University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

12-1991

A self configuring high-order neural network

Ronald Brett Michaels

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Recommended Citation

Michaels, Ronald Brett, "A self configuring high-order neural network. " Master's Thesis, University of Tennessee, 1991.
https://trace.tennessee.edu/utk_gradthes/12475

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Ronald Brett Michaels entitled "A self configuring high-order neural network." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Engineering Science.

Lefteri Tsoukalas, Major Professor

We have read this thesis and recommend its acceptance:

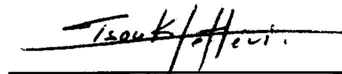
Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

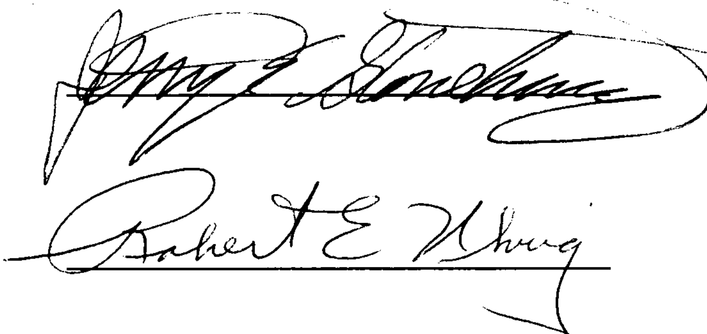
To the Graduate Council:

I am submitting herewith a thesis written by Ronald Brett Michaels entitled "A Self Configuring High-Order Neural Network." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Engineering Science.



Lefteri Tsoukalas, Major Professor

We have read this thesis
and recommend its acceptance:



Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, providing that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor or, in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Ronald Brett Michaels

Date November 26, 1991

A Self Configuring High–Order Neural Network

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Ronald Brett Michaels

December 1991

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Lefteri Tsoukalas, for his help and enthusiasm during the development of the ideas which are expressed herein.

I acknowledge the Tennessee Valley Authority and thank them for the use of the simulated power plant data used in this thesis.

ABSTRACT

The functional link net of Yo–Han Pao and the high–order neural network of Giles and Maxwell require that the user select the expansion terms to suit the particular data set.

For the two category classification problem a method of finding adaptively appropriate expansion terms for a one layer functional link net is presented and discussed. In the training phase input vectors **x** are expanded using Hebbian learning in the form of second order neurons (or outerproduct expansion). A new network layer is then created by multiplying the expanded vectors by a matrix determined by applying the Karhunen– Loève expansion to those expanded vectors. This removes all correlation from the features of the expanded vectors and may reduce their dimensionality. If the Ho–Kashyap algorithm indicates linear separability, quit; otherwise, expand the current layer and repeat above steps until linear separability is obtained. It is then possible to pass a vector **x** having symbolic terms through the multilayer network. The result is a polynomial in the components of **x**. The symbolic portion of each term of the polynomial represents one expansion function of an equivalent one layer functional link net, and the numerical coefficient of the term is the associated weight.

Contents

1 Introduction **1**

1.1 A Neural Network Taxonomy 1

1.2 A Review of Supervised Learning 1

1.2.1 Nonlinear Separability in Pattern Recognition 2

1.2.2 Nonlinear Separability in Neural Networks 7

2 Statement of the Problem **12**

3 A Description of the Algorithm **14**

3.1 The Outerproduct Expansion 17

3.2 The Ho–Kashyap Algorithm 19

3.3 The Karhunen–Loève Expansion 21

3.4 The Working of the Overall Algorithm 22

4 Results and Conclusions **31**

4.1 General Characteristics of the Algorithm 31

4.2 Results from Theoretical Problems 37

4.3 Results from Nuclear Power Plant Simulator Data 40

4.4 Conclusions 45

4.5 Possible Further Study 46

List of References 48

Appendices 52

Appendix A

Simulated Power Plant Data 53

Appendix B

Source Code for High–Order Neural Network Program 63

Appendix C

Source Code for Classifier Program 90

Vita 111

List of Figures

1 Nonlinear vs. Linear Separability 3
2 Basic Model for a Pattern Classifier, after Nilsson 3
3 Basic Model for a Pattern Dichotomizer, after Nilsson 4
4 A Quadric Discriminator, after Nilsson 5
5 Schematic Illustration of a Functional Link, after Pao 8
6 Expansion Term Grouping 11
7 Flow Chart for Self Configuring Network 15
8 Multilayer Net as Constructed by Successive Expansions and Transformations 16
9 The XOR Problem 26
10 Program Output for XOR Problem 27
11 Multilayer Network Solution to XOR Problem 28
12 Equivalent Single Layer Functional Link Solution to XOR Problem 30
13 XOR Problem Mapped Onto 3 Dimensions 33
14 XOR Problem Showing Class Regions 34
15 Size of Matrices for Multilayer Solution of Loop Problem 35
16 Loop Problem Showing Class Regions 36
17 Test2-d Problem Showing Class Regions 38
18 12 Point Problem Showing Class Regions 41
19 Maze Problem Showing Class Regions 42
20 Program Output for MS1 vs. TH5 Problem 44

1 Introduction

What do computers do? Very generally, they process information. Computers have increased by several orders of magnitude society's ability to store and process information. In a historical sense, the first major thrust in information processing was system and information theory, which is about database processing. A later major thrust was the development of artificial intellegence for knowledgebase processing. We are now in the midst of a third major thrust, that is Neural Networks [26], which holds the promise of further increases in information processing power.

1.1 A Neural Network Taxonomy

Neural Networks are structures made up of many small identical processing elements, called neurons, operating in parallel. The intelligence of a neural network is contained in the architecture or arrangement of the neurons and in the adaptation rules under which the neurons operate. Neural networks have the capability to learn or adapt. Adaptive neural networks can be divided into three types based on the learning procedure employed [26]:

Supervised Learning, where the network is presented patterns and then told by a teacher whether its response is correct or incorrect.

Unsupervised Learning, where data is presented and the task of the neural network is to categorize the data into several clusters.

Self-supervised training is used by automata which are able to generate error signals internally without any external teacher.

In supervised learning we may divide networks into two types, those made up of first order or linear neurons and those made up of higher order or nonlinear neurons.

1.2 A Review of Supervised Learning

The single layer supervised learning network with linear neurons, while having its origins in the field of Psychology, has been studied extensively over the years by the pattern recog-

nition community, and its utility for linear pattern separation is well known. Networks with nonlinear neurons have appeared in the literature for many years; however, authors have typically discussed nonlinear classifiers in terms of feasibility rather than utility.

The perceptron was introduced by Rosenblatt [17] in 1958 as an attempt to model certain brain functions. The phenomenon which Rosenblatt wished to model was learning, and this learning took the form of classifying stimuli into categories. Initially stimuli would be presented to a perceptron, and positive or negative reinforcement would be applied depending upon whether or not the classification was correct. The reinforcements would alter the connections (later called weights) between neurons . After some period of training the perceptron could correctly classify not only the stimuli used in the learning process, but also stimuli which were "similar" to the training stimuli.

