	50.00	50.00	50.00	50.00	50.00	50.29	0.88
3.00	50.00	50.00	50.00	50.00	50.98	50.00	50.00	50.00	50.00	50.00	50.10	0.29

Table E.18. PNN Test Performance vs Sigma

Sigma	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Run 10 %crct	Avg %crct	Std %crct
0.25	74.00	78.00	82.00	77.00	78.00	84.00	82.00	84.00	86.00	86.00	81.10	3.91
0.50	73.00	73.00	81.00	75.00	80.00	80.00	87.00	86.00	88.00	89.00	81.20	5.83
0.75	70.00	68.00	80.00	72.00	74.00	80.00	83.00	86.00	84.00	89.00	78.60	6.83
1.00	66.00	69.00	72.00	64.00	67.00	76.00	73.00	76.00	76.00	78.00	71.70	4.67
1.25	59.00	63.00	62.00	59.00	64.00	68.00	67.00	69.00	65.00	66.00	64.20	3.31
1.50	56.00	56.00	56.00	54.00	58.00	59.00	56.00	58.00	59.00	57.00	56.90	1.51
1.75	51.00	52.00	53.00	51.00	54.00	54.00	51.00	52.00	54.00	51.00	52.30	1.27
2.00	50.00	50.00	53.00	50.00	52.00	50.00	50.00	51.00	53.00	51.00	51.00	1.18
2.25	50.00	50.00	52.00	50.00	51.00	50.00	50.00	50.00	51.00	51.00	50.50	0.67
2.50	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	51.00	50.00	50.10	0.30
2.75	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	0.00
3.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	0.00

For the software parameters, the output threshold was set to one. The training vectors were loaded by class with 51 vectors from each class and a data seed of six. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 was 102 and the number of nodes in layer 2 was two. The transfer function for the nodes in layer 1 was gaussian while the transfer function for the nodes in layer 2 was linear. The weights linking layer 1 nodes to the layer 2 nodes were set using the PNN algorithm.

Tables E.19 and E.20 show the categorization performance of an RBF Kernel Classifier network as the sigma, or spread, varied. The weights in the hidden layer were trained using the Nodes

at Data Points algorithm.

Table E.19. RBF Network Training Performance vs Sigma

Sigma	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Run 10 %crct	Avg %crct	Std %crct
0.25	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
0.50	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
0.75	100.00	100.00	100.00	100.00	98.04	100.00	100.00	100.00	100.00	98.04	99.61	0.78
1.00	100.00	91.18	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	99.12	2.65
1.25	100.00	100.00	100.00	100.00	96.08	100.00	100.00	100.00	100.00	100.00	99.61	1.18
1.50	100.00	100.00	100.00	100.00	87.25	100.00	100.00	100.00	100.00	94.12	98.14	4.03
1.75	100.00	60.78	100.00	91.18	100.00	83.33	56.86	100.00	98.04	100.00	86.37	18.67
2.00	82.35	100.00	100.00	57.84	48.09	75.49	87.25	52.94	50.98	81.37	74.81	19.30
2.25	50.00	50.00	97.06	61.76	50.00	50.00	60.78	57.84	52.94	100.00	64.12	17.78
2.50	53.92	50.00	50.00	50.98	50.00	50.00	50.00	50.98	54.90	50.00	51.08	1.72
2.75	50.00	50.00	50.00	50.98	50.00	50.00	50.00	50.00	51.96	50.00	50.29	0.63
3.00	50.00	53.92	50.00	50.00	50.00	50.98	50.00	50.98	70.59	50.98	52.65	6.09

Table E.20. RBF Network Test Performance vs Sigma

Sigma	Run 1 %crct	Run 2 %crct	Run 3 %crct	Run 4 %crct	Run 5 %crct	Run 6 %crct	Run 7 %crct	Run 8 %crct	Run 9 %crct	Run 10 %crct	Avg %crct	Std %crct
0.25	74.00	72.00	82.00	77.00	79.00	84.00	84.00	85.00	86.00	86.00	80.90	4.85
0.50	78.00	73.00	82.00	79.00	80.00	84.00	88.00	85.00	88.00	88.00	82.50	4.78
0.75	82.00	73.00	85.00	79.00	76.00	87.00	88.00	86.00	86.00	91.00	83.30	5.40
1.00	83.00	70.00	89.00	80.00	80.00	89.00	87.00	82.00	85.00	89.00	83.40	5.61
1.25	82.00	74.00	90.00	80.00	77.00	87.00	88.00	81.00	84.00	87.00	83.00	4.88
1.50	82.00	74.00	90.00	79.00	69.00	87.00	90.00	81.00	85.00	84.00	82.10	6.39
1.75	79.00	47.00	89.00	67.00	80.00	69.00	53.00	83.00	82.00	87.00	73.60	13.57
2.00	70.00	71.00	89.00	51.00	49.00	69.00	79.00	50.00	50.00	87.00	63.50	13.39
2.25	50.00	50.00	89.00	61.00	50.00	50.00	58.00	44.00	52.00	86.00	59.00	14.94
2.50	50.00	45.00	50.00	52.00	50.00	50.00	48.00	53.00	57.00	59.00	51.40	3.90
2.75	50.00	49.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	50.00	49.90	0.30
3.00	50.00	53.00	50.00	50.00	50.00	57.00	47.00	52.00	64.00	45.00	51.80	5.10

For the software parameters, the output threshold was set at one. The training vectors were loaded by class with 51 vectors from each class and a data seed of six. The number of nodes in layer 0 was 50, while the number of nodes in layer 1 was 102 and the number of nodes in layer 2 was two. The transfer function for the nodes in layer 1 was gaussian while the transfer function for the nodes in layer 2 was linear. The weights linking layer 1 nodes to the layer 2 nodes were set using the global MSE minimization algorithm.

E.3 Radar System Characterization

This data consisted of 300 training and 1990, 6 dimension pattern vectors. These pattern vectors represented one of ten radar platforms. Only Kernel Classifier networks were developed to categorize this data. The networks consisted of a standard RBF network and an RBF-based arbitrator network.

E.3.1 RBF Network The topology for this network is shown in figure 3.7. Table E.21 shows the categorization performance of the RBF network trained to categorize the ten radar platforms. This network consisted of 6 nodes in layer 0, and 155 nodes in layer 1, and 10 nodes in layer 2. For the software parameters, the transfer function for each node in layer 0 was the identity transfer

Table E.21. RBF Network Performance

Classification Threshold	% Correct	
	Training	Test
.8	87.00	72.29
.6	89.67	77.91
.4	93.67	82.88
.2	93.67	82.88
.0	93.67	82.88

Table E.22. Network A Performance

Classification Threshold	% Correct	
	Training	Test
.8	93.33	81.40
.6	94.67	88.50
.4	95.00	90.05
.2	95.00	90.05
.0	95.00	90.05

function, while the transfer function for each node in layer 1 was gaussian and layer 2 was linear. The weights for the layer 1 nodes were set via the Center at Class-Cluster Averages algorithm with the average threshold set at .03. The spreads were set using the Scale Sigmas According to Class Interference algorithm with the interference threshold set at .5. The weights linking the layer 1 nodes to the layer 2 nodes were set via global minization of the MSE.

E.3.2 Arbitrator The topology for this network is shown in figure 5.15. Table E.22 shows the performance of Network A trained to categorize five radar platforms. This network consisted of 6 nodes in layer 0, and 175 nodes in layer 1, and 6 nodes in layer 2. For the software parameters, the transfer function for each node in layer 0 was the identity transfer function, while the transfer function for each node in layer 1 was gaussian and layer 2 was linear. The weights for the layer 1 nodes were set via the Center at Class-Cluster Averages algorithm with the average threshold set at .03. The spreads were set using the Scale Sigmas According to Class Interference algorithm with the interference threshold set at .5. The weights linking the layer 1 nodes to the layer 2 nodes were set via global minization of the MSE.

Table E.23 shows the performance of Network B trained to categorize five radar platforms. This network consisted of 6 nodes in layer 0, and 173 nodes in layer 1, and 5 nodes in layer 2. For the software parameters, the transfer function for each node in layer 0 was the identity transfer function, while the transfer function for each node in layer 1 was gaussian and layer 2 was linear.

Table E.23. Network B Performance

Classification Threshold	% Correct	
	Training	Test
.8	87.00	78.33
.6	95.67	88.31
.4	98.67	92.64
.2	98.67	92.64
.0	98.67	92.64

The weights for the layer 1 nodes were set via the Center at Class-Cluster Averages algorithm with the average threshold set at .01. The spreads were set using the P-Neighbors algorithm with the number of neighbors set at .5. The weights linking the layer 1 nodes to the layer 2 nodes were set via global minization of the MSE.

Table E.24 shows the performance of the total Arbitrator network trained to categorize the ten radar platforms. This network consisted of 6 nodes in layer 0 feeding networks A and B. The

Table E.24. Arbitration Network Performance

Classification Threshold	% Correct	
	Training	Test
.8	95.00	73.90
.6	98.33	81.43
.4	99.33	85.79
.2	99.33	86.35
.0	99.33	86.35

outputs of these networks were passed into Network C which consisted of 10 nodes in its input layer, 75 RBF nodes and 10 output nodes. For the software parameters, the transfer function for each RBF node was gaussian and each output node was linear. The weights for the RBF nodes were set via the Center at Class-Cluster Averages algorithm with the average threshold set at .03. The spreads were set using the Scale Sigmas According to Class Interference algorithm with the interference threshold set at .5. The weights linking the RBF nodes to the output nodes were set via global minization of the MSE.

Appendix F. *Software Analysis*

F.1 Introduction

The software to be described in this chapter was designed according to the object-oriented approach presented in Chapter 4. This chapter begins with a description of the data structures implemented for the software and concludes with a discussion of the software modules and the mapping of the training algorithms, developed in chapter 3, into software functions.

F.2 Object Oriented Structure

The structure of the software centered on the attributes of the nodes' weights, sigmas, connections, transfer function and class listed in Chapter 4.

F.2.1 Weights A given node's weights are defined as the factor by which an output signal from another node is multiplied before being processed by that node. For example, as shown in figure F.1, node[2]-weight[1] would be the weight which multiplies the output of node 1 prior to entering node 2 for processing. Conversely, node[1]-weight[2] would be the weight which multiplies the output of node 2 prior to entering node 1 for processing. Finally, node[1]-weight[1] would be the amplification factor for the output of a node[1] feeding back into node[1].

F.2.2 Sigmas A given node's sigmas are defined as the offset factor for a node with a sigmoidal transfer function or, conversely, the spread factor, in a particular direction, for the gaussian transfer function. This factor thus controls a node's response to a given input. For example, as shown in figure F.2, node[2]-sigma[1] would be the sigma factor for a signal passing from the output of node 1 to the input of node 2. Conversely, node[1]-sigma[2] would be the sigma factor for a signal passing from the output of node 2 to the input of node 1. Finally, node-record[1]-sigma[1] would be internal sigma for node 1.

F.2.3 Connect A node's connections to other nodes in the network are defined as the connection by which an output signal from another node is allowed to pass to that node. For example, as shown in figure F.3, node[2]-connect[1] would be set to a 1 if node 2 received output from node 1. Conversely, if node 2 did not receive node 1's output, node[2]-connect[1] would be set to a 0. Similarly, node[1]-connect[2] would be 1 if node 1 received node 2's output and 0 if node 1 did not receive node 2's output. Finally, node[1]-connect[1] will be set to a 1 if node 1 to received input from itself.

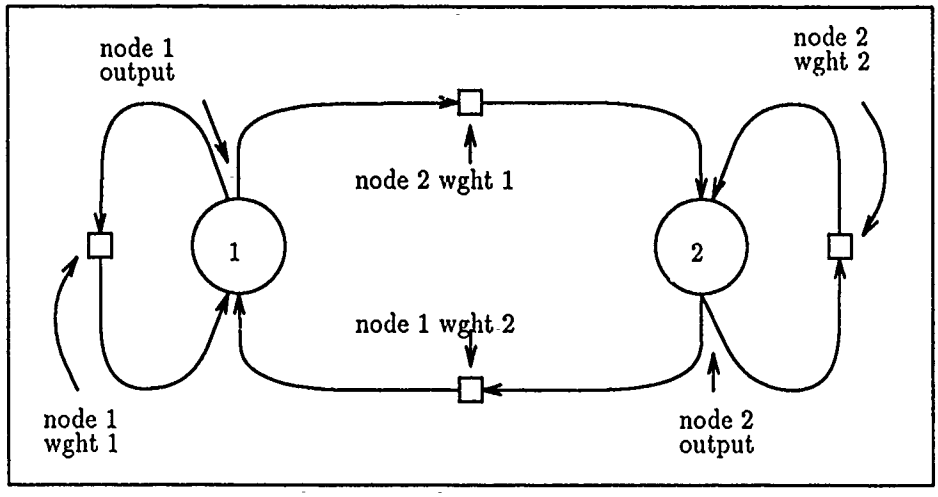

Figure F.1. Node Weight Structure

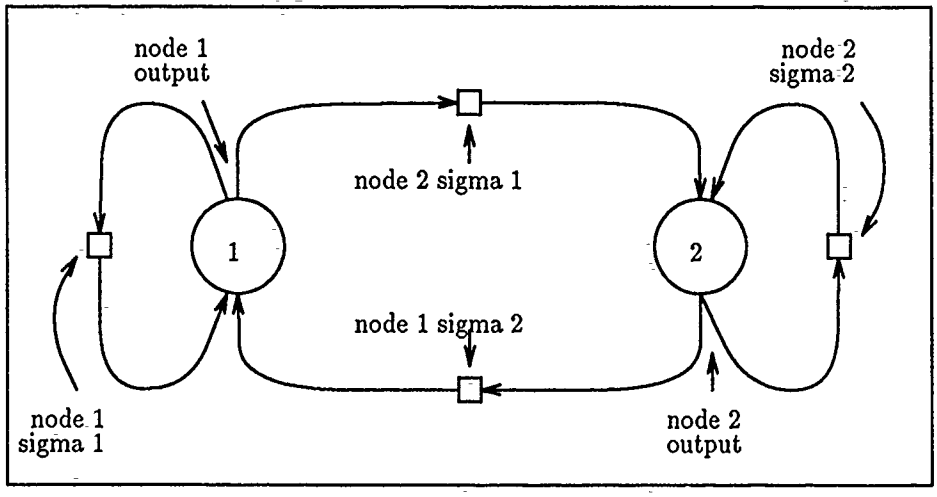

Figure F.2. Node Sigma Structure

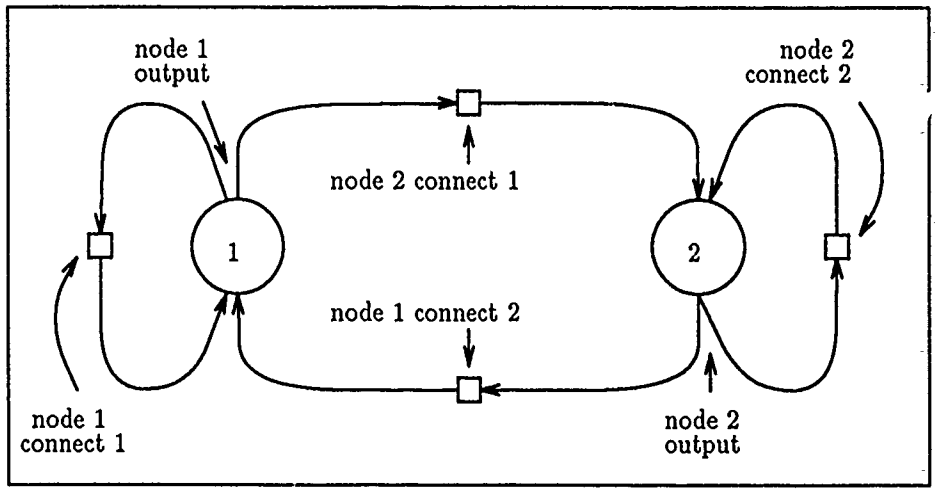

Figure F.3. Node Connection Structure

F.2.4 Transfer Function A given nodes output will be a function of the inputs, weights, and sigmas connecting the node to the other nodes. Letting $\bar{x}$, be the input vector for a given node, $\bar{w}$ the weight vector, and $\bar{\sigma}$ be the offset vector then the possible transfer functions will be the linear transfer function of the form:

$$f(\bar{x}) = \bar{x}\bar{w} \tag{F.1}$$

or a sigmoidal function of the form:

$$f(\bar{x}) = [1 + e^{-(\bar{x}\cdot\bar{w}+\sigma_i)}]^{-1} \tag{F.2}$$

or a gaussian function of the form:

$$f(\bar{x}) = e^{-\frac{\|x-w\|^2}{2\sigma^2}} \tag{F.3}$$

or the identity function of the form:

$$f(x_i) = x_i \tag{F.4}$$

However, any other applicable transfer function could also be assigned.

F.2.5 Class A given node can be assigned a class to which the node is responsible for responding. This is applicable to nodes whose transfer function is the gaussian function as well as to output nodes.

F.3 Software Analysis

The software modules implemented in this theses are described in detail in the following sections. The code for these modules is found in Appendix G.

F.3.1 NETMENU This module is the overall controlling module of the network. It provides the user interface to the software via the SUN terminal and keyboard and calls the appropriate modules to execute the network decision. NETMENU allows the selection of the type of network to be configured, the number of layers and nodes in the network, and the training rule of for the weights of each node. Currently, the allowed training rules for the weights in layer 1 are to train the weights to match the features, center the weights at class averages, establish the weights via the K-means algorithm, establish the weights via Kohonen training, or train the weights via backpropagation using the MSE, CE and CFE algorithms. Allowed training rules for the weights in layer 2 are matrix inversion and backpropagation using the MSE, CE and CFE algorithms and the PNN Implementation. Allowed training rules for the weights in layer 3 are backpropagation using the MSE, CE and CFE algorithms.

Once the training rules are established, NETMENU will allow the transfer function, for the nodes in each layer, to be selected. At this time, the only valid transfer functions are the identity, the gaussian, the sigmoidal, and the linear transfer functions. Also, each node in the same layer will be assigned the same transfer function.

If any of the layers are assigned the gaussian transfer function, NETMENU allows the selection of a training rule for the thresholds or sigmas. At this time, the thresholds may be trained by setting them to a constant, scaling the sigmas based on class interference, and setting the sigmas equal to the P-Neighbor average distances.

Once these parameters have been established, NETMENU will then preset the weights, sigmas, or outputs for the network, if desired, or randomly initialize the weights and sigmas. Any preset data will be read from input data files specifying the appropriate reference number for the node and the preset value.

After reading in both the training and test data files, NETMENU will then configure the network by establishing the appropriate nodal connections and assigning the nodes their appropriate

transfer function.

From the selected training rule, NETMENU will then call the appropriate training function, in NETTRAIN, to train the weights in each of the layers. Once the network has been fully trained, NETMENU then establishes the accuracy of the network by calling NETERROR to apply both the test and training data to the network, calculate the network output, and perform a percentage calculation. This percentage calculation is made for the training data, test data and total data. A correct response occurs when the appropriate node produces an output, above some user predefined threshold, which is greater than all the other output nodes.

F.3.2 NETERROR This module contains the functions necessary to determine the networks classification of a data vector and the error performance of the network.

F.3.2.1 Test The Network This function takes an unknown data vector from the calling module, applies the data vector to the network and calculates the network output. If their largest output for any node is below a predefined threshold, the unknown pattern will be assigned a class of zero to indicate an unclassified pattern. If the largest output is above the threshold, the unknown pattern will be assigned the same classification as that of the winning node. For classifications in which the network class does not match the input data class, an error count is updated.

F.3.2.2 Determine Class as Largest This function classifies the input data vector as belonging to one of the possible output classes providing the highest output is above some user selected threshold. This allows the user to determine the point at which the network is allowed to consider an input data pattern is classified.

F.3.2.3 Update Errors This function increments an error count for each time the network classification does not match the proper classification of the data.

F.3.3 NETTRAIN NETTRAIN contains the submodules necessary to establish the network weights via the following training procedures:

a) Global MSE Minimization

b) Make Nodes at Data Points

c) Center Weights at Class Averages

d) K-means Cluster

e) Train via Kohonen

f) MSE Remaining Layers

g) CE Remaining Layers

h) CFM Remaining Layers

i) PNN Last Layer

j) Scale Sigma by Class Interference

k) Set Sigmas to Constant

l) Set Sigma at P Neighbor Avg

Each of these submodules accomplishes its training routine by executing the specialized functions contained in NETAUX.

F.3.3.1 Global MSE Minimization Global MSE Minimization performs the optimization of the weights linking the radial basis function nodes in the hidden layer to the nodes in the output layer using the equations established in section 3.5. This submodule first calls the function "Determine Y Matrix". This function applies each data record in the training set and calculates the output for each of the L nodes in layer 1. That is, the output for each of the nodes due to the p^{th} pattern will be

$$y_{out} = [y_{p1}, y_{p2}, y_{p3}, \cdots, y_{pL}] \tag{F.5}$$

This process is accomplished for each of the P patterns in the training set to produce the matrix:

$$Y = \begin{vmatrix} y_{11} & y_{12} & \cdots & y_{1L} \\ y_{21} & y_{22} & \cdots & y_{2L} \\ \vdots & \vdots & \vdots & \vdots \\ y_{P1} & y_{P2} & \cdots & y_{PL} \end{vmatrix} \tag{F.6}$$

Global MSE Minimization then computes the S matrix by calling the function "Determine S Matrix" to determine the required outputs for each of the M nodes in the output layer due to an input pattern. This function tests the classification assigned to a data record. If the class of the data record p is N, then the output of the $(N-1)^{th}$ node in the output layer is assigned a value of 1 while all other nodes are assigned a value of zero:

$$s_p = [d_{p1}, d_{p2}, \ldots, d_{pN-1}, d_{pN}, d_{pN+1}, \ldots, d_{pM}] \tag{F.7}$$

which translates to

$$s_p = [0, 0, \ldots, 1, 0, 0, \ldots, 0] \tag{F.8}$$

This process is accomplished for each of the P records in the training set until a P by M matrix is established where each p^{th} row contains the desired output for the last layer nodes due to p^{th} input pattern.

$$S = \begin{vmatrix} d_{11} & d_{12} & \ldots & d_{1M} \\ d_{21} & d_{22} & \ldots & d_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ d_{P1} & d_{P2} & \ldots & d_{PL} \end{vmatrix} \tag{F.9}$$

"Global MSE Minimization" then calls the function "Determine M Matrix" to compute the summation of the product of the B^{th} output of a node in the first layer with the l^{th} output of a node in the first layer over all P patterns. That is

$M_{lB} = \sum_{p=1}^{P} y_{pl} y_{pB}$ as defined in equation C.61. Assuming there are L nodes in the layer, the M matrix then becomes:

$$M = \begin{vmatrix} M_{11} & M_{12} & \ldots & M_{1L} \\ M_{21} & M_{22} & \ldots & M_{2L} \\ \vdots & \vdots & \vdots & \vdots \\ M_{L1} & M_{L2} & \ldots & M_{LL} \end{vmatrix} \tag{F.10}$$

After executing the function "Make Identity Matrix" to form an L by L identity matrix as:

$$N = \begin{vmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 \end{vmatrix} \tag{F.11}$$

"Global MSE Minimization" then calls the function "Determine Matrix Transpose" which takes the M matrix and returns its transpose back as a new M matrix. "Global MSE Minimization" then uses the function "Invert a matrix" to invert the transpose of M. This inverted matrix is returned to as the N matrix. "Global MSE Minimization" then computes the optimal value for each weight by calling the function "Caiculate Weight Matrix". This function uses the values in the Y, S and inverted M matrix to perform the optimal weight linking node B in layer 1 to node D in layer 2, w_{BD}, as:

$$w_{BD} = \sum_{l=1}^{L}(\sum_{p=1}^{P} y_{pl} d_{pD}) N_{Bl} \tag{F.12}$$

This calculation is made for each weight linking a node in layer 1 to a node in layer 2.

F.3.3.2 Make Nodes at Data Points This module sets the weights of the nodes in the first layer equal to the features of the input patterns. This module begins by applying a data record to the network. "Nodes at Data Points" then calls the function "Calculate Layer 1 Output" to determine the output for each node in layer 1. "Make Nodes at Data Points" then compares the output of each node in layer 1 to the predefined threshold T_{max} and the class of each node in layer 1 with the class of the input pattern. If the output for a node is greater than T_{max} and the class for that node is the same as that of the input pattern, no new node is added. If not, a new node is added whose weights match the input pattern features. This process continues for each training vector.

F.3.3.3 Center Weights at Class Averages This module incrementally adapts the weights of the nodes in layer 1 to the running average of the data records lying within a threshold T_{max} of the current center. This module begins by applying a training vector to the network. "Center Weights at Class Averages" then calculates the distance from all the nodes in layer 1 to that training vector by the equation:

$$d_l = \sqrt{\sum_{k=1}^{K}(x_k - w_{lk})^2} \tag{F.13}$$

If this distance for a node less than the T_{max}, "Center Weights at Class Averages" calls the function "Update Average" to update the current weights of that node by the equation:

$$w_{kl}^{+} = w_{kl}^{-} + \frac{x_k - w_{kl}^{-}}{N+1} \tag{F.14}$$

If the distance between a training vector and all the nodes in layer 1 is greater than T_{max}, "Center Weights as Class Averages" will create a new node and assign the weights of the new node the features of the training vector. This process will continue until all records in the training set have been processed. "Center Weights at Class Averages" will repeat this process of cycling through the training data and updating and creating new nodes until no new nodes are created. At this point, the entire feature space is now covered by the nodes in layer 1.

F.3.3.4 K-Means Cluster This module adapts the weights of the K nodes in layer 1 to be the centers of K clusters. K-Means Cluster begins by the setting the weights of the K cluster nodes in layer 1 equal to the features of the first K vectors. "K-means cluster" then applies a training vector to the network and calls the function "Find Nearest Neighbor". This function computes the distance from each input record to each of the K nodes in layer 1 by the equation

$$d_l = \sum_{k=1}^{K} (x_k - w_{kl})^2 \tag{F.15}$$

This function returns the reference number of the node in layer 1 with the closest distance to the input vector. "K-Means Cluster" then assigns the input vector to the cluster of the closest node in layer 1. This process of inputting a training vector, finding the node whose weights are the closest to that particular training vector, and assigning this training vector to the cluster whose node has the closest weights is repeated until all training records have been processed. For each cluster node in layer 1, "K-means cluster" then takes the training vectors assigned to that node's cluster and repeatedly calls the function "Update Average". This function updates each weight to a new average by the equation

$$w_{kl}^{+} = w_{kl}^{-} + \frac{x_k - w_{kl}^{-}}{N+1} \tag{F.16}$$

After all training patterns assigned to the cluster of a node have been processed, each of the node's weights will contain the average of the features of the pattern vectors assigned to that cluster. After this adjustment, "K-means cluster" tests the difference between each of the cluster's old weights and new weights via the equation

$$\delta = \mid w_{kl}^{-} - w_{kl}^{+} \mid \tag{F.17}$$

If the difference is not less than some small value ϵ, for all nodes' weights, the clustering algorithm

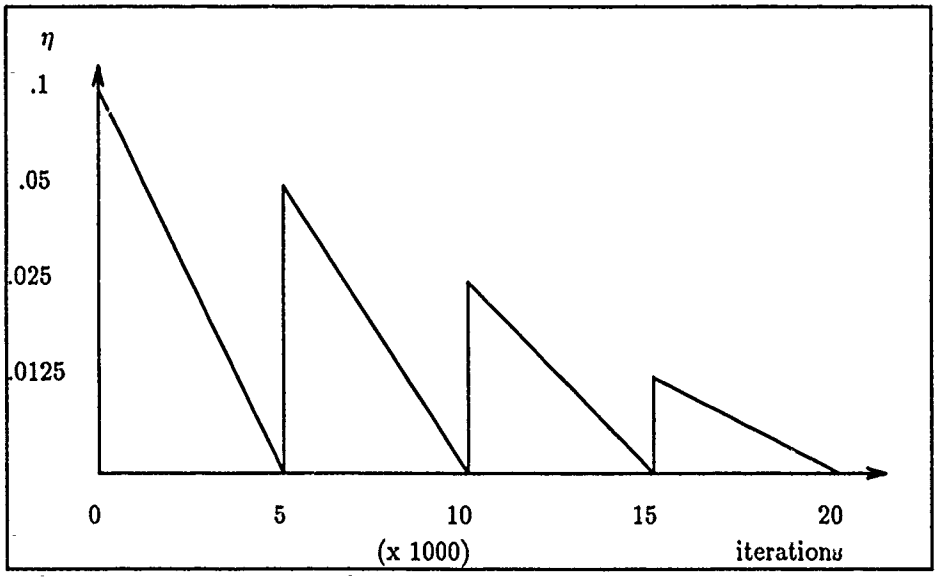

Figure F.4. Kohonen Training Eta Adaption

has not converged and this process of applying a training vector, finding the closest cluster node, and updating the weights of the cluster nodes to be the average of the input pattern vectors assigned to the cluster will then be repeated. If the difference is less than ϵ, the algorithm has converged.

F.3.3.5 Train Via Kohonen This module adapts the weights in the first layer by using the Kohonen algorithm defined in section 3.5. "Train Via Kohonen" first calls the function "Get Random Class Record" to return a training vector from the training set with an equal probability according to a uniform probability distribution. "Train Via Kohonen" then applies the data record to the network and calls the function "Calculate Distance From Outputs To Next Layer". Assuming there are K nodes in layer 0 and J nodes in layer 1, this function performs the calculation

$$d_j = \sum_{k=1}^{K} (x_k - w_{kj})^2 \tag{F.18}$$

These distance values are stored and returned to "Train Via Kohonen" which then calls the function "Find Nearest Element" to return the number of the node which had its weights "closest" to the features of the input pattern. The module "Train Via Kohonen" then calls the function "Get Linear Training Eta" to return the learning factor based on the saw tooth function as shown in figure F.4.

$$\eta = \frac{\eta_{max}}{i_0 - i_{max}}(i - i_0) + \eta_{max} \tag{F.19}$$

Here

η = the learning constant.

η_{max} = the maximum η which occurs at the beginning of the interval.

i_0 = the iteration number at the beginning of the interval.

i_{max} = the iteration number at the end of the interval.

i = the current iteration number.

Once the η is returned "Train Via Kohonen" then executes the function "Get Kohonen Neighborhood". This function returns the neighborhood number of the nodes which are in the neighborhood of the winning node. The neighborhood is determined by the iteration number as follows:

neighborhood = 7 for $0 < iterations < 5000$

neighborhood = 5 for $5000 < iterations < 10000$

neighborhood = 3 for $10000 < iterations < 15000$

neighborhood = 1 for $15000 < iterations < 20000$

"Train Via Kohonen" then calls function "Find Kohonen Boundaries" to check the kohonen layer boundaries to ensure that the length of the neighborhood does not exceeded the boundaries. "Train Via Kohonen" will determine which nodes will be updated by invoking the function "Determine Neighborhood Elements" and update each of these node weights by executing the function "Train Kohonen Weights". This function updates each of the required nodes by the equation

$$w_i^+ = w_i^- + \eta(x_i - w_i^-) \tag{F.20}$$

"Train Via Kohonen" repeats this functional execution for the selected number of iterations.

F.3.3.6 MSE Remaining Layers This module performs the backpropagation of the error to adjust the network parameters for the weights and sigmas for each node in the network. "MSE Remaining Layers" first calls the function "Get Random Class Record" to obtain a random training pattern from one of the classes according to uniform probability distribution. "MSE Remaining Layers" then invokes "Feed Forward Network Output" to calculate the network output.

After the network output is calculated, "MSE Remaining Layers" determines the difference between the desired network output and the actual network output, or error, by calling the function "Calculate Errors In Output". This function compares the output of each node in the output layer to their desired output for that pattern. This comparison is done by the following equation:

$$| y_m - d_m | > t_{max} \tag{F.21}$$

Here t_{max} is the maximum threshold, currently set to .9, y_m is the output for node m in the output layer and d_m is the desired output for node m in the output layer. If all the output nodes meet this criteria no weights are updated and this process is repeated for another random record. If any of the output nodes don't meet this criteria, "MSE Remaining Layers" will then update the remaining network layers parameters via backpropagation according to the MSE algorithm. If the transfer function for the last layer nodes are sigmoidal, their weights and sigma will be updated by the equation

$$w_{MN}^{+} = w_{MN}^{-} + \frac{\eta}{N}(d_N - y_N)y_N(1 - y_N)(y_M) \tag{F.22}$$

and

$$\sigma_N^{+} = \sigma_N^{-} + \frac{\eta}{N}(d_N - y_N)y_N(1 - y_N) \tag{F.23}$$

If the transfer function is the linear transfer function, the node weights and threshold in the last layer will be updated via the equations

$$w_{MN}^{+} = w_{MN}^{-} + \frac{\eta}{N}(d_N - y_N)(y_M) \tag{F.24}$$

"MSE Remaining Layers" will then call the function "MSE Mid Layer" to update the weights and sigmas for the nodes in the next to the last layer of the network via the equations

$$w_{LM}^{+} = w_{LM}^{-} + \frac{\eta}{N}\sum_{n=1}^{N}(d_n - y_n)y_n(1 - y_n)w_{Mn}y_M(1 - y_M)y_L \tag{F.25}$$

and

$$\sigma_M^+ = \sigma_M^- + \frac{\eta}{N}\sum_{n=1}^{N}(d_n - y_n)y_n(1-y_n)w_{Mn}y_M(1-y_M) \qquad \text{(F.26)}$$

After the weights for the next to last layer have been updated, "MSE Remaining Layer" will then call function "MSE 1st Layer", if required, to update the weights of the first layer. Currently, for a three layer network, all nodes must have the sigmoidal transfer function. Therefore "MSE 1st Layer" will update the node weights and thresholds via the equations

$$w_{KL}^+ = w_{KL}^- + \frac{\eta}{N}\sum_{n=1}^{N}(d_n - y_n)y_n(1-y_n)[\sum_{m=1}^{M} w_{mn}y_m(1-y_m)w_{Lm}y_L(1-y_L)y_K] \qquad \text{(F.27)}$$

and

$$\sigma_L^+ = \sigma_L^- + \frac{\eta}{N}\sum_{n=1}^{N}(d_n - y_n)y_n(1-y_n)[\sum_{m=1}^{M} w_{mn}y_m(1-y_m)w_{Lm}y_L(1-y_L)] \qquad \text{(F.28)}$$

After the first layer parameters have been updated "MSE Remaining Layers" will continue this backpropagation until the required number of iterations are achieved.

F.3.3.7 CE Remaining Layers This module performs the backpropagation of the error according to the CE algorithm to adjust the network parameters for the weights and sigmas for each node in the network. "CE Remaining Layers" first calls the function "Get Random Class Record" to obtain a random training pattern from one of the classes according to a uniform probability distribution. "CE Remaining Layers" then invokes "Feed Forward Network Output" to calculate the network output. After the network output is calculated, "CE Remaining Layers" determines the difference between the desired network output and the actual network output, or error, by calling the function "Calculate Errors In Output". This function compares the output of each node in the output layer to their desired output for that pattern. This comparison is done by the following equation.

$$| y_m - d_m | > t_{max} \qquad \text{(F.29)}$$

Here t_{max} is the maximum threshold, currently set to .9, y_m is the output for node m in the output layer and d_m is the desired output for node m in the output layer. If all the output nodes meet this

criteria no weights are updated and this process is repeated for another random record. If any of the output nodes don't meet this criteria, "CE Remaining Layers" will then update the remaining network layers parameters via backpropagation according to the CE algorithm. "CE Last Layer" is called first to update the nodes in the last layer by the equation

$$w_{MN}^{+} = w_{MN}^{-} + \frac{\eta}{2.3N}(d_N - y_N)(y_M) \tag{F.30}$$

and

$$\sigma_N^{+} = \sigma_N^{-} + \frac{\eta}{2.3N}(d_N - y_N) \tag{F.31}$$

The factor of 1/2.3 is due to the multiplicative factor $1/\ln(10)$ from the derivative of the CFE objective function. "CE Remaining Layers" will then call the function "CE Mid Layer" to update the weights and sigmas for the nodes in the next to the last layer of the network. "CE Mid Layer" will update the weights via the equation

$$w_{LM}^{+} = w_{LM}^{-} + \frac{\eta}{2.3N}\sum_{n=1}^{N}(d_n - y_n)w_{Mn}y_M(1 - y_M)y_L \tag{F.32}$$

and

$$\sigma_M^{+} = \sigma_M^{-} + \frac{\eta}{2.3N}\sum_{n=1}^{N}(d_n - y_n)w_{Mn}y_M(1 - y_M) \tag{F.33}$$

After the weights for the next to last layer have been updated, "CE Remaining Layers" will then call function "CE Last Layer", if required, to update the weights of the first layer. This function updates the via the equations

$$w_{KL}^{+} = w_{KL}^{-} + \frac{\eta}{2.3N}\sum_{n=1}^{N}(d_n - y_n)[\sum_{m=1}^{M} w_{mn}y_m(1 - y_m)w_{Lm}y_L(1 - y_L)y_K] \tag{F.34}$$

and

$$\sigma_L^{+} = \sigma_L^{-} + \frac{\eta}{2.3N}\sum_{n=1}^{N}(d_n - y_n)[\sum_{m=1}^{M} w_{mn}y_m(1 - y_m)w_{Lm}y_L(1 - y_L)] \tag{F.35}$$

After the first layer parameters have been updated "CE Remaining Layers" will continue this backpropagation until the required number of iterations are achieved.

F.3.3.8 CFM Remaining Layers This module performs backpropagation according to the CFM algorithm to adjust the network parameters for the weights and sigmas for each node in the network. "CFM Remaining Layers" first calls the function "Get Random Class Record" to obtain a random training pattern from one of the classes according to a uniform probability distribution. "CFM Remaining Layers" then invokes "Feed Forward Network Output" to calculate the network output. After the network output is calculated, "CFM Remaining Layers" determines the node in the output layer with the largest output by calling the function "Determine Network Class". This function assigns the network's classification as the largest value of the nodes in the output layer by the equation

$$class = y_{t_{max}} \tag{F.36}$$

where t_{max} = the output threshold. If the network's class is not the class of the training vector or if the difference between the correct node's output and the next highest node's output is less than the user predefined difference, the network nodes are updated according to the CFM algorithm. "CFM Remaining Layers" will then update the remaining network layers parameters via backpropagation according to the CFM algorithm. "CFM Last Layer" is called first to update incorrect node in the last layer by the equatio:

$$w_{MN}^{+} = w_{MN}^{-} - \frac{\eta\beta\alpha}{N-1} z_N(1-z_N)y_N(1-y_N)(y_M) \tag{F.37}$$

and

$$\sigma_N^{+} = \sigma_N^{-} - \frac{\eta\beta\alpha}{N-1} z_N(1-z_N)y_N(1-y_N) \tag{F.38}$$

while the correct node's weights and threshold are updated according to the equation

$$w_{MC}^{+} = w_{MC}^{-} + \frac{\eta\beta\alpha}{N-1} \sum_{n=1}^{N} z_n(1-z_n)y_C(1-y_C)(y_M) \tag{F.39}$$

and

$$\sigma_C^+ = w_C^- + \frac{\eta\beta\alpha}{N-1}\sum_{n=1}^{N} z_n(1-z_n)y_C(1-y_C) \tag{F.40}$$

"CFM Remaining Layers" will then call the function "CFM Mid Layer" to update the weights and sigmas for the nodes in the next to the last layer of the network. "CFM Mid Layer" will update the weights and threshold via the equations

$$w_{LM}^+ = w_{LM}^- + \frac{\eta\beta\alpha}{N-1}\sum_{n=1}^{N} z_n(1-z_n)[y_C(1-y_C)w_{MC} - y_n(1-y_n)w_{Mn}]y_M(1-y_M)y_L \tag{F.41}$$

and

$$\sigma_M^+ = \sigma_M^- + \frac{\eta\beta\alpha}{N-1}\sum_{n=1}^{N} z_n(1-z_n)[y_C(1-y_C)w_{MC} - y_n(1-y_n)w_{Mn}]y_M(1-y_M) \tag{F.42}$$

After the weights for the next to last layer have been updated, "CFM Remaining Layers" will then call function "CFM First Layer", if required, to update the weights of the first layer. This function updates the weights and thresholds via the equations

$$\begin{aligned} w_{KL}^+ \;=\; & w_{KL}^- + \frac{\eta\beta\alpha}{N-1}\sum_{n=1}^{N} z_n(1-z_n)[y_C(1-y_C)\sum_{m=1}^{M} w_{mC} \\ & - y_n(1-y_n)\sum_{m=1}^{M} w_{mn}]y_m(1-y_m)y_L(1-y_L)y_K \end{aligned} \tag{F.43}$$

and

$$\begin{aligned} \sigma_L^+ \;=\; & \sigma_L^- + \frac{\eta\beta\alpha}{N-1}\sum_{n=1}^{N} z_n(1-z_n)[y_C(1-y_C)\sum_{m=1}^{M} w_{mC} \\ & - y_n(1-y_n)\sum_{m=1}^{M} w_{mn}]y_m(1-y_m)y_L(1-y_L) \end{aligned} \tag{F.44}$$

After the first layer parameters have been updated "CFM Remaining Layers" will continue this backpropagation until the required number of iterations are achieved.

F.3.3.9 Scale Sigmas by Class Interference This module scales the sigmas, for nodes in layer 1 containing radial basis functions, if required, by a constant. This module begins by applying the training vectors to the network and calling the function "Calculate Layer 1 Output" to calculate the outputs from the nodes in layer 1. "Scale Sigmas by Class Interference" then compares the output of each node in layer 1 with the threshold T_{max}. If the output for a node is greater than T_{max} and that node is not assigned the same class as that of the input pattern, that nodes sigma is scaled by the equation:

$$\sigma^{+} = \sigma^{-}(1 - Constant) \tag{F.45}$$

"Scale Sigmas by Class Interference" will continue to reduce the σ for that node until the output is less then T_{max}. This process is repeated for all the training vectors in the training set.

F.3.3.10 Set Sigma at P Neighbor Avg This module sets the sigma for each node in the first layer equal to the root mean square of the distance between that node and its P nearest neighbors. This module begins by calling the function "Find Distance Between Nodes". This function calculates the distance between the nodes in layer 1 by the equation:

$$d_{ij} = \sqrt{\sum_{k=1}^{K}(w_{ki} - w_{kl})^2} \tag{F.46}$$

After the distances between all the nodes have been calculated, "Set Sigma at P Neighbor Average" will then call the function "Sort 2 Dimensional Array" to find the P shortest distances for each node. "Set Sigma at P Neighbor Avg" will then calculate the sigma for each node by the equation:

$$\sigma_l = \sqrt{\frac{1}{P}\sum_{p=1}^{P} d_{lp}^2} \tag{F.47}$$

F.3.3.11 Set Sigmas to Constant This module sets the sigma for each node in layer 1 to a predefined constant C by the following equation:

$$\sigma_l = Constant \tag{F.48}$$

F.3.4 NETINPUT This module contains the functions necessary to load the training and test data files. This data may be loaded from a separate training and test file or a single file. This method of loading the data is controlled by the function "load input patterns".

F.3.4.1 Load Input Patterns This function allows the user to control how the test and training data is to be loaded. Currently, the function "Load Separate Files" allows the user to load the training and test data from separate files either in sequence or randomly. The function "Load From Single File" allows the user to load the training and test data from a single file either in sequence of randomly. The function "Load By Classes" allows the user to control the number of training vectors assigned to each class. This is important if the number of training vectors must reflect the a priori probability of the input data. These data vectors will be randomly selected from a single file of input data.

F.3.4.2 Get Data This function loads the input data into the training and test data structures. The training and test data structures contain an array of the features of the input patterns, the class of the pattern, and the sequence number in the input file.

F.3.4.3 Normalize Data This function allows the user to normalize the input data via the equation:

$$x_i = \frac{x_i}{\|\bar{x}\|} \tag{F.49}$$

F.3.4.4 Get Weights, Sigmas, Classes, and Outputs These functions load initial weights, sigmas, classes and outputs from an input file. This is important for applications in which the initial conditions of the network are known a priori.

F.3.5 NETINIT This module contains the functions which allocate memory for the nodes and data records, correct node weights and connections, and initialize the node weights, sigmas, transfer functions and network connections.

F.3.5.1 Initialize Node Weights This function initializes the node weights between each connected node in the network to a value between 0 and 1. For nodes not connected, the weights are set to 0.

F.3.5.2 Initialize Node Connections This function connects the nodes in the network as defined by the network type. Currently, the only network type allowed is the feedforward network. For this network, the nodes in layer 0 will receive inputs from no other nodes and there connections to all other nodes will be assigned 0. The nodes in layer 1 will only be connected to and receive inputs from the nodes in layer 0. The nodes in layer 2 will only be connected to and

receive inputs from the nodes in layer 1. The nodes in layer 3 will only be connected to and receive inputs from the nodes in layer 2.

F.3.5.3 Initialize Node Sigmas This function initializes the sigma between each node in the network to a value between 0 and 1;

F.3.5.4 Initialize Node Outputs This function initializes the outputs for each node in the network to a value of 0.

F.3.5.5 Initialize Node Transfer Functions This function initializes the transfer functions, as defined by the user, for each node in the network.

F.3.5.6 Create Node This function allocates enough memory to hold the "Node Data" data structures for each node in the network.

F.3.5.7 Create Data Record This function allocates enough memory to hold the "Data" data structures for each training and test input data vector.

F.3.5.8 Disconnect Node This function will disconnect any nodes in the network from all other nodes if the node is not being used. This is important for applications in which the number of nodes in the network is not known apriori and pruned as a result of network learning.

F.3.5.9 Correct Node Weights This function limits the range of a node's weights to a value between -100 and 100. This is important to prevent backpropagation training algorithms from saturating the transfer functions during training.

F.3.6 NETSHOW This module contains the output functions necessary to display and file the performance and parameters of the network.

F.3.6.1 Print/File Network Output These functions show the current output for the final layer of nodes in the network.

F.3.6.2 Print/File Data Parameters These functions show the name of the training and test data files, the number of data vectors in each file, and the dimension of the data vectors.

F.3.6.3 Print/File Randomization Rule These functions show the randomization rule selected to load the input data. If the randomization rule is to "Load By Class", this function will also display the number of training patterns selected from each class.

F.3.6.4 Print/File Seeds These functions show the initial weight, sigma, data, and record seeds used to set the randomization rules for the network parameters. The weight and sigma seeds control the initialization of the network weights and thresholds. The data seed controls the randomization of the input data vectors while the record seed controls the presentation of the data vectors for the backpropagation algorithms.

F.3.6.5 Print/File Net Topology These functions show the current topology of the network, including the type of network, the number of layers in the network, and the number of nodes in each layer. This is important in the monitoring of the network topology for networks in which the number of nodes is adapted according to the data parameters.

F.3.6.6 Print/File Transfer Functions These functions show the transfer functions for each layer of nodes in the network.

F.3.6.7 Print/File Node at Data Points Data These functions show the number of nodes, output threshold, and the initial sigmas for the nodes in layer 1 when trained using this training rule.

F.3.6.8 Print/File Center at Class-Cluster Averages Data These functions show the initial parameters of the maximum number of initial nodes, the average threshold and sigma threshold used to train the nodes in layer 1 using this training rule.

F.3.6.9 Print/File K Means Data These functions show the number of clusters used to establish the node 1 weights when the K-Means algorithm is used to train the weights for the layer 1 nodes.

F.3.6.10 Print/File Kohenen Data These functions show the parameters for the number of nodes in x and y direction, the number of training iterations, the neighborhoods, and scaling factors used to train a layer of nodes using the Kohonen algorithm.

F.3.6.11 Print/File MSE Data These functions show the parameters for the number of iterations, the learning factor, and the momentum factor used to train a layer of nodes using the

backpropagation algorithm according to the MSE objective function.

F.3.6.12 Print/File CFM Data These functions show the parameters for the number of iterations, the learning factor, the momentum factor, the amplification factor, the offset factor and the lateral shift factor used to train a layer of nodes using the backpropagation algorithm according to the CFM objective function.

F.3.6.13 Print/File CE Data These functions show the parameters for the number of iterations, the learning factor, and the momentum factor used to train a layer of nodes using the backpropagation algorithm according to the CFM objective function.

F.3.6.14 Print/File Sigma Data These functions show the training rules and parameters of the sigma threshold, sigma constant, and number of nearest neighbors used to train the spreads of thresholds of nodes having the gaussian transfer function.

F.3.7 NETOUT This module contains the functions necessary to compute the outputs for each node in the network, the outputs for each layer of a feedforward network, and the output for the entire network due to a given input pattern.

F.3.7.1 Calculate Feedforward Network Output This function calculates the network output due to a given input vector by calling the applicable output functions for each layer of the network.

F.3.7.2 Calculate Layer 0 Output This function establishes the range of nodes in layer 0 and calls "Calculate Node Output" to calculate the output for each node in layer 0. This output will depend on the transfer function assigned to each node in layer 0.

F.3.7.3 Calculate Layer 1 Output This function establishes the range of nodes in layer 1 and calls "Calculate Node Output" to calculate the output for each node in layer 1. This output will depend on the transfer function assigned to each node in layer 1.

F.3.7.4 Calculate Layer 2 Output This function establishes the range of nodes in layer 2 and calls "Calculate Node Output" to calculate the output for each node in layer 2. This output will depend on the transfer function assigned to each node in layer 2.

F.3.7.5 Calculate Layer 3 Output This function establishes the range of nodes in layer 3 and calls "Calculate Node Output" to calculate the output for each node in layer 3. This output will depend on the transfer function assigned to each node in layer 3.

F.3.7.6 Calculate Node Output This module computes the output for each node in the network as determined by the transfer function and the other nodes from which the node receives input. Currently, a node's transfer function can either be the sigmoidal transfer function of

$$y_{out} = [1 + e^{-\sum_{l=1}^{L} w_l y_l(in) + \sigma}]^{-1} \tag{F.50}$$

or the gaussian radial basis function of

$$y_{out} = e^{-\sum_{l=1}^{L} [y_l(in) - w_l]^2} \tag{F.51}$$

or the linear transfer function of

$$y_{out} = \sum_{l=1}^{L} w_l y_l(in) \tag{F.52}$$

or the identity transfer function in which

$$y_{out} = y_{in} \tag{F.53}$$

F.3.8 NETAUX This module contains the functions called by NETTRAIN to set the network parameters. Below is the list of functions maintained in this module.

1) Determine Y Matrix

2) Determine S Matrix

3) Determine M Matrix

4) Calculate Weight Matrix

5) MSE Last Layer

6) MSE Last Layer Linear

7) MSE Last Layer Sigmoid

8) MSE Mid Layer

9) MSE 1st Layer

10) Calculate Errors in Output

11) Get Linear Training Eta

12) Get Kohonen Neighborhood

13) Calc Dist Outputs to Nxt Lyr

14) Find Nearest Element

15) Find Kohonen Weights

16) Determine Neighborhood Elements

17) Train Kohonen Weights

18) Find Distance Between Nodes

19) Sort 2 Dim Array

20) CE Last Layer

21) CE Mid Layer

22) CE First Layer

23) Calculate Zn

24) CFM Last Layer

25) CFM Mid Layer

26) Find Second Highest Node

27) CFM First Layer

28) Find Nearest Neighbor

For a detailed discussion of these functions, see NETTRAIN or Appendix G.

F.3.9 NETMATH This module contains the mathematical functions used by the various training algorithms within the module NETTRAIN.

F.3.9.1 Make Identity Matrix This function returns a square identity matrix to the calling routine. The number of rows and columns in the matrix and the address of the matrix must be passed to this function for proper execution.

F.3.9.2 Determine Matrix Transpose This function returns the transpose of a matrix. The calling function must provide the addresses of the matrix to be transposed and the matrix transpose, along with the number of rows and columns of the matrix.

F.3.9.3 Invert A Matrix This function returns the inverse of a square matrix. The calling function must provide the addresses of the matrix to be inverted and the final inverted matrix along with the number of rows in the matrix. The matrix inversion is completed via gaussian elimination.

F.3.9.4 Update Average This function computes the running average of a series of data points. The calling function must provide the current average, the next data point to be incorporated into the average and the number of data points within the average. The update equation is as follows:

$$\bar{u}_i^+ = \bar{u}_i^- + \frac{1}{N}(x_i - \bar{u}_i^-) \tag{F.54}$$

F.3.9.5 Update Sigma This function computes the running standard deviation of a series of data points. The calling function must provide the current average, the next data point to be incorporated into the standard deviation, the current standard deviation and the number of data points in the calculation. The update equation is as follows:

$$\sigma_+ = \sqrt{\sigma_-^2 + \frac{1}{N}((1 - \frac{1}{N})(x_i - \bar{u})^2 - \sigma_-^2)} \tag{F.55}$$

F.3.9.6 Calculate Percentage This function computes the percentage a ratio of numbers. The calling function must provide the numerator and the denominator of the numbers.

F.3.9.7 Get Random Class Record This function randomly selects, according to a uniform probability distribution, one of the possible classes and randomly returns a data vector from the selected class.

F.3.9.8 Get Random Record This function randomly selects, according to a uniform probability distribution, a data vector from the set of training data.

Appendix G. *Software Code*

G.1 NETMENUE

```
/*****************************************************************/
/* Module Name: NETMENUE     Number:1                            */
/* Description: This module is the overall controlling module of */
/*              of the software.  It provides the user interface */
/*              to the software via a SUN workstation terminal   */
/*              and keyboard.  This module calls the appropriate */
/*              modules to execute user desisions.               */
/* Modules Called: NETERROR, NETTRAIN, NETINPUT, NETINIT, NETSHOW */
/* Functions Contained: None                                     */
/* Date: 11 Nov 90     Revision: 1.0                             */
/*****************************************************************/

#include "netvrble.h"
#include "netfnctn.h"

main()
{
     FILE *fptr, *train_ptr, *test_ptr, *user_ptr;
     FILE *MSE_ptr, *CE_ptr, *CFM_ptr;
     struct data *training_data[TRAIN_SET];
     struct data *test_data[TEST_SET];
     struct Node_data *Node_record[TOTAL_NODES];
     float output_threshold = 1;
     float average_threshold = 2;
     float sigma_threshold = 4;
     float sigma_factor = .5;
     float interference_threshold = .8;
     float MSE_error_delta = .2;
     float class_threshold =0;
     float per_cent_correct = 0;
     float sigma_constant = 2;
     float MSE_eta = .3;
     float CFM_alpha = 1;
     float CFM_beta = 4;
     float CFM_eta = 3;
     float CFM_zeta = 0;
     float CFM_momentum = .1;
     float CFM_delta = .4;
     float CE_epsilon = .1;
     float CE_iterations = 20000;
     float CE_momentum = .1;
     float CE_eta = 2.76;
     float MSE_momentum = .1;
     static int neighborhoods[6] = {7,5,3,1};
     static int train_width[6] = {0, 5000, 10000, 15000, 20000};
     static float train_scale[6] = {.1, .05, .025, .0125};
     int misclassified[200];
     int train_error = 0;
     int test_error = 0;
     int total_error = 0;
     int correct_class = 0;
     int network_class = 0;
     int width_no = 5;
     int nodes_x = 10;
     int nodes_y = 0;
     int p_neighbors = 0;
     int train_set = TRAIN_SET;
     int test_set = TEST_SET;
     int classes = CLASSES;
```

```
int dimension = DIMENSION;
int node;
int record, row, x, y, layer, starting_node_in_layer[4];
int number_of_layers;
int nodes_in_layer[4], training_rule[4], transfer_function[4];
int network_type = 0;
int total_nodes = 0;
int preset = 0;
int sigma_rule = 0;
int int_buffer = 0;
int total_iterations = 0;
int current_node = 0;
int MSE_iterations = 0;
int kohonen_iterations = 0;
int nodes_1_max = 0;
int nodes_2_max = 0;
int nodes_3_max = 0;
int error = 0;
int CFM_successes = 100;
int CFM_iterations = 20000;
int CE_successes = 100;
int randomization_rule = 0;
int training_patterns_in_class[CLASSES];
int MSE_successes = 100;
int class = 0;
int find_the_distance = 0;
int normalize_the_data = 0;
unsigned data_seed = 0;
unsigned sigma_seed = 0;
unsigned wght_seed = 0;
unsigned record_seed = 0;

/*****************TEST STUFF ********************************************/

static char train_file[] = "class2.in";
static char test_file[] = "class2.in";
static char output_file[] = "nodes_test.out";
static char selection_file[] = "nodes_test.sel";
static char MSE_file[] = "MSE_data.out";
static char CFM_file[] = "beta4data";
static char CE_file[] = "CE_data_final";

normalize_the_data = 0;                    /* 1 = yes */
find_the_distance = 0;                        /* 1 = yes */

dimension = 50;
train_set = 102;
test_set = 100;
classes = 2;                               /* Randomization Rule        */
randomization_rule = 3;                    /* 1 - load separate files   */
                                           /* 2 - load from single file */
                                           /* 3 - load by class         */
training_patterns_in_class[1] = 51;
training_patterns_in_class[2] = 51;
training_patterns_in_class[3] = 0;
training_patterns_in_class[4] = 0;
training_patterns_in_class[5] = 0;
training_patterns_in_class[6] = 0;
training_patterns_in_class[7] = 0;
training_patterns_in_class[8] = 0;
training_patterns_in_class[9] = 0;
training_patterns_in_class[10] = 0;

wght_seed = 0;  sigma_seed = 0; data_seed = 1; record_seed = 1;
```

```
  network_type = 1; number_of_layers = 2;

  nodes_in_layer[0]=dimension;
  nodes_in_layer[1]=60;
  nodes_in_layer[2]=2;
  nodes_in_layer[3]=0;

  training_rule[0]=0;

  training_rule[1]=1;
/* 1-nodes at data points   2-center class average 3-  K-means */
/* sig-thres, out-thres     avg-thresh sigthresh   sig rule 3or4 */

/* 4-kohonen   5-MSE backprop     6-CFM backprop  7-CE backprop  */
/*   nodes_x      MSE stuff          CFM stuff       CE stuff    */

  training_rule[2] = 1;

/* 1 - matrix invert  2 - MSE backprop 3-CFM backprop 4-CE backprop */
/* 5 - Parzen window        MSE stuff      CFM stuff      CE stuff  */

  training_rule[3] = 0;

/* 1 - MSE backprop  2 -CFM backprop  3-CE backprp 4-Parzen window */
/*     MSE stuff          CFM stuff      CE stuff                  */

  sigma_threshold = 4;            kohonen_iterations = 20000;
  output_threshold = 1;           nodes_x = 7;
  average_threshold = 2;

  MSE_iterations = 30000;  CFM_alpha = 1.0;       CE_epsilon = .05;
  MSE_error_delta = .1;    CFM_beta = 4.0;        CE_iterations = 30000;
  MSE_momentum = .1;       CFM_eta = .14;          CE_momentum = .05;
  MSE_eta = .3;            CFM_zeta = 0;          CE_eta = 1.75;
  MSE_successes = 100;     CFM_successes = 100;   CE_successes = 15000;
                           CFM_iterations = 150000;
                           CFM_momentum = .1;
                           CFM_delta = 1.0;


  transfer_function[0] = 0;     /* 1- sigmoidal */
  transfer_function[1] = 2;     /* 2 -rbf       */
  transfer_function[2] = 3;     /* 3- linear    */
  transfer_function[3] = 0;

  sigma_rule = 1;              /* Sigma rules  1 - scale by constant    */
                               /*  interference_threshold = .8;         */
                               /*          sigma_factor = .5;           */
  interference_threshold = .4;
  sigma_factor = .1;           /*     2 - half nearest neighbor         */
  sigma_constant = .5;
  p_neighbors = 6;             /*     3 - constant                      */
                               /*          sigma_constant = 2;          */

                               /*     4 - p neighbr average             */
                               /*            p_neighbors = 4;          */

/*************************************************************************/

  for (x = 0; x < train_set; x++)
  {
     create_data_record(training_data,
                        x,
                        &error);
     if(error != 0)
```

```
   {
      printf("\n***** out of memory for training_data *******\n");
      exit();
   }
 }

for (x = 0; x < test_set; x++)
{
   create_data_record(test_data,
                      x,
                      &error);
   if(error != 0)
   {
      printf("\n**** out of memory for test data *****\n");
      exit();
   }
 }


 train_ptr = fopen(train_file,"r");
 test_ptr = fopen(test_file,"r");

 load_input_patterns (training_data,
                      test_data,
                      train_set,
                      test_set,
                      dimension,
                      classes,
                      training_patterns_in_class,
                      randomization_rule,
                      data_seed,
                      train_ptr,
                      test_ptr);
  fclose(train_ptr);
  fclose(test_ptr);

if(normalize_the_data == 1)
{
  normalize_data(training_data,
                 train_set,
                 dimension);

  normalize_data(test_data,
                 test_set,
                 dimension);
}

if(find_the_distance == 1)
{

  fptr = fopen("nrmtrain.dat","w");
  fprintf(fptr,"\n normalized Distances for Training data");
  calculate_euclidean_distance_between_inputs(training_data,
                                              train_set,
                                              dimension,
                                              fptr);

  fclose(fptr);

  fptr = fopen("nrmtest.dat","w");
  fprintf(fptr,"\n normalized Distances for Test data");
  calculate_euclidean_distance_between_inputs(test_data,
                                              test_set,
                                              dimension,
                                              fptr);
```

```
    fclose(fptr);
  }

user_ptr = fopen(selection_file,"w");

nodes_1_max = nodes_in_layer[1];
nodes_2_max = nodes_in_layer[2];
nodes_3_max = nodes_in_layer[3];

total_nodes = nodes_in_layer[0];
starting_node_in_layer[0] = 0;

for (layer = 1; layer < number_of_layers +1; layer++)
{
   starting_node_in_layer[layer] = starting_node_in_layer[layer-1]
                                 + nodes_in_layer[layer-1];
   total_nodes = total_nodes + nodes_in_layer[layer];
}

error = 0;
for (node = 0; node < total_nodes; node++)
{
  create_node (Node_record,
               node,
               &error);

  if (error != 0)
  {
     printf("\nout of memory");
     exit();
  }
}


  initialize_node_connections(Node_record,
                              number_of_layers,
                              nodes_in_layer,
                              starting_node_in_layer,
                              network_type,
                              total_nodes);

  initialize_node_weights(Node_record,
                          total_nodes,
                          wght_seed);

  initialize_node_sigmas(Node_record,
                         total_nodes,
                         sigma_seed);

  initialize_node_outputs(Node_record,
                          total_nodes);

  initialize_node_transfer_function(Node_record,
                                    number_of_layers,
                                    nodes_in_layer,
                                    starting_node_in_layer,
                                    transfer_function);


  file_data_parameters(train_file,
            test_file,
               train_set,
               test_set,
               dimension,
               classes,
```

```
            user_ptr);

 file_randomization_rule(randomization_rule,
                         training_patterns_in_class,
                         classes,
                         user_ptr);

  file_seeds(wght_seed,
           sigma_seed,
           data_seed,
           record_seed,
           user_ptr);

  fprintf(user_ptr,"\nstarting network topology");
  file_net_topology(network_type,
                    number_of_layers,
                    nodes_in_layer,
                    user_ptr);

  file_transfer_functions(network_type,
                          number_of_layers,
                          starting_node_in_layer,
                          Node_record,
                          user_ptr);



for (layer = 1; layer < number_of_layers +1; layer++)
{
  if(layer == 1)
  {
    switch (training_rule[layer])
    {
      case 1:
        make_nodes_at_data_points (training_data,
                                   Node_record,
                                   train_set,
                                   nodes_in_layer,
                                   sigma_threshold,
                                   output_threshold,
                                   starting_node_in_layer,
                                   total_nodes,
                                   nodes_1_max);

          file_nodes_at_data_points_info(layer,
                                 output_threshold,
                                 sigma_threshold,
                                 user_ptr);

          break;

        case 2:
            center_weights_at_class_averages(training_data,
                                             Node_record,
                                             train_set,
                                             nodes_in_layer,
                                             average_threshold,
                                             sigma_threshold,
                                             starting_node_in_layer,
                                             total_nodes,
                                             layer,
                                             nodes_1_max);

       file_center_at_class_avgs_data(layer,
```

```
                        average_threshold,
                        sigma_threshold,
                        user_ptr);


        break;

    case 3:
        k_means_cluster(training_data,
                        Node_record,
                        train_set,
                        nodes_in_layer[1],
                        nodes_in_layer,
                        starting_node_in_layer,
                        layer);

  file_k_means_data(layer,
              nodes_in_layer[1],
             user_ptr);

        break;

    case 4:
      nodes_y = nodes_in_layer[1]/nodes_x;

      train_via_kohonen(training_data,
                        Node_record,
                        nodes_in_layer,
                        starting_node_in_layer,
                        neighborhoods,
                        train_width,
                        train_scale,
                        width_no,
                        nodes_x,
                        nodes_y,
                        layer,
                        train_set,
                        kohonen_iterations,
                        record_seed);

    file_kohonen_data(layer,
              nodes_x,
              nodes_y,
              user_ptr);


      break;

case 5:
    MSE_ptr = fopen(MSE_file,"w");

    MSE_remaining_layers(Node_record,
                         training_data,
                         test_data,
                         transfer_function,
                         nodes_in_layer,
                         starting_node_in_layer,
                         number_of_layers,
                         layer,
                         train_set,
                         test_set,
                         MSE_eta,
                         total_nodes,
                         MSE_successes,
                         MSE_error_delta,
```

```
                        MSE_iterations,
                        MSE_momentum,
                        classes,
                        record_seed,
                        MSE_ptr);

    fclose(MSE_ptr);

    file_MSE_data(layer,
                  MSE_iterations,
                  MSE_error_delta,
                  MSE_momentum,
                  MSE_successes,
                  MSE_eta,
                  user_ptr);

     layer = number_of_layers;

 break;

case 6:
   CFM_ptr = fopen(CFM_file,"w");
   CFM_remaining_layers(Node_record,
                        training_data,
                        test_data,
                        nodes_in_layer,
                        starting_node_in_layer,
                        number_of_layers,
                        layer,
                        train_set,
                        test_set,
                        CFM_eta,
                        total_nodes,
                        CFM_successes,
                        CFM_iterations,
                        CFM_alpha,
                        CFM_beta,
                        CFM_zeta,
                        CFM_momentum,
                        CFM_delta,
                        classes,
                        record_seed,
                        CFM_ptr);

   fclose(CFM_ptr);

 file_CFM_data(layer,
         CFM_alpha,
         CFM_beta,
         CFM_eta,
         CFM_zeta,
         CFM_successes,
         CFM_iterations,
         CFM_momentum,
         CFM_delta,
         user_ptr);

     layer = number_of_layers;

 break;

case 7:
  CE_ptr = fopen(CE_file,"w");

  CE_remaining_layers (Node_record,
```

```
                        training_data,
                        test_data,
                        nodes_in_layer,
                        starting_node_in_layer,
                        number_of_layers,
                        layer,
                        train_set,
                        test_set,
                        CE_eta,
                        total_nodes,
                        CE_successes,
                        CE_epsilon,
                        CE_iterations,
                        CE_momentum,
                        classes,
                        record_seed,
                        CE_ptr);

      fclose(CE_ptr);

      file_CE_data(layer,
               CE_epsilon,
               CE_iterations,
               CE_momentum,
               CE_eta,
               CE_successes,
               user_ptr);

       layer = number_of_layers;

     break;

    default:
        break;

   }
    if (nodes_in_layer[1] < nodes_1_max)
    {
       int_buffer = nodes_1_max - nodes_in_layer[layer];
       for (x = 0; x < int_buffer; x++)
       {
          current_node = starting_node_in_layer[layer]
                       + nodes_in_layer[layer] + x;
          disconnect_node(Node_record,
                          current_node,
                          total_nodes);
       }
    }

if(Node_record[starting_node_in_layer[1]]->transfer_function == 2)
{
 file_sigma_data(layer,
                 sigma_rule,
                 interference_threshold,
                 sigma_factor,
                 sigma_constant,
                 p_neighbors,
                 user_ptr);

  if(sigma_rule == 1)
  {
      scale_sigmas_by_class_interference(training_data,
                         Node_record,
                         train_set,
                         nodes_in_layer,
```

```
                        starting_node_in_layer,
                        total_nodes,
                        layer,
                        interference_threshold,
                        sigma_factor);
      }

      else if(sigma_rule == 2)
      {
         printf("\n error in sigma rule selection");
      exit();
       }
       else if(sigma_rule == 3)
       {
            set_sigmas_to_constant(Node_record,
                                   nodes_in_layer,
                                   starting_node_in_layer,
                                   layer,
                                   sigma_constant);
       }

       else if(sigma_rule == 4)
       {
        set_sigma_at_P_neighbor_avg(Node_record,
                                    nodes_in_layer,
                                    starting_node_in_layer,
                                    layer,
                                    total_nodes,
                                    p_neighbors);

       }
    }
  }
  else if (layer == 2)
  {
    switch (training_rule[layer])
    {
      case 1:
            global_MSE_minimization(training_data,
                                    Node_record,
                                    train_set,
                                    nodes_in_layer,
                                    starting_node_in_layer,
                                    total_nodes,
                                    layer);


       file_matrix_data(layer,
                        user_ptr);

       break;
       case 2:
          MSE_ptr = fopen(MSE_file,"w");
            MSE_remaining_layers(Node_record,
                                 training_data,
                                 test_data,
                                 transfer_function,
                                 nodes_in_layer,
                                 starting_node_in_layer,
                                 number_of_layers,
                                 layer,
                                 train_set,
                                 test_set,
                                 MSE_eta,
                                 total_nodes,
```

```
                                MSE_successes,
                                MSE_error_delta,
                                MSE_iterations,
                                MSE_momentum,
                                classes,
                                record_seed,
                                MSE_ptr);
        fclose(MSE_ptr);

    file_MSE_data(layer,
                  MSE_iterations,
                  MSE_error_delta,
                  MSE_momentum,
                  MSE_successes,
                  MSE_eta,
                  user_ptr);

    layer = number_of_layers;

    break;

case 3:
      CFM_ptr = fopen(CFM_file,"w");

      CFM_remaining_layers(Node_record,
                           training_data,
                           test_data,
                           nodes_in_layer,
                           starting_node_in_layer,
                           number_of_layers,
                           layer,
                           train_set,
                           test_set,
                           CFM_eta,
                           total_nodes,
                           CFM_successes,
                           CFM_iterations,
                           CFM_alpha,
                           CFM_beta,
                           CFM_zeta,
                           CFM_momentum,
                           CFM_delta,
                           classes,
                           record_seed,
                           CFM_ptr);

    fclose(CFM_ptr);

    file_CFM_data(layer,
                  CFM_alpha,
                  CFM_beta,
                  CFM_eta,
                  CFM_zeta,
                  CFM_successes,
                  CFM_iterations,
                  CFM_momentum,
                  CFM_delta,
                  user_ptr);

     layer = number_of_layers;

    break;

   case 4:
     CE_ptr = fopen(CE_file,"w");
```

```
    CE_remaining_layers (Node_record,
                         training_data,
                         test_data,
                         nodes_in_layer,
                         starting_node_in_layer,
                         number_of_layers,
                         layer,
                         train_set,
                         test_set,
                         CE_eta,
                         total_nodes,
                         CE_successes,
                         CE_epsilon,
                         CE_iterations,
                         CE_momentum,
                         classes,
                         record_seed,
                         CE_ptr);

    fclose(CE_ptr);

    file_CE_data(layer,
                 CE_epsilon,
                 CE_iterations,
                 CE_momentum,
                 CE_eta,
                 CE_successes,
                 user_ptr);

     layer = number_of_layers;

   break;


   case 5:
         PNN_last_layer(Node_record,
                                nodes_in_layer,
                                starting_node_in_layer,
                                layer);

        file_parzen_window_data (Node_record,
                                 nodes_in_layer,
                                 starting_node_in_layer,
                                 layer,
                                 user_ptr);
         break;


   default:
         break;
   }
}
else if(layer == 3)
{
 switch (training_rule[layer])
 {
   case 1:
      MSE_ptr = fopen(MSE_file,"w");

      MSE_remaining_layers(Node_record,
                           training_data,
                           test_data,
                           transfer_function,
                           nodes_in_layer,
```

```
                        starting_node_in_layer,
                        number_of_layers,
                        layer,
                        train_set,
                        test_set,
                        MSE_eta,
                        total_nodes,
                        MSE_successes,
                        MSE_error_delta,
                        MSE_iterations,
                        MSE_momentum,
                        classes,
                        record_seed,
                        MSE_ptr);

   fclose(MSE_ptr);

   file_MSE_data(layer,
                 MSE_iterations,
                 MSE_error_delta,
                 MSE_momentum,
                 MSE_successes,
                 MSE_eta,
                 user_ptr);

   layer = number_of_layers;
   break;

case 2:
   CFM_ptr = fopen(CFM_file,"w");

   CFM_remaining_layers(Node_record,
                        training_data,
                        test_data,
                        nodes_in_layer,
                        starting_node_in_layer,
                        number_of_layers,
                        layer,
                        train_set,
                        test_set,
                        CFM_eta,
                        total_nodes,
                        CFM_successes,
                        CFM_iterations,
                        CFM_alpha,
                        CFM_beta,
                        CFM_zeta,
                        CFM_momentum,
                        CFM_delta,
                        classes,
                        record_seed,
                        CFM_ptr);

fclose(CFM_ptr);

file_CFM_data(layer,
              CFM_alpha,
              CFM_beta,
              CFM_eta,
              CFM_zeta,
              CFM_successes,
              CFM_iterations,
              CFM_momentum,
              CFM_delta,
              user_ptr);
```

```
            layer = number_of_layers;

          break;

         case 3:
           CE_ptr = fopen(CE_file,"w");

           CE_remaining_layers (Node_record,
                                training_data,
                                test_data,
                                nodes_in_layer,
                                starting_node_in_layer,
                                number_of_layers,
                                layer,
                                train_set,
                                test_set,
                                CE_eta,
                                total_nodes,
                                CE_successes,
                                CE_epsilon,
                                CE_iterations,
                                CE_momentum,
                                classes,
                                record_seed,
                                CE_ptr);

            fclose(CE_ptr);

           file_CE_data(layer,
                        CE_epsilon,
                        CE_iterations,
                        CE_momentum,
                        CE_eta,
                        CE_successes,
                        user_ptr);

            layer = number_of_layers;

          break;

      case 4:
            PNN_last_layer(Node_record,
                                      nodes_in_layer,
                                      starting_node_in_layer,
                                      layer);

            file_parzen_window_data (Node_record,
                                     nodes_in_layer,
                                     starting_node_in_layer,
                                     layer,
                                     user_ptr);
            break;

          default:
            break;
        }
      }
}

fprintf(user_ptr,"\n Final topology");
file_net_topology(network_type,
                  number_of_layers,
                  nodes_in_layer,
                  user_ptr);
```

```
fptr = fopen(output_file,"w");
file_network_parameters(Node_record,
                        network_type,
                        number_of_layers,
                        nodes_in_layer,
                        training_rule,
                        transfer_function,
                        sigma_rule,
                        total_nodes,
                        fptr);


 for (node = 0; node < total_nodes; node++)
   file_node_data(Node_record,
                  node,
                  total_nodes,
                  fptr);

for (x = 0; x < classes+1; x++)
  training_patterns_in_class[x] = 0;

train_error = 0;
for (x = 0; x < train_set; x++)
{
  class = training_data[x]->class;
  training_patterns_in_class[class] +=1;

  test_the_network(training_data,
                   Node_record,
                   nodes_in_layer,
                   starting_node_in_layer,
                   number_of_layers,
                   x,
                   total_nodes,
                   class_threshold,
                   misclassified,
                   &train_error);

  file_last_layer_output(training_data,
                         Node_record,
                         x,
                         nodes_in_layer,
                         starting_node_in_layer,
                         number_of_layers,
                         fptr);

}
correct_class = train_set - train_error;
calculate_percentage((float)correct_class,
                     (float)train_set,
                     &per_cent_correct);

fprintf(user_ptr,"\ntraining data");
file_error_data(train_error,
                per_cent_correct,
                misclassified,
                user_ptr);


file_class_count(training_patterns_in_class,
                 classes,
                 user_ptr);
```

```
  for (x = 0; x < classes+1; x++)
    training_patterns_in_class[x] = 0;

  test_error = 0;
  for (x = 0; x < test_set; x++)
  {
    class = test_data[x]->class;
    training_patterns_in_class[class] += 1;

    test_the_network(test_data,
                     Node_record,
                     nodes_in_layer,
                     starting_node_in_layer,
                     number_of_layers,
                     x,
                     total_nodes,
                     class_threshold,
                     misclassified,
                     &test_error);

    file_last_layer_output(test_data,
                           Node_record,
                           x,
                           nodes_in_layer,
                           starting_node_in_layer,
                           number_of_layers,
                           fptr);

  }
  correct_class = test_set - test_error;
  calculate_percentage((float)correct_class,
                       (float)test_set,
                       &per_cent_correct);

  fprintf(user_ptr,"\ntest data");
  file_error_data(test_error,
                  per_cent_correct,
                  misclassified,
                  user_ptr);


  file_class_count (training_patterns_in_class,
                    classes,
                    user_ptr);

   total_error = train_error + test_error;
   correct_class = train_set + test_set - total_error;

   calculate_percentage((float)correct_class,
                        (float)(test_set+train_set),
                        &per_cent_correct);

   fprintf(fptr,"\ntotal per cent correct = %f",per_cent_correct);
   fprintf(user_ptr,"\n total per cent correct = %f",per_cent_correct);

fclose(fptr);

for (node = 0; node < total_nodes; node++)
  free(*Node_record[node]);
fclose(user_ptr);
}
```