Rosenblatt's perceptrons were linear perceptrons and were set in a neural context. Over the next few years perceptrons were abstracted, generalized, and cut loose from their neural context. Perceptrons and their variants along with the Bayesian Classifier formed the basis for a field of study called Pattern Recognition.

1.2.1 Nonlinear Separability in Pattern Recognition

One of the limitations of linear perceptrons is that they can make only linear separations. There are, however, many interesting problems which require a separation which cannot be accomplished by a line or plane. One way of separating those categories is by use of a curved line or plane. Please refer to Figure 1 for a simple comparison of linear and nonlinear separability.

Nilsson [13] presented in 1965 a basic pattern classifier which encompassed both linear and nonlinear classification. Figure 2 shows Nilsson's model for a pattern classifier. In this model a pattern vector, $\mathbf{x}$, is fed into the discriminators, g_i. The discriminators compute the values of discriminant functions, $g_i(\mathbf{x})$, which functions, in theory, might be of any arbitrary form. The values output by the discriminators are called discriminants. The discriminants are fed to a maximum selector which chooses the maximum discriminant. If the ith discriminant is chosen as the maximum then the input pattern is classified as being in the ith pattern class. In current neural network terminology the discriminators

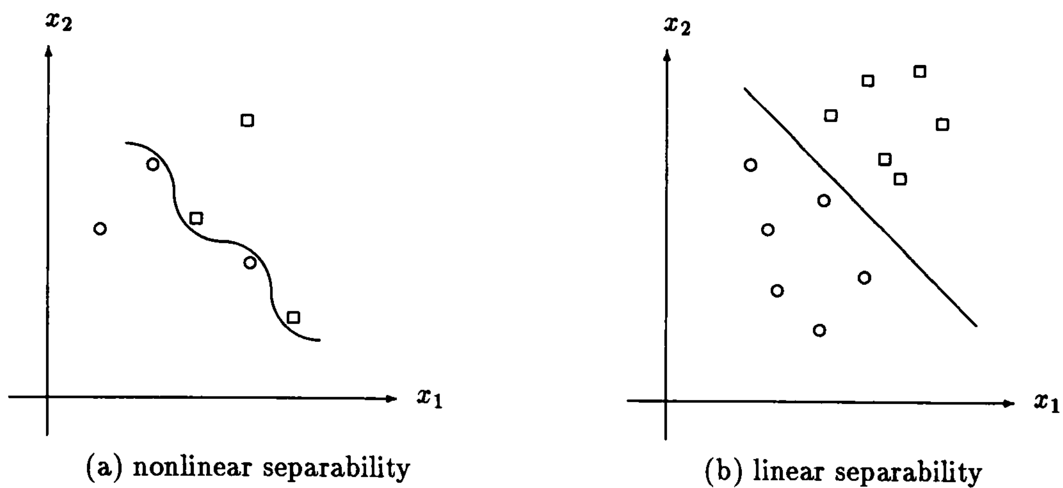


Figure 1: Nonlinear vs. Linear Separability

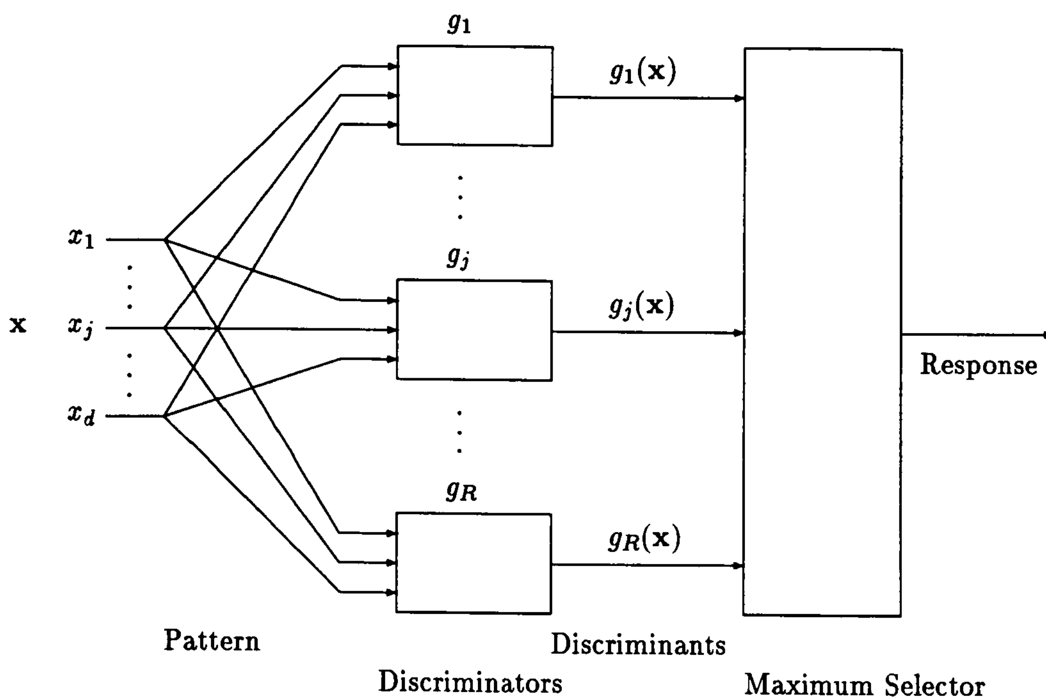


Figure 2: Basic Model for a Pattern Classifier, after Nilsson

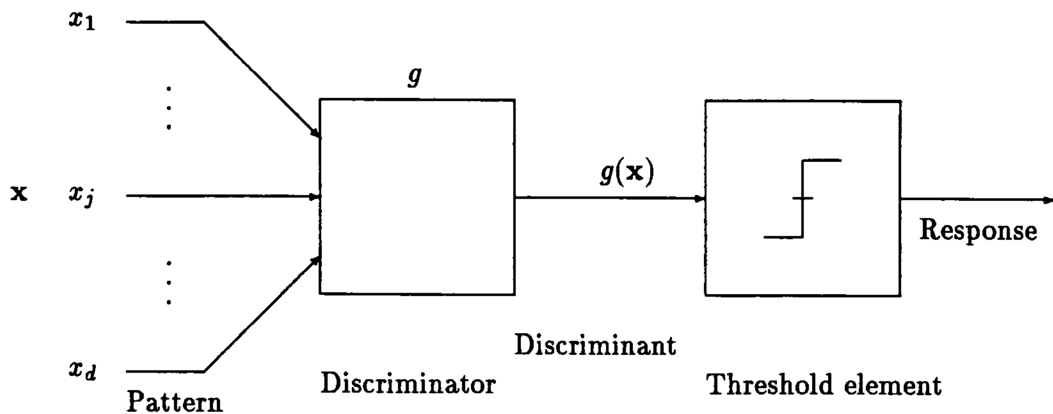


Figure 3: Basic Model for a Pattern Dichotomizer, after Nilsson

would be called neurons, and the neurons would be characterized as linear or nonlinear based on the type of discriminant function employed.