G.2 NETERROR

```
/*********************************************************************/
/* Module Name: NETERROR.C              Number: 2.0                  */
/* Description: This module contains the functions which provide     */
/*              error accounting of the network performance.         */
/*                                                                   */
/* Modules Called: NETOUT                                            */
/* Functions Contained: 2.1 test_the_network                         */
/*                      2.2 determine_class_as_largest               */
/*                      2.3 update_errors                            */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

#include "netvrble.h"
#include "netfnctn.h"

/*********************************************************************/
/* Function Name: test_the_network                    Number: 2.1    */
/* Description: This function calculates the errors from a           */
/*              feed forward network due to an input record          */
/*                                                                   */
/* Functions Called: 7.1 calculate_feed_forware_network_output       */
/*                   2.2 determine_class_as_largest                  */
/*                   2.3 update_errors                               */
/*                                                                   */
/* Variables Passed In: training or test_data - Structure array      */
/*                      Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer             */
/*                      number_of_layers - Integer                   */
/*                      total_nodes - Integer                        */
/*                      class_threshold - Float                      */
/*                      misclassified - Integer array                */
/*                      *error - Integer pointer                     */
/*                                                                   */
/* Variables Returned: *error - Integer pointer                      */
/*                     misclassifier - Integer array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void test_the_network (struct data *data_record[],
                       struct Node_data *node_record[],
                       int nodes_in_layer[],
                       int starting_node_in_layer[],
                       int number_of_layers,
                       int record,
                       int total_nodes,
                       float class_threshold,
                       int misclassified[],
                       int *error)

{
   int network_class = 0;
   int error_buffer;
   error_buffer = *error;

   calculate_feed_forward_network_output(data_record,
                                         node_record,
                                         number_of_layers,
                                         nodes_in_layer,
                                         starting_node_in_layer,
                                         record,
```

```
                                        total_nodes);

   determine_class_as_largest(node_record,
                              nodes_in_layer,
                              starting_node_in_layer,
                              &network_class,
                              number_of_layers,
                              class_threshold);

    update_errors(data_record,
                  record,
                  network_class,
                  misclassified,
                  &error_buffer);

    *error = error_buffer;
}

/*********************************************************************/
/* Function Name: determine_class_as_largest    Number: 2.2          */
/* Description: This function determines the class as the largest    */
/*              output node if the output is above some level        */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      *network_class - Integer pointer             */
/*                      last_layer - Integer                         */
/*                      class_threshold - Float                      */
/*                                                                   */
/* Variables Returned: *network_class - Integer pointer              */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void determine_class_as_largest(struct Node_data *node_record[],
                                int nodes_in_layer[],
                                int starting_node_in_layer[],
                                int *network_class,
                                int last_layer,
                                float class_threshold)

{
  int x, last_layer_node;
  float largest = 0;
  for (x = 0; x < nodes_in_layer[last_layer]; x++)
  {
     last_layer_node = starting_node_in_layer[last_layer] + x;
     if ((node_record[last_layer_node]->output > largest)
         && (node_record[last_layer_node]->output > class_threshold))
     {
        largest = node_record[last_layer_node]->output;
        *network_class = x + 1;
     }
   }
}

/*********************************************************************/
/* Function Name: update_errors                    Number: 2.3       */
/* Description: This function compares the actual class of the       */
/*              data record with the network class.  If the two      */
/*              are not the same an error is recorded and the        */
/*              record number of the data record is stored.          */
/*                                                                   */
/* Functions Called: None                                            */
```

```
/* Variables Passed In: training or test_data - Structure array    */
/*                      record - Integer                           */
/*                      network_class - Integer                    */
/*                      misclassified - Integer array              */
/*                      *class_error - Integer pointer             */
/*                                                                 */
/* Variables Returned: *class_error - Integer pointer              */
/*                     misclassifier - Integer array               */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*******************************************************************/


void update_errors(struct data *data_record[],
                   int record,
                   int network_class,
                   int misclassified[],
                   int *class_error)
{
  int x;
  x = *class_error;
  if(data_record[record]->class != network_class)
  {
     misclassified[x] = data_record[record]->number;
     *class_error = *class_error + 1;
  }
}
```

G.3 NETTRAIN

```
/*******************************************************************/
/* Module Name:NETTRAIN     Number:3.0                             */
/* Description: This module contains the functions necessary to    */
/*              establish the network parameters.                  */
/* Modules Called: NETOUT, NETAUX, NETMATH, NETSHOW                */
/* Functions Contained: 3.1 global_MSE_minimization                */
/*                      3.2 make_nodes_at_data_points              */
/*                      3.3 center_weights_at_class_averages       */
/*                      3.4 k_means_cluster                        */
/*                      3.5 train_via_kohonen                      */
/*                      3.6 MSE_remaining_layers                   */
/*                      3.7 CE_remaining_layers                    */
/*                      3.8 CFM_remaining_layers                   */
/*                      3.9 PNN_last_layer                         */
/*                      3.10 scale_sigmas_by_class_interference    */
/*                      3.11 set_sigmas_to_constant                */
/*                      3.12 set_sigma_at_P_neighbor_avg           */
/*                                                                 */
/* Date:10 Nov 90     Revision: 1.0                                */
/*******************************************************************/


#include "netvrble.h"
#include "netfnctn.h"

FILE *fileptr;

/************************************************


/*******************************************************************/
```

```
/* Function Name: global_MSE_minimization     Number: 3.1          */
/* Description: This function performs a global minimization of    */
/*              the MSE to find network weights.  The equation is  */
/*              W = [M(transpose)](inverse)*Y(transpose)*S         */
/*                                                                 */
/* Functions Called: 8.1 determine_Y_matrix                        */
/*                   8.2 determine_S_matrix                        */
/*                   9.1 make_identity_matrix                      */
/*                   8.3 determine_M_matrix                        */
/*                   9.2 determine_matrix_transpose                */
/*                   9.3 invert_a_matrix                           */
/*                   8.4 calculate_weight_matrix                   */
/*                                                                 */
/* Variables Passed In:  Training Data - Structure                 */
/*                       Node Record - Structure                   */
/*                       nodes in layer - integer array            */
/*                       starting node in layer - integer array    */
/*                       total nodes - integer                     */
/*                       current layer - integer                   */
/*                                                                 */
/* Variables Returned:   Node Record - Structure                   */
/* Date: 10 Nov 90      Revision: 1.0                              */
/*******************************************************************/


void global_MSE_minimization (struct data *training_data[],
                              struct Node_data *Node_record[],
                              int train_set,
                              int nodes_in_layer[],
                              int starting_node_in_layer[],
                              int total_nodes,
                              int current_layer)
{

     int row, nodes;

     float MT[TRAIN_SET][TRAIN_SET], N[TRAIN_SET][TRAIN_SET],
           Y[TRAIN_SET][TRAIN_SET], M[TRAIN_SET][TRAIN_SET],
           weight[TRAIN_SET][CLASSES], S[TRAIN_SET][CLASSES];

     float *MTptr[TRAIN_SET], *Nptr[TRAIN_SET],
           *Yptr[TRAIN_SET], *Mptr[TRAIN_SET],
           *weightptr[TRAIN_SET], *Sptr[TRAIN_SET];

     for (row = 0; row < train_set; row++)
     {
           Nptr[row] = &N[row][0];
           MTptr[row] = &MT[row][0];
           Yptr[row] = &Y[row][0];
           Mptr[row] = &M[row][0];
           Sptr[row] = &S[row][0];
           weightptr[row] = &weight[row][0];
     }

     determine_Y_matrix(Node_record,
                        training_data,
                        train_set,
                        nodes_in_layer,
                        starting_node_in_layer,
                        total_nodes,
                        Yptr,
                        current_layer);

     nodes = nodes_in_layer[current_layer-1];
```

```
    determine_S_matrix(training_data,
                       train_set,
                       nodes_in_layer,
                       Sptr,
                       current_layer);

      nodes = nodes_in_layer[current_layer];

    nodes = nodes_in_layer[current_layer-1];

    make_identity_matrix (Mptr,
                          nodes);

    determine_M_matrix( Yptr,
                        Mptr,
                        nodes_in_layer,
                        train_set,
                        current_layer);

    determine_matrix_transpose(MTptr,
                               Mptr,
                               nodes);


    invert_a_matrix (Mptr,
                     Mptr,
                     nodes);

    calculate_weight_matrix(Node_record,
                                  weightptr,
                            Mptr,
                            Yptr,
                            Sptr,
                                  nodes_in_layer,
                                  starting_node_in_layer,
                                  train_set,
                                  current_layer);
}
/********************** End Global_MSE_Minimization ************/

/*********************************************************************/
/* Function Name: Make_nodes_at_data_points     Number:3.2           */
/* Doscription: This module sets the weights equal to the exemplar   */
/*              values. The equation is w(1) = x(1)                  */
/*                                                                   */
/* Functions Called: 7.2 calculate_layer_0_output                    */
/*                   7.3 calculate_layer_1_output                    */
/*                                                                   */
/* Variables Passed In: Training Data - Structure                    */
/*                      Node Data - Structure                        */
/*                      Train_set - Integer                          */
/*                      Nodes_in_layer - Integer array               */
/*                      Sigma_Max  - float                           */
/*                      Output_Max - float                           */
/*                      Starting_Node_in_layer - Integer array       */
/*                      Total_Nodes - Integer                        */
/*                      Nodes_1 - Integer                            */
/*                                                                   */
/* Variables Returned: Node_record - Structure                       */
/*                                                                   */
/* Date:10 Nov 90      Revision:1.0                                  */
/*********************************************************************/
```

```
void make_nodes_at_data_points(struct data *data_record[],
                               struct Node_data *n_record[],
                               int record_no,
                               int nodes_in_layer[],
                               float sigma_max,
                               float output_max,
                               int starting_node_in_layer[],
                               int total_nodes,
                               int nodes_1)

{

 int record, y, node1, covered, current_node, new_node_number, xfer_function;
 record = 0;
 nodes_in_layer[ ] = 0;
 for (record = 0; record < record_no; record++)
 {
    if(nodes_in_layer[1] < nodes_1 )
    {
       covered = 0;
       calculate_layer_0_output (data_record,
                                 n_record,
                                 nodes_in_layer,
                                 record);

       calculate_layer_1_output (data_record,
                                 n_record,
                                 nodes_in_layer,
                                 starting_node_in_layer,
                                 total_nodes);


       for (node1 = 0; node1 < nodes_in_layer[1]; node1++)
       {
            current_node = starting_node_in_layer[1] + node1;
            if ((n_record[current_node]->output > output_max)
            &&
              (n_record[current_node]->class == data_record[record]->class))
                covered = covered +1;
        }
       if (covered == 0)
       {
           new_node_number = starting_node_in_layer[1] + nodes_in_layer[1];
           for (y = 0; y < nodes_in_layer[0]; y++)
           {
              n_record[new_node_number]->weight[y] = data_record[record]->vector[y];
              n_record[new_node_number]->sigma[y] = sigma_max;
           }
           n_record[new_node_number]->class = data_record[record]->class;
          nodes_in_layer[1] = nodes_in_layer[1] +1;
       }
     }
   }
}
/***************** End Make Nodes at Data Points*******/


/*********************************************************************/
/* Function Name: Center_Weights_at_Class_Averages      Number:3.3 */
/* Description: This function sets the node weights equal to the   */
/*              averages of clusters of the same class             */
/*              w(+) = w(-) + [x-w(-)]/(N+1)                       */
/*                                                                 */
/* Functions Called: 9.4 update_average                            */
/*                                                                 */
```

```
/* Variables Passed In: Training data - Structure                  */
/*                      Node_record - Structure                    */
/*                      record_no - integer                        */
/*                      nodes_in_layer - integer array             */
/*                      average_threshold - float                  */
/*                      sigma_threshold - float                    */
/*                      starting_node_in_layer - integer array     */
/*                      total_nodes - integer                      */
/*                      current_layer - integer                    */
/*                      nodes_1_maximum - integer                  */
/*                                                                 */
/* Variables Returned:  Node_record - structure                    */
/*                                                                 */
/* Date:10 Nov 90        Revision: 1.0                             */
/*******************************************************************/

void center_weights_at_class_averages(struct data *data_record[],
                                      struct Node_data *node_record[],
                                      int record_no,
                                      int nodes_in_layer[],
                                      float threshold,
                                      float sigma_max,
                                      int starting_node_in_layer[],
                                      int total_nodes,
                                      int current_layer,
                                      int nodes_max)
{
 int nearest_node = 0;
 int iteration;
 double min_distance = 1000;
 double distance, buffer;
 double exponent_1 = 2;
 double exponent_2 = .5;
 int new_node = 0;
 float new_average;
 int record, y, x, covered, elements[TRAIN_SET+TEST_SET], current_node, previous_layer_node, new_node_number;
 nodes_in_layer[current_layer] = 0;
 record = 0;


 for (x=0; x < TRAIN_SET + TEST_SET; x++)
    elements[x]=0;
 do
 {
  new_node = 0;
  for (record = 0; record < record_no; record++)
  {
    if (nodes_in_layer[current_layer] < nodes_max)
    {
      min_distance = 1000;
      covered = 0;
      calculate_layer_0_output(data_record,
                               node_record,
                               nodes_in_layer,
                               record);

      for (x = 0; x < nodes_in_layer[current_layer]; x++)
      {
        current_node = starting_node_in_layer[current_layer] + x;
        if (node_record[current_node]->class == data_record[record]->class)
        {
            buffer = 0;
            for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
            {
              previous_layer_node = starting_node_in_layer[current_layer-1]+y;
```

```
            distance = node_record[current_node]->weight[previous_layer_node]
                     -node_record[previous_layer_node]->output;
            distance = pow(distance, exponent_1);
            buffer = buffer + distance;
           }
           distance = pow(buffer,exponent_2);
           if (distance < min_distance)
           {
              nearest_node = current_node;
              min_distance = distance;
           }
        }
      }

      if (min_distance < threshold)
      {
         x = nearest_node-starting_node_in_layer[current_layer];
         elements[x] = elements[x]+1;
         for ( y = 0; y < nodes_in_layer[current_layer-1]; y++)
         {
           previous_layer_node = starting_node_in_layer[current_layer-1] +y;
           update_average(node_record[nearest_node]->weight[previous_layer_node],
                          elements[x],
                          node_record[previous_layer_node]->output,
                          &new_average);

           node_record[nearest_node]->weight[previous_layer_node] = new_average;
           }
         x = nodes_in_layer[current_layer];
         covered = covered +1;
      }

      else
      {
         new_node_number = starting_node_in_layer[current_layer] + x;
         for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
         {
           previous_layer_node = starting_node_in_layer[current_layer-1] + y;
           node_record[new_node_number]->weight[previous_layer_node] = node_record[previous_layer_node]->output;
           node_record[new_node_number]->sigma[previous_layer_node] = sigma_max;
         }
          node_record[new_node_number]->class = data_record[record]->class;
          elements[x] = 1;
          nodes_in_layer[current_layer] = nodes_in_layer[current_layer]+1;
        /*   printf("\ncreated node  %d",new_node_number); */
          new_node = 1;
      }
     }
    }
   }

  while (new_node == 1);
}

/***************End Center Weights at Class Centers *******/


/*********************************************************************/
/* Function Name:K-Means Cluster              Number:3.4             */
/* Description: This function implements the K-Means Clustering     */
/*              algorithm to set the weights.                        */
/*              w(+) = (1/M)sum [x(n)]                               */
/*                                                                   */
/* Functions Called: 7.2  calculate_layer_0_output                   */
/*                   8.28 find_nearest_neighbor                      */
```

```
/*                  9.4  update_average                              */
/*                                                                   */
/* Variables Passed In: Training Data - Structure                    */
/*                      Node_record - Structure                      */
/*                      Train_set - Structure                        */
/*                      Nodes_1_Maximum Integer                      */
/*                      Nodes_in_Layer - Integer array               */
/*                      Starting_Node_in_layer - Integer array       */
/*                      Current_Layer - Integer                      */
/*                                                                   */
/* Variables Returned:  Node_record - Structure                      */
/* Date: 10 Nov 90     Revision:1.0                                  */
/*********************************************************************/

void k_means_cluster(struct data *data_record[],
                     struct Node_data *node_record[],
                     int record_no,
                     int number_of_clusters,
                     int nodes_in_layer[],
                     int starting_node_in_layer[],
                     int current_layer)
{
 int record, x, y, z, current_node, nearest_node, previous_layer_node, update;
 int total_elements, current_record, elements;
 float new_average, current_avg[TEST_SET];
 int element[TEST_SET][TEST_SET], elements_in_cluster[TEST_SET];
 double distance, nearest_distance, buffer, difference;
 new_average = 0;
 for (record = 0; record < number_of_clusters; record++)
 {
   calculate_layer_0_output(data_record,
                            node_record,
                            nodes_in_layer,
                            record);

   current_node = starting_node_in_layer[current_layer] + record;
   for (y = 0; y < nodes_in_layer[current_layer-1];y++)
   {
     previous_layer_node = starting_node_in_layer[current_layer-1]+y;
     node_record[current_node]->weight[previous_layer_node] = node_record[previous_layer_node]->output;
   }
 }

 do
 {
    for (x = 0; x < number_of_clusters; x++)
      elements_in_cluster[x] = 0;
    update = 0;
    for (record = 0; record < record_no; record++)
    {
      find_nearest_neighbor(data_record,
                            node_record,
                            record,
                            nodes_in_layer,
                            starting_node_in_layer,
                            current_layer,
                            &nearest_node);
      nearest_node = nearest_node - starting_node_in_layer[current_layer];
      current_record = elements_in_cluster[nearest_node];
      element[nearest_node][current_record] = record;
      elements_in_cluster[nearest_node] =   elements_in_cluster[nearest_node]+1;
    }

    for (x = 0; x < nodes_in_layer[current_layer]; x++)
    {
```

```
      current_node = starting_node_in_layer[current_layer] + x;
      for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
        current_avg[y] = 0;
      total_elements = elements_in_cluster[x];
      for (z = 0; z < total_elements; z++)
      {
         record = element[x][z];
         elements = z + 1;
         calculate_layer_0_output(data_record,
                                  node_record,
                                  nodes_in_layer,
                                  record);

         for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
         {
           previous_layer_node = starting_node_in_layer[current_layer-1] + y;
           update_average(current_avg[y],
                          elements,
                          node_record[previous_layer_node]->output,
                          &new_average);

           current_avg[y] = new_average;
         }
      }
      for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
      {
         previous_layer_node = starting_node_in_layer[current_layer-1] + y;
         difference = node_record[current_node]->weight[previous_layer_node]-current_avg[y];
         if (fabs(difference) > .00001)
           update = 1;
         node_record[current_node]->weight[previous_layer_node] = current_avg[y];
       }
    }
  }
  while(update != 0);
}
/***************** End K-Means Cluster **********************/


/*********************************************************************/
/* Function Name: Kohonen Training      Number:3.5                   */
/* Description: This function updates the weights via the kohonen    */
/*              training algorithm                                   */
/*              w(+) = w(-) + a[x-w(-)]                              */
/*                                                                   */
/* Functions Called: 4.9  get_random_record                          */
/*                   7.2  calculate_layer_0_output                   */
/*                   8.13 calc_dist_ontput_to_nxt_lyr                */
/*                   8.14 find_nearest_element                       */
/*                   8.11 get_linear_training_eta                    */
/*                   8.12 get_kohonen_neighborhood                   */
/*                   8.15 find_kohonen_boundaries                    */
/*                   8.16 determine_neighborhood_elements            */
/*                   8.17 train_kohonen_weights                      */
/*                                                                   */
/* Variables Passed In: Training_Data - Structure                    */
/*                      Node_Record - Structure                      */
/*                      Nodes_in_layer - integer array               */
/*                      Starting_Node_in_layer - integer array       */
/*                      Neighborhoods - integer array                */
/*                      Train Width - integer array                  */
/*                      Train Scale - float array                    */
/*                      Width_number - integer                       */
/*                      Nodes_x - integer                            */
/*                      Nodes_y - integer                            */
```

```
/*                    Current_layer - integer                    */
/*                    Train_set - integer                        */
/*                                                               */
/* Variables Returned: Node_record - Structure                   */
/* Date: 10 Nov 90      Revision: 1.0                            */
/*****************************************************************/

void train_via_kohonen(struct data *data_record[],
                       struct Node_data *node_record[],
                       int nodes_in_layer[],
                       int starting_node_in_layer[],
                       int neighborhoods[],
                       int train_width[],
                       float train_scale[],
                       int width_no,
                       int nodes_x,
                       int nodes_y,
                       int current_layer,
                       int train_set,
                            int kohonen_iterations,
                            unsigned seed)

 {
   int x, y, current_node, iterations, record;
   int winner_node;
   int neighbors, left, right, up, down, nodes_to_update;
   int update_node[100];
   float eta;
   float distance[100];

   record = 0;
   iterations = 0;
   srand(seed);
   do
   {
      get_random_record(train_set,
                        &record);

      calculate_layer_0_output(data_record,
                               node_record,
                               nodes_in_layer,
                               record);

      calc_dist_outputs_to_nxt_lyr(node_record,
                                   nodes_in_layer,
                                   starting_node_in_layer,
                                   current_layer,
                                   distance);

       find_nearest_element(distance,
                            nodes_in_layer[1],
                            &winner_node);

       winner_node = starting_node_in_layer[current_layer]
                   + winner_node;

       get_linear_training_eta(train_width,
                               train_scale,
                               iterations,
                               &eta,
                               width_no);


        get_kohonen_neighborhood(train_width,
                                 iterations,
```

```
                                neighborhoods,
                                width_no,
                                &neighbors);


          find_kohonen_boundaries(winner_node,
                                          starting_node_in_layer,
                                          current_layer,
                                          nodes_x,
                                          nodes_y,
                                          neighbors,
                                          &left,
                                          &right,
                                          &up,
                                          &down);

            determine_neighborhood_elements(left,
                                          right,
                                          up,
                                          down,
                                          &nodes_to_update,
                                          update_node,
                                          starting_node_in_layer,
                                          nodes_x,
                                          current_layer);

            train_kohonen_weights(node_record,
                                        nodes_in_layer,
                                        starting_node_in_layer,
                                        current_layer,
                                        nodes_to_update,
                                        update_node,
                                        eta);

       iterations = iterations + 1;
      }
      while (iterations < kohonen_iterations);
}
/********** End train via kohonen **********************/

/*****************************************************************/
/* Function Name: MSE Remaining Layers     Number:3.6            */
/* Description: The function performs backpropagation to optimize */
/*              the MSE objective function.                      */
/*                                                               */
/* Functions Called: 4.8  get_random_class_record                */
/*                   7.1  calculate_feedforward_network_output   */
/*                   8.10 calculate_errors_in_output             */
/*                   8.5  MSE_last_layer                         */
/*                   8.8  MSE_mid_layer                          */
/*                   8.9  MSE_1st_layer                          */
/*                   5.9  correct_node_weights                   */
/*                   2.1  test_the_network                       */
/*                   9.6  calculate_percentage                   */
/*                                                               */
/* Variables Passed In: Node_Record - Structure                  */
/*                      Training_Data - Structure                */
/*                      Test_Data - Structure                    */
/*                      Transfer_Function - Integer array        */
/*                      Nodes_in_Layer - Integer array           */
/*                      Starting_Node_in_Layer - Integer array   */
/*                      Number_of_Layers - Integer               */
/*                      Current_Layer - Integer                  */
/*                      Train_Set - Integer                      */
/*                      Test_Set - Integer                       */
```

```
/*                      MSE_Eta - Float                                 */
/*                      Total_Nodes - Integer                           */
/*                      MSE_Successes - Integer                         */
/*                      MSE_Epsilon - float                             */
/*                      MSE_Iterations - Integer                        */
/*                      MSE_momentum - float                            */
/*                      Classes - integer                               */
/*                      Record_Seed - unsigned                          */
/*                      File_Ptr - File pointer                         */
/*                                                                      */
/* Variables Returned: Node_Record - Structure                          */
/* Date: 10 Nov 90      Revision: 1.0                                   */
/************************************************************************/

void MSE_remaining_layers(struct Node_data *node_record[],
                          struct data *data_record[],
                             struct data *test_record[],
                          int transfer_function[],
                          int nodes_in_layer[],
                          int starting_node_in_layer[],
                          int number_of_layers,
                          int current_layer,
                          int train_set,
                             int test_set,
                      float eta,
                          int total_nodes,
                             int MSE_successes,
                     float epsilon,
                             int backprop_iterations,
                     float alpha,
                             int classes,
                             unsigned seed,
                             FILE *file_ptr)

{
   float desired_output[CLASSES];
   int x, y, error, record, layer, node;
   int success = 0;
   int iteration = 0;
   int error_interval_count = 0;
   int error_count = 0;
   int misclassified[TRAIN_SET];
   int correct_class = 0;
   float class_threshold = 0;
   float per_cent_correct = 0.0;
   float old_wght[TOTAL_NODES][TOTAL_NODES];
   float *wght_ptr[TOTAL_NODES];

   class_threshold = 1-epsilon;

   for (x = 0; x < total_nodes; x++)
   {
     for (y = 0; y < total_nodes; y++)
       old_wght[x][y] = 0;
     wght_ptr[x] = &old_wght[x][0];
   }

  srand(seed);
  do
  {
     error = 0;
     get_random_class_record(data_record,
                             train_set,
                             classes,
                             &record);
```

```
for (x = 0; x < nodes_in_layer[number_of_layers]; x++)
{
  if (x == data_record[record]->class -1)
    desired_output[x] = 1.00;
  else
    desired_output[x] = 0.00;
}

calculate_feed_forward_network_output(data_record,
                                      node_record,
                                      number_of_layers,
                                      nodes_in_layer,
                                      starting_node_in_layer,
                                      record,
                                      total_nodes);

calculate_errors_in_output(node_record,
                           dosired_output,
                           nodes_in_layer,
                           starting_node_in_layer,
                           number_of_layers,
                           &error,
              epsilon);

if (error != 0)
{
  success = 0;
  for (layer = number_of_layers; layer > current_layer -1; layer--)
  {
     if (layer == number_of_layers)
        MSE_last_layer(node_record,
                           desired_output,
                           nodes_in_layer,
                           starting_node_in_layer,
                           layer,
                           eta,
                  epsilon,
                           wght_ptr,
                           alpha);

     else if (layer == number_of_layers-1)
        MSE_mid_layer(node_record,
                          desired_output,
                          nodes_in_layer,
                          starting_node_in_layer,
                          layer,
                          eta,
                          wght_ptr,
                          alpha);

    else if (layer == number_of_layers-2)
        MSE_1st_layer(node_record,
                          desired_output,
                          nodes_in_layer,
                          starting_node_in_layer,
                          layer,
                          eta,
                          wght_ptr,
                          alpha);
  }
  }
  else
    success = success + 1;
  iteration = iteration +1;
```

```
    correct_node_weights(node_record,
                                 total_nodes);

    error_interval_count = error_interval_count + 1;
    if (error_interval_count == 1000)
    {
        error_count = 0;
        for (x = 0; x < train_set; x++)
          test_the_network(data_record,
                           node_record,
                           nodes_in_layer,
                           starting_node_in_layer,
                           number_of_layers,
                           x,
                           total_nodes,
                           class_threshold,
                           misclassified,
                           &error_count);

      correct_class = train_set - error_count;
         calculate_percentage((float)correct_class,
                                      (float)train_set,
                                      &per_cent_correct);
      fprintf(file_ptr,"\niteration = %d training correct = %f",
                           iteration, per_cent_correct);

         error_count = 0;
         for (x = 0; x < test_set; x++)
             test_the_network(test_record,
                                        node_record,
                                        nodes_in_layer,
                                        starting_node_in_layer,
                                        number_of_layers,
                                        x,
                                        total_nodes,
                                        class_threshold,
                                        misclassified,
                                        &error_count);

         correct_class = test_set - error_count;
         calculate_percentage((float)correct_class,
                              (float)test_set,
                              &per_cent_correct);
         fprintf(file_ptr,"  test percent = %f",per_cent_correct);
         error_interval_count = 0;
       }
  }
  while((iteration < backprop_iterations) && (success < MSE_successes));
 }


/********************** End MSE Remaining Layers **********/

/*********************************************************************/
/* Function Name: CE_Remaining_Layers    Number:3.7                  */
/* Description: The function sets parameters by backpropagation     */
/*              according to the CE objective function              */
/*                                                                   */
/* Functions Called: 4.8  get_random_class_record                    */
/*                   7.1  calculate_feedforward_network_output       */
/*                   8.10 calculate_errors_in_output                 */
/*                   8.20 CE_last_layer                              */
/*                   8.21 CE_mid_layer                               */
/*                   8.22 CE_first_layer                             */
/*                   5.9  correct_node_weights                       */
```

```
/*                  2.1  test_the_Network                          */
/*                  9.6  calculate_percentage                      */
/*                                                                 */
/* Variables Passed In: Node_Record - Structure                    */
/*                      Training_Data - Structure                  */
/*                      Test_Data - Structure                      */
/*                      Nodes_in_Layer - Integer array             */
/*                      Number_of_Layers - Integer                 */
/*                      Starting_Node_in_Layer - Integer           */
/*                      Current_Layer - Integer                    */
/*                      Train_Set - Integer                        */
/*                      Test_Set - Integer                         */
/*                      CE_eta - Float                             */
/*                      Total_Nodes - Integer                      */
/*                      CE_Successes - Integer                     */
/*                      CE_Epsilon - Float                         */
/*                      CE_Iterations - Integer                    */
/*                      CE_Momentum - Float                        */
/*                      Classes - Integer                          */
/*                      Record_Seed - Unsigned                     */
/*                      File_Ptr - File pointer                    */
/*                                                                 */
/* Variables Returned: Node_Record - Structure                     */
/* Date:10 Nov 90      Revision: 1.0                               */
/*******************************************************************/

void CE_remaining_layers(struct Node_data *node_record[],
                         struct data *data_record[],
                         struct data *test_record[],
                         int nodes_in_layer[],
                         int starting_node_in_layer[],
                         int number_of_layers,
                         int current_layer,
                         int train_set,
                         int test_set,
                         float eta,
                         int total_nodes,
                         int CE_successes,
                         float epsilon,
                         int CE_iterations,
                         float momentum,
                         int classes,
                         unsigned seed,
                         FILE *file_ptr)

{
  float desired_output[CLASSES];
  int x, y, error, record, layer, node;
  int success = 0;
  int iteration = 0;
  int error_interval_count = 0;
  int error_count = 0;
  int misclassified[TRAIN_SET];
  int correct_class = 0;
  float per_cent_correct = 0.0;
  float old_wght[TOTAL_NODES][TOTAL_NODES];
  float *wght_ptr[TOTAL_NODES];

  float class_threshold = 1-epsilon;
  float new_eta = eta/(nodes_in_layer[number_of_layers]*2.3);

  for (x = 0; x < total_nodes; x++)
  {
    for (y = 0; y < total_nodes; y++)
      old_wght[x][y] = 0;
```

```
  wght_ptr[x] = &old_wght[x][0];
}
srand(seed);

do
{
   error = 0;
   get_random_class_record(data_record,
                           train_set,
                           classes,
                           &record);

   for (x = 0; x < nodes_in_layer[number_of_layers]; x++)
   {
      if (x == data_record[record]->class-1)
         desired_output[x] = 1.00;
      else
         desired_output[x] = 0.00;
   }

   calculate_feed_forward_network_output(data_record,
                                         node_record,
                                         number_of_layers,
                                         nodes_in_layer,
                                         starting_node_in_layer,
                                         record,
                                         total_nodes);

   calculate_errors_in_output(node_record,
                              desired_output,
                              nodes_in_layer,
                              starting_node_in_layer,
                              number_of_layers,
                              &error,
                              epsilon);

   if (error != 0)
   {
     success = 0;
     for (layer = number_of_layers; layer > current_layer -1; layer--)
     {
       if (layer == number_of_layers)
         CE_last_layer(node_record,
                       nodes_in_layer,
                       starting_node_in_layer,
                       layer,
                       wght_ptr,
                       new_eta,
                       momentum,
                       desired_output);

       else if (layer == number_of_layers -1)
          CE_mid_layer(node_record,
                       nodes_in_layer,
                       starting_node_in_layer,
                       layer,
                       wght_ptr,
                       new_eta,
                       momentum,
                       desired_output);

       else if (layer == number_of_layers-2)
          CE_first_layer(node_record,
                         nodes_in_layer,
                         starting_node_in_layer,
```

```
                        layer,
                        wght_ptr,
                        new_eta,
                        momentum,
                        desired_output,
                        total_nodes);
            }
        }
        else
          success = success + 1;
        iteration = iteration + 1;
        correct_node_weights(node_record,
                             total_nodes);

        error_interval_count = error_interval_count + 1;
        if (error_interval_count == 1000)
        {
           error_count = 0;
           for (x = 0; x < train_set; x++)
             test_the_network(data_record,
                        node_record,
                        nodes_in_layer,
                        starting_node_in_layer,
                        number_of_layers,
                        x,
                        total_nodes,
                        class_threshold,
                        misclassified,
                        &error_count);

           correct_class = train_set - error_count;
           calculate_percentage((float)correct_class,
                                (float)train_set,
                                &per_cent_correct);
           fprintf(file_ptr,"\niteration = %d training correct = %f",
                           iteration, per_cent_correct);

           error_count = 0;
           for (x = 0; x < test_set; x++)
             test_the_network(test_record,
                        node_record,
                        nodes_in_layer,
                        starting_node_in_layer,
                        number_of_layers,
                        x,
                        total_nodes,
                        class_threshold,
                        misclassified,
                        &error_count);

           correct_class = test_set - error_count;
           calculate_percentage((float)correct_class,
                                (float)test_set,
                                &per_cent_correct);
           fprintf(file_ptr,"  test correct = %f",per_cent_correct);

           error_interval_count = 0;
        }
    }
    while((iteration < CE_iterations) && (success < CE_successes));
}

/*************************************************************/
/*    End CE Remaining Layers                                */
/*************************************************************/
```

```
/*************************************************************************/
/* Function Name: CFM_Remaining_Layers    Number:3.8                     */
/* Description: The function sets parameters by backpropagation          */
/*              according to the CE objective function                   */
/* Functions Called: 4.8  get_random_class_record                        */
/*                   7.1  calculate_feedforward_network_output           */
/*                   2.2  determine_class_as_largest                     */
/*                   8.26 find_second_highest_node                       */
/*                   8.23 calculate_zn                                   */
/*                   8.20 CFM_last_layer                                 */
/*                   8.21 CFM_mid_layer                                  */
/*                   8.22 CFM_first_layer                                */
/*                   5.9  correct_node_weights                           */
/*                   2.1  test_the_network                               */
/*                   9.6  calculate_percentage                           */
/*                                                                       */
/* Variables Passed In: Node_Record - Structure                          */
/*                      Training_Data - Structure                        */
/*                      Test_Data - Structure                            */
/*                      Nodes_in_Layer - Integer array                   */
/*                      Starting_Node_in_layer - Integer array           */
/*                      Number_of_Layers - Integer                       */
/*                      Current_Layer - Integer                          */
/*                      Train_Set - Integer                              */
/*                      Test_Set - Integer                               */
/*                      CE_eta - Float                                   */
/*                      Total_Nodes - Integer                            */
/*                      CFM_Successes - Integer                          */
/*                      CFM_Iterations - Integer                         */
/*                      CFM_alpha - Float                                */
/*                      CFM_beta - Float                                 */
/*                      CFm_zeta - Float                                 */
/*                      CE_Momentum - Float                              */
/*                      CE_delat - Float                                 */
/*                      Classes - Integer                                */
/*                      Record_Seed - Unsigned                           */
/*                      File_Ptr - File pointer                          */
/*                                                                       */
/* Variables Returned: Node_Record - Structure                           */
/* Date:10 Nov 90      Revision: 1.0                                     */
/*************************************************************************/



/*************************************************************/
/*   CFM Remaining Lyrs                                      */
/*************************************************************/


void CFM_remaining_layers(struct Node_data *node_record[],
                          struct data *data_record[],
                          struct data *test_record[],
                          int nodes_in_layer[],
                          int starting_node_in_layer[],
                          int number_of_layers,
                          int current_layer,
                          int train_set,
                          int test_set,
                          float eta,
                          int total_nodes,
                          int CFM_successes,
                          int CFM_iterations,
                          float alpha,
```

```
                    float beta,
                    float zeta,
                    float momentum,
                    float delta,
                    int classes,
                    unsigned seed,
                    FILE *file_ptr)

{
  int x, y, record, layer, correct_node, winner_node;
  int network_class = 0;
  int success = 0;
  int iteration = 0;
  int error_interval_count = 0;
  int error_count = 0;
  int misclassified[TRAIN_SET];
  int correct_class = 0;
  float per_cent_correct = 0;
  int next_highest_node;
  float class_threshold = 0;
  float new_eta = 0;
  float zn[CLASSES];
  float old_wght[TOTAL_NODES][TOTAL_NODES];
  float *wght_ptr[TOTAL_NODES];
  float epsilon = .9;

  for (x = 0; x < total_nodes; x++)
  {
    for (y = 0; y < total_nodes; y++)
      old_wght[x][y] = 0;
    wght_ptr[x] = &old_wght[x][0];
  }

  new_eta = eta * beta * alpha/(nodes_in_layer[number_of_layers]-1);
  srand(seed);

  do
  {
    iteration = iteration + 1;

    get_random_class_record(data_record,
                            train_set,
                            classes,
                            &record);

    calculate_feed_forward_network_output(data_record,
                                          node_record,
                                          number_of_layers,
                                          nodes_in_layer,
                                          starting_node_in_layer,
                                          record,
                                          total_nodes);

    correct_node = starting_node_in_layer[number_of_layers]
                 + data_record[record]->class - 1;


    determine_class_as_largest(node_record,
                               nodes_in_layer,
                               starting_node_in_layer,
                               &network_class,
                               number_of_layers,
                               class_threshold);

    winner_node = starting_node_in_layer[number_of_layers]
```

```
            + network_class * 1;

   if (winner_node == correct_node)
      find_second_highest_node (node_record,
                                nodes_in_layer,
                                starting_node_in_layer,
                                number_of_layers,
                                winner_node,
                                &next_highest_node);


   if ((winner_node != correct_node)
      || (node_record[correct_node]->output
         - node_record[next_highest_node]->output < delta))
   {
      success = 0;
      calculate_zn (node_record,
                    nodes_in_layer,
                    starting_node_in_layer,
                    number_of_layers,
                    correct_node,
                    zn,
                    beta,
                    zeta);

      for (layer = number_of_layers; layer > current_layer-1; layer--)
      {
        if (layer == number_of_layers)
           CFM_last_layer(node_record,
                          nodes_in_layer,
                          starting_node_in_layer,
                          layer,
                          zn,
                          correct_node,
                          new_eta,
                          wght_ptr,
                          momentum);

         else if (layer == number_of_layers-1)
              CFM_mid_layer(node_record,
                          nodes_in_layer,
                          starting_node_in_layer,
                          layer,
                          zn,
                          correct_node,
                          new_eta,
                          wght_ptr,
                          momentum);

          else if (layer == number_of_layers-2)
            CFM_first_layer(node_record,
                          nodes_in_layer,
                          starting_node_in_layer,
                          layer,
                          zn,
                          correct_node,
                          new_eta,
                          wght_ptr,
                          momentum,
                          total_nodes);
      }
}
   else
     success = success + 1;
```

```
      error_interval_count = error_interval_count + 1;
      if (error_interval_count == 1000)
      {
         error_count = 0;
         for (x = 0; x < train_set; x++)
            test_the_network(data_record,
                             node_record,
                             nodes_in_layer,
                             starting_node_in_layer,
                             number_of_layers,
                             x,
                             total_nodes,
                             epsilon,
                             misclassified,
                             &error_count);

         correct_class = train_set - error_count;
            calculate_percentage((float)correct_class,
                                 (float)train_set,
                                 &per_cent_correct);

         fprintf(file_ptr,"\niterations = %d training correct = %f",
                                 iteration, per_cent_correct);

         error_count = 0;
         for (x = 0; x < test_set; x++)
           test_the_network(test_record,
                            node_record,
                            nodes_in_layer,
                            starting_node_in_layer,
                            number_of_layers,
                            x,
                            total_nodes,
                            epsilon,
                            misclassified,
                            &error_count);

         correct_class = test_set - error_count;
         calculate_percentage((float)correct_class,
                              (float)test_set,
                              &per_cent_correct);

         fprintf(file_ptr,"  test correct = %f",per_cent_correct);
         error_interval_count = 0;
       }
      }
      while(success < CFM_successes && iteration < CFM_iterations);
}

/*********************************************************/
/*  End CFM Remaining Lyrs                               */
/*********************************************************/

/*******\  ***************************************************************/
/* Function Name:PNN_last_layer          Number:3.9                      */
/* Description: This function sets the network weights in the            */
/*              output layer equal to 1 and connects the output          */
/*              layer nodes only to the nodes of the same class          */
/*              in the hidden layer                                      */
/*                                                                       */
/* Functions Called: None                                                */
/* Variables Passed In: Node_Record - Structure                          */
/*                      Nodes_in_Layer - Integer array                   */
/*                      Starting_Node_in_Layer - Integer array           */
/*                      Current_Layer - Integer                          */
```

```
/*                                                                    */
/* Variables Returned: Node_Record - Structure                        */
/* Date:10 Nov 90      Revision:1.0                                   */
/**********************************************************************/


void PNN_last_layer(struct Node_data *node_record[],
                                int nodes_in_layer[],
                                int starting_node_in_layer[],
                                int current_layer)

{
   int x, y, current_node, previous_lyr_node;
   float nodes_of_class;

   for (x = 0; x < nodes_in_layer[current_layer]; x++)
   {
     current_node = starting_node_in_layer[current_layer]+x;
     node_record[current_node]->class = x + 1;
     nodes_of_class = 0;
     for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
     {
        previous_lyr_node = starting_node_in_layer[current_layer-1]+y;
        if (node_record[current_node]->class ==
                          node_record[previous_lyr_node]->class)
        {
           nodes_of_class += 1;
           node_record[current_node]->connect[previous_lyr_node] = 1;
        }
        else
        {
           node_record[current_node]->connect[previous_lyr_node] = 0;
           node_record[current_node]->weight[previous_lyr_node] = 0;
        }
     }
     for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
     {
       previous_lyr_node = starting_node_in_layer[current_layer-1] + y;
       node_record[current_node]->weight[previous_lyr_node] = 1;
     }

   }
}

/**********************************************************/
/*   End Connect Nodes to Class Nodes                     */
/**********************************************************/

/**********************************************************************/
/* Function Name:  Scale_Sigmas_by_Class_Interference Number:3.10  */
/* Description: This function scales the size of the RBF sigmas by */
/*              by a constant if a data point causes more than 1   */
/*              RBF node to be excited past some threshold and the */
/*              RBF nodes are not detecting the same classes.      */
/*                                                                 */
/* Functions Called: 7.2 calculate_layer_0_output                  */
/*                   7.3 calculate_layer_1_output                  */
/*                                                                 */
/* Variables Passed In: Training_Data - Structure                  */
/*                      Node_Record - Structure                    */
/*                      Train_Set - Integer                        */
/*                      Nodes_in_Layer - Integer array             */
/*                      Starting_Node_in_Layer - Integer array     */
/*                      Total_Nodes - Integer                      */
/*                      Current_Layer - Integer                    */
```

```
/*                    Output_Threshold - Float                    */
/*                    Scale_Factor - Float                        */
/*                                                                */
/* Variables Returned: Node_Record - Structure                    */
/* Date: 10 Nov 90      Revision: 1.0                             */
/******************************************************************/

void scale_sigmas_by_class_interference(struct data *data_record[],
                                        struct Node_data *N_record[],
                                   int record_no,
                                        int nodes_in_layer[],
                                        int starting_node_in_layer[],
                                        int total_nodes,
                                        int current_layer,
                                   float out_max,
                                        float scale_factor)
{
   int record_ptr[TOTAL_NODES];
   int x, y, node, record, z, current_node;
   for (record = 0; record < record_no; record++)
   {
     calculate_layer_0_output(data_record,
                              N_record,
                              nodes_in_layer,
                              record);

     calculate_layer_1_output(data_record,
                              N_record,
                              nodes_in_layer,
                              starting_node_in_layer,
                              total_nodes);

      x = 0;
      for (node = 0; node < nodes_in_layer[current_layer]; node++)
      {
         current_node = starting_node_in_layer[current_layer] + node;
         if ((N_record[current_node]->output > out_max) &&
               (N_record[current_node]->class != data_record[record]->class))
         {
               record_ptr[x] = current_node;
               x = x+1;
          }
     }
       if (x > 0)
       {
         for (y = 0; y < x; y++)
         {
           current_node = record_ptr[y];
           do
           {
             for (z=0; z < total_nodes; z++)
              N_record[current_node]->sigma[z]=N_record[current_node]->sigma[z]
                        -scale_factor * (N_record[current_node]->sigma[z]);
              calculate_node_output(data_record,
                                    N_record,
                                    current_node,
                                    total_nodes);
            }
            while(N_record[current_node]->output > out_max);
         }
       }
    }
  }
/******************** End Optimize Sigmas ****************/
```

```
/*********************************************************************/
/* Function Name: Set_Sigma_to_Constant     Number: 3.11             */
/* Description: This function sets the RBF sigmas to a constant     */
/* Functions Called: None                                            */
/* Variables Passed In: Node_Record - Structure                      */
/*                      Nodes_in_Layer - Integer array               */
/*                      Starting_Node_in_Layer - Integer array       */
/*                      Current_Layer - Integer                      */
/*                      Sigma_Constant - Float                       */
/*                                                                   */
/* Variables Returned: Node_Record - Structure                       */
/* Date: 10 Nov 90              Revision:1.0                         */
/*********************************************************************/

void set_sigmas_to_constant(struct Node_data *node_record[],
                            int nodes_in_layer[],
                            int starting_node_in_layer[],
                            int current_layer,
                            float sigma_constant)

{
  int x, y, current_node, previous_layer_node;
  for (x= 0; x < nodes_in_layer[current_layer]; x++)
  {
     current_node = starting_node_in_layer[current_layer]+x;
     for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
     {
        previous_layer_node = starting_node_in_layer[current_layer-1]+y;
        node_record[current_node]->sigma[previous_layer_node] = sigma_constant;
     }
   }
 }
/******************End Set Sigmas to a Constant************/

/*********************************************************************/
/* Function Name:Set_Sigmas_at_P_Neighbors_Avg      Number:3.12      */
/* Description: This function sets the sigmas of the RBFs equal to   */
/*              the root mean square distances of the closest P      */
/*              Neighbors                                            */
/*              sigma = sqrt[(1/P)sum(dp)]                           */
/*                                                                   */
/* Functions Called: 8.18 find_distance_between_nodes                */
/*                   8.19 sort_2_dim_array                           */
/*                                                                   */
/* Variables Passed In: Node_Record - Structure                      */
/*                      Nodes_in_Layer - Integer array               */
/*                      Starting_Node_in_Layer - Integer array       */
/*                      Current_Layer - Integer                      */
/*                      Total_Nodes - Integer                        */
/*                      P_Neighbors - Integer                        */
/*                                                                   */
/* Variables Returned: Node_Record - Structure                       */
/* Date: 10 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void set_sigma_at_P_neighbor_avg(struct Node_data *node_record[],
                                 int nodes_in_layer[],
                                 int starting_node_in_layer[],
                                 int current_layer,
                                 int total_nodes,
                                 int p_neighbors)
{
   int x, y, z, current_node, next_node, previous_layer_node;
   float distance_between[TRAIN_SET][TRAIN_SET];
   float *distance_ptr[TRAIN_SET];
```

```
   double avg, distance, buffer;
   double exponent_1 = 2;
   double exponent_2 = .5;
   for (x = 0; x < nodes_in_layer[current_layer]; x++)
     distance_ptr[x] = &distance_between[x][0];
   for (x= 0; x < nodes_in_layer[current_layer]; x++)
   {
      current_node = starting_node_in_layer[current_layer] + x;
      for (y = 0; y < nodes_in_layer[current_layer]; y++)
      {
        next_node = starting_node_in_layer[current_layer] + y;
        find_distance_between_nodes(node_record,
                                    nodes_in_layer,
                                    starting_node_in_layer,
                                    current_node,
                                    next_node,
                                    current_layer,
                                    distance_ptr);
      }
          sort_2_dim_array(distance_ptr,
                           nodes_in_layer[current_layer],
                           x);
        distance = 0;
        for (y = 0; y < p_neighbors +1; y++)
        {
           buffer = distance_between[x][y];
           distance = distance + pow(buffer,exponent_1);
        }
        avg = distance/p_neighbors;
        for (z = 0; z < total_nodes; z++)
          node_record[current_node]->sigma[z] = pow(avg,exponent_2);
      }

}

/*************************************************************/
/* End Set Sigma at P neighbor average                       */
/*************************************************************/
```