In the two category case where $R = 2$ an interesting form results if we define

$$g(\mathbf{x}) = g_1(\mathbf{x}) - g_2(\mathbf{x}) \tag{1}$$

as a single discriminant function. If the pattern vector to be classified falls into class 1 then $g(\mathbf{x})$ is greater than zero. Conversely if the pattern vector falls into class 2 then $g(\mathbf{x})$ is less than zero. Nilsson referred to this two category pattern classifier as a pattern dichotomizer. This model is illustrated in Figure 3. A threshold element having a threshold of zero is employed to place the input pattern into class 1 or class 2 depending on the sign of $g(\mathbf{x})$.

For the linear classifier the function $g(\mathbf{x})$, which is linear in $\mathbf{x}$, takes the following form

$$g(\mathbf{x}) = w_1x_1 + w_2x_2 + \cdots + w_dx_d + w_{d+1}. \tag{2}$$

The x_i are the components of the pattern vector $\mathbf{x}$ and the w_i are called weights. The term w_{d+1} (denoted the threshold by some authors) is necessary if the hyperplane $g(\mathbf{x}) = 0$ does not pass through the origin of the d dimensional space containing the pattern vectors. This hyperplane is called a decision surface because points lying on one side of the surface are in one class and points lying on the other side are in the other class. Figure 1(b) illustrates two pattern classes which are linearly separable.

For the nonlinear classifier the most general form of the discriminant function was

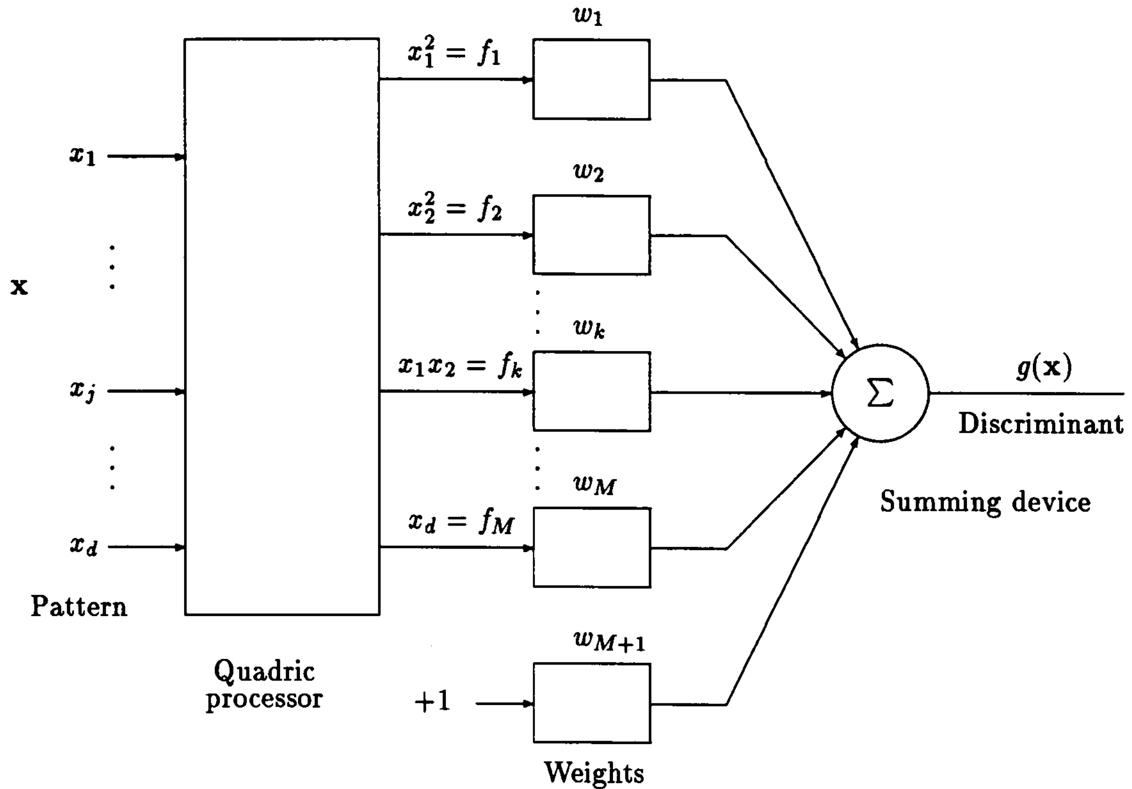


Figure 4: A Quadric Discriminator, after Nilsson

called the Φ function by Nilsson. The general form of this function is

$$\Phi(\mathbf{x}) = w_1 f_1(\mathbf{x}) + w_2 f_2(\mathbf{x}) + \ldots + w_M f_M(\mathbf{x}) + w_{M+1} \tag{3}$$

where the $f_i(\mathbf{x})$ are linearly independent, real, single valued functions which have no dependence upon the weights.

As an illustration, let us look at Nilsson's quadric discriminator. He defined the M functions making up the Φ function as follows: the first d components are $f_1(\mathbf{x}) = x_1^2$, $f_2(\mathbf{x}) = x_2^2, \cdots, f_d(x) = x_d^2$; the next group of functions are all of the product terms $x_i x_j$; the last d components are the x_i terms. The quadric discriminator is illustrated in Figure 4. The pattern vector $\mathbf{x}$ is thus mapped by the Φ function into an M dimensional space which Nilsson called Φ space. $\Phi(\mathbf{x}) = 0$ constitutes a hyperplane in Φ space and is a decision surface. Figure 1(a) illustrates two pattern classes which, while not linearly separable, are nonlinearly separable [20]. The purpose of a Φ function is to map the points to be classified into another space, usually of higher dimensionality, where linear

separability is possible. It is important to note that Φ functions are linear in $f_i(\mathbf{x})$. This means that the same training procedures which might be used for the linear classifier are directly applicable to the nonlinear classifier.

Nilsson gives no indication as to how to choose the appropriate $f_i(\mathbf{x})$ functions for a particular nonlinear classification problem.

Specht [22] in his 1966 Ph.D. Thesis demonstrates a method of finding appropriate polynomial discriminant functions for use in a pattern dichotomizer. Specht finds a decision surface using a Bayesian Classifier method and then approximates that surface with a polynomial. Each of these polynomial terms then becomes a function in a Φ function. This method is still in use today and is referred to as a polynomial adaline or "padaline". It is the direct predecessor of the probabilistic neural network [23]. The decision surfaces of the polynomial adaline are guaranteed to approach the Bayes–optimal decision surfaces as the number of training patterns is increased. The shape of the decision surfaces may be made as complex as necessary to make the desired separation. This must, however, be adjusted by the human operator.

Minsky and Papert [12] in their 1969 book, *Perceptrons*, discuss the limitations of perceptrons. Minsky and Papert refer to "predicates" or "partial functions", which are in effect the $f_i(\mathbf{x})$ in one of Nilsson's Φ functions. That these predicates need not be linear is clearly illustrated by, as an example, the predicate ψ_{PARITY}. Minsky and Papert give no guidance as to the selection of predicates for a problem whose topological properties are not known in advance. Rather, "Our purpose is to explain why there is little chance of much good coming from giving a high–order problem to a quasi–universal perceptron whose partial functions have not been chosen with any particular task in mind." [12]

Duda and Hart [4] in 1973 present the generalized discriminant function which can employ any arbitrary function of the pattern vectors to create decision surfaces of any desired shape. "By selecting these functions judiciously and letting $\hat{d}$ [the dimensionality of the enhanced vector] be sufficiently large, one can approximate any desired discriminant function by such a series expansion." Their presentation is essentially the same as Nilsson's; and, again, no indication is given as to how to choose the appropriate functions.