G.4 NETINPUT

```
/*********************************************************************/
/* Module Name: NETINPUT.C     Number:4.0                            */
/* Description: This module provides the functions necessary to      */
/*              randomly load the input data and to preset the       */
/*              network parameters                                   */
/*                                                                   */
/* Modules Called: NETINIT.C                                         */
/* Functions Contained: 4.1  load_input_patterns                     */
/*                      4.2  load_separate_files                     */
/*                      4.3  load_from_single_file                   */
/*                      4.4  load_by_classes                         */
/*                      4.5  get_data                                */
/*                      4.6  normalize_data                          */
/*                      4.7  randomize_records                       */
/*                      4.8  get_random_class_record                 */
/*                      4.9  get_random_record                       */
/*                      4.10 calulate_euclidean_distance_between     */
/*                             _inputs                               */
/*                      4.11 get_weights                             */
```

```
/*                      4.12 get_sigmas                                 */
/*                      4.13 get_outputs                                */
/*                                                                      */
/* Date: 11 Nov 90      Revision: 1.0                                   */
/************************************************************************/

#include "netvrble.h"
#include "netfnctn.h"
#include <time.h>
#include <stdlib.h>

/************************************************************************/
/* Function Name: load_input_patterns     Number: 4.1                   */
/* Description: This function determines wether the data should         */
/*              be loaded randomly from separate files, from a          */
/*              single file for by class                                */
/*                                                                      */
/* Functions Called: 4.2 load_separate_files                            */
/*                   4.3 load_from_single_file                          */
/*                   4.4 load_by_classes                                */
/*                                                                      */
/* Variables Passed In: training_data - Structure array                 */
/*                      test_data - Structure array                     */
/*                      train_set - Integer                             */
/*                      test_set - Integer                              */
/*                      dimension - Integer                             */
/*                      classes - Integer                               */
/*                      training_patterns_in_class - Integer array      */
/*                      randomization_rule - Integer                    */
/*                      data_seed - Unsigned                            */
/*                      *train_ptr - FILE pointer                       */
/*                      *test_ptr - FILE pointer                        */
/*                                                                      */
/* Variables Returned: *train_ptr - FILE pointer                        */
/*                     *test_ptr - FILE pointer                         */
/* Date: 11 Nov 90      Revision:  1.0                                  */
/************************************************************************/


void load_input_patterns(struct data *training_data[],
                         struct data *test_data[],
                         int train_set,
                         int test_set,
                         int dimension,
                         int classes,
                         int training_patterns_in_class[],
                         int randomization_rule,
                         unsigned data_seed,
                         FILE *train_ptr,
                                 FILE *test_ptr)

{
  int random_record[TRAIN_SET+TEST_SET];

    if (randomization_rule == 1)
    {
          load_separate_files (training_data,
                               test_data,
                               train_set,
                               test_set,
                               dimension,
                               random_record,
                               data_seed,
                               train_ptr,
                               test_ptr);
```

```
    }
    else if (randomization_rule == 2)
    {
        load_from_single_file (training_data,
                               test_data,
                               train_set,
                               test_set,
                               dimension,
                               random_record,
                               data_seed,
                               train_ptr);

    }
    else if (randomization_rule == 3)
    {
       load_by_classes (training_data,
                        test_data,
                        training_patterns_in_class,
                        random_record,
                        train_set,
                        test_set,
                        dimension,
                        classes,
                        data_seed,
                        train_ptr);
    }
   else
   {
        printf("\nerror in randomization rule");

   }
}

/*************************************************************/
/*    End Load Input patterns                                */
/*************************************************************/

/*************************************************************/
/*  Functions called by Load Input patterns                  */
/*************************************************************/

/*********************************************************************/
/* Function Name: load_separate_files    Number: 4.2                 */
/* Description: This function loads the input data from a separate */
/*              test and training file randomly.                     */
/*                                                                   */
/* Functions Called: 4.5 get_data                                    */
/*                   4.7 randomize_records                           */
/*                                                                   */
/* Variables Passed In: training_data - Structure array              */
/*                      test_data - Structure array                  */
/*                      train_set - Integer                          */
/*                      test_set - Integer                           */
/*                      dimension - Integer                          */
/*                      random_record - Integer array                */
/*                      data_seed - Unsigned                         */
/*                      *train_ptr - FILE pointer                    */
/*                      *test_ptr - FILE pointer                     */
/*                                                                   */
/* Variables Returned: training_data - Structure array               */
/*                     test_data - Structure array                   */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/
```

```
void load_separate_files (struct data *training_data[],
                          struct data *test_data[],
                          int train_set,
                          int test_set,
                          int dimension,
                          int random_record[],
                          unsigned seed,
                          FILE *train_ptr,
                          FILE *test_ptr)
{

     randomize_records(train_set,
                       random_record,
                       seed);

     get_data(training_data,
              train_set,
              dimension,
              train_ptr,
              random_record);


     randomize_records(test_set,
                       random_record,
                       seed);

     get_data(test_data,
              test_set,
              dimension,
              test_ptr,
              random_record);

}

/*********************************************************************/
/* Function Name: load_from_single_file     Number:4.3               */
/* Description: This function loads the training and test data       */
/*              randomly from a single file                          */
/*                                                                   */
/* Functions Called: 5.7 create_data_record                          */
/*                   4.5 get_data                                    */
/*                   4.7 randomize_records                           */
/*                                                                   */
/* Variables Passed In: training_data - Structure array              */
/*                      test_data - Structure array                  */
/*                      train_set - Integer                          */
/*                      test_set - Integer                           */
/*                      dimension - Integer                          */
/*                      random_record - Integer array                */
/*                      data_seed - Unsigned                         */
/*                      *train_ptr - FILE pointer                    */
/*                      *test_ptr - FILE pointer                     */
/*                                                                   */
/* Variables Returned: training_data - Structure array               */
/*                     test_data - Structure array                   */
/*                                                                   */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void load_from_single_file (struct data *training_data[],
                            struct data *test_data[],
                            int train_set,
                            int test_set,
                            int dimension,
                            int random_record[],
```

```
                        unsigned seed,
                        FILE *train_ptr)

{
   int x, y;
   int error = 0;
   struct data *temp_data[TEST_SET+TRAIN_SET];

   for (x = 0; x < train_set+test_set; x++)
   {
      create_data_record(temp_data,
                         x,
                         &error);
      if (error != 0)
      {
         printf("\n ** Out of memory for temp data ** \n");
      }
   }

   randomize_records(train_set + test_set,
                     random_record,
                     seed);

   get_data (temp_data,
             test_set + train_set,
             dimension,
             train_ptr,
             random_record);

    for (x = 0; x < train_set; x++)
      training_data[x] = temp_data[x];

    for (x = 0; x < test_set; x++)
    {
          y = train_set + x;
          test_data[x] = temp_data[y];
    }
}

/*************************************************************************/
/* Function Name: load_by_classes     Number: 4.4                        */
/* Description: This function loads a user selected number of            */
/*              training patterns for each class randomly from a          */
/*              single file.  The remaining patterns are loaded           */
/*              as test patterns                                          */
/*                                                                        */
/* Functions Called: 5.7 create_data_record                               */
/*                   4.5 get_data                                         */
/*                   4.7 randomize_records                                */
/*                   4.9 get_random_record                                */
/*                                                                        */
/* Variables Passed In: training_data - Structure array                   */
/*                      test_data - Structure array                       */
/*                      training_patterns_in_class - Integer array        */
/*                      random_record - Integer array                     */
/*                      train_set - Integer                               */
/*                      test_set - Integer                                */
/*                      dimension - Integer                               */
/*                      classes - Integer                                 */
/*                      data_seed - Unsigned                              */
/*                      *train_ptr - FILE pointer                         */
/*                                                                        */
/* Variables Returned: training_data - Structure array                    */
/*                     test_data - Structure array                        */
/*                                                                        */
```

```
/* Date: 11 Nov 90      Revision: 1.0                                  */
/*********************************************************************/

void load_by_classes(struct data *training_data[],
                     struct data *test_data[],
                     int training_patterns_in_class[],
                     int random_record[],
                     int train_set,
                     int test_set,
                     int dimension,
                     int classes,
                     unsigned seed,
                     FILE *fptr)
{
   struct data *temp_data[TRAIN_SET+TEST_SET];
   int number_in_class[CLASSES];
   int x, y, record;
   int error = 0;
   int class = 0;

   srand(seed);
   for (x = 0; x < classes+1; x++)
     number_in_class[x] = 0;

   for (x = 0; x < train_set + test_set; x++)
   {
      create_data_record(temp_data,
                         x,
                         &error);
      if (error != 0)
      {
         printf("\n ** Out of memory for temp data ** \n");
      }
   }

   randomize_records(train_set+test_set,
                     random_record,
                     seed);
   get_data(temp_data,
            train_set+test_set,
            dimension,
            fptr,
            random_record);
   x = 0;
   do
   {
      get_random_record(classes,
                        &class);

      class = class + 1;
      number_in_class[class] = number_in_class[class]+1;
      if (training_patterns_in_class[class]+1 > number_in_class[class])
      {
        do
          get_random_record(train_set+test_set,
                            &record);
        while(temp_data[record]->class != class);

        training_data[x] = temp_data[record];
        temp_data[record] = temp_data[train_set + test_set -1];
        train_set = train_set -1;
        x = x + 1;
      }
      else
        number_in_class[class] = training_patterns_in_class[class];
```

```
   }
   while (train_set > 0);

   for (x = 0; x < test_set; x++)
    test_data[x] = temp_data[x];
}

/*************************************************************/
/*      End Functions Called by Load Input Patterns          */
/*************************************************************/

/*********************************************************************/
/* Function Name: get_data    Number: 4.5                            */
/* Description: This function loads the data into an array given     */
/*              by the user.                                         */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: training_data, test_data - Structure array   */
/*                      train_set or test_set - Integer              */
/*                      dimension - Integer                          */
/*                      *train_ptr or *test_ptr - FILE pointer       */
/*                      random_record - Integer array                */
/*                                                                   */
/* Variables Returned: training_data, test_data - Structure array    */
/* Date: 11 Nov 90      Revision: 1.0  */
/*********************************************************************/

void get_data (struct data *data_record[],
               int record_no,
               int dimension,
               FILE *fptr,
               int random_record[])

{
   int x, y, record;
   float vector_data;
   int known_class;
   for (x = 0; x < record_no; x++)
   {
      record = random_record[x];
      for (y = 0; y < dimension; y++)
      {
         fscanf(fptr, "%f", &vector_data);
         data_record[record]->vector[y] = vector_data;
      }
      fscanf(fptr, "%d",&known_class);
      data_record[record]->class = known_class;
      data_record[record]->number = x;
   }
}
/*********************************************************************/
/* Function Name: normalize_data      Number: 4.6                    */
/* Description: This function energy normalizes each component of   */
/*              the input data by                                    */
/*              x(k) = sqrt{[x(k)^2]/||x||^2}                        */
/* Functions Called: None                                            */
/*                                                                   */
/* Variables Passed In: training_data, test_data - Structure array   */
/*                      train_set or test_set - Integer              */
/*                      dimension - Integer                          */
/*                                                                   */
/* Variables Returned: training_data, test_data - Structure array    */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/
```

```
void normalize_data (struct data *data_record[],
                     int record_no,
                     int dimension)

{
   double buffer;
   float distance;
   double exponent_2 = 2;
   double exponent_1 = .5;
   int x, y;
   for (x = 0; x < record_no; x++)
   {
      buffer = 0;
      for (y = 0; y < dimension; y++)
        buffer = buffer + pow((double) (data_record[x]->vector[y]),exponent_2);
      distance = pow(buffer, exponent_1);

      for (y = 0; y < dimension; y++)
        data_record[x]->vector[y] = data_record[x]->vector[y]/distance;

      buffer = 0;
      for ( y = 0; y < dimension; y++)
        buffer = buffer + pow((double) (data_record[x]->vector[y]),exponent_2);
      distance = pow(buffer,exponent_1);

   }
}

/*********************************************************************/
/* Function Name: randomize_records                  Number: 4.7     */
/* Description: This function returns an array containing random     */
/*              from 0 to the number of input patterns   numbers     */
/*                                                                   */
/* Functions Called: 4.9 get_random_record                           */
/*                                                                   */
/* Variables Passed In: train_set or test_set - Integer              */
/*                      random_record - Integer array                */
/*                      data_seed - Unsigned                         */
/*                                                                   */
/* Variables Returned: random_record - Integer array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void randomize_records(int max_no,
                       int random_record[],
                       unsigned seed)
{
  int x, record, temp[TRAIN_SET+TEST_SET];
  for (x = 0; x < max_no; x++)
  {
     random_record[x] = x;
     temp[x] = x;
  }

  x = 0;
  if (seed != 0)
  {
    srand(seed);
    do
    {
       get_random_record(max_no,
                         &record);

       random_record[x] = temp[record];
       temp[record] = temp[max_no -i];
```

```
      max_no = max_no-1;
      x = x + 1;
    }
    while(max_no > 0);
  }
}

/*********************************************************************/
/* Function Name: get_random_class_record    Number: 4.8             */
/* Description: This function returns the random class and record    */
/*              number for a pattern with that class                 */
/*                                                                   */
/* Functions Called: 4.9 get_random_record                           */
/* Variables Passed In: training or test_data - Structure array      */
/*                      train_set or test_set - Integer              */
/*                      classes - Integer                            */
/*                      *record - Integer pointer                    */
/*                                                                   */
/* Variables Returned: *record - Integer pointer                     */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void get_random_class_record(struct data *data_record[],
                             int max_record,
                             int max_classes,
                             int *record)

{
 int class;
 int record_number;
 get_random_record(max_classes,
                   &class);
  class = class + 1;
 do
   get_random_record(max_record,
                     &record_number);
 while(data_record[record_number]->class != class);
 *record = record_number;
}

/*********************************************************************/
/* Function Name: get_random_record    Number: 4.9                   */
/* Description: This function returns a random number between 0      */
/*              and maximum number -1                                */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: max_number - Integer                         */
/*                      *record - Integer pointer                    */
/*                                                                   */
/* Variables Returned: *record - Integer pointer                     */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

 void get_random_record(int max_number,
                        int *record)

{
   float x;
   int buffer;
   x = ((float)rand())/(32767* 65534);
   buffer = (x * max_number + .5);
   if (buffer > max_number -.5)
     *record = 0;
   else if (buffer < 0)
     *record = 0;
```

```
   else
     *record = buffer;
}

/*********************************************************************/
/* Function Name: calculate_euclidean_distance_between  Number:4.10*/
/*                _inputs                                            */
/* Description: This function calculates the distance between       */
/*              each of the input data records by                    */
/*              d(ij) = sqrt{sum[x(i)- x(j)]^2}                      */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: training or test_data - Structure array      */
/*                      train_set or test_set - Integer              */
/*                      dimension - Integer                          */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0  */
/*********************************************************************/

void calculate_euclidean_distance_between_inputs(struct data *data_record[],
                                                 int record_no,
                                                 int dimension,
                                                 FILE *fptr)
{
   float distance;
   double buffer = 0;
   double exponent_1 = .5;
   double exponent_2 = 2;
   int x, y, z;
   for (x = 0; x < record_no; x++)
     for (y = 0; y < record_no; y++)
     {
       buffer = 0;
       for (z = 0; z < dimension; z++)
         buffer = buffer + pow((double)(data_record[x]->vector[z] -
                                        data_record[y]->vector[z]),exponent_2);
       distance = pow(buffer,exponent_1);
       fprintf(fptr, "\n %d %d distance = %f classes = %d %d ",
                     x,y,distance,data_record[x]->class,
                     data_record[y]->class);
     }
}

/*********************************************************************/
/* Function Name: get_weights     Number: 4.11                       */
/* Description: This function reads the initial network weights      */
/*              from a file                                          */
/*                                                                   */
/* Functions Called:None                                             */
/* Variables Passed In:Node_record - Structure array                 */
/*                      *fptr - FILE pointer                         */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void get_weights (struct Node_data *node_record[],
                  FILE *fptr)

{
   int x, y;
   float weight;
   while(fscanf(fptr,"%d %d %f", &x, &y, &weight) != EOF)
      node_record[x]->weight[y] = weight;
```

```
}

/*********************************************************************/
/* Function Name: get_sigmas      Number: 4.12                       */
/* Description: This function reads the initial network sigmas       */
/*              from a file                                          */
/*                                                                   */
/* Functions Called:None                                             */
/* Variables Passed In:Node_record - Structure array                 */
/*                     *fptr - FILE pointer                          */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void get_sigmas(struct Node_data *node_record[],
                FILE *fptr)

{
   int x, y;
   float sigma;
   while(fscanf(fptr, "%d %f", &x, &y, &sigma) != EOF)
      node_record[x]->sigma[y] = sigma;
}

/*********************************************************************/
/* Function Name: get_outputs     Number: 4.13                       */
/* Description: This function reads the initial network outputs      */
/*              from a file                                          */
/*                                                                   */
/* Functions Called:None                                             */
/* Variables Passed In:Node_record - Structure array                 */
/*                     *fptr - FILE pointer                          */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void get_outputs(struct Node_data *node_record[],
                 FILE *fptr)

{
   int x;
   float output;
   while(fscanf(fptr, "%d %f",&x, &output) != EOF)
      node_record[x]->output = output;
}
```

G.5 NETINIT

```
/*********************************************************************/
/* Module Name: NETINIT.C    Number:5.0                              */
/* Description: This module provides the initialization routines     */
/*              for the nodes of a neural network                    */
/* Modules Called: None                                              */
/*                                                                   */
/* Functions Contained: 5.1 initialize_node_weights                  */
/*                      5.2 initialize_node_connections              */
/*                      5.3 initialize_node_sigmas                   */
/*                      5.4 initialize_node_outputs                  */
```

```
/*                      5.5 initialize_noe_transfer_function      */
/*                      5.6 create_node                           */
/*                      5.7 create_data_record                    */
/*                      5.8 disconnect_node                       */
/*                      5.9 correct_node_weights                  */
/*                                                                */
/* Date: 10 Nov 90      Revision: 1.0                             */
/******************************************************************/

#include "netvrble.h"
#include "netfnctn.h"

/******************************************************************/
/* Function Name: initialize_node_weights    Number: 5.1          */
/* Description: This function initializes the weights between     */
/*              connected nodes to the range -1 to 1.  For un-    */
/*              connected nodes, the weights are set to 0         */
/*                                                                */
/* Functions Called: None                                         */
/* Variables Passed In: Node_record - Structure array             */
/*                      total_nodes - Integer                     */
/*                      weight_seed - Unsigned                    */
/*                                                                */
/* Variables Returned:  Node_record - Structure array             */
/* Date: 11 Nov 90      Revision: 1.0                             */
/******************************************************************/

void initialize_node_weights (struct Node_data *node_record[],
                              int total_nodes,
                              unsigned seed)

{
  int x, y;
  float fanout = 0;
  double z = -1.0;
  double w = 1.0;

  srand(seed);
  for (x = 0; x < total_nodes; x++)
  {
    fanout  = 0;
    for (y = 0; y < total_nodes; y++)
      if (node_record[x]->connect[y] == 1)
         fanout = fanout + 1;
    for (y = 0; y < total_nodes; y++)
      if (node_record[x]->connect[y] == 1)
      {
        w = (double)rand();
        node_record[x]->weight[y] = (pow(z,w)*((float)rand())/(65534*32767));
      }
     else
        node_record[x]->weight[y] = 0;
   }
}

/******************************************************************/
/* Function Name: initialize_node_connections    Number: 5.2      */
/* Description: This function initalizes the connections between  */
/*              nodes which should be connected to 1.  The nodes  */
/*              which shouldn't be connected are set to 0.  These */
/*              connections are dependent on network topology     */
/*                                                                */
/* Functions Called: None                                         */
/* Variables Passed In: Node_record - Structure array             */
/*                      number_of_layers - Integer                */
```

```
/*                      nodes_in_layer - Integer array            */
/*                      starting_node_in_layer - Integer array    */
/*                      network_type - Integer                    */
/*                      total_nodes - Integer                     */
/*                                                                */
/* Variables Returned: Node_record - Structure array              */
/* Date: 11 Nov 90      Revision: 1.0                             */
/******************************************************************/

void initialize_node_connections(struct Node_data *node_record[],
                                 int number_of_layers,
                                 int nodes_in_layer[],
                                 int starting_node_layer[],
                                 int network_type,
                                 int total_nodes)

{
   int layer_no, x, y, layer;

   switch (network_type)
   {
      case 1:
        for (x = 0; x < total_nodes; x++)
          for ( y = 0; y < total_nodes; y++)
            node_record[x]->connect[y] = 0;

        for (layer_no = 1; layer_no < number_of_layers +1; layer_no++)
          for (x = starting_node_layer[layer_no]; x < starting_node_layer[layer_no]+nodes_in_layer[layer_no]; x++)
            for (layer = 0; layer < number_of_layers + 1; layer ++)
              for ( y = starting_node_layer[layer]; y < starting_node_layer[layer]+nodes_in_layer[layer]; y++)
                if (layer == layer_no -1)
                  node_record[x]->connect[y] = 1;
                else
                  node_record[x]->connect[y] = 0;
         break;

       default:
          printf("\n Error in network selection");
            break;
   }
}

/******************************************************************/
/* Function Name: initialize_node_sigmas     Number: 5.3          */
/* Description: This function initializes the node sigmas to a    */
/*              random value between 0 and 1                      */
/*                                                                */
/* Functions Called: None                                         */
/* Variables Passed In: Node_record - Structure array             */
/*                      total_nodes - Integer                     */
/*                      sigma_seed - Unsigned                     */
/*                                                                */
/* Variables Returned: Node_record - Structure array              */
/* Date: 11 Nov 90      Revision: 1.0                             */
/******************************************************************/

void initialize_node_sigmas(struct Node_data *node_record[],
                            int total_nodes,
                            unsigned seed)

{
   int x, y;
   srand(seed);
   for (x = 0; x < total_nodes; x++)
     for (y = 0; y < total_nodes; y++)
```

```
      node_record[x]->sigma[y] = (((float)rand())/(32767*65534));
}

/*********************************************************************/
/* Function Name: initialize_node_outputs    Number: 5.4             */
/* Description: This function initializes the outputs for all the    */
/*              nodes to 0.                                          */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      total_nodes - Integer                        */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void initialize_node_outputs(struct Node_data *node_record[],
                             int total_nodes)

{
   int node;
   for (node = 0; node < total_nodes; node++)
     node_record[node]->output = 0;
}

/*********************************************************************/
/* Function Name: initialize_node_transfer_function   Number:  5.5 */
/* Description: This function initializes the transfer function    */
/*              for each node in the network.  These transfer      */
/*              functions are depenedent on the layer the node     */
/*              is assigned.                                       */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: Node_record - Structure array              */
/*                      number_of_layers - Integer                 */
/*                      nodes_in_layer - Integer array             */
/*                      starting_node_in_layer - Integer array     */
/*                      transfer_function - Integer array          */
/*                                                                 */
/* Variables Returned: Node_record - Structure array               */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*********************************************************************/

void initialize_node_transfer_function(struct Node_data *node_record[],
                                       int number_of_layers,
                                       int nodes_in_layer[],
                                       int starting_node_layer[],
                                       int transfer_function[])

{
   int layer, node;
   for (layer = 0; layer < number_of_layers +1; layer++)
     for (node = starting_node_layer[layer]; node < starting_node_layer[layer]+nodes_in_layer[layer]; node++)
       node_record[node]->transfer_function = transfer_function[layer];
}

/*********************************************************************/
/* Function Name: create_node    Number: 5.6                         */
/* Description: This function creates a data structure for each      */
/*              node in the network                                  */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      new_node - Integer                           */
/*                      *error - Integer pointer                     */
```

```
/*                                                                    */
/* Variables Returned: *error - Integer pointer                       */
/* Date: 11 Nov 90      Revision: 1.0                                 */
/**********************************************************************/

void create_node(struct Node_data *node_record[],
                 int new_node,
                 int *error)

{
     if((node_record[new_node] = (struct Node_data *)malloc(sizeof(struct Node_data))) == NULL)
       *error = 1;
}

/**********************************************************************/
/* Function Name: create_data_record     Number: 5.7                  */
/* Description: This function creates a data structure for each       */
/*              training and test pattern used in the network         */
/*                                                                    */
/* Functions Called: None                                             */
/* Variables Passed In: Node_record - Structure array                 */
/*                      new_record - Integer                          */
/*                      *error - Integer pointer                      */
/*                                                                    */
/* Variables Returned: *error - Integer pointer                       */
/* Date: 11 Nov 90      Revision: 1.0                                 */
/**********************************************************************/

void create_data_record(struct data *data_record[],
                        int new_record,
                        int *error)
{
   if((data_record[new_record]= (struct data *)malloc(sizeof(struct data)))
                                                              ==NULL)
      *error = 1;
}

/**********************************************************************/
/* Function Name: disconnect_node     Number: 5.8                     */
/* Description: This function disconnects any nodes which are         */
/*              no longer required by the network                     */
/*                                                                    */
/* Functions Called: None                                             */
/* Variables Passed In: Node_record - Structure array                 */
/*                      current_node - Integer                        */
/*                      total_nodes - Integer                         */
/*                                                                    */
/* Variables Returned: Node_record - Structure array                  */
/* Date: 11 Nov 90      Revision: 1.0                                 */
/**********************************************************************/

void disconnect_node(struct Node_data *node_record[],
                     int current_node,
                     int total_nodes)
{
   int x;
   for (x = 0; x < total_nodes; x++)
   {
     node_record[x]->connect[current_node] = 0;
     node_record[current_node]->connect[x] = 0;
   }
}

/**********************************************************************/
/* Function Name: correct_node_weights     Number: 5.9                */
```

```
/* Description: This function ensures the weights always range     */
/*              between -100 and 100                               */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: Node_record - Structure array              */
/*                      total_nodes - Integer                      */
/*                                                                 */
/* Variables Returned: Node_record - Structure array               */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*******************************************************************/

void correct_node_weights(struct Node_data *node_record[],
                          int total_nodes)
{
   int x, y;
   for (x = 0; x < total_nodes; x++)
     for (y = 0; y < total_nodes; y++)
       if(node_record[x]->weight[y] > 0)
       {
         if(node_record[x]->weight[y] > 100)
           node_record[x]->weight[y] = 100;
         else if (node_record[x]->weight[y] < .0001)
           node_record[x]->weight[y] = .0001;
       }
       else
       {
          if(node_record[x]->weight[y] < -100)
            node_record[x]->weight[y] = -100;
          else if (node_record[x]->weight[y] > -.0001)
            node_record[x]->weight[y] = -.0001;
       }
  }
```

G.6 NETSHOW

```
/*******************************************************************/
/* Module Name: NETSHOW.C    Number: 6.0                           */
/* Description: This module contains the functions which display   */
/*              or file network data.                              */
/*                                                                 */
/* Modules Called: None                                            */
/* Functions Contained:                                            */
/*  6.1  file_data               6.2  file_randomization_rule      */
/*  6.3  file_seeds              6.4 file_net_topology             */
/*  6.5  file_transfer_functions 6.6 file_nodes_at_data_points_info*/
/*  6.7  file_center_at_avgs_data 6.8 file_k_means_data            */
/*  6.9  file_kohonen_data       6.10 file_MSE_data                */
/*  6.11 file_CFM_data           6.12 file_CE_data                 */
/*  6.13 file_matrix_data        6.14 file_parzen_window_data      */
/*  6.15 file_sigma_data         6.16 print_last_layer_output      */
/*  6.17 file_last_layer_output  6.18 print_data                   */
/*  6.19 file_data               6.20 print_node_data              */
/*  6.21 file_node_data          6.22 print_node_weights           */
/*  6.23 file_node_weights       6.24 print_node_sigma             */
/*  6.25 print_node_output       6.26 print_node_transfer_function */
/*  6.27 file_network_parameters 6.28 file_error_data              */
/*  6.29 file_class_count                                          */
/*                                                                 */
/* Date: 11 Nov 90      Revision: 1.0                              */
```

```
/*********************************************************************/

#include "netvrble.h"
#include "netfnctn.h"

/*********************************************************************/
/* Function Name: file_data_parameters     Number: 6.1               */
/* Description: This function files the data parameters such as      */
/*              name of data files, length, dimension and classes.   */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: train_file - Character array                 */
/*                      test_file - Character array                  */
/*                      train_set - Integer                          */
/*                      test_set - Integer                           */
/*                      dimension - Integer                          */
/*                      classes - Integer                            */
/*                      *fptr - File pointer                         */
/* Variables Returned:                                               */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_data_parameters(char train_file[],
                          char test_file[],
                          int train_set,
                          int test_set,
                          int dimension,
                          int classes,
                          FILE *fptr)

{
   fprintf(fptr,"\n Training file = %s",train_file);
   fprintf(fptr,"   Test file = %s", test_file);
   fprintf(fptr,"\n with %d  training vectors and %d  test vectors",
                  train_set,test_set);
   fprintf(fptr,"\n dimension = %d classes = %d",
                    dimension, classes);
   fprintf(fptr,"\n");
}

/*********************************************************************/
/* Function Name: file_randomization_rule     Number: 6.2            */
/* Description: This function files the method by which the data     */
/*              was loaded.                                          */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: randomization_rule - Integer                 */
/*                      training_patterns_in_class - Integer array   */
/*                      classes - Integer                            */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_randomization_rule(int randomization_rule,
                             int training_patterns_in_class[],
                             int classes,
                             FILE *fptr)
{
  int x;

  switch(randomization_rule)
  {
    fprintf(fptr,"\nrandomization rule is ");
```

```
    case 1:
        fprintf(fptr," load from separate files");
        break;

    case 2:
        fprintf(fptr," load from single files");
        break;

    case 3:
        fprintf(fptr,"load by class ");
        for (x = 1; x < classes + 1; x++)
          fprintf(fptr,"\n training patterns in class %d = %d",
                        x, training_patterns_in_class[x]);
    }
    fprintf(fptr,"\n");
}

/*********************************************************************/
/* Function Name: file_seeds                    Number: 6.3          */
/* Description: This function files the seeds used to randomly set   */
/*              network parameters and load the data.                */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: wght_seed - Unsigned                         */
/*                      sigma_seed - Unsigned                        */
/*                      data_seed - Unsigned                         */
/*                      record_seed - Unsigned                       */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_seeds(unsigned wght_seed,
                unsigned sigma_seed,
                unsigned data_seed,
                unsigned record_seed,
                FILE *fptr)

{
   fprintf(fptr,"\nweight seed = %u  sigma seed = %u  data seed = %u  record seed = %u",
                 wght_seed, sigma_seed, data_seed, record_seed);
    fprintf(fptr,"\n");
}

/*********************************************************************/
/* Function Name: file_net_topology             Number: 6.4          */
/* Description: This function files the topology of the network      */
/*              such as feedforward with number of layers            */
/*                                                                   */
/* Functions Calles: None                                            */
/* Variables Passed In: network_type - Integer                       */
/*                      number_of_nodes - Integer                    */
/*                      nodes_in_layer - Integer array               */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_net_topology(int network_type,
                       int number_of_layers,
                       int nodes_in_layer[],
                       FILE *fptr)
```

```
{
   int x;
   if(network_type == 1)
     fprintf(fptr,"\n network type = feedforward with number of layers = %d",
                                                    number_of_layers);
   for (x = 0; x < number_of_layers+1; x++)
     fprintf(fptr,"\n nodes in layer %d = %d",x, nodes_in_layer[x]);
   fprintf(fptr,"\n");

}

/*********************************************************************/
/* Function Name: file_tranfer_functions     Number: 6.5             */
/* Description: This function files the transfer function for each */
/*              node in the network.                                 */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: network_type - Integer                       */
/*                      number_of_layers - Integer                   */
/*                      starting_node_in_layer - Integer array       */
/*                      Node_record - Structure array                */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/


void file_transfer_functions(int network_type,
                             int number_of_layers,
                             int starting_node_in_layer[],
                             struct Node_data *node_record[],
                             FILE *fptr)

{
   int x;
   if (network_type == 1)
   {
      for (x = 0; x < number_of_layers+1; x++)
      switch(node_record[starting_node_in_layer[x]]->transfer_function)
      {
         case 1:
           fprintf(fptr,"\n layer %d transfer function = sigmoid",x);
           break;

         case 2:
           fprintf(fptr,"\n layer %d transfer function = rbf",x);
           break;

         case 3:
           fprintf(fptr,"\n layer %d transfer function = linear",x);
           break;

         default:
           break;
     }
   }
   fprintf(fptr,"\n");
}


/*********************************************************************/
/* Function Name: file_nodes_at_data_points_info      Number: 6.6    */
/* Description: This function files the parameters used to train     */
/*              a layer of nodes using the nodes_at_data_points      */
```

```
/*              algorithm                                              */
/*                                                                     */
/* Functions Called: None                                              */
/* Variables Passed In: current_layer - Integer                        */
/*                      output_threshold - Float                       */
/*                      sigma_threshold - Float                        */
/*                      *fptr - File pointer                           */
/*                                                                     */
/* Variables Returned: None                                            */
/* Date: 11 Nov 90      Revision: 1.0                                  */
/***********************************************************************/

void file_nodes_at_data_points_info(int layer,
                                    float output_threshold,
                                    float sigma_threshold,
                                    FILE *fptr)

{
  fprintf(fptr,"\nlayer %d nodes at the data points",layer);
  fprintf(fptr,"\n output threshold = %f sigma threshold = %f",
                                    output_threshold, sigma_threshold);
}

/***********************************************************************/
/* Function Name: file_center_at_class_avgs_data      Number: 6.7      */
/* Description: This function files the parameters used to train       */
/*              a layer of nodes using the center_at_class_avgs        */
/*              algorithm                                              */
/*                                                                     */
/* Functions Called: None                                              */
/* Variables Passed In: current_layer - Integer                        */
/*                      average_threshold - Float                      */
/*                      sigma_threshold - Float                        */
/*                      *fptr - File pointer                           */
/*                                                                     */
/* Variables Returned: None                                            */
/* Date: 11 Nov 90      Revision: 1.0                                  */
/***********************************************************************/

void file_center_at_class_avgs_data(int layer,
                                    float average_threshold,
                                    float sigma_threshold,
                                    FILE *fptr)
{
          fprintf(fptr,"\n layer %d center at class avgs",layer);
          fprintf(fptr,"\n average threshold = %f sigma threshold = %f",
                       average_threshold, sigma_threshold);
}

/***********************************************************************/
/* Function Name: file_k_means_data                   Number: 6.8      */
/* Description: This function files the parameters used to train       */
/*              a layer of nodes using the k_means_cluster             */
/*              algorithm                                              */
/*                                                                     */
/* Functions Called: None                                              */
/* Variables Passed In: current_layer - Integer                        */
/*                      clusters  - Integer                            */
/*                      *fptr - File pointer                           */
/*                                                                     */
/* Variables Returned: None                                            */
/* Date: 11 Nov 90      Revision: 1.0                                  */
/***********************************************************************/

void file_k_means_data(int layer,
```

```
                    int clusters,
                    FILE *fptr)
{
          fprintf(fptr,"\n layer %d K means cluster",layer);
          fprintf(fptr,"\n number of clusters = %d", clusters);
}

/*********************************************************************/
/* Function Name: file_kohonen_data                  Number: 6.9     */
/* Description: This function files the parameters used to train     */
/*              a layer of nodes using the train_via_kohonen         */
/*              algorithm                                            */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: current_layer - Integer                      */
/*                      nodes_x - Integer                            */
/*                      nodes_y - Integer                            */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_kohonen_data(int layer,
                    int nodes_x,
                    int nodes_y,
                    FILE *fptr)
{
          fprintf(fptr,"\nlayer %d Kohonen Training",layer);
          fprintf(fptr,"\n nodes in x direction = %d",nodes_x);
          fprintf(fptr,"\n nodes in y direction = %d", nodes_y);
}

/*********************************************************************/
/* Function Name: file_MSE_data                    Number: 6.10      */
/* Description: This function files the parameters used to train     */
/*              a the remaining layers of nodes using the            */
/*              MSE_remaining_layers algorithm                       */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: current_layer - Integer                      */
/*                      MSE_iterations - Integer                     */
/*                      MSE_error_delta - Float                      */
/*                      MSE_momentum - Float                         */
/*                      MSE_successes - Integer                      */
/*                      MSE_eta - Float                              */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_MSE_data(int layer,
                  int MSE_iterations,
                  float MSE_error_delta,
                  float MSE_momentum,
                  int MSE_successes,
                  float MSE_eta,
                  FILE *fptr)
{
   fprintf(fptr,"\nMSE layer %d and all others",layer);
   fprintf(fptr,"\n iterations = %d error delta = %f momemtum = %f",
                       MSE_iterations, MSE_error_delta, MSE_momentum);
   fprintf(fptr,"\nsuccess = %d eta = %f",MSE_successes,MSE_eta);
   fprintf(fptr,"\n");
```

```
}

/*********************************************************************/
/* Function Name: file_CFM_data                    Number: 6.11      */
/* Description: This function files the parameters used to train     */
/*              a the remaining layers of nodes using the            */
/*              CFM_remaining_layers algorithm                       */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: current_layer - Integer                      */
/*                      CFM_alpha - Float                            */
/*                      CFM_beta - Float                             */
/*                      CFM_eta - Float                              */
/*                      CFM_zeta - Float                             */
/*                      CFM_successes - Integer                      */
/*                      CFM_iterations - Integer                     */
/*                      CFM_momentum - Float                         */
/*                      CFM_delta - Float                            */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_CFM_data(int layer,
                   float CFM_alpha,
                   float CFM_beta,
                   float CFM_eta,
                   float CFM_zeta,
                   int CFM_successes,
                   int CFM_iterations,
                   float CFM_momentum,
                   float CFM_delta,
                   FILE *fptr)
{
   fprintf(fptr,"\nCFM layer %d and all others",layer);
   fprintf(fptr, "\nalpha = %f beta = %f eta = %f zeta = %f",
                  CFM_alpha, CFM_beta, CFM_eta, CFM_zeta);
   fprintf(fptr,"\nsuccesses = %d iterations = %d momentum = %f delta = %f",
                  CFM_successes, CFM_iterations, CFM_momentum, CFM_delta);
   fprintf(fptr,"\n");
}

/*********************************************************************/
/* Function Name: file_CE_data                     Number: 6.12      */
/* Description: This function files the parameters used to train     */
/*              a the remaining layers of nodes using the            */
/*              CE_remaining_layers algorithm                        */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: current_layer - Integer                      */
/*                      CE_epsilon - Float                           */
/*                      MSE_iterations - Integer                     */
/*                      MSE_momentum - Float                         */
/*                      MSE_eta - Float                              */
/*                      MSE_successes - Integer                      */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_CE_data(int layer,
                  float CE_epsilon,
                  int CE_iterations,
```

```
                float CE_momentum,
                float CE_eta,
                int CE_successes,
                FILE *fptr)
{
   fprintf(fptr,"\nCE layer %d and all others",layer);
   fprintf(fptr, "\neplison = %f iterations = %d momentum = %f",
                CE_epsilon, CE_iterations, CE_momentum);
   fprintf(fptr,"\neta = %f errors = %d",
                CE_eta, CE_successes);
   fprintf(fptr,"\n");
}

/*********************************************************************/
/* Function Name: file_matrix_data                    Number: 6.13 */
/* Description: This function files the parameters used to train   */
/*              a the remaining layers of nodes using the          */
/*              global_MSE_algorithm                               */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: current_layer - Integer                    */
/*                      *fptr - File pointer                       */
/*                                                                 */
/* Variables Returned: None                                        */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*********************************************************************/

void file_matrix_data(int layer,
                      FILE *fptr)
{
            fprintf(fptr,"\nLayer %d Linear by Matrix Inversion",layer);
}

/*********************************************************************/
/* Function Name: file_parzen_window_data          Number: 6.14    */
/* Description: This function files the parameters used to train   */
/*              a the remaining layers of nodes using the          */
/*              PNN_implementation algorithm                       */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: Node_record - Structure array              */
/*                      nodes_in_layer - Integer array             */
/*                      starting_node_in_layer - Integer array     */
/*                      current_layer - Integer                    */
/*                      *fptr - File pointer                       */
/*                                                                 */
/* Variables Returned: None                                        */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*********************************************************************/

void file_parzen_window_data(struct Node_data *node_record[],
                      int nodes_in_layer[],
                      int starting_node_in_layer[],
                      int layer,
                      FILE *fptr)

{
  int x, y, current_node, previous_layer_node;
  int total_nodes = 0;
  fprintf(fptr,"\nParzen window for layer %d",layer);
  for (x = 0; x < nodes_in_layer[layer]; x++)
  {
    current_node = starting_node_in_layer[layer]+x;
    fprintf(fptr,"\nnode %d with class %d connected to: \n",
        current_node, node_record[current_node]->class);
```

```
    total_nodes = 0;
    for (y = 0; y < nodes_in_layer[layer-1]; y++)
    {
      previous_layer_node = starting_node_in_layer[layer-1]+y;
      if(node_record[current_node]->connect[previous_layer_node] ==1)
      {
         fprintf(fptr,"%d ",previous_layer_node);
         total_nodes = total_nodes+1;
      }
    }
      fprintf(fptr,"\n total nodes for this node is %d ",total_nodes);
  }
}

/*********************************************************************/
/* Function Name: file_sigma_data                    Number: 6.15    */
/* Description: This function files the parameters used to train     */
/*              a the sigmas of an RBF network by any of the         */
/*              following algorithms                                 */
/*              a) scale_according_to_interference                   */
/*              b) set_sigmas_to_constant                            */
/*              c) set_sigma_at_P_neighbor_avg                       */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: current_layer - Integer                      */
/*                      sigma_rule - Integer                         */
/*                      interference_threshold - Float               */
/*                      sigma_factor - Float                         */
/*                      sigma_constant - Float                       */
/*                      p_neighbors - Integer                        */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_sigma_data(int layer,
                     int sigma_rule,
                     float interference_threshold,
                     float sigma_factor,
                     float sigma_constant,
                     int p_neighbors,
                     FILE *fptr)

{
      fprintf(fptr,"\nsigmas in layer %d",layer);
      switch(sigma_rule)
      {
        case 1:
             fprintf(fptr,"\nsigmas scaled by constant");
             fprintf(fptr,"interference threshold = %f sigma factor = %f",
                          interference_threshold, sigma_factor);
             break;
        case 2:
             fprintf(fptr,"\nerror, no training rule");
             break;

        case 3:
             fprintf(fptr,"\nsigmas set to a constant");
             fprintf(fptr," sigma constant = %f",sigma_constant);
             break;
        case 4:
             fprintf(fptr,"\n P neighbors");
             fprintf(fptr," p_neighors = %d",p_neighbors);
             break;
```

```
        default:
            break;
       }
    fprintf(fptr,"\n");
}

/*****************************************************************/
/* Function Name: print_last_layer_output          Number: 6.16  */
/* Description: This function prints the outputs of the last     */
/*              layer of the network, giving the node number     */
/*              node output.                                     */
/*                                                               */
/* Functions Called: None                                        */
/* Variables Passed In: Node_record - Structure array            */
/*                      nodes_in_layer - Integer array           */
/*                      starting_node_in_layer - Integer array   */
/*                      last_layer - Integer                     */
/*                                                               */
/* Variables Returned: None                                      */
/* Date: 11 Nov 90      Revision: 1.0                            */
/*****************************************************************/

void print_last_layer_output(struct Node_data *node_record[],
                             int nodes_in_layer[],
                             int starting_node_in_layer[],
                             int last_layer)
{
   int x, last_layer_node;
   for (x = 0; x < nodes_in_layer[last_layer]; x++)
   {
     last_layer_node = starting_node_in_layer[last_layer] + x;
     printf("\n node %d output = %f",last_layer_node
                                    ,node_record[last_layer_node]->output);
   }
}

/*****************************************************************/
/* Function Name: file_last_layer_output           Number: 6.17  */
/* Description: This function files  the outputs of the last     */
/*              layer of the network, giving the node number     */
/*              node output, data record number and class.       */
/*                                                               */
/* Functions Called: None                                        */
/* Variables Passed In: training or test_data - Structure array  */
/*                      Node_record - Structure array            */
/*                      record - Integer                         */
/*                      nodes_in_layer - Integer array           */
/*                      starting_node_in_layer - Integer array   */
/*                      last_layer - Integer                     */
/*                      *fptr - File pointer                     */
/*                                                               */
/* Variables Returned: None                                      */
/* Date: 11 Nov 90      Revision: 1.0                            */
/*****************************************************************/

void file_last_layer_output(struct data *data_record[],
                            struct Node_data *node_record[],
                            int record,
                            int nodes_in_layer[],
                            int starting_node_in_layer[],
                            int last_layer,
                            FILE *fptr)
{
   int x, last_layer_node;
```

```
   fprintf(fptr,"\n\n element %d ",record);
   fprintf(fptr,"is data record %d with class %d",data_record[record]->number
                                                 ,data_record[record]->class);
   for (x = 0; x < nodes_in_layer[last_layer]; x++)
   {
      last_layer_node = starting_node_in_layer[last_layer] + x;
      fprintf(fptr,"\n node %d output = %f",last_layer_node
                                          ,node_record[last_layer_node]->output);
    }
}

/*********************************************************************/
/* Function Name: print_data                          Number: 6.18   */
/* Description: This function prints the data of the input           */
/*              files as read by the software                        */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: training or test_data - Structure array      */
/*                      train_set or test_set - Integer              */
/*                      dimension - Integer                          */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void print_data(struct data *data_record[],
                int record_no,
                int dimension)

{
     int x, y;
     for (x =0; x < record_no; x++)
     {
       for (y = 0; y < dimension; y++)
          printf(" %lf",data_record[x]->vector[y]);
       printf("\n %d \n\n ", data_record[x]->class);
      }
 }

/*********************************************************************/
/* Function Name: file_data                           Number: 6.19   */
/* Description: This function files the data of the input            */
/*              files as read by the software                        */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: training or test_data - Structure array      */
/*                      train_set or test_set - Integer              */
/*                      dimension - Integer                          */
/*                      *fptr - File pointer                         */
/*                                                                   */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************** ***********************************************/

 void file_data(struct data *data_record[],
                int record_no,
                int dimension,
                FILE *fptr)

 {
    int x, y;
    for (x = 0; x < record_no; x++)
    {
       for (y = 0; y < dimension; y++)
```

```
            fprintf(fptr," %f",data_record[x]->vector[y]);
        fprintf(fptr,"\n %d \n \n ", data_record[x]->class);
    }
 }

/*********************************************************************/
/* Function Name: print_node_data                    Number: 6.20    */
/* Description: This function prints the data structure of each      */
/*              node in the network including the weights,           */
/*              sigmas, and the transfer functions                   */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      current_node - Integer                       */
/*                      total_nodes - Integer                        */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void print_node_data(struct Node_data *node_record[],
                     int current_node,
                     int total_nodes)

{
   int x;
   printf("\ndata for node %d",current_node);
   for (x = 0; x < total_nodes; x++)
    if (node_record[current_node]->connect[x] == 1)
      printf("\nweight %d = %f",x,node_record[current_node]->weight[x]);

   if (node_record[current_node]->transfer_function ==2)
     for (x = 0; x < total_nodes; x++)
     {
       if (node_record[current_node]->connect[x] == 1)
         printf("\nsigma = %f",node_record[current_node]->sigma[x]);
     }
   else
     printf("\n sigma %d = %f",current_node
                              ,node_record[current_node]->sigma[current_node]);
   printf("\ntransfer function = %d",node_record[current_node]->transfer_function);
}

/*********************************************************************/
/* Function Name: file_node_data                     Number: 6.21    */
/* Description: This function files the data structure of each       */
/*              node in the network including the weights,           */
/*              sigmas, and the transfer functions                   */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      current_node - Integer                       */
/*                      total_nodes - Integer                        */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_node_data(struct Node_data *node_record[],
                    int current_node,
                    int total_nodes,
                    FILE *fptr)
{
   int x;
```

```
   fprintf(fptr,"\ndata for node %d",current_node);
   for (x = 0; x < total_nodes; x++)
   {
     if (node_record[current_node]->connect[x] == 1)
     {
       fprintf(fptr,"\n wght %d = %f ",x,node_record[current_node]->weight[x]);
       if (node_record[current_node]->transfer_function == 2)
          fprintf(fptr," sigma %d = %f ",x,node_record[current_node]->sigma[x]);
     }
   }
     if (node_record[current_node]->transfer_function == 1)
      fprintf(fptr,"\nsigma %d = %f",current_node
                              ,node_record[current_node]->sigma[current_node]);

   fprintf(fptr,"\ntransfer function = %d",node_record[current_node]->transfer_function);
}

/*********************************************************************/
/* Function Name: print_node_weights              Number: 6.22       */
/* Description: This function prints the weights for each            */
/*              node in the network.                                 */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      current_node - Integer                       */
/*                      total_nodes - Integer                        */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void print_node_weights(struct Node_data *node_record[],
                        int current_node,
                        int total_nodes)
{
  int x;
  printf("\nNode %d",current_node);
  for (x = 0; x < total_nodes; x++)
    if (node_record[current_node]->connect[x] == 1)
      printf("\nweight %d = %f",x,node_record[current_node]->weight[x]);
}

/*********************************************************************/
/* Function Name: file_node_weights b             Number: 6.23       */
/* Description: This function files the weights for each             */
/*              node in the network.                                 */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      current_node - Integer                       */
/*                      total_nodes - Integer                        */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_node_weights(struct Node_data *node_record[],
                       int current_node,
                       int total_nodes,
                       FILE *fptr)
{
 int x;
 fprintf(fptr,"\nNode %d",current_node);
 for (x = 0; x < total_nodes; x++)
```

```
   if (node_record[current_node]->connect[x] == 1)
     fprintf(fptr,"\nweight %d = %f",x,node_record[current_node]->weight[x]);
}

/*********************************************************************/
/* Function Name: print_node_sigma                  Number: 6.24     */
/* Description: This function prints the sigmas for each             */
/*              node in the network.                                 */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      current_node - Integer                       */
/*                      total_nodes - Integer                        */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void print_node_sigma(struct Node_data *node_record[],
                      int current_node,
                      int total_nodes)

{
  int x;
  printf("\nNode %d",current_node);
  if (node_record[current_node]->transfer_function == 2)
    for (x = 0; x < total_nodes; x++)
    {
      if (node_record[current_node]->connect[x] == 1)
        printf("\n sigma %d = %f",x, node_record[current_node]->sigma[x]);
    }
  else
     printf("\n sigma %d = %f",current_node,
                              node_record[current_node]->sigma[current_node]);

}

/*********************************************************************/
/* Function Name: print_node_output                 Number: 6.25     */
/* Description: This function prints the output for a given          */
/*              node in the network.                                 */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      current_node - Integer                       */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void print_node_output(struct Node_data *node_record[],
                       int current_node)

{
   printf("\n Node %d output = %f",current_node, node_record[current_node]->output);
}

/*********************************************************************/
/* Function Name: print_node_transfer_function      Number: 6.26     */
/* Description: This function prints the transfer function  for a    */
/*              given node in the network.                           */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      current_node - Integer                       */
```

```
/*                      total_nodes - Integer                          */
/*                                                                     */
/* Variables Returned: None                                            */
/* Date: 11 Nov 90      Revision: 1.0                                  */
/***********************************************************************/

void print_node_transfer_function(struct Node_data *node_record[],
                                  int current_node)

{
   printf("\n Node %d transfer function = %d",current_node,node_record[current_node]->transfer_function);
}

/***********************************************************************/
/* Function Name: file_network_parameters          Number: 6.27        */
/* Description: This function files the network parameters,            */
/*              network type, number of layers, nodes in each          */
/*              layer, node transfer functions, training rules         */
/*              and the interconnection topology for the network       */
/*              nodes                                                  */
/*                                                                     */
/* Functions Called: None                                              */
/* Variables Passed In: Node_record - Structure array                  */
/*                      network_type - Integer                         */
/*                      number_of_layer - Integer                      */
/*                      nodes_in_layer - Integer array                 */
/*                      training_rule - Integer array                  */
/*                      transfer_function - Integer array              */
/*                      sigma_rule - Integer                           */
/*                      total_nodes - Integer                          */
/*                      *fptr - File pointer                           */
/*                                                                     */
/* Variables Returned: None                                            */
/* Date: 11 Nov 90      Revision: 1.0                                  */
/***********************************************************************/

void file_network_parameters(struct Node_data *node_record[],
                             int network_type,
                             int number_of_layers,
                             int nodes_in_layer[],
                             int training_rule[],
                             int transfer_function[],
                             int sigma_rule,
                             int total_nodes,
                             FILE *fptr)
{
   int x, y;
   fprintf(fptr,"\nNetwork Parameters");
   fprintf(fptr,"\nNetwork type = %d",network_type);
   fprintf(fptr,"\nNumber of layer = %d",number_of_layers);
   for (x = 0; x < number_of_layers + 1; x++)
   {
      fprintf(fptr,"\n nodes in layer %d = %d",x,nodes_in_layer[x]);
      fprintf(fptr,"  transfer function = %d",transfer_function[x]);
      fprintf(fptr,"  training rule = %d",training_rule[x]);
   }
   fprintf(fptr,"\n sigma rule = %d",sigma_rule);
   for (x = 0; x < total_nodes; x++)
   {
     fprintf(fptr,"\n\n node %d receives input from the following nodes\n",x);
     for (y = 0; y < total_nodes; y++)
     {
        if (node_record[x]->connect[y] == 1)
          fprintf(fptr,"%d  ",y);
     }
```

```
   }
}

/*********************************************************************/
/* Function Name: file_error_data                    Number: 6.28    */
/* Description: This function files the total number ofch            */
/*              errrors, the percentage correct, and the record      */
/*              misclassified                                        */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: class_error - Integer                        */
/*                      per_cent_correct - Float                     */
/*                      misclassifier - Integer Array                */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void file_error_data(int class_error,
                     float per_cent_correct,
                     int misclassified[],
                     FILE *fptr)
{
  int x;
  fprintf(fptr,"\n total errors = %d",class_error);
  fprintf(fptr,"\n per cent correct = %f",per_cent_correct);
  for (x = 0; x < class_error; x++)
    fprintf(fptr,"\n record %d misclassified ",misclassified[x]);
}

/*********************************************************************/
/* Function Name: file_class_count                   Number: 6.28    */
/* Description: This function files the number of training           */
/*              patterns from each class                             */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: training_patterns_in_class - Integer array   */
/*                      classes - Integer                            */
/*                      *fptr - File pointer                         */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/


void file_class_count (int training_patterns_in_class[],
                       int classes,
                       FILE *fptr)

{
   int x;
   for (x = 1; x < classes +1; x++)
     fprintf(fptr,"\n records in class %d = %d",x, training_patterns_in_class[x]);
}
```

G.7 NETOUT

```
/*********************************************************************/
/* Module Name: NETOUT.C                              Number: 7.0    */
/* Description: This module contains the functions which             */
/*              calculate the output for each node in the network    */
/*              due to a given input pattern                         */
/*                                                                   */
/* Modules Called: None                                              */
/* Functions Contained: 7.1 calculate_feed_forward_network_output    */
/*                      7.2 calculate_layer_0_output                 */
/*                      7.3 calculate_layer_1_output                 */
/*                      7.4 calculate_layer_2_output                 */
/*                      7.5 calculate_layer_3_output                 */
/*                      7.6 calculate_node_output                    */
/*                      7.7 calculate_output_as_input                */
/*                      7.8 calculate_linear_output                  */
/*                      7.9 calculate_rbf_output                     */
/*                      7.10 calculate_sigmoid_output                */
/*                                                                   */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

#include"netvrble.h"
#include"netfnctn.h"

/*********************************************************************/
/* Function Name: calculate_feed_forward_network_output  Number:7.1*/
/* Description: This function calculates the output of a feed        */
/*              forward network due to an input pattern              */
/*                                                                   */
/* Functions Called: 7.2 calculate_layer_0_output                    */
/*                   7.3 calculate_layer_1_output                    */
/*                   7.4 calculate_layer_2_output                    */
/*                   7.5 calculate_layer_3_output                    */
/*                                                                   */
/* Variables Passed In: training or test_data - Structure array      */
/*                      Node_record - Structure array                */
/*                      number_of_layers - Integer                   */
/*                      nodes_in_layer - Integer array               */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void calculate_feed_forward_network_output (struct data *data_record[],
                                            struct Node_data *node_record[],
                                            int number_of_layers,
                                            int nodes_in_layer[],
                                            int starting_node_in_layer[],
                                            int record,
                                            int total_nodes)

{
   int layer;
   for (layer = 0; layer < number_of_layers + 1; layer++)
   {
     switch (layer)
     {
       case 0:
         calculate_layer_0_output(data_record,
                                  node_record,
                                  nodes_in_layer,
```

```
                                record);
          break;

        case 1:
          calculate_layer_1_output(data_record,
                                node_record,
                                nodes_in_layer,
                                starting_node_in_layer,
                                total_nodes);
          break;

        case 2:
           calculate_layer_2_output(data_record,
                                node_record,
                                nodes_in_layer,
                                starting_node_in_layer,
                                total_nodes);

           break;

        case 3:
           calculate_layer_3_output(data_record,
                                node_record,
                                nodes_in_layer,
                                starting_node_in_layer,
                                total_nodes);
           break;

        default:
           printf("\nerror");
     }
   }
}

/*********************************************************************/
/* Function Name: calculate_layer_0_output    Number: 7.2            */
/* Description: This function calculates the output for each node    */
/*              in layer 0 of a feed forward network                 */
/*                                                                   */
/* Functions Called: 7.6 calculate_node_output                       */
/* Variables Passed In: training or test_data - Structure array      */
/*                      Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      record - Integer                             */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/


void calculate_layer_0_output (struct data *data_record[],
                               struct Node_data *node_record[],
                               int nodes_in_layer[],
                               int record)

{
   int node0;
   for (node0 = 0; node0 < nodes_in_layer[0]; node0++)
      calculate_node_output (data_record,
                             node_record,
                             node0,
                             record);
}

/*********************************************************************/
```

```
/* Function Name: calculate_layer_1_output    Number: 7.3          */
/* Description: This function calculates the output for each node  */
/*              in layer 1 of a feed forward network               */
/*                                                                 */
/* Functions Called: 7.6 calculate_node_output                     */
/* Variables Passed In: training or test_data - Structure array    */
/*                      Node_record - Structure array              */
/*                      nodes_in_layer - Integer array             */
/*                      starting_node_in_layer - Integer array     */
/*                      total_nodes - Integer                      */
/*                                                                 */
/* Variables Returned: Node_record - Structure array               */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*******************************************************************/

void calculate_layer_1_output (struct data *data_record[],
                               struct Node_data *node_record[],
                               int nodes_in_layer[],
                               int starting_node_layer[],
                               int total_nodes)

{
   int node1,current_node;
   for (node1 = 0; node1 < nodes_in_layer[1]; node1++)
   {
      current_node = starting_node_layer[1] + node1;
      calculate_node_output (data_record,
                             node_record,
                             current_node,
                             total_nodes);
   }
}

/*******************************************************************/
/* Function Name: calculate_layer_2_output    Number: 7.4          */
/* Description: This function calculates the output for each node  */
/*              in layer 2 of a feed forward network               */
/*                                                                 */
/* Functions Called: 7.6 calculate_node_output                     */
/* Variables Passed In: training or test_data - Structure array    */
/*                      Node_record - Structure array              */
/*                      nodes_in_layer - Integer array             */
/*                      starting_node_in_layer - Integer array     */
/*                      total_nodes - Integer                      */
/*                                                                 */
/* Variables Returned: Node_record - Structure array               */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*******************************************************************/

void calculate_layer_2_output (struct data *data_record[],
                               struct Node_data *node_record[],
                               int nodes_in_layer[],
                               int starting_node_layer[],
                               int total_nodes)

{
   int node2, current_node;
   for (node2 = 0; node2 < nodes_in_layer[2]; node2++)
   {
      current_node = starting_node_layer[2] + node2;
      calculate_node_output (data_record,
                             node_record,
                             current_node,
                             total_nodes);
   }
```

```
}

/*********************************************************************/
/* Function Name: calculate_layer_3_output    Number: 7.5            */
/* Description: This function calculates the output for each node    */
/*              in layer 3 of a feed forward network                 */
/*                                                                   */
/* Functions Called: 7.6 calculate_node_output                       */
/* Variables Passed In: training or test_data - Structure array      */
/*                      Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      total_nodes - Integer                        */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void calculate_layer_3_output (struct data *data_record[],
                               struct Node_data *node_record[],
                               int nodes_in_layer[],
                               int starting_node_layer[],
                               int total_nodes)


{
   int node3, current_node;
   for (node3 = 0; node3 < nodes_in_layer[3]; node3++)
   {
      current_node = starting_node_layer[3] + node3;
      calculate_node_output (data_record,
                             node_record,
                             current_node,
                             total_nodes);
   }
}

/*********************************************************************/
/* Function Name: calculate_node_output       Number: 7.6            */
/* Description: This function calculates the output for a single     */
/*              node in a network by testing the transfer            */
/*              function and sending control to the appropriate      */
/*              function                                             */
/*                                                                   */
/* Functions Called: 7.7 calculate_output_as_input                   */
/*                   7.8 calculate_linear_output                     */
/*                   7.9 calculate_rbf_output                        */
/*                   7.10 calculate_sigmoid_output                   */
/*                                                                   */
/* Variables Passed In: training or test_data - Structure array      */
/*                      Node_record - Structure array                */
/*                      current_node - Integer array                 */
/*                      total_nodes - Integer                        */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/


void calculate_node_output(struct data *data_record[],
                           struct Node_data *node_record[],
                           int current_node,
                           int total_nodes)

{
```

```
   switch (node_record[current_node]->transfer_function)
   {

      case 0:
        calculate_output_as_input (data_record,
                                   node_record,
                                   current_node,
                                   total_nodes);

        break;

      case 1 :
         calculate_sigmoid_output(node_record,
                                  current_node,
                                  total_nodes);
         break;

      case 2 :
         calculate_rbf_output(node_record,
                              current_node,
                              total_nodes);

         break;

      case 3 :
         calculate_linear_output(node_record,
                                  current_node,
                                  total_nodes);

         break;
      default:
         break;
   }
}

/*********************************************************************/
/* Function Name: calculate_output_as_input   Number: 7.7            */
/* Description: This function calculates the output for node         */
/*              with the identity transfer function as               */
/*              y(out) = x(in)                                       */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: training or test_data - Structure array      */
/*                      Node_record - Structure array                */
/*                      current_node - Integer                       */
/*                      record - Integer                             */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 11 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void calculate_output_as_input(struct data *data_record[],
                               struct Node_data *n_record[],
                               int node,
                               int record)
{
   n_record[node]->output = data_record[record]->vector[node];
   n_record[node]->class = data_record[record]->class;
}

/*********************************************************************/
/* Function Name: calculate_linear_output     Number: 7.8            */
/* Description: This function calculates the output for node         */
/*              with the linear transfer function as                 */
/*              y(l) = sum[w(kl)x(k)]                                */
```

```
/*                                                                    */
/* Functions Called: None                                             */
/* Variables Passed In: Node_record - Structure array                 */
/*                      current_node - Integer                        */
/*                      total_nodes - Integer                         */
/*                                                                    */
/* Variables Returned: Node_record - Structure array                  */
/* Date: 11 Nov 90      Revision: 1.0                                 */
/**********************************************************************/

void calculate_linear_output(struct Node_data *node_record[],
                             int current_node,
                             int total_nodes)

 {
    int node;
    node_record[current_node]->output = 0;
    for(node = 0; node < total_nodes; node++)
     {
          node_record[current_node]->output = node_record[current_node]->output
                                    + node_record[current_node]->weight[node]
                                    * node_record[node]->output
                                    * node_record[current_node]->connect[node];

    }
 }

/**********************************************************************/
/* Function Name: calculate_rbf_output     Number: 7.9                */
/* Description: This function calculates the output for node          */
/*              with the linear transfer function as                  */
/*              y(1) = exp(-[1/2]sum{[x(k)-w(kl)]^2/@^2]              */
/*                                                                    */
/* Functions Called: None                                             */
/* Variables Passed In: Node_record - Structure array                 */
/*                      current_node - Integer                        */
/*                      total_nodes - Integer                         */
/*                                                                    */
/* Variables Returned: Node_record - Structure array                  */
/* Date: 11 Nov 90      Revision: 1.0                                 */
/**********************************************************************/

void calculate_rbf_output(struct Node_data *node_record[],
                          int current_node,
                          int total_nodes)

{
   double buffer =0;
   double weight_offset = 0;
   double x = 2.0;
   int node;
   for (node = 0; node < total_nodes; node++)
   {
       if(node_record[current_node]->connect[node] != 0)
       {
          weight_offset = (node_record[current_node]->weight[node]-
                           node_record[node]->output);
          buffer = buffer
                + pow((weight_offset/node_record[current_node]->sigma[node]),x);
        }
   }
   node_record[current_node]->output = exp(-(double)(buffer/2));
}

/**********************************************************************/
```

```
/* Function Name: calculate_sigmoid_output     Number: 7.10        */
/* Description: This function calculates the output for node       */
/*              with the sigmoidal transfer function as            */
/*              y(l) = 1/{1+exp[-sum[w(kl)x(k)]+@]}                */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: Node_record - Structure array              */
/*                      current_node - Integer                     */
/*                      total_nodes - Integer                      */
/*                                                                 */
/* Variables Returned: Node_record - Structure array               */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*******************************************************************/


void calculate_sigmoid_output(struct Node_data *node_record[],
                              int current_node,
                              int total_nodes)
{
   double buffer = 0;
   int node;
   for (node = 0; node < total_nodes; node++)
   {
      if (node_record[current_node]->connect[node] != 0)
        buffer = buffer + node_record[current_node]->weight[node]
                        * node_record[node]->output;
   }
   buffer = buffer + node_record[current_node]->sigma[current_node];
   node_record[current_node]->output = 1/(1 + exp(-buffer));
}
```