Similarly Tou and Gonzalez [24] in 1974 present generalized decision functions. No

indication is given as to how to choose the appropriate functions.

1.2.2 Nonlinear Separability in Neural Networks

With the rapid growth of the Neural Network field in the late 1980's the classification problem has been recast in a neural context. The backpropagation algorithm is a new algorithm, but many of the ideas of Neural Networks have their roots in Pattern Recognition.

Backpropagation Networks The introduction of the backpropagation algorithm by Rumelhart, Hinton, and Williams [18] in 1986 created new interest in classification problems because the multilayer backpropagation algorithm was capable of, among other things, learning the nonlinear decision surfaces required for correct classification of non-linearly separable data.

The backpropagation algorithm is a multilayer neural network which has a nonlinear transfer function between layers. The learning algorithm used with the backpropagation network is the generalized delta rule; this is a gradient descent algorithm, from which spring two of the disadvantages of backpropagation. One disadvantage is that backpropagation is slow to converge to a solution; another is that it can get caught in a local minimum. Another problem of backpropagation is that the user must specify the number of neurons in the hidden layers as well as the number of hidden layers; if too few neurons are specified the net will not learn; if too many neurons are specified the network cannot generalize properly.

Sigma–Pi Units Rumelhart and McClelland [19] presented the sigma–pi unit, a neuron which sums the weighted products of two or more elements of the original pattern vector. This may be stated formally by the below equation for the output or activation, $g(\mathbf{x})$, of a neuron having an n dimension input vector $\mathbf{x}$ as follows:

$$g(\mathbf{x}) = \sum_{S_j \in \mathbf{P}} w_j \prod_{i \in S_j} x_i \tag{4}$$

where $\mathbf{P}$ is the power set of $\{1, \cdots, n\}$. From a mathematical point of view, the sigma–pi unit may be seen as a special case of the Φ function. It is therefore obvious that a network

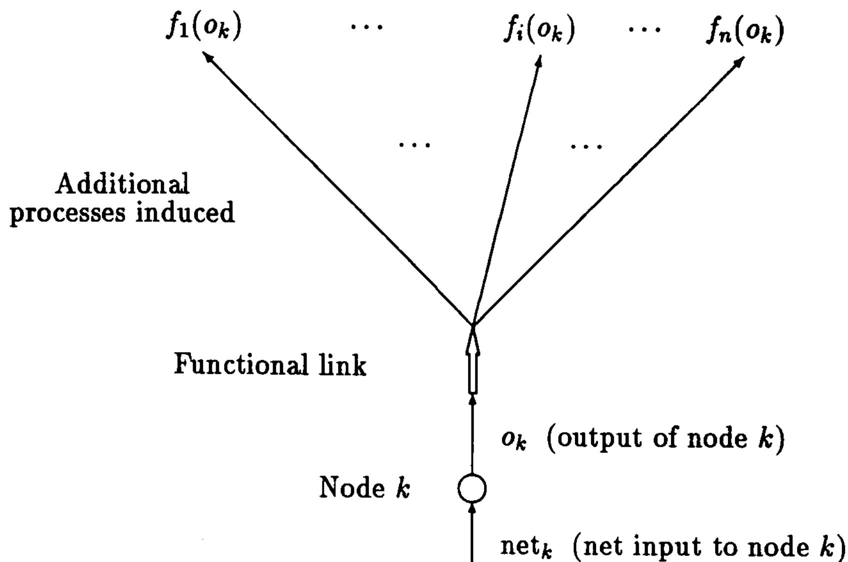


Figure 5: Schematic Illustration of a Functional Link, after Pao

employing sigma–pi units would be capable of making nonlinear separations.

Functional Link Networks As a response to the problems of the backpropagation algorithm, Pao [14] has reintroduced the idea of the nonlinear discriminant function as the functional link network. This is a single layer neural network employing nonlinear discriminant functions. Here the rationale is that components are added to the original pattern vectors so as to create vectors in a higher dimension space in which separation can be obtained by a hyperplane. While, mathematically, Pao's approach is very similar to the earlier work in nonlinear separation described above, his nomenclature is substantially different.

Pao describes the functional link as follows, "The overall concept is that of a *functional* link. Thus, in this mathematically based conceptual model of a net suitable for parallel distributed processing, different additional functionalities may be activated once a node is activated. As illustrated in Figure 8.1 [reproduced as Figure 5], activation of node k offers the possibility that processes $f_0(o_k)$, $f_1(o_k), \cdots, f_n(o_k)$ may also be activated."

As an example, consider input pattern vector component net_k, where net_k is the input

from some lower neural layer to node k. That is,

$$net_k = \sum_{i=1}^{N} w_i x_i + \Theta \tag{5}$$

where Θ is called the threshold and is directly comparable to the w_{d+1} in Equation 2 above. net_k is normally run through a "squashing function", the most commonly used being the logistic function

$$o_k = \frac{1}{1 + e^{-\alpha net_k}} \tag{6}$$

where α is a user controlled constant. o_k is thus the output of the k_{th} node or the k_{th} component in an output vector **x**. The position of the squashing function corresponds to that of Nilsson's threshold element in Figure 3 above. Now that o_k has been determined, so may various functions of o_k, for example, $f_j(o_k) = sin(j\pi o_k)$.

It is instructive to compare the functional link as shown in Figure 5 with Nilsson's Φ function in Equation 3 above.

Pao considers two models, a functional expansion model and an outerproduct model. In the functional expansion model, each vector component is treated as the argument of a function. For an N dimension output vector **x** the first N functions might be of the form $f_i(x_i) = x_i$ where x_i is the same as the o_k above. For each of the x_i there might be a series of functions f_j of the form $f_j(x_i) = sin(j\pi x_i)$. Any other scalar valued functions could just as easily be used, for example $f_j(x_i) = x_i^j$. It is important to note that in Pao's functional expansion model the functions must be functions of a single component of an output vector.

In the outerproduct model, additional components are created by taking the outer product of **x** with itself. If we consider **x** to be an N dimension row vector and augment it with a 1 in order to retain the components of the original vector we get the following result:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \\ 1 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_N & 1 \end{bmatrix} = \begin{bmatrix} x_1x_1 & x_1x_2 & \cdots & x_1x_N & x_1 \\ x_2x_1 & x_2x_2 & \cdots & x_2x_N & x_2 \\ \vdots & & \ddots & & \vdots \\ x_Nx_1 & x_Nx_2 & \cdots & x_Nx_N & x_N \\ x_1 & x_2 & \cdots & x_N & 1 \end{bmatrix} \tag{7}$$

It is not necessary to retain duplicate terms, and square terms are not considered to be outerproduct terms, so they are not retained. As a generalization, the outerproduct may be applied recursively to generate polynomials of higher degree. Figure 6 summarizes three of the most frequently used expansion types.