G.8 NETAUX

```
/*******************************************************************/
/* Module Name: NETAUX     Number: 8.0                             */
/* Description: This module contains the training functions called */
/*              by NETTRAIN                                        */
/*                                                                 */
/* Modules Called: None                                            */
/* Functions Contained: 8.1  determine Y matrix                    */
/*                      8.2  determine S matrix                    */
/*                      8.3  determine M matrix                    */
/*                      8.4  calculate weight matrix               */
/*                      8.5  MSE_last_layer                        */
/*                      8.6  MSE_last_layer_linear                 */
/*                      8.7  MSE_last_layer_sigmoid                */
/*                      8.8  MSE_mid_layer                         */
/*                      8.9  MSE_1st_layer                         */
/*                      8.10 calculate_errors_in_output            */
/*                      8.11 get_linear_training_eta               */
/*                      8.12 get_kohonen_neighborhood              */
/*                      8.13 calc_dist_outputs_to_nxt_lyr          */
/*                      8.14 find_nearest_element                  */
/*                      8.15 find_kohonen_boundaries               */
/*                      8.16 determine_neighborhood_elements       */
/*                      8.17 train_kohonen_weights                 */
/*                      8.18 find_distance_between_nodes           */
/*                      8.19 sort_2_dim_array                      */
/*                      8.20 CE_last_layer                         */
/*                      8.21 CE_mid_layer                          */
/*                      8.22 CE_first_layer                        */
```

```
/*                    8.23 calculate_zn                          */
/*                    8.24 CFM_last_layer                        */
/*                    8.25 CFM_mid_layer                         */
/*                    8.26 find_second_highest_node              */
/*                    8.27 CFM_first_layer                       */
/*                    8.28 find_nearest_neighbor                 */
/*                                                               */
/* Date: 10 Nov 90      Revision: 1.0                            */
/*****************************************************************/

#include "netvrble.h"
#include "netfnctn.h"

/*****************************************************************/
/* Function Name: determine_Y_matrix              Number:8.1     */
/* Description: This function calculates the outputs for each    */
/*              node in a given layer and stores these outputs in */
/*              a matrix.  Each row will represent the outputs for */
/*              that layer due to an input pattern               */
/* Functions Called: calculate_layer_0_output                    */
/*                   calculate_layer_1_output                    */
/*                   calculate_layer_2_output                    */
/*                   calculate_layer_3_output                    */
/*                                                               */
/* Variables Passed In: Node_record - Structure array            */
/*                      training_data - Structure array          */
/*                      train_set - integer                      */
/*                      nodes_in_layer - integer array           */
/*                      starting_node_in_layer - integer array   */
/*                      total_nodes - integer                    */
/*                      *Y - float array pointer                 */
/*                      current_layer - integer                  */
/*                                                               */
/* Variables Returned: *Y - float array pointer                  */
/* Date: 10 Nov 90     Revision:1.0                              */
/*****************************************************************/

void determine_Y_matrix(struct Node_data *N_record[],
                        struct data *data_record[],
                        int record_no,
                        int nodes_in_layer[],
                        int starting_node_in_layer[],
                        int total_nodes,
                        float *Y[],
                        int current_layer)

{
  int row, column, current_node;
  for (row = 0; row < record_no; row++)
  {
    switch(current_layer)
    {
    case 1:
       calculate_layer_0_output(data_record,
                                N_record,
                                nodes_in_layer,
                                row);
       break;

    case 2:
       calculate_layer_0_output(data_record,
                                N_record,
                                nodes_in_layer,
                                row);
```

```
      calculate_layer_1_output(data_record,
                               N_record,
                               nodes_in_layer,
                               starting_node_in_layer,
                               total_nodes);

      break;

   case 3:
      calculate_layer_0_output(data_record,
                               N_record,
                               nodes_in_layer,
                               row);

      calculate_layer_1_output(data_record,
                               N_record,
                               nodes_in_layer,
                               starting_node_in_layer,
                               total_nodes);

      calculate_layer_2_output(data_record,
                               N_record,
                               nodes_in_layer,
                               starting_node_in_layer,
                               total_nodes);

   default:
      break;
  }

   for (column = 0; column < nodes_in_layer[current_layer-1]; column++)
   {
      current_node = starting_node_in_layer[current_layer - 1] + column;
      *((Y[row])+column) = N_record[current_node]->output;
   }
  }
}

/*********************************************************************/
/* Function Name: determine_S_matrix    Number: 8.2                  */
/* Description: This function sets the desired outputs for the       */
/*              last layer to a 1 for the node responsible for the   */
/*              class and 0 for the remaining nodes                  */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: training_data - Structure array              */
/*                      train_set - Integer                          */
/*                      nodes_in_layer - Integer array               */
/*                      *S - Float array pointer                     */
/*                      current_layer - integer                      */
/*                                                                   */
/* Variables Returned: *S array                                      */
/* Date: 10 Nov 90     Revision: 1.0                                 */
/*********************************************************************/


void determine_S_matrix(struct data *data_record[],
                        int record_no,
                        int nodes_in_layer[],
                        float *S[],
                        int current_layer)
{
   int row, column;
   for (row = 0; row < record_no; row ++)
```

```
   for (column = 0; column < nodes_in_layer[current_layer]; column++)
    *((S[row])+column) = 0;
  for (row = 0; row < record_no; row ++)
  {
     column =  data_record[row]->class;
   *((S[row])+column-1) = 1;
  }
 }

/*********************************************************************/
/* Function Name: determine_M_matrix     Number: 8.3                 */
/* Description: This function determines the cross products of the */
/*              outputs from all the nodes in a given layer with a */
/*              specific node in that layer                        */
/*              M(lB) = sum[y(pl)y(pB)]                            */
/* Functions Called: None                                          */
/* Variables Passed In: *Y - Float array pointer                   */
/*                      *M - Float array pointer                   */
/*                      nodes_in_layer - Integer array             */
/*                      train_set - Integer                        */
/*                      current_layer - Integer                    */
/*                                                                 */
/* Variables Returned: *M - Float array pointer                    */
/* Date: 10 Nov 90      Revision: 1.0                              */
/*********************************************************************/

void determine_M_matrix(float *Y[],
                        float *M[],
                        int nodes_in_layer[],
                        int patterns,
                        int current_layer)
{
  int row, column, p;
   for (row = 0; row < nodes_in_layer[current_layer-1]; row++)
     for (column = 0; column < nodes_in_layer[current_layer-1]; column++)
     {
       *((M[row])+column) = 0;
       for (p = 0; p < patterns; p++)
         *((M[row])+column) = *((Y[p])+row) *  *((Y[p])+column) + *((M[row])+column);
     }
}

/*********************************************************************/
/* Function Name: calculate_weight_matrix      Number: 8.4           */
/* Description: This function calculates the output layer weights  */
/*              by w(BD) = sum{sum[y(pl)d(pD)]M(Bl)}               */
/* Functions Called: None                                          */
/* Variables Passed In: Node_record - Structure array              */
/*                      *W - float array pointer                   */
/*                      *M - float array pointer                   */
/*                      *Y - float array pointer                   */
/*                      *S - float array pointer                   */
/*                      nodes_in_layer - Integer array             */
/*                      starting_node_in_layer - Integer array     */
/*                      train_set - Integer                        */
/*                      current_layer - Integer                    */
/*                                                                 */
/* Variables Returned: *W - float array pointer                    */
/* Date: 10 Nov 90      Revision: 1.0                              */
/*********************************************************************/

 void calculate_weight_matrix(struct Node_data *M_record[],
                              float *W[],
                              float *M[],
                              float *Y[],
```

```
                          float *S[],
                          int nodes_in_layer[],
                          int starting_node_in_layer[],
                          int patterns,
                          int current_layer)
{
   float sum = 0;
   int row, column,P, L, current_node, previous_layer_node;
   for (row = 0; row < nodes_in_layer[current_layer-1]; row++)
   {
     previous_layer_node = starting_node_in_layer[current_layer-1] + row;
     for (column = 0; column < nodes_in_layer[current_layer]; column++)
     {
       *((W[row])+column) = 0;
       for (L = 0; L < nodes_in_layer[current_layer-1]; L++)
       {
          sum = 0;
          for (P = 0; P < patterns; P++)
            sum = sum + *((Y[P])+L) * *((S[P])+column);
          *((W[row])+column) = *((W[row])+column) + *((H[row])+L) * sum;
       }
       current_node = starting_node_in_layer[current_layer] + column;
       H_record[current_node]->weight[previous_layer_node] = *((W[row])+column);
     }
   }
}

/*************************************************************/
/** End Functions Called by Optimize Weights by Matrix ***/
/*************************************************************/

/*********************************************************************/
/* Function Name: MSE_last_layer    Number: 8.5                      */
/* Description: The function determines wether to backpropate the    */
/*              parameter by the sigmoidal or linear update          */
/*              equations                                            */
/*                                                                   */
/* Functions Called: 8.6 MSE_last_layer_linear                       */
/*                   8.7 MSE_last_layer_sigmoid                      */
/*                                                                   */
/* Variables Passed In: Node_record - Structure array                */
/*                      desired_output - Float array                 */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      last_layer - Integer                         */
/*                      MSE_eta - Float                              */
/*                      MSE_epsilon - Float                          */
/*                      *wght - Float array pointer                  */
/*                      MSE_momentem - Float                         */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 10 Nov 90      Revision: 1.0                                */
/*********************************************************************/

 void MSE_last_layer(struct Node_data *node_record[],
                     float desired_output[],
                     int nodes_in_layer[],
                     int starting_node_in_layer[],
                     int last_layer,
                     float eta,
                     float epsilon,
                     float *wght[],
                     float alpha)
 {
    int x, y, last_layer_node, previous_layer_node;
```

```
   for (x = 0; x < nodes_in_layer[last_layer]; x++)
   {
      last_layer_node = starting_node_in_layer[last_layer]+x;
      switch (node_record[last_layer_node]->transfer_function)
      {
         case 1:
              MSE_last_layer_sigmoid(node_record,
                                     desired_output,
                                     nodes_in_layer,
                                      starting_node_in_layer,
                                      last_layer,
                                      last_layer_node,
                                      eta,
                                      wght,
                                      alpha);
               break;
          case 2:
              /* reserved for rbf */
          printf("\n error, improper transfer function");
              break;
          case 3:
              MSE_last_layer_linear(node_record,
                                      desired_output,
                                      nodes_in_layer,
                                      starting_node_in_layer,
                                      last_layer,
                                      last_layer_node,
                                      eta,
                                      wght,
                                      alpha);
              break;
             default:
              break;
        }
    }
 }

/*************************************************************************/
/* Function Name: MSE_last_layer_linear    Number: 8.6                   */
/* Description: This function implements the following update rule       */
/*              for a node  forming a linear combination of the          */
/*              outputs from the previous layer:                         */
/*              w+ = w- + eta * (d-z) * y                                */
/*              @+ = @- + eta * (d-z)                                    */
/*   where w+ - next weight                                              */
/*         eta - training factor (.01 - .99)                             */
/*         d - desired output                                            */
/*         z - actual output of the last layer node                      */
/*         y - actual output of the node in the previous layer           */
/*         @ - the sigma (theta) for that node                           */
/*                                                                       */
/*  Functions Called: None                                               */
/*  Variables Passed In: Node_record - Structure Array                   */
/*                       desired_output - Float array                    */
/*                       nodes_in_layer - Integer array                  */
/*                       starting_node_in_layer - Integer array          */
/*                       number_of_layers - Integer                      */
/*                       total_nodes - Integer                           */
/*                       MSE_eta - Float                                 */
/*                       *wght - Float array pointer                     */
/*                       MSE_momentum - Float                            */
/*                                                                       */
/*  Variables Returned: Node_record - Structure Array                    */
/*  Date: 10 Nov 90     Revision: 1.0                                    */
/*************************************************************************/
```

```
void MSE_last_layer_linear(struct Node_data *node_record[],
                           float desired_output[],
                           int nodes_in_layer[],
                           int starting_node_in_layer[],
                           int last_layer,
                           int last_layer_node,
                           float eta,
                  float *wght[],
                  float alpha)
{
   float delta_1, old_wght;
   int y, previous_layer_node;
   int x = 0;

   x = last_layer_node - starting_node_in_layer[last_layer];
   delta_1 = desired_output[x] - node_record[last_layer_node]->output;
   for (y = 0; y < nodes_in_layer[last_layer-1]; y++)
   {
     previous_layer_node = starting_node_in_layer[last_layer-1]+y;
     old_wght = *((wght[last_layer_node])+previous_layer_node);
     *((wght[last_layer_node])+previous_layer_node) =
                  node_record[last_layer_node]->weight[previous_layer_node];

     node_record[last_layer_node]->weight[previous_layer_node] =
      node_record[last_layer_node]->weight[previous_layer_node]
      + eta * delta_1 * node_record[previous_layer_node]->output
      + alpha * (node_record[last_layer_node]->weight[previous_layer_node]
            - old_wght);

   }
   old_wght = *((wght[last_layer_node])+last_layer_node);
   *((wght[last_layer_node])+last_layer_node) =
                     node_record[last_layer_node]->sigma[last_layer_node];
   node_record[last_layer_node]->sigma[last_layer_node] =
     node_record[last_layer_node]->sigma[last_layer_node] +
     eta * delta_1
    + alpha * (node_record[last_layer_node]->sigma[last_layer_node]
            - old_wght);
}

/*********************************************************************/
/* Function Name: MSE_last_layer_sigmoid      Number: 8.7            */
/* Description:  This function implements the update rule for a      */
/*               sigmoidal transfer function in the last layer:      */
/*   w+ = w- + eta * (d - z) * (1-z) * z * y                         */
/*   @+ = @- + eta * (d - z) * (1-z) * z                             */
/*   where                                                           */
/*   w - weight between node in last layer and previous layer        */
/*   @ - sigma or theta of last layer node                           */
/*   eta - training coefficient                                      */
/*   d - desired output for the last layer node                      */
/*   z - actual output for the last layer node                       */
/*   y - actual output for the previous layer node                   */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure Array                */
/*                      desired_output - Float array                 */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_layer    - Integer                   */
/*                      current_node - Integer                       */
/*                      MSE_eta - Float                              */
/*                      *wght - Float array pointer                  */
/*                      MSE_momentum - Float                         */
```

```
/*                                                                    */
/* Variables Returned: Node_record - Structure Array                  */
/* Date: 10 Nov 90       Revision: 1.0                                */
/**********************************************************************/


void MSE_last_layer_sigmoid(struct Node_data *node_record[],
                            float desired_output[],
                            int nodes_in_layer[],
                            int starting_node_in_layer[],
                            int current_layer,
                            int current_node,
                            float eta,
                            float *wght[],
                            float alpha)
{
  float delta_1, delta_2, old_wght;
  int y, previous_layer_node, x;

  x = current_node - starting_node_in_layer[current_layer];
  delta_1 = desired_output[x]-node_record[current_node]->output;
  delta_2 = (1-node_record[current_node]->output)
          * node_record[current_node]->output;
  for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
  {
  previous_layer_node = starting_node_in_layer[current_layer-1] + y;
  old_wght = *((wght[current_node])+previous_layer_node);
  *((wght[current_node])+previous_layer_node) =
                      node_record[current_node]->weight[previous_layer_node];

  node_record[current_node]->weight[previous_layer_node] =
       node_record[current_node]->weight[previous_layer_node]
       + eta * delta_1 * delta_2 * node_record[previous_layer_node]->output
       + alpha * (node_record[current_node]->weight[previous_layer_node]
             - old_wght);

  }
  old_wght = *((wght[current_node])+current_node);
  *((wght[current_node])+current_node) =
                      node_record[current_node]->sigma[current_node];
  node_record[current_node]->sigma[current_node] =
     node_record[current_node]->sigma[current_node] +
     eta * delta_1 * delta_2
     + alpha * (node_record[current_node]->sigma[current_node]
             - old_wght);
}

/**********************************************************************/
/* Function Name: MSE_mid_layer     Number: 8.8                       */
/* Description:  This function implements update rule for sigmoidal   */
/*   transfer function in the middle layer and last layer:            */
/*   w''+ = w''- + eta * sum(d - z)* w' * (1-z) * z * y * (1-y) * x   */
/*   @+ = @- + eta * sum(d - z) * w' * (1-z) * z * y * (1-y)          */
/*   where                                                            */
/*   w - weight between node in last layer and previous layer         */
/*   @ - sigma or theta of last layer node                            */
/*   w'- weight linking node in layer 3 to node in layer 2            */
/*   w'' - weight linking node in layer 2 to node in layer 1          */
/*   eta - training coefficient                                       */
/*   d - desired output for the last layer node                       */
/*   z - actual output for the last layer node                        */
/*   y - actual output for the layer 2 node                           */
/*   x - actual output for the layer 1 node                           */
/*                                                                    */
/*  Functions Called: None                                            */
```

```
/*  Variables Passed Into: Node_record - Structure Array              */
/*                         desired_output - Float array               */
/*                         nodes_in_layer - Integer array             */
/*                         starting_node_in_layer - Integer array     */
/*                         current_layer - Integer                    */
/*                         MSE_eta - Float                            */
/*                         *wght - Float array pointer                */
/*                         MSE_momentum - Float                       */
/*                                                                    */
/*  Variables Returned: Node_record - Structure Array                 */
/*  Date: 10 Nov 90       Revision: 1.0                               */
/**********************************************************************/


void MSE_mid_layer(struct Node_data *node_record[],
                   float desired[],
                   int nodes_in_layer[],
                   int starting_node_in_layer[],
                   int current_layer,
                   float eta,
                   float *wght[],
                   float alpha)

{
   int z, last_layer_node, y, previous_layer_node, current_node;
   int x;
   float sum, delta_1, delta_2, old_wght;
   sum = 0;

  for (x = 0; x < nodes_in_layer[current_layer]; x++)
  {
    sum = 0;
    current_node = starting_node_in_layer[current_layer] +x;
    for (z = 0; z < nodes_in_layer[current_layer+1]; z++)
    {
      last_layer_node = starting_node_in_layer[current_layer+1] + z;
      delta_1 = desired[z] - node_record[last_layer_node]->output;
      delta_2 = (1 - node_record[last_layer_node]->output)
               * node_record[last_layer_node]->output
               * node_record[last_layer_node]->weight[current_node];
      sum = sum + delta_1 * delta_2;
     }
   delta_1 = (1 - node_record[current_node]->output)
            * node_record[current_node]->output;

   for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
   {
     previous_layer_node = starting_node_in_layer[current_layer-1] + y;
      old_wght = *((wght[current_node])+previous_layer_node);
     *((wght[current_node])+previous_layer_node) =
                       node_record[current_node]->weight[previous_layer_node];

     node_record[current_node]->weight[previous_layer_node] =
               node_record[current_node]->weight[previous_layer_node]
             + eta * delta_1 * sum * node_record[previous_layer_node]->output
          + alpha * (node_record[current_node]->weight[previous_layer_node]
             - old_wght);

   }
   old_wght = *((wght[current_node])+current_node);
     *((wght[current_node])+current_node) =
                       node_record[current_node]->sigma[current_node];
   node_record[current_node]->sigma[current_node] =
     node_record[current_node]->sigma[current_node] +
     eta * delta_1 * sum
```

```
    + alpha * (node_record[current_node]->sigma[current_node]
          - old_wght);
 }
}
/*************************************************************************/
/* Function Name: MSE_1st_layer     Number: 8.9                          */
/* Description:  This function implements update rule for sigmoidal      */
/*   transfer function in the first, middle layer and last layer:        */
/*   w''' = w''' + eta*sum(d-z)*(1-z)*z*sum(w*y*(1-y)*w'')*x*(1-x)*a     */
/*   @+ = @- + eta * sum(d - z)*(1-z)*z*sum(w*y*(1-y)*w'')*x*(1-x)       */
/*   where                                                               */
/*   w - weight between node in last layer and previous layer            */
/*   @ - sigma or theta of last layer node                               */
/*   w'- weight linking node in layer 3 to node in layer 2               */
/*   w'' - weight linking node in layer 2 to node in layer 1             */
/*   w''' - weight linking node in layer 1 to node in layer 0            */
/*   eta - training coefficient                                          */
/*   d - desired output for the last layer node                          */
/*   z - actual output for the last layer node                           */
/*   y - actual output for the layer 2 node                              */
/*   x - actual output for the layer 1 node                              */
/*   a - actual output for the layer 0 node                              */
/*                                                                       */
/*  Variables Passed into: Node_record - Structure Array                 */
/*                         desired_output - Float array                  */
/*                         nodes_in_layer - Integer array                */
/*                         starting_node_in_layer - Integer array        */
/*                         current_layer - Integer                       */
/*                         MSE_eta - Float                               */
/*                         *wght - Float array pointer                   */
/*                         MSE_momentum - Float                          */
/*                                                                       */
/*  Variables Returned: Node_record - Structure Array                    */
/*  Date: 10 Nov 90     Revision: 1.0                                    */
/*************************************************************************/

void MSE_1st_layer(struct Node_data *node_record[],
                           float desired_output[],
                           int nodes_in_layer[],
                           int starting_node_in_layer[],
                           int current_layer,
                           float eta,
                           float *wght[],
                           float alpha)

{
   int next_layer_node, last_layer_node, y, z, previous_layer_node;
   int current_node, x;
   float sum_1, sum_2, delta_1, delta_2, delta_3, delta_4;
   float old_wght;
   sum_1 = 0;

 for (x = 0; x < nodes_in_layer[current_layer]; x++)
 {
   sum_1 = 0;
   current_node = starting_node_in_layer[current_layer] + x;
   for ( z = 0; z < nodes_in_layer[current_layer+2]; z++)
   {
     last_layer_node = starting_node_in_layer[current_layer+2] + z;
     delta_1 = desired_output[z]-node_record[last_layer_node]->output;
     delta_2 = (1-node_record[last_layer_node]->output)
               * node_record[last_layer_node]->output;
     sum_2 = 0;
     for (y = 0; y < nodes_in_layer[current_layer + 1]; y++)
     {
```

```
        next_layer_node = starting_node_in_layer[current_layer + 1] + y;
        delta_3 = (1-node_record[next_layer_node]->output)
                  * node_record[next_layer_node]->output;
        delta_4 = node_record[last_layer_node]->weight[next_layer_node]
                  * node_record[next_layer_node]->weight[current_node];
        sum_2 = sum_2 + delta_3 * delta_4;
      }
      sum_1 = sum_1 + delta_1 * delta_2 * sum_2;
    }
    delta_1 = (1 - node_record[current_node]->output)
              * node_record[current_node]->output;
    for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
    {
      previous_layer_node = starting_node_in_layer[current_layer-1] + y;
      old_wght = *((wght[current_node])+previous_layer_node);
      *((wght[current_node])+previous_layer_node) =
                        node_record[current_node]->weight[previous_layer_node];
      node_record[current_node]->weight[previous_layer_node] =
         node_record[current_node]->weight[previous_layer_node]
         + eta * delta_1 * sum_1 * node_record[previous_layer_node]->output
      + alpha * (node_record[current_node]->weight[previous_layer_node]
              - old_wght);
    }
     old_wght = *((wght[current_node])+current_node);
      *((wght[current_node])+current_node) =
                        node_record[current_node]->sigma[current_node];

    node_record[current_node]->sigma[current_node] =
      node_record[current_node]->sigma[current_node] +
      eta * delta_1 * sum_1
      + alpha * (node_record[current_node]->sigma[current_node]
              - old_wght);
  }
}

/*********************************************************************/
/* Function Name:calculate_errors_in_output     Number: 8.10         */
/* Description: This function compares the output of each node in    */
/*              in the output layer with the desired output.         */
/*              If the difference between the desired and the        */
/*              actual is greater than some delta, an error          */
/*              is returned                                          */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure Array                */
/*                      desired_output - Float array                 */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      number_of_layers - Integer                   */
/*                      *error - Integer pointer                     */
/*                      epsilon - Float                              */
/*                                                                   */
/* Variables Returned: *error - Integer pointer                      */
/* Date: 10 Nov 90     Revision: 1.0                                 */
/*********************************************************************/

void calculate_errors_in_output(struct Node_data *node_record[],
                          float desired_output[],
                          int nodes_in_layer[],
                          int starting_node_in_layer[],
                          int number_of_layers,
                                  int *error,
                          float epsilon)
{
  int x, last_layer_node;
  for (x = 0; x < nodes_in_layer[number_of_layers]; x++)
```

```
  {
    last_layer_node = starting_node_in_layer[number_of_layers]+x;
    if (fabs(desired_output[x]-node_record[last_layer_node]->output) > epsilon)
    {
      *error = *error + 1;
      x = nodes_in_layer[number_of_layers];
    }
  }
}

/************************************************************/
/***** End functions called by backprp remaining lyrs *****/
/************************************************************/




/************************************************************/
/* The following functions called by train via kohonen    */
/************************************************************/

/*********************************************************************/
/* Function Name: get_linear_training_eta     Number: 8.11           */
/* Description: This function determinse the training eta by         */
/*              n = {n(max)/[i(o)-i(max)]}[i-i(o)]+n(max)             */
/* Functions Called: None                                            */
/* Variables Passed In: train_width - Integer array                  */
/*                      train_scale - Float array                    */
/*                      iterations - Integer                         */
/*                      *eta - Float pointer                         */
/*                      width_no - Integer                           */
/*                                                                   */
/* Variables Returned: *eta - Float pointer                          */
/* Date: 10 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void get_linear_training_eta (int train_width[],
                              float train_scale[],
                              int iterations,
                              float *eta,
                              int width_no)
{
   int n, x;
   n = 0;
   for (x = 1; x < width_no; x++)
     if (iterations > train_width[x])
       n = x;
     *eta = (train_scale[n]/(train_width[n] - train_width[n+1]))
          * (iterations - train_width[n]) + train_scale[n];
}

/*********************************************************************/
/* Function Name: get_kohonen_neighborhood          Number: 8.12     */
/* Description: This function provide the neighborhood used in       */
/*              the update of the nodes                              */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: train_width - Integer array                  */
/*                      iterations - Integer                         */
/*                      neighborhoods - Integer array                */
/*                      width_no - Integer                           */
/*                      *neighbor - Integer pointer                  */
/*                                                                   */
```

```
/* Variables Returned: *neighbor - Integer pointer                */
/* Date: 10 Nov 90      Revision: 1.0                             */
/*****************************************************************/

void get_kohonen_neighborhood (int train_width[],
                               int iterations,
                               int neighborhoods[],
                               int width_no,
                               int *neighbor)
{
   int n, x;
   n = 0;
   for (x = 1; x < width_no; x++)
     if (iterations > train_width[x])
       n = x;
     *neighbor = neighborhoods[n];
}

/*****************************************************************/
/* Function Name: calc_dist_outputs_to_nxt_lyr       Number:8.13  */
/* Description: This function finds the euclidean distance between */
/*              the outputs of one layer and the weights of the  */
/*              next layer                                       */
/*              d(ij) = sqrt{sum[y(i)-w(j)]^2}                   */
/* Functions Called: None                                        */
/* Variables Passed In: Node_record - Structure array            */
/*                      nodes_in_layer - Integer array           */
/*                      starting_node_in_layer - Integer array   */
/*                      current_layer - Integer                  */
/*                      distance - Float array                   */
/*                                                               */
/* Variables Returned: distance - Float array                    */
/* Date: 10 Nov 90      Revision: 1.0                            */
/*****************************************************************/

void calc_dist_outputs_to_nxt_lyr(struct Node_data *node_record[],
                                  int nodes_in_layer[],
                                  int starting_node_in_layer[],
                                  int current_layer,
                                  float distance[])

{
  int x, y, current_node, previous_layer_node;
  double buffer;
  double exponent_1 = 2;
  for (x = 0; x < nodes_in_layer[current_layer]; x++)
  {
     current_node = starting_node_in_layer[current_layer] + x;
     buffer = 0;
     distance[x] = 0;
     for ( y = 0; y < nodes_in_layer[current_layer-1]; y++)
     {
        previous_layer_node = starting_node_in_layer[current_layer-1] + y;
        buffer = node_record[current_node]->weight[previous_layer_node]
               - node_record[previous_layer_node]->output;
        distance[x] = distance[x] + pow(buffer,exponent_1);
     }
  }
}

/*****************************************************************/
/* Function Name: find_nearest_element           Number: 8.14    */
/* Description: This function finds the node in the kohonen layer */
/*              nearest to the input pattern                     */
/* Functions Called: None                                        */
```

```
/* Variables Passed In: min - Float array                              */
/*                      array_max - Integer                            */
/*                      *nearest_element - Integer pointer             */
/*                                                                     */
/* Variables Returned: *nearest_element - Integer pointer              */
/*                                                                     */
/* Date: 10 Nov 90      Revision: 1.0                                  */
/***********************************************************************/

void find_nearest_element(float min[],
                          int array_max,
                          int *nearest_element)

{
   int x;
   float temp = 1000;
   *nearest_element = 0;
   for (x = 0; x < array_max; x++)
       if (temp > min[x])
       {
          temp = min[x];
          *nearest_element = x;
       }
}

/***********************************************************************/
/* Function Name: find_kononen_boundaries          Number: 8.15        */
/* Description: This function finds the valid boundaries of the        */
/*              rectangular kohonen layer.  Theses boundaries are      */
/*              centered at the winning node in the network            */
/*                                                                     */
/* Functions Called: None                                              */
/* Variables Passed In: winner_node - Integer                          */
/*                      starting_node_in_layer - Integer array         */
/*                      current_layer - Integer                        */
/*                      nodes_x - Integer                              */
/*                      nodes_y - Integer                              */
/*                      neighbors - Integer                            */
/*                      *boundary_left - Integer pointer               */
/*                      *boundary_right - Integer pointer              */
/*                      *boundary_up - Integer pointer                 */
/*                      *boundary_down - Integer pointer               */
/*                                                                     */
/* Variables Returned:  *boundary_left - Integer pointer               */
/*                      *boundary_right - Integer pointer              */
/*                      *boundary_up - Integer pointer                 */
/*                      *boundary_down - Integer pointer               */
/*                                                                     */
/* Date: 10 Nov 90      Revision: 1.0                                  */
/***********************************************************************/

void find_kohonen_boundaries (int winner_node,
                              int starting_node_in_layer[],
                              int current_layer,
                              int nodes_x,
                              int nodes_y,
                              int neighbors,
                              int *boundary_left,
                              int *boundary_right,
                              int *boundary_up,
                              int *boundary_down)
{
   int neighbors_x, neighbors_y, winner_node_y, winner_node_x;
   neighbors_x = neighbors - 1;
   neighbors_y = neighbors - 1;
```

```
  winner_node_y = (winner_node-starting_node_in_layer[current_layer])
                  /nodes_x;
  winner_node_x = (winner_node-starting_node_in_layer[current_layer])
                  - winner_node_y * nodes_x;

  *boundary_left = winner_node_x - neighbors_x/2;
  *boundary_right = winner_node_x + neighbors_x/2;
  *boundary_up = winner_node_y + neighbors_y/2;
  *boundary_down = winner_node_y - neighbors_y/2;

  if (*boundary_left < 0)
    *boundary_left = 0;
  if (*boundary_right > nodes_x - 1)
    *boundary_right = nodes_x - 1;
  if (*boundary_up > nodes_y - 1)
    *boundary_up = nodes_y - 1;
  if (*boundary_down < 0)
    *boundary_down = 0;

}

/*********************************************************************/
/* Function Name: determine_neighborhood_elements     Number: 8.16 */
/* Description: This function returns the node numbers of the      */
/*              nodes whose weights will be updated                */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: boundary_left - Integer                    */
/*                      boundary_right - Integer                   */
/*                      boundary_up - Integer                      */
/*                      boundary_down - Integer                    */
/*                      *nodes_to_update - Integer pointer         */
/*                      update_node - Integer array                */
/*                      starting_node_in_layer - Integer array     */
/*                      nodes_x - Integer                          */
/*                      current_layer - Integer                    */
/*                                                                 */
/* Variables Returned: *nodes_to_update - Integer pointer          */
/*                      update_node - Integer array                */
/* Date: 10 Nov 90      Revision: 1.0                              */
/*********************************************************************/

void determine_neighborhood_elements (int boundary_left,
                                      int boundary_right,
                                      int boundary_up,
                                      int boundary_down,
                                      int *nodes_to_update,
                                      int update_node[],
                                      int starting_node_in_layer[],
                                      int nodes_x,
                                      int current_layer)

{
  int x, y, z;
  z = 0;
  for (y = boundary_down; y < boundary_up + 1; y++)
    for (x = boundary_left; x < boundary_right + 1; x++)
    {
      update_node[z] = x + nodes_x * y +
 starting_node_in_layer[current_layer];
      z = z + 1;

    }
  *nodes_to_update = z;
```

```
}
/*********************************************************************/
/* Function Name: train_kohonen_weights            Number: 8.17      */
/* Description: This function updates the winning node and the       */
/*              neighborhood nodes by the equation                   */
/*              w(+) = w(-) + n[x-w(-)]                              */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_layer - Integer array                */
/*                      nodes_to_update - Integer                    */
/*                      update_node - Integer array                  */
/*                      eta - Float                                  */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 10 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void train_kohonen_weights(struct Node_data *node_record[],
                           int nodes_in_layer[],
                           int starting_node_in_layer[],
                           int current_layer,
                           int nodes_to_update,
                           int update_node[],
                           float eta)

{
   int x, y, current_node, previous_layer_node;
   float buffer;
   for (x = 0; x < nodes_to_update; x++)
   {
     current_node = update_node[x];
     for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
     {
        previous_layer_node = starting_node_in_layer[current_layer-1] + y;
        buffer = node_record[previous_layer_node]->output -
                 node_record[current_node]->weight[previous_layer_node];
        node_record[current_node]->weight[previous_layer_node] =
               node_record[current_node]->weight[previous_layer_node]
               + eta * buffer;
     }
   }
}
/*********************************************************/
/*  End functions called by train vi kohonen             */
/*********************************************************/

/*********************************************************/
/*  The following functions are called by set sigma at   */
/*  P neighbors avg.                                     */
/*********************************************************/


/*********************************************************************/
/* Function Name: find_distance_between_nodes      Number:8.18       */
/* Description: This function finds the euclidean distance between   */
/*              nodes in the same layer                              */
/*              d(ij) = sqrt{sum[w(i)-w(j)]^2}                       */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_node - Integer                       */
```

```
/*                    next_node - Integer                          */
/*                    current_layer - Integer                      */
/*                    *distance_between - Float array pointer      */
/*                                                                 */
/* Variables Returned: *distance_between - pointer                 */
/* Date: 10 Nov 90      Revision: 1.0                              */
/*******************************************************************/


void find_distance_between_nodes(struct Node_data *node_record[],
                                 int nodes_in_layer[],
                                 int starting_node_in_layer[],
                                 int current_node,
                                 int next_node,
                                 int current_layer,
                                 float *distance_between[])
{
   int x, y, z, previous_layer_node;
   double distance, buffer;
   double exponent_1 = 2;
   double exponent_2 = .5;
   x = current_node - starting_node_in_layer[current_layer];
   y = next_node - starting_node_in_layer[current_layer];
   buffer = 0;
   for (z = 0; z < nodes_in_layer[current_layer-1]; z++)
   {
      previous_layer_node = starting_node_in_layer[current_layer-1] + z;
      distance = node_record[current_node]->weight[previous_layer_node]
               - node_record[next_node]->weight[previous_layer_node];
      distance = pow(distance,exponent_1);
      buffer = buffer + distance;
   }
   *((distance_between[x])+y) = pow(buffer,exponent_2);
}

/*******************************************************************/
/* Function Name: sort_2_dim_array                  Number: 8.19 */
/* Description: This function returns an array sorted in          */
/*              descending order                                   */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: *M - Float array pointer                   */
/*                      array_max - Integer                        */
/*                      row - Integer                              */
/*                                                                 */
/* Variables Returned: *M - Float array pointer                    */
/* Date: 10 Nov 90      Revision: 1.0                              */
/*******************************************************************/


void sort_2_dim_array(float *M[],
                      int array_max,
                      int row)
{
   int y, z;
   float temp;
   for (y = 0; y < array_max; y++)
     for (z = y; z < array_max; z++)
      if (*((M[row])+z) < *((M[row])+y) )
      {
         temp = *((M[row])+y);
         *((M[row])+y) = *((M[row])+z);
         *((M[row])+z) = temp;
      }
}
```

```
/*************************************************************/
/*  End functions called by set sigma at P neighbor avg **/
/*************************************************************/


/*************************************************************/
/*  Functions Called by CE Remaining Layers                  */
/*************************************************************/

/*************************************************************************/
/* Function Name: CE_last_layer                          Number:8.20     */
/* Description: This function updates the last layer weights of          */
/*              and offsets of a sigmoidal network by                    */
/*              w(mn)+ = w(mn)- + n/(2.3N)[d(n)-y(n)]y(m)                */
/*               @(n) = @(n) - n/(2.3N)[d(n)-y(n)]                       */
/*                                                                       */
/* Functions Called: None                                                */
/* Variables Passed In: Node_record - Structure array                    */
/*                      nodes_in_layer - Integer array                   */
/*                      starting_node_in_layer - Integer array           */
/*                      current_layer - Integer                          */
/*                      *wght - Float array                              */
/*                      CE_eta - Float                                   */
/*                      CE_momentum - Float                              */
/*                      desired - Float array                            */
/*                                                                       */
/* Variables Returned: Node_record - Structure array                     */
/* Date: 10 Nov 90       Revision: 1.0                                   */
/*************************************************************************/


void CE_last_layer(struct Node_data *node_record[],
                   int nodes_in_layer[],
                   int starting_node_in_layer[],
                   int current_layer,
                   float *wght[],
                   float eta,
                   float alpha,
                   float desired[])

{
   int x, y, last_lyr_node, mid_lyr_node;
   float old_wght, buffer_1;

   for (x = 0; x < nodes_in_layer[current_layer]; x++)
   {
      last_lyr_node = starting_node_in_layer[current_layer]+x;
      buffer_1 = desired[x] - node_record[last_lyr_node]->output;
      {
         for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
         {
           mid_lyr_node = starting_node_in_layer[current_layer-1] + y;
           old_wght = *((wght[last_lyr_node])+mid_lyr_node);
           *((wght[last_lyr_node])+mid_lyr_node) =
                    node_record[last_lyr_node]->weight[mid_lyr_node];
           node_record[last_lyr_node]->weight[mid_lyr_node] +=
                    eta * buffer_1 * node_record[mid_lyr_node]->output
                    + alpha
                    * (node_record[last_lyr_node]->weight[mid_lyr_node]
                       - old_wght);
         }
      old_wght = *((wght[last_lyr_node])+last_lyr_node);
      *((wght[last_lyr_node])+last_lyr_node) =
                    node_record[last_lyr_node]->sigma[last_lyr_node];
```

```
      node_record[last_lyr_node]->sigma[last_lyr_node] +=
                  eta * buffer_1
                  + alpha
                  * (node_record[last_lyr_node]->sigma[last_lyr_node]
                    - old_wght);
      }
   }
}

/*********************************************************************/
/* Function Name: CE_mid_layer     Number: 8.21                      */
/* Description: This function updates the weights in the middle      */
/*              layer of a sigmoidal network by                      */
/*  w(LM)+ = w(LM)- + n/(2.3M)sum[d(n)-y(n)]w(Mn)y(M)[1-y(M)]y(L)     */
/*  @(LM)+ = @(LM)- + n/(2.3M)sum[d(n)-y(n)]w(Mn)y(M)[1-y(M)]        */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_layer - Integer                      */
/*                      *wght - Float array                          */
/*                      CE_eta - Float                               */
/*                      CE_momentum - Float                          */
/*                      desired - Float array                        */
/*                                                                   */
/* Variables Returned: None                                          */
/* Date: 10 Nov 90     Revision: 1.0                                 */
/*********************************************************************/

void CE_mid_layer(struct Node_data *node_record[],
                  int nodes_in_layer[],
                  int starting_node_in_layer[],
                  int current_layer,
                  float *wght[],
                  float eta,
                  float alpha,
                  float desired[])

{
  int x, y, z, mid_lyr_node, first_lyr_node, last_lyr_node;
  float buffer_1, buffer_2, old_wght;
  for (x = 0; x < nodes_in_layer[current_layer]; x++)
  {
     mid_lyr_node = starting_node_in_layer[current_layer]+x;
     buffer_1 = (1-node_record[mid_lyr_node]->output)
              * node_record[mid_lyr_node]->output;
     buffer_2 = 0;
     for (y = 0; y < nodes_in_layer[current_layer+1]; y++)
     {
        last_lyr_node = starting_node_in_layer[current_layer+1] + y;
        buffer_2 += (desired[y]-node_record[last_lyr_node]->output)
                 * node_record[last_lyr_node]->weight[mid_lyr_node];
     }
     for (z = 0; z < nodes_in_layer[current_layer-1]; z++)
     {
        first_lyr_node = starting_node_in_layer[current_layer-1] + z;
        old_wght = *((wght[mid_lyr_node])+first_lyr_node);
        *((wght[mid_lyr_node])+first_lyr_node) =
                node_record[mid_lyr_node]->weight[first_lyr_node];
        node_record[mid_lyr_node]->weight[first_lyr_node] +=
                eta * buffer_2 * buffer_1 * node_record[first_lyr_node]->output
                + alpha
                * (node_record[mid_lyr_node]->weight[first_lyr_node]- old_wght);
     }
```

```
    old_wght = *((wght[mid_lyr_node])+mid_lyr_node);
    *((wght[mid_lyr_node])+mid_lyr_node) =
                node_record[mid_lyr_node]->sigma[mid_lyr_node];
    node_record[mid_lyr_node]->sigma[mid_lyr_node]+=
              eta * buffer_2 * buffer_1
              + alpha
              * (node_record[mid_lyr_node]->sigma[mid_lyr_node] - old_wght);
  }
}

/*********************************************************************/
/* Function Name: CE_first_layer     Number: 8.22                    */
/* Description: This function updates the weights in the first       */
/*              layer of a three layer sigmoidal network by          */
/* w(KL)+ = w(KL)+n/(2.3N)sum[d(n)-y(n)]{sum[w(mn)y(m)(1-y(m))       */
/*                       w(Lm)y(L)(1-y(L)y(K)]                       */
/* w(KL)+ = w(KL)+n/(2.3N)sum[d(n)-y(n)]{sum[w(mn)y(m)(1-y(m))       */
/*                       w(Lm)y(L)(1-y(L)y(K)]                       */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_layer - Integer                      */
/*                      *wght - Float array                          */
/*                      CE_eta - Float                               */
/*                      CE_momentum - Float                          */
/*                      desired - Float array                        */
/*                                                                   */
/* Variables Returned: Node_record - Structure array                 */
/* Date: 10 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void CE_first_layer(struct Node_data *node_record[],
                    int nodes_in_layer[],
                    int starting_node_in_layer[],
                    int current_layer,
                    float *wght[],
                    float eta,
                    float alpha,
                    float desired[],
                    int total_nodes)

{
   float old_wght, buffer_1, buffer_2, output[TOTAL_NODES];
   int x, y, z, last_lyr_node, mid_lyr_node, first_lyr_node, input_lyr_node;

   for (x = 0; x < total_nodes; x++)
      output[x] = (1-node_record[x]->output)
                * node_record[x]->output;

   for (x = 0; x < nodes_in_layer[current_layer]; x++)
   {
      first_lyr_node = starting_node_in_layer[current_layer] + x;
      buffer_1 = 0;
      for (y = 0; y < nodes_in_layer[current_layer+2]; y++)
      {
         last_lyr_node = starting_node_in_layer[current_layer+2] + y;
         buffer_2 = 0;
         for (z = 0; z < nodes_in_layer[current_layer+1]; z++)
         {
           mid_lyr_node = starting_node_in_layer[current_layer+1]+z;
           buffer_2 += node_record[last_lyr_node]->weight[mid_lyr_node]
                    * output[mid_lyr_node]
                    * node_record[mid_lyr_node]->weight[first_lyr_node];
```

```
        }
        buffer_1 += buffer_2 * (desired[y]-node_record[last_lyr_node]->output);
      }

      for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
      {
        input_lyr_node = starting_node_in_layer[current_layer-1]+y;
        old_wght = *((wght[first_lyr_node])+input_lyr_node);
        *((wght[first_lyr_node])+input_lyr_node) =
                    node_record[first_lyr_node]->weight[input_lyr_node];
        node_record[first_lyr_node]->weight[input_lyr_node] +=
                   eta * buffer_1 * output[first_lyr_node]
                   * node_record[input_lyr_node]->output
                   + alpha
                  * (node_record[first_lyr_node]->weight[input_lyr_node]
                    - old_wght);

      }
      old_wght = *((wght[first_lyr_node])+first_lyr_node);
      *((wght[first_lyr_node])+first_lyr_node) =
              node_record[first_lyr_node]->sigma[first_lyr_node];

      node_record[first_lyr_node]->sigma[first_lyr_node] +=
         eta * buffer_1 * output[first_lyr_node]
         + alpha * (node_record[first_lyr_node]->sigma[first_lyr_node]
                   - old_wght);
    }
}

/*********************************************************/
/*     End Functions Called by CE Remaining Lyrs         */
/*********************************************************/

/*********************************************************/
/*   Functions called by CFM Remaining Lyrs              */
/*********************************************************/

/*********************************************************************/
/* Function Name: calculate_zn     Number: 8.23                      */
/* Description: This function calculates the CFM sigmoid output     */
/*              for each of the incorrect nodes by                   */
/*              z(n) = 1/[1+exp(-By(c)+By(n)+zeta)]                  */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_layer - Integer                      */
/*                      correct_node - Integer                       */
/*                      zn - Float array                             */
/*                      CFM_beta - Float                             */
/*                      CFM_zeta - Float                             */
/*                                                                   */
/* Variables Returned: zn - Float array                              */
/* Date: 10 Nov 90      Revision: 1.0                                */
/*********************************************************************/


void calculate_zn (struct Node_data *node_record[],
                   int nodes_in_layer[],
                   int starting_node_in_layer[],
                   int current_layer,
                   int correct_node,
                   float zn[],
                   float beta,
```

```
                float zeta)
{
  int x, current_node;
  double buffer = 0;
  for (x = 0; x < nodes_in_layer[current_layer]; x++)
  {
     current_node = starting_node_in_layer[current_layer]+x;
     buffer = zeta - beta
               *(node_record[correct_node]->output
               - node_record[current_node]->output);
     zn[x] = 1/(1+exp(buffer));
   }
}

/*********************************************************************/
/* Function Name: CFM_last_layer     Number: 8.24                    */
/* Description: This function updates the weights of the last        */
/*              layer of a sigmoidal network by                      */
/*     w(MN)+ = w(MN)-anB/(N-1)z(N)[1-z(N)]y(N)[1-y(N)]y(M)          */
/*     @(N)+ = @(N)-anB/(N-1)z(N)[1-z(N)]y(N)[1-y(N)]                */
/*     w(MC)+ = w(MC)-anB/(N-1)sum{z(N)[1-z(N)]}y(C)[1-y(C)]y(M)     */
/*     @(C)+ = @(C)-anB/(N-1)sum{z(N)[1-z(N)]}y(C)[1-y(C)]           */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      last_layer - Integer                         */
/*                      zn - Float array                             */
/*                      correct_node - Integer                       */
/*                      CFM_eta - Float                              */
/*                      *wght - Float array                          */
/*                      CFM_momentum - Float                         */
/*                                                                   */
/* Variables Returned:  Node_record - Structure Array                */
/* Date: 10 Nov 90      Revision: 1.0                                */
/*********************************************************************/

void CFM_last_layer(struct Node_data *node_record[],
                    int nodes_in_layer[],
                    int starting_node_in_layer[],
                    int last_layer,
                    float zn[],
                    int correct_node,
                    float eta,
                    float *wght[],
                    float alpha)

{
   float buffer_2, buffer_1, sum_zn, old_wght;
   int x, y, n, current_node, previous_layer_node;
   for (x = 0; x < nodes_in_layer[last_layer]; x++)
   {
      current_node = starting_node_in_layer[last_layer]+x;
      buffer_1 = (1-node_record[current_node]->output)
            * node_record[current_node]->output;
      if (current_node != correct_node)
      {
         for (y = 0; y < nodes_in_layer[last_layer-1]; y++)
         {
            previous_layer_node = starting_node_in_layer[last_layer-1] + y;
            buffer_2 = node_record[previous_layer_node]->output;
            old_wght = node_record[current_node]->weight[previous_layer_node];
            node_record[current_node]->weight[previous_layer_node] +=
                              -eta*(1-zn[x])*zn[x]
```

```
                                * buffer_2 * buffer_1
                                + alpha
                                * (node_record[current_node]->weight[previous_layer_node]
                                   - *((wght[current_node])+previous_layer_node));

            *((wght[current_node])+previous_layer_node) =  old_wght;
          }
          old_wght = node_record[current_node]->sigma[current_node];
          node_record[current_node]->sigma[current_node] +=
                                  -eta*(1-zn[x])*zn[x]* buffer_1
                                  + alpha * (node_record[current_node]->sigma[current_node]
                                          - *((wght[current_node])+current_node));
          *((wght[current_node])+current_node) = old_wght;
        }
       else
       {
          sum_zn = 0;
          for (n = 0; n < nodes_in_layer[last_layer]; n++)
            if (n != correct_node-starting_node_in_layer[last_layer])
              sum_zn = sum_zn + zn[n]* (1-zn[n]);

          for (y = 0; y < nodes_in_layer[last_layer-1]; y++)
          {
            previous_layer_node = starting_node_in_layer[last_layer-1]+y;
            buffer_2 = node_record[previous_layer_node]->output;
            old_wght = node_record[current_node]->weight[previous_layer_node];
            node_record[current_node]->weight[previous_layer_node] +=
                                  eta * buffer_1 * buffer_2 * sum_zn
                                  + alpha
                                  * (node_record[current_node]->weight[previous_layer_node]
                                   - *((wght[current_node])+previous_layer_node));
            *((wght[current_node])+previous_layer_node) = old_wght;
          }
          old_wght = node_record[current_node]->sigma[current_node];
          node_record[current_node]->sigma[current_node] +=
                           eta * buffer_1 * sum_zn
                           + alpha
                           * (node_record[current_node]->sigma[current_node]
                              - *((wght[current_node])+current_node));
          *((wght[current_node])+current_node) = old_wght;
       }
     }
}

/*********************************************************************/
/* Function Name: CFM_mid_layer                    Number: 8.25      */
/* Description: This function updates the second hidden layer        */
/*              parameter of a sigmoidal network by                  */
/* w(LM)+ = w(LM)-anB/(N-1)sum{z(n)[1-z(n)]}y(c)[1-y(c)]w(Mc)         */
/*                          - y(n)[1-y(n)]w(Mn)y(M)[1-y(M)]y(L)      */
/* @(M)+ = @(M)-anB/(N-1)sum{z(n)[1-z(n)]}y(c)[1-y(c)]w(Mc)          */
/*                          - y(n)[1-y(n)]w(Mn)y(M)[1-y(M)]          */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_layer - Integer                      */
/*                      zn - Float array                             */
/*                      correct_node - Integer                       */
/*                      CFM_eta - Float                              */
/*                      *wght - Float array                          */
/*                      CFM_momentum - Float                         */
/*                                                                   */
/* Variables Returned: Node_record - Structure Array                 */
```

```
/* Date: 10 Nov 90     Revision: 1.0                                    */
/*********************************************************************/

void CFM_mid_layer(struct Node_data *node_record[],
                   int nodes_in_layer[],
                   int starting_node_in_layer[],
                   int current_layer,
                   float zn[],
                   int correct_node,
                   float eta,
                   float *wght[],
                   float alpha)

{
   int x, y, z, n;
   int current_node, previous_layer_node, last_layer_node;
   float buffer_1, buffer_2, buffer_3, buffer_4, old_wght;
   for (x = 0; x < nodes_in_layer[current_layer]; x++)
   {
      current_node = starting_node_in_layer[current_layer]+x;
      buffer_1 = node_record[current_node]->output
               * (1-node_record[current_node]->output);
      buffer_3 = 0;
      for (z = 0; z < nodes_in_layer[current_layer+1]; z++)
      {
        last_layer_node = starting_node_in_layer[current_layer+1] + z;
        if (last_layer_node != correct_node)
           buffer_3 += zn[z]*(1-zn[z])
                    *(1-node_record[last_layer_node]->output)
                    * node_record[last_layer_node]->output
                    * node_record[last_layer_node]->weight[current_node];
        else
        {
           buffer_4 = 0;
           for (n = 0; n < nodes_in_layer[current_layer+1]; n++)
             if (n != correct_node-starting_node_in_layer[current_layer+1])
                buffer_4 = buffer_4 + zn[n]*(1-zn[n]);
             buffer_4 = buffer_4 * node_record[correct_node]->output
                      * (1 -node_record[correct_node]->output)
                      * (node_record[correct_node]->weight[current_node]);
        }
      }
        for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
        {
           previous_layer_node = starting_node_in_layer[current_layer-1]+y;
           buffer_2 = node_record[previous_layer_node]->output;
           old_wght = node_record[current_node]->weight[previous_layer_node];
           node_record[current_node]->weight[previous_layer_node] +=
                         eta * (buffer_4-buffer_3)
                         * buffer_1 * buffer_2
                         + alpha
                         * (node_record[current_node]->weight[previous_layer_node]
                            - *((wght[current_node])+previous_layer_node));
           *((wght[current_node])+previous_layer_node) = old_wght;
        }
        old_wght = node_record[current_node]->sigma[current_node];
        node_record[current_node]->sigma[current_node] +=
                         eta * (buffer_4-buffer_3)
                         * buffer_1
                         + alpha
                         * (node_record[current_node]->sigma[current_node]
                            - *((wght[current_node])+current_node));
        *((wght[current_node])+current_node) = old_wght;

   }
```

```
}

/*********************************************************************/
/* Function Name: find_second_highest_node          Number: 8.26   */
/* Description:  This function returns the incorrect node with the */
/*               highest output value                                */
/* Functions Called:None                                             */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      last_layer - Integer                         */
/*                      winner_node - Integer                        */
/*                      *next_highest_node - Integer pointer         */
/*                                                                   */
/* Variables Returned: *next_highest_node - Integer pointer          */
/* Date: 10 Nov 90     Revision: 1.0                                 */
/*********************************************************************/

void find_second_highest_node (struct Node_data *node_record[],
                               int nodes_in_layer[],
                               int starting_node_in_layer[],
                               int last_layer,
                               int winner_node,
                               int *next_highest_node)

{
   int x, current_node;
   float outmax = 0;
   for (x = 0; x < nodes_in_layer[last_layer]; x++)
   {
     current_node = starting_node_in_layer[last_layer]+x;
     if (current_node != winner_node)
     {
        if (node_record[current_node]->output > outmax)
        {
           outmax = node_record[current_node]->output;
           *next_highest_node = current_node;
        }
     }
   }
}

/*********************************************************************/
/* Function Name: CFM_first_layer     Number: 8.27                   */
/* Description: This function updates the first hidden layer of a  */
/*              three layer sigmoidal network by                     */
/* w(KL)+ = w(KL)-anB/(N-1)sum{z(n)[1-z(n)]}y(c)[1-y(c)]sum{w(mc}  */
/*          -y(n)[1-y(n)]sum{w(mn)y(m)[1-y(m)]w(Lm)y(L)[1-y(L)y(K) */
/* @(KL)+ = @(KL)-anB/(N-1)sum{z(n)[1-z(n)]}y(c)[1-y(c)]sum{w(mc}  */
/*          -y(n)[1-y(n)]sum{w(mn)y(m)[1-y(m)]w(Lm)y(L)[1-y(L)     */
/*                                                                   */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_layer - Integer                      */
/*                      zn - Float array                             */
/*                      correct_node - Integer                       */
/*                      CFM_eta - Float                              */
/*                      *wght - Float array                          */
/*                      CFM_momentum - Float                         */
/*                      total_nodes - Integer                        */
/*                                                                   */
/* Variables Returned: Node_record - Structure Array                 */
/* Date: 10 Nov 90     Revision: 1.0                                 */
```

```
/*********************************************************************/

void CFM_first_layer (struct Node_data *node_record[],
                      int nodes_in_layer[],
                      int starting_node_in_layer[],
                      int current_layer,
                      float zn[],
                      int correct_node,
                      float eta,
                      float *wght[],
                      float alpha,
                      int total_nodes)

{
   int x, y, z;
   float buffer_1, buffer_2, old_wght;
   int first_lyr_node, last_lyr_node, mid_lyr_node, input_lyr_node;
   float output[TOTAL_NODES];

   for (x = 0; x < total_nodes; x++)
     output[x] = (1-node_record[x]->output)*node_record[x]->output;

   for (x = 0; x < nodes_in_layer[current_layer]; x++)
   {
      buffer_2 = 0;
      first_lyr_node = starting_node_in_layer[current_layer] + x;
      for (y = 0; y < nodes_in_layer[current_layer+2]; y++)
      {
        last_lyr_node = starting_node_in_layer[current_layer+2] + y;
        if (last_lyr_node != correct_node)
        {
          buffer_1 = 0;
          for (z = 0; z < nodes_in_layer[current_layer+1]; z++)
          {
            mid_lyr_node = starting_node_in_layer[current_layer+1]+z;
            buffer_1 = buffer_1
                  + (output[correct_node]*node_record[correct_node]->weight[mid_lyr_node]
                   - output[last_lyr_node]*node_record[last_lyr_node]->weight[mid_lyr_node])
                  * output[mid_lyr_node]*node_record[mid_lyr_node]->weight[first_lyr_node];
          }
          buffer_2 = buffer_2 + zn[y]*(1-zn[y])* buffer_1;
        }
      }

      for (z = 0; z < nodes_in_layer[current_layer-1]; z++)
      {
         input_lyr_node = starting_node_in_layer[current_layer-1] + z;
         old_wght = *((wght[first_lyr_node])+input_lyr_node);
         *((wght[first_lyr_node])+input_lyr_node) =
                       node_record[first_lyr_node]->weight[input_lyr_node];

         node_record[first_lyr_node]->weight[input_lyr_node] +=
                        eta * buffer_2 * output[first_lyr_node]
                        * node_record[input_lyr_node]->output
                        + alpha
                        * (node_record[first_lyr_node]->weight[input_lyr_node]
                           - old_wght);

      }
      old_wght = *((wght[first_lyr_node])+first_lyr_node);
      *((wght[first_lyr_node])+first_lyr_node) =
                 node_record[first_lyr_node]->sigma[first_lyr_node];
      node_record[first_lyr_node]->sigma[first_lyr_node] +=
                        eta * buffer_2 * output[first_lyr_node]
                        + alpha
```

```
                        * (node_record[first_lyr_node]->sigma[first_lyr_node]
                          - old_wght);
   }
}

/*****************************************************/
/*  End Functions Called by CFM Remaining Lyrs       */
/*****************************************************/

/*********************************************************************/
/* Function Name: find_nearest_neighbor              Number:8.28     */
/* Description: This function finds the euclidean distance between   */
/*              nodes in the same layer                              */
/*              d(ij) = sqrt{sum[w(i)-w(j)]^2}                       */
/* Functions Called: None                                            */
/* Variables Passed In: Node_record - Structure array                */
/*                      nodes_in_layer - Integer array               */
/*                      starting_node_in_layer - Integer array       */
/*                      current_node - Integer                       */
/*                      next_node - Integer                          */
/*                      current_layer - Integer                      */
/*                      *distance_between - Float array pointer      */
/*                                                                   */
/* Variables Returned: *distance_between - pointer                   */
/* Date: 10 Nov 90      Revision: 1.0                                */
/*********************************************************************/


void find_nearest_neighbor(struct data *data_record[],
                           struct Node_data *node_record[],
                                    int record,
                           int nodes_in_layer[],
                           int starting_node_in_layer[],
                           int current_layer,
                           int *nearest_node)

{

   int x, y, current_node, previous_layer_node;
   float buffer, distance, nearest_distance;
   double exponent = 2;
   calculate_layer_0_output(data_record,
                            node_record,
                            nodes_in_layer,
                            record);

       nearest_distance = 1000;
       for (x = 0; x < nodes_in_layer[current_layer]; x++)
       {
            current_node = starting_node_in_layer[current_layer] + x;
            distance = 0;
            for (y = 0; y < nodes_in_layer[current_layer-1]; y++)
            {
              previous_layer_node = starting_node_in_layer[current_layer-1]+y;
              buffer = node_record[current_node]->weight[previous_layer_node]
                         - node_record[previous_layer_node]->output;
              distance = distance + pow(buffer,exponent);
            }
            if (distance < nearest_distance)
            {
            nearest_distance = distance;
              *nearest_node = current_node;
            }
       }
```

```
}
```