High–Order Neural Networks Independently of Pao, high–order neural networks have been investigated by Giles and Maxwell [6]. In their terminology, "First–order units are units which are linear in the sense that that they can capture only first–order correlations." First–order units may be represented as

$$y_i(x) = S\left[\sum_j^N W(i,j)x(j)\right] \tag{8}$$

where $\mathbf{x} = \{x(j)\}$ is an input vector, $W(i,j)$ are the weights, N is the dimensionality of $\mathbf{x}$ and S is some unspecified sigmoid function which could, for example, be Equation 6 above.

Giles and Maxwell are interested in problems which have a high–order correlational structure; therefore, they have investigated high–order units. Their general equation for high–order units is

$$y_i(x) = S\left[W_0(i) + \sum_j W_1(i,j)x(j) + \sum_j \sum_k W_2(i,j,k)x(j)x(k) + \ldots\right] \tag{9}$$

where the higher order weights capture higher order correlations. A unit which includes terms involving the product of k distinct components of $\mathbf{x}$ is referred to as a kth order unit. Note the similarity between Equation 9 and Equation 3 above. We may thus see the high–order units of Giles and Maxwell as special cases of Nilsson's Φ function with the substitution of a sigmoid function for the threshold function of Figure 3. There is also a direct comparison between the high–order unit and the sigma–pi unit as represented in Equation 4 above.

Outerproduct expansion (OP)

original vector	$x_1\ x_2\ x_3\ \cdots\ x_d$
first expansion group	$\{x_i x_j\}\ i < j$
second expansion group	$\{x_i x_j x_k\}\ i < j < k$
third expansion group	$\{x_i x_j x_k x_m\}\ i < j < k < m$
etc.	

Polynomial expansion (PL)

original vector	$x_1\ x_2\ x_3\ \cdots\ x_d$
first expansion group	$\{x_i x_j\}\ i \leq j$
second expansion group	$\{x_i x_j x_k\}\ i \leq j \leq k$
third expansion group	$\{x_i x_j x_k x_m\}\ i \leq j \leq k \leq m$
etc.	

Trigonometric expansion (T)

original vector	$x_1\ x_2\ x_3\ \cdots\ x_d$
first expansion group	$\{\sin(\pi x_i)\ \cos(\pi x_i)\}$
second expansion group	$\{\sin(2\pi x_i)\ \cos(2\pi x_i)\}$
third expansion group	$\{\sin(4\pi x_i)\ \cos(4\pi x_i)\}$
etc.	

Figure 6: Expansion Term Grouping

2 Statement of the Problem

The problem with networks using nonlinear functions to make nonlinear separations lies in the choice of the functions used to enhance the pattern vectors. Pao [14] discusses his experience in using different expansions and states that different expansions give different results for different types of problem. Qian et al [16], in the context of using a network for functional approximation, give the following opinion: "Although we have a great freedom to choose a complete orthogonal basis, some basis functions may perform better than others, depending on the actual problem. For instance, cosines are usually more accurate than sine/cosines for most problems." Maxwell et al [11], in the context of the contiguity problem, sum up the situation. "By choosing a set of high order terms which embodies prior knowledge about the problem domain, we are able to construct a network which can learn and generalize very efficiently within its designated environment. By providing the network with the tools it needs to solve the problems it expects to encounter, we liberate the network from the difficult task of deciding which tools it will need and then creating those tools. This process constitutes a major part of the learning procedure for networks utilizing back propagation."

Giles and Maxwell [6] summarize the problem of finding a set of nonlinear expansion functions and suggest four possible approaches.

Their first approach is to match the order of the network to the order of the problem (as defined in Minsky and Papert [12]). Where the order of a problem is known, a network of suitable order can be created to solve it. Where the order of a problem is unknown, it can be estimated.

The second suggested approach is implementation of invariances. If we have *a priori* knowledge that a problem has a given set of invariances, they may be implemented by the correct choice of expansion terms.

Their third suggested approach is the use of correlation calculations to find which input terms have the highest correlation with the output of the network. Terms having low correlations are dropped.

They also suggest generating representations adaptively. In this approach the network

is progressively adapted to suit the problem. Backpropagation is cited as an example of this approach. Another suggested method is the use of genetic search algorithms to find suitable high–order representations.

In summary, the problem of finding a particular set of functions which could be used to implement Equation 3 above has been around since the 1960's and, while the nomenclature of the nonlinear classification problem has changed as it moved from the pattern recognition domain to the neural network domain, the finding of the correct functions to use for a particular problem has remained a serious obstacle to the implementation of the single layer nonlinear classifier.

The problem considered below may be stated as follows: given a training set of pattern vectors consisting of patterns from two separable classes, construct adaptively and without regard for any special features of the problem (for example, order in Minsky and Papert's [12] sense) a single layer network capable of separating the points of the two classes.

3 A Description of the Algorithm

We describe a high-order network of nonlinear neurons which can be created by the training process. In this algorithm a multilayer network architecture is constructed adaptively. In the event that the algorithm decides to extend the network to include another layer, the requisite expansions and transformations are made deterministically. Only in the final layer, after linear separability has been obtained, are weights determined by an adaptive process. Therefore, much of the learning carried out under this algorithm is done through adaptive network architecture rather than through adaptive network connections.

The proposed network is based on two well known techniques from pattern recognition: the Ho–Kashyap algorithm (HK) [8] and the Karhunen–Loève expansion (KL) [25]. The introduction of nonlinearity at each level is done using the outerproduct expansion (OP) [14]. The flow chart shown in Figure 7 outlines the combination of these methods into an algorithm. The multilayer network constructed by the proposed algorithm can be expressed in the standard network format, as shown in Figure 8. Since there are, in general, no restrictions on the nature of the expansion terms no connectivity is shown for the expansion terms.

Once the network has been completely constructed, it may be reduced to a single layer functional link network by passing a vector $\mathbf{x}$ having symbolic components x_i through the multilayer network. The result is a polynomial in the x_i. The symbolic portion of each term of the polynomial represents one expansion function of an equivalent one layer functional link net, and the numerical coefficient of that term is the associated weight.

In order to understand the overall algorithm, a review of the components may be helpful. Each component of the algorithm is discussed briefly below, then the overall algorithm is discussed. Note that each of these components is off the shelf technology. The originality of this approach lies in the combination of components and in the repeated use of the outerproduct expansion on the expanded and transformed pattern vector at each stage, not on the original pattern vector.

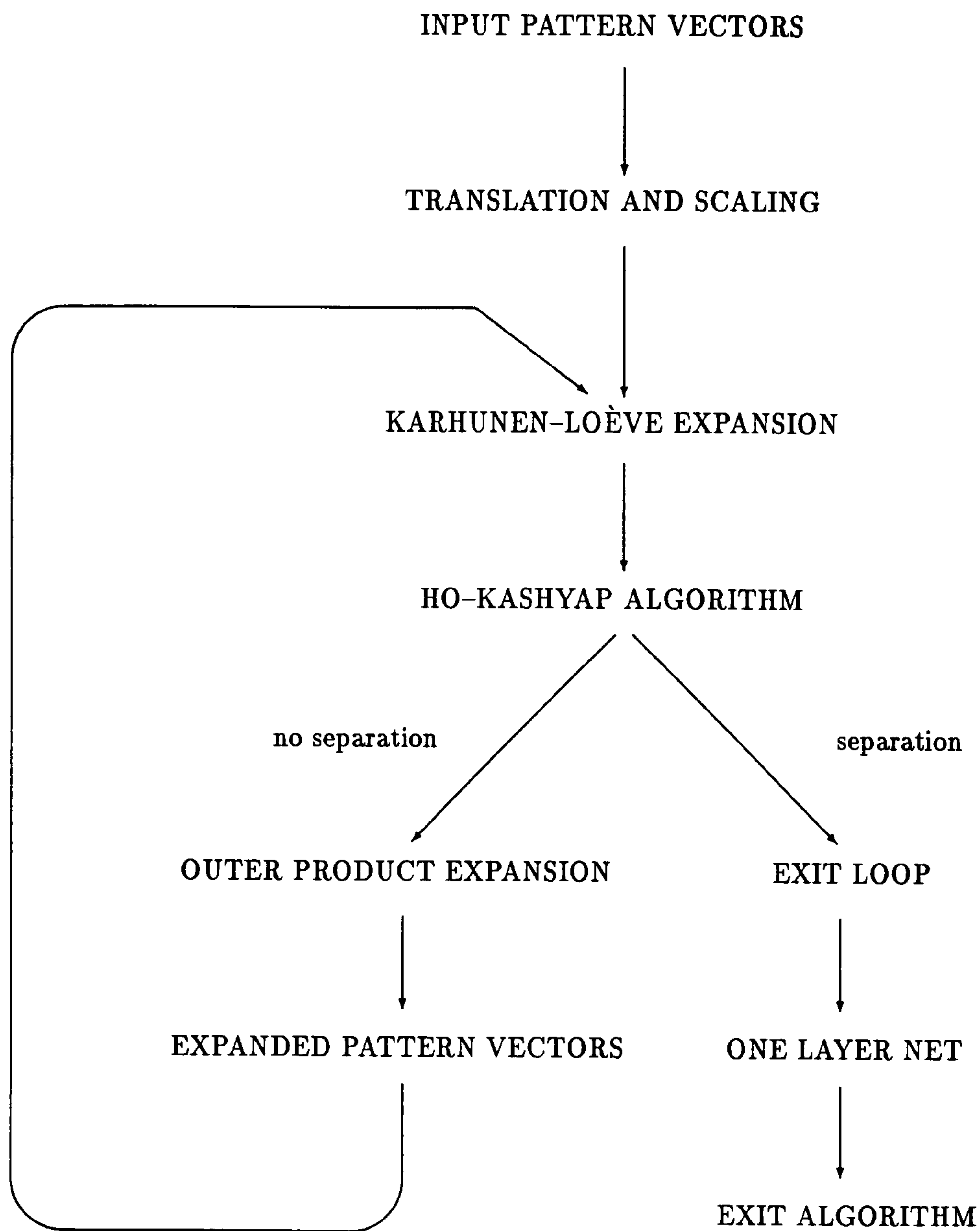


Figure 7: Flow Chart for Self Configuring Network

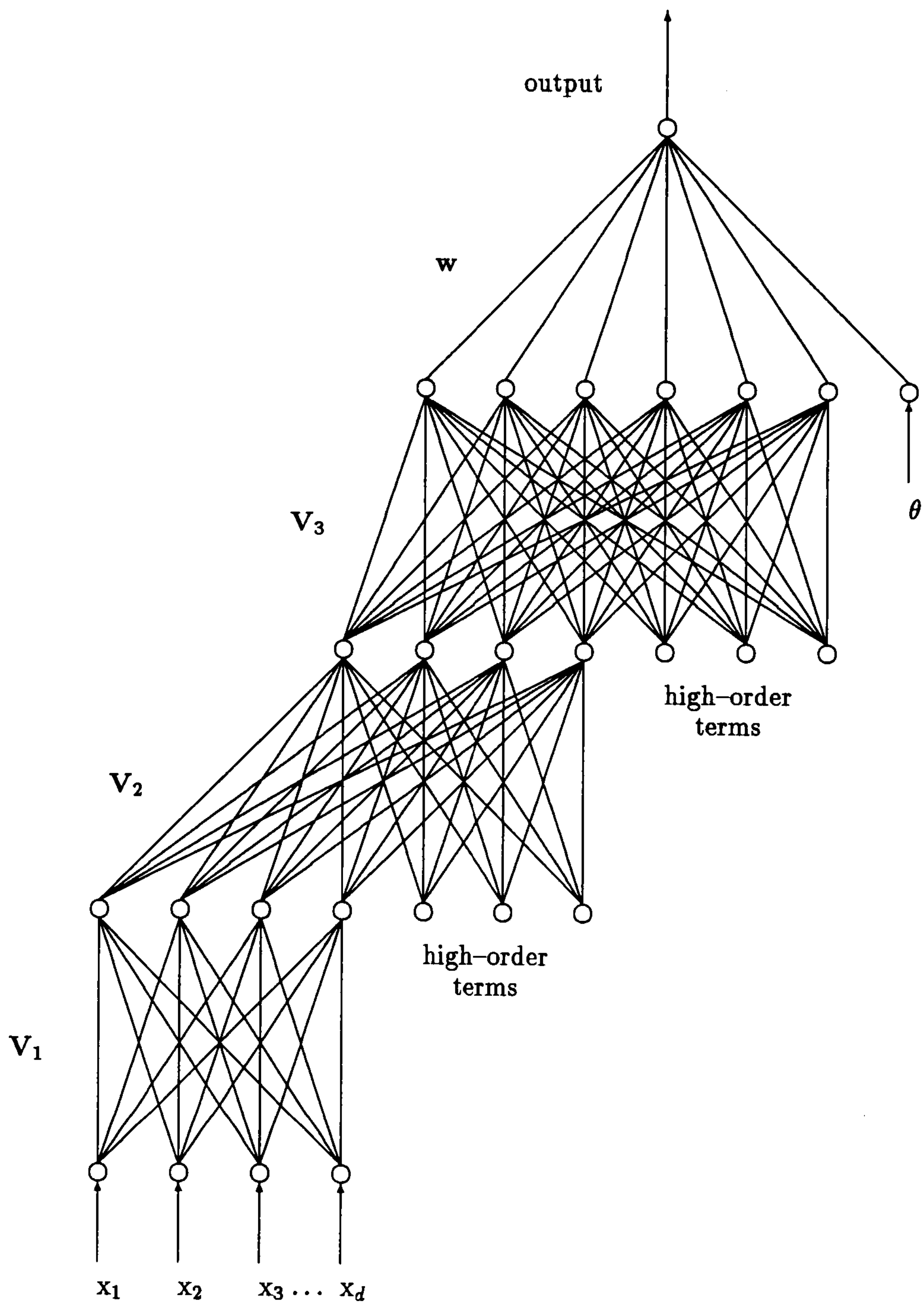


Figure 8: Multilayer Net as Constructed by Successive Expansions and Transformations

3.1 The Outerproduct Expansion

The outerproduct expansion (OP) [14] consists of terms made up of two or more components of the original pattern vector multiplied together and takes its name from the outerproduct of two vectors, as illustrated in Equation 7. The outerproduct expansion is illustrated in Table 6 along with the polynomial and trigonometric expansions for comparison. The outerproduct terms $x_i x_j$ may be considered to capture some correlation between the ith and the jth components of the pattern vectors. Higher–order terms such as $x_i x_j x_k$ may be thought of as capturing correlations between three pattern components. The order of terms is limited only by the dimensionality of the pattern vector; however, in our algorithm, only the second order terms are generated.

For the Functional Link Net, an original pattern vector of dimension d is augmented by appending one or more terms, each such term consisting of some outerproduct expansion of the original pattern vector. A training set of pattern vectors consisting of patterns from two categories is augmented by using the same outerproduct terms for the augmentation of each pattern vector.

Since the outerproduct concept has its roots in associative memory models, it is instructive to briefly review associative memory. One of the models of memory in living organisms is that memory is the result of changes in the characteristics of the synaptic junctions between neurons. The most commonly accepted explanation for changes in synaptic connectivity between two neurons is that postulated by D. O. Hebb [7] in 1949:

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells, such that A's efficiency, as one of the cells firing B, is increased."

Hebb's statement was interpreted by Teuvo Kohonen [9] in a mathematical sense. Given an assortment of synaptic junctions, let m_{ij} be the strength of the ij^{th} junction, let f_i be the strength of the ith forcing stimulus, let s_j be the strength of the jth response stimulus, then,

$$\frac{d}{dt} m_{ij} = p f_i s_j \tag{10}$$

where p is the plasticity of the synaptic connection. For simplicity, p is normally considered to be a constant.

If it is assumed that $m_{ij} = 0$ at $t = 0$ then the above equation has the following solution:

$$m_{ij} = pf_i s_j t. \tag{11}$$

In words, the synaptic strength of a junction is a function of the product of the strength of the ith forcing stimulus times the strength of the jth response stimulus times the plasticity of the neuronal connection times the duration of application of the stimuli.

It is convenient to assume that the stimuli are applied for a time equal to $1/p$; then at time $1/p$ the strength of the synaptic connection is:

$$m_{ij} = f_i s_j. \tag{12}$$

If the stimuli are considered to be respectively the ith and j^{th} components of vectors $\mathbf{f}$ and $\mathbf{s}$ and m_{ij} is considered to be the ij^{th} component of a matrix M the above equation can be restated in matrix algebra terminology as:

$$\mathbf{M} = \mathbf{f}\mathbf{s}^T \tag{13}$$

If we further assume that N different $\mathbf{f}$ $\mathbf{s}$ pairs are sequentially presented to the neurons and that their effects on the synaptic connections are additive, Equation 14 results. This equation defines the Distributed Associative Memory Model of Teuvo Kohonen.

$$\mathbf{M} = \sum_{i=1}^{N} \mathbf{f}_i \mathbf{s}_i^T \tag{14}$$

In the special case where $\mathbf{f_i} = \mathbf{s_i}$ for $i = 1, \ldots, N$ we have autoassociative memory and Equation 14 becomes

$$\mathbf{M} = \mathbf{X}\mathbf{X}^T \tag{15}$$

where the $\mathbf{X}$ is a matrix made up of pattern vectors in column vector format. This is the familiar autocorrelation matrix. In associative memory applications the $m_i m_i$ terms are then set to zero because of the observed lack of self–feedback in biological neural systems [10].

3.2 The Ho–Kashyap Algorithm

The Ho–Kashyap algorithm (HK) [8] is a pattern classification algorithm for making linear separations. It has two very important properties which set it apart from other, similar algorithms. One, it converges rapidly to a solution if one exists; and two, if no solution exists then the algorithm signals nonseparability.

We present below an outline of the Ho–Kashyap algorithm. Complete convergence proofs may be found in Ho and Kashyap [8], Duda and Hart [4], and Tou and Gonzales [24].

Suppose that we have a matrix $\mathbf{X}$ of pattern vectors made up of row vectors $\mathbf{x}_i^j$ where the superscript represents the class number and the subscript indicates the vector number. The pattern vectors are augmented with an additional element which has the value 1, and class 2 patterns are negated. Then $\mathbf{X}$ has the following form.

$$\mathbf{X} = \begin{bmatrix} 1 & \mathbf{x}_1^1 \\ 1 & \mathbf{x}_2^1 \\ \vdots & \vdots \\ -1 & -\mathbf{x}_{m-1}^2 \\ -1 & -\mathbf{x}_m^2 \end{bmatrix} \tag{16}$$

The standard two category classification problem may then be stated as finding a column vector $\mathbf{w}$, the weight vector, such that

$$\mathbf{X}\mathbf{w} > 0. \tag{17}$$

This matrix equation combines the scalar equations, Equation 2, for all of the pattern vectors and by the negation of class 2 patterns includes the threshold element of Figure 3.

For the Ho–Kashyap algorithm a slight variation is introduced

$$\mathbf{X}\mathbf{w} = \mathbf{b} \tag{18}$$

where $\mathbf{b}$ is a vector whose components are all positive. A loss function for gradient descent is then introduced which is a function of $\mathbf{w}$ and $\mathbf{b}$.

$$J(\mathbf{w}, \mathbf{b}) = \frac{1}{2} \| \mathbf{X}\mathbf{w} - \mathbf{b} \|^2 \tag{19}$$

Note that $\mathbf{b}$ must remain positive throughout the gradient descent. There are then two gradients which may be brought to zero.

$$\nabla_{\mathbf{w}} J(\mathbf{w}, \mathbf{b}) = \mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{b} \tag{20}$$

$$\nabla_{\mathbf{b}} J(\mathbf{w}, \mathbf{b}) = \mathbf{b} - \mathbf{X}\mathbf{w} \tag{21}$$

Since there are no constraints on $\mathbf{w}$ we may set Equation 20 to zero.

$$\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{b} = 0 \tag{22}$$

Equation 22 is then the familiar least squares fit problem which we may solve for $\mathbf{w}$ as follows.

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{b} \tag{23}$$

The overall Ho–Kashyap algorithm may now be summarized as follows [8].

1. Assume that $\mathbf{b}$ is fixed and use Equation 23, the least squares fit, to compute $\mathbf{w}$.

2. Assume that $\mathbf{w}$ is fixed and use Equation 21, gradient descent, to minimize $J(\mathbf{w}, \mathbf{b})$ by varying $\mathbf{b}$.

Steps 1 and 2 are alternated until convergence is achieved or nonseparability is signaled.

It should be noted that in the case where the classes are not linearly separable the HK algorithm is not guaranteed to signal nonseparability in finite time [20]. We have found that in practice it is necessary to set some upper limit to the number of iterations the HK algorithm is allowed to make. If, within that limit, the HK algorithm has signalled neither separability nor nonseparability we consider the classes nonseparable and move on to the outerproduct expansion, as shown in Figure 7.

The HK algorithm also has a serious drawback: it requires the calculation of the pseudoinverse of the matrix $\mathbf{X}$, Equation 23. However, the use of the Karhunen–Loève expansion to reduce the dimensionality of the pattern vectors, as an added benefit, produces a set of transformed pattern vectors which are uncorrelated. Thus, the matrix $\mathbf{X}^T\mathbf{X}$ is, in theory, diagonal and may be inverted easily.

3.3 The Karhunen–Loève Expansion

The Karhunen–Loève expansion (KL) [25] is a powerful tool for compressing information and is useful in pattern recognition for the preselection of variables. The pattern vectors are reexpressed in terms of a set of orthogonal basis vectors. Not all basis vectors will have the same weight or importance; those of minor importance can be dropped with very little error.

We present below an outline of the KL expansion and its principal characteristics. A complete proof of the KL expansion may be found in Fukunaga [5].

Let $\mathbf{x}$ be an n dimension random column vector, then we may express $\mathbf{x}$ as a linear transformation of $\mathbf{y}$

$$\mathbf{x} = \mathbf{\Phi}\,\mathbf{y} \tag{24}$$

where $\mathbf{\Phi} = [\phi_1\ \phi_2\ \cdots\ \phi_n]$ and $\mathbf{\Phi}$ is nonsingular. We may further assume that the ϕ_i form an orthonormal set of basis vectors made up of the normalized eigenvectors of the autocorrelation matrix of the pattern vectors. Then if we premultiply Equation 24 by $\mathbf{\Phi}^T$

$$\mathbf{\Phi}^T\,\mathbf{x} = \mathbf{y} \tag{25}$$

results. $\mathbf{y}$ is now an orthonormal transformation of $\mathbf{x}$. Let $\mathbf{R}$ be the autocorrelation matrix, then

$$\mathbf{R} = \mathbf{X}\mathbf{X}^T \tag{26}$$

where the $\mathbf{X}$ is a matrix made up of pattern vectors in column vector format. $\mathbf{\Phi}$ must then satisfy

$$\mathbf{R}\mathbf{\Phi} = \Lambda\,\mathbf{\Phi} \tag{27}$$

where Λ is the diagonal matrix of eigenvalues.

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \tag{28}$$

We may consider each component of $\mathbf{y}$ as a feature which contributes to representing the original vector $\mathbf{x}$. Now suppose that we wish to reduce the dimensionality of $\mathbf{y}$ by

leaving off some row or rows of Φ^T. The contribution of each component of $\mathbf{y}$ is measured by its corresponding eigenvalue. If we delete a component of $\mathbf{y}$, say y_i, by deleting the corresponding eigenvector, ϕ_i, from Φ then the mean–square error increases by λ_i. In order to minimize error, the component of $\mathbf{y}$ having the smallest corresponding eigenvalue should be dropped first. It can be shown that in order to minimize the error from deleting a component of $\mathbf{y}$ the best choice for Φ is a matrix containing the normalized eigenvectors of the autocorrelation matrix of the pattern vectors [5].

It should be noted that the components of the pattern vectors, y_i, are uncorrelated; therefore, in theory, the autocorrelation matrix of $\mathbf{Y}$ should be diagonal. This feature of the KL expansion has potential to ease the calculation of the pseudoinverse matrix in the HK algorithm.

3.4 The Working of the Overall Algorithm

The learning algorithm, as described in Figure 7, has been implemented for the most part by a C language program, fl1, which has been tested on a 80386 based IBM AT clone running under MS–DOS and a on VAX running under VMS. The source code for this program is listed in Appendix B. The reduction of the multilayer network to the equivalent single layer network has been done by hand calculation and the use of the MATLAB software package.

Appendix C contains the classifier program, class, which reads the output file from the fl1 program and implements the multilayer form of the high–order network. This multilayer network may be used to make classifications of pattern vectors which are randomly generated or derived from some other source. In the case of 2–dimensional problems, these patterns are then plotted on the screen either as white dots on black or black dots on black. The result is a speckled region which represents class 1 and a black region which represents class 2. The resulting screen display may then be printed in reverse colors using the MS-DOS screen printing utility.

Input Pattern Vectors The first step of the algorithm is to input the pattern vectors from a disk file. The required format for this disk file is shown in the introductory

comments of source code file fl1.c in Appendix B.

Translation and Scaling The pattern vectors are translated so that the mean vector of all patterns is **0**. The patterns are then scaled so that the mean of the absolute values of all pattern components is 1.0. The translation and scaling factors are written to an output file for later use. The patterns are then augmented by the appending of a 1 to each pattern vector. Class 2 patterns are multiplied by -1. The purpose of the translation and scaling is to reduce every problem to the same format.

Karhunen–Loève Expansion The translated and scaled patterns now enter the main loop of the algorithm with a KL expansion to reexpress the pattern vectors in terms of a new orthogonal basis set, hopefully of reduced dimensionality, and to remove any correlation between vectors.

It is at this step that an important decision must be made: how many eigenvectors should be dropped (if any) in the reduction of the dimensionality of the pattern vectors. This decision may be expressed as a KL threshold, or minimum value for the eigenvalues; that is, if any eigenvalue is below the KL threshold its corresponding eigenvector should be dropped.

If the KL threshold is too low, features will be retained in the pattern vectors which contribute almost nothing to separability. This in itself is not bad, but these retained features are included in the following outerproduct expansion. This results in larger matrices and more lengthy calculations at every subsequent step. Without any dimensionality reduction, successive second order outerproduct expansions produce a growth in dimensionality of order d^2 where d is the dimensionality of the pattern vector before the expansion. The penalty for keeping unnecessary features is, at best, slower execution of the algorithm and, at worst, a combinatorial explosion which exhausts all computer memory.

Conversely, if the KL threshold is set too high, some features will be lost which contribute significantly to separability. This means that the main algorithm must go through additional iterations in order to generate a set of expansion terms which will yield linear separability. In some cases the algorithm is unable to generate suitable features having large enough eigenvalues to exceed the KL threshold and thus cycles endlessly without

finding a solution.

Two approaches to setting the KL threshold have been tried; one approach is to use a fixed KL threshold. A value of 0.0001 works well for problems which require higher order correlations for their solution, for example, parity problems or the 12 point problem shown in Figure 18.

The other approach tried is to use a variable KL threshold based on a multiple of the sum of the eigenvalues. This approach allows us to set the percentage of mean–square error we are willing to tolerate at each invocation of the KL expansion [5]. A KL threshold of 0.0001 times the sum of all of the eigenvalues works well for problems which are complex but of low order as defined by Minsky and Papert [12]. An example of this type of problem is the loop problem shown in Figure 16.

The resultant matrix of eigenvectors is written to a disk file for future use.

Ho–Kashyap Algorithm The Ho–Kashyap algorithm is then used to