G.9 NETMATH

```
/*************************************************************************/
/* Module Name: NETMATH.C     Number: 9.0                                */
/* Description: This module contains the basic mathematical              */
/*              used to train the networks                               */
/*                                                                       */
/* Modules Called: None                                                  */
/* Functions Contained:                                                  */
/*  9.1 make_identity_matrix       9.2 determine_matrix_transpose        */
/*  9.3 invert_a_matrix            9.4 update_average                    */
/*  9.5 update_sigma               9.6 calculate_percentage              */
/*                                                                       */
/* Date: 11 Nov 90     Revision: 1.0                                     */
/*************************************************************************/


#include "netvrble.h"
#include "netfnctn.h"

/*************************************************************************/
/* Function Name: make_identity_matrix       Number: 9.1                 */
/* Description: This function returns a matrix whose elements are       */
/*              are 1 on the diagonal and 0 elsewhere                    */
/*                                                                       */
/* Functions Called: None                                                */
/* Variables Passed In:*M - Float array pointer                          */
/*                     nodes_2 - Integer                                 */
/*                                                                       */
/* Variables Returned: *M - Float array pointer                          */
/* Date: 11 Nov 90      Revision: 1.0                                    */
/*************************************************************************/

void make_identity_matrix (float *M[],
                           int nodes_2)
{
   int row, column;
   for (row = 0; row < nodes_2; row++)
     for (column = 0, column < nodes_2; column++)
       if (row == column)
        *((M[row])+column) = 1;
       else
        *((M[row])+column) = 0;
}

/*************************************************************************/
/* Function Name: determine_matrix_transpose      Number: 9.2            */
/* Description: This function returns the transpose of a square         */
/*              matrix by                                                */
/*              MT[x][y] = M[y][x]                                       */
/*                                                                       */
/* Functions Called: None                                                */
/* Variables Passed In: *MT - Float array pointer                        */
/*                      *M - Float array pointer                         */
/*                      total_rbfs                                       */
/*                                                                       */
/* Variables Returned: *MT - Float array pointer                         */
/* Date: 11 Nov 90      Revision: 1.0                                    */
```

```
/*********************************************************************/

void determine_matrix_transpose(float *MT[],
                                float *M[],
                                int total_rbfs)
{
  int row, column;
  for (row = 0; row < total_rbfs; row ++)
    for (column = 0; column < total_rbfs; column++)
      *((MT[row])+column) = *((M[column])+row);
  for (row = 0; row < total_rbfs; row++)
    for (column = 0; column < total_rbfs; column++)
      *((M[row])+column) = *((MT[row])+column);
}

/*********************************************************************/
/* Function Name: invert_a_matrix              Number: 9.3           */
/* Description: This function returns inverts a square matrix        */
/*              via Gaussian elmination                               */
/*                                                                    */
/* Functions Called: None                                             */
/* Variables Passed In: *M[] - Float array pointer                    */
/*                      *N[] - Float array pointer                    */
/*                       nodes_2 - Integer                            */
/*                                                                    */
/* Variables Returned:  *N[] - Float array pointer                    */
/* Date: 11 Nov 90      Revision: 1.0                                 */
/*********************************************************************/

void invert_a_matrix (float *M[],
                      float *N[],
                      int nodes_2)
{
   float beta = 0;
   float alpha = 0;
   int xx, yy, row, column;
   for (row = 0; row < nodes_2; row++)
   {
     beta = *((M[row])+row);
     for (column = 0; column < nodes_2; column++)
     {
        *((M[row])+column) = *((M[row])+column)/beta;
        *((N[row])+column) = *((N[row])+column)/beta;
     }
     for (xx = 0; xx < nodes_2; xx++)
       if (xx != row)
       {
          alpha = *((M[xx])+row);
          for (yy = 0; yy < nodes_2; yy++)
          {
            *((M[xx])+yy) = -alpha * *((M[row])+yy) + *((M[xx])+yy);
            *((N[xx])+yy) = -alpha * *((N[row])+yy) + *((N[xx])+yy);
               }
       }
   }
}

/*********************************************************************/
/* Function Name: update_average               Number: 9.4           */
/* Description: This function maintains a running average             */
/*              avg(+) = avg(-)+[x-avg(-)]/[N(-)+1]                   */
/*                                                                    */
/* Functions Called: None                                             */
/* Variables Passed In: current_average - Float                       */
/*                      elements - Integer                            */
```

```
/*                    x - Float                                    */
/*                    *next_average - Float pointer                */
/*                                                                 */
/* Variables Returned: *next_average - Float pointer               */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*******************************************************************/

void update_average (float current_average,
                     int elements,
                     float x,
                     float *next_average)

{
    float y;
    *next_average = current_average + (1/(float)elements)*(x-current_average);
}

/*******************************************************************/
/* Function Name: update_sigma                      Number:9.5     */
/* Description: This function maintains a running sigma by         */
/*  sigma(+) = sqrt{sigma(-)^2+(1/N)[1-(1/N)][x-avg(-)]^2-sigma(-)^2} */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: current_sigma - Float                      */
/*                    elements - Integer                           */
/*                    x - Integer                                  */
/*                    current_average - Float                      */
/*                    *next_sigma - Float pointer                  */
/*                                                                 */
/* Variables Returned: *next_sigma - Float pointer                 */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*******************************************************************/

void update_sigma (float current_sigma,
                   int elements,
                   float x,
                   float current_average,
                   float *next_sigma)

{
   *next_sigma = pow((double)(current_sigma),2)+(1/(float)elements)
                * ((1-1/(float)elements) * pow((double)(x-current_average),2)
                 - pow((double)(current_sigma),2));
   *next_sigma = pow((double)(*next_sigma),.5);
}

/*******************************************************************/
/* Function Name: calculate_percentage              Number: 9.6    */
/* Description: This function returns the percentage value         */
/*                                                                 */
/* Functions Called: None                                          */
/* Variables Passed In: numerator - Float                          */
/*                    denominator - Float                          */
/*                    *per_cent - Float pointer                    */
/*                                                                 */
/* Variables Returned:  *per_cent - Float pointer                  */
/* Date: 11 Nov 90      Revision: 1.0                              */
/*******************************************************************/

void calculate_percentage(float numerator,
                          float denominator,
                          float *per_cent)
{
  *per_cent = 100 * numerator/denominator;
}
```

G.10 NETVRBLE

```
#include <stdio.h>
#include <math.h>

/*************************************************************/
/* Name: NETVRBLE                              Number: 10.0  */
/* Description: This module contains the data structures     */
/*              for the nodes and the data along with        */
/*              the maximum values for the global            */
/*              variables                                    */
/*                                                           */
/* Modules Called: None                                      */
/* Functions Contained: None                                 */
/* Variables Passed In: None                                 */
/* Variables Returned: None                                  */
/* Date: 10 Nov 90              Revision: 1.0                */
/*************************************************************/

#define DIMENSION 100        /* Length of Feature Vector    */
#define TRAIN_SET 200        /* Number of Training Patterns */
#define TEST_SET 200         /* Number of Test Patterns     */
#define CLASSES 20           /* Number of Classes           */
#define TOTAL_NODES 250      /* Number of Nodes             */

struct data
{
   float vector[DIMENSION];
   int class;
   int number;
};

struct Node_data
{
   float weight[TOTAL_NODES];
   float sigma[TOTAL_NODES];
   int class;
   int transfer_function;
   float output;
   int connect[TOTAL_NODES];
};
```

Bibliography

1. Barmore, Gary D. *Speech Recognition Using Neural Nets and Dynamic Time Warping.* MS thesis, AFIT/GEO/GENG/88D-1, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.
2. Beastall, W.D. "Recognition of Radar Signals by Neural Networks." In *First IEE International Conference on Artificial Neural Networks*, pages 139–142, London, UK: IEEE Press, 1989.
3. Brady, M. and others. *Gradient Descent Fails to Separate.* Technical Report O-9740/B202, Palo Alto CA: Lockheed R and DD, July 1988.
4. Broomhead, D.S. and David Lowe. *Radial Basis Functions, Multi-Variable Functional Interpolation and Adaptive Networks.* Memorandum 4148, London, UK: Royal Signals and Radar Establishment, March 1988.
5. Cybenko, G. "Approximation of Superpositions of a Sigmoidal Function," *Mathematics of Control, Signals and Systems*, *2*:303–314 (March 1989).
6. Dayhoff, Judith E. *Neural Network Architectures.* New York: Van Nostrand Reinhold, 1990.
7. Fukunaga, Keinosuke. *Introduction to Statistical Pattern Recognition.* New York: Academic Press, 1972.
8. James, Mike. *Classification Algorithms.* New York: John Wiley and Sons, Inc., 1985.
9. Kohonen, Teuvo. *Self-Organizing Maps.* Tutorial 132, Laboratory of Computer and Information Science: Helsinki University of Technology, January 1982.
10. Kohonen, Teuvo. *Self-Organization and Associative Memory.* New York: Springer-Verlag Berlin Heidelberg, 1989.
11. Lippman, Richard P. "Pattern Classification Using Neural Networks," *IEEE Communications Magazine*, *2*:47–63 (November 1990).
12. Marchette, D.J. and J.E. Priebe. *The Adaptive Kernel Neural Network.* Technical Document 1676, San Diego CA: Naval Ocean Systems Center, October 1989 (A-217-230).
13. Moody, John and Christian Darken. "Learning with Localized Receptive Fields." In *Proceedings of the 1988 Connectionist Models Summer School*, pages 133–143, New Haven CT: Yale Computer Science, 1988
14. Nowlan, Steven J. *Max Likelihood Competition in RBF Networks.* Technical Report CRG-TR-90-2, Toronto Canada. Department of Computer Science, University of Toronto, February 1990.
15. Poggio, Thomaso and F. Girosi. *A Theory of Networks for Approximation and Learning.* A.I. Memo 1140, Cambridge MA: MIT and Center for Biological Information Processing, July 1989.
16. Poggio, Thomaso and F. Girosi. "Regularization Algorithms for Learning Equivalent to Multilayer Networks," *Science*, *247*:978–982 (February 1990).
17. Powell, M.J.D. and others. *Algorithms for Approximation.* Oxford: Clarendon, 1987.
18. Renals, S. "Radial Basis Function Network for Speech Pattern Classification," *Electronic Letters*, *25*:437–439 (March 1990).
19. Rogers, Steven K. and others. *An Introduction to Biological and Artificial Neural Networks.* wpafb. afit, 1990.

20. Ruck, D and others. "Classification of Tactical Targets with Neural Netowrks." In *Biologicial and Artificial Neural Networks for Pattern Recognition*, pages 268–290, wpafb: afit, 1990.

21. Ruck, D and others. "The Multilayer Perceptron as an Approximation to a Bayes Optimal Discriminant Function." In *International Joint Conference on Neural Networks*, pages 863–873, Ann Arbor, MI: IEEE Press, 1990.

22. Rummelhart, David E. and others. *Parallel Distributed Processing*. Cambridge MA: The MIT Press, 1986.

23. S., Renals and Richard Rohwer. "Phoneme Classification Experiments Using Radial Basis Functions." In *Proceedings of the International Joint Conference on Neural Networks*, pages 461–467, 1989.

24. Specht, Donald F. "Probabalistic Neural Networks for Classification, Mapping, or Associative-Memory." In *IEEE International Conference on Neural Networks*, pages 525–532, San Diego, CA: IEEE Press, 1988.

25. Stremler, Ferrel G. *Introduction to Communication Systems*. Philippines: Addison-Wesley Publishing Company, 1982.

26. Tarr, Gregory L. *Dynamic Analysis of Feedforward Neural Networks Using Simulated and Measured Data*. MS thesis, AFIT/GE/GENG/88D-54, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

27. Tou, Julius T. and Rafael C. Gonzalez. *Pattern Recognition Principles*. Reading MA: Addison-Wesley Publishing Co., 1974.

28. Waibel, Alexander H. and John B. Hampshire. "A Novel Objective Function for Improved Phoneme Recognition," *IEEE Transactions On Neural Networks*, *1*:216–227 (June 1990).

29. Wasserman, Philip D. *Neural Computing, Theory and Practice*. New York: Van Nostrand Reinhold, 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE
CHARACTERIZATION OF RADAR SIGNALS USING NEURAL NETWORKS

5. FUNDING NUMBERS

6. AUTHOR(S)
Daniel R. Zahirniak, Capt

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/90D-69

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Electronic Warfare Division
WRDC/AAWP-1
WPAFB OH 45433-6543

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Recent work concerning artificial neural networks has focused on decreasing network training times. Kernel Classifier networks, using radial basis functions (RBFs) as the kernel function, can be trained quickly with little performance degradation. Short training times are critical for systems which must adapt to changing environments. The function of Kernel Classifier networks is based on the principle that multivariate functions can be approximated via linear combinations of RBFs. RBFs can also perform probability density estimations, making classifications approximating a Baye's optimal descriminant. Methods used to set the RBF centers included matching the training data, Kohonen Training, K-Means Clustering and placement at averages of data clusters of the same class. Test results indicate the performance of these networks was equal to that of Hyperplane Classifier networks trained, via backpropagation, to optimize the Mean Square Error, Cross Entropy, and Classification Figure of Merit objective functions. However, the RBF networks trained much faster. The RBF networks also outperformed the Probability Neural Networks,(PNN) indicating the weights in the output layer offset the choice of non-optimal spreads. This ability to train quickly while obtaining high classification accuracies make RBF Kernel Classifier networks an attractive option for systems which must adapt quickly to changing environments.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Neural Networks, Artificial Intelligence, Signal Classification, Pattern Recognition, Backward Error Propagation, Radial Basis Function	289
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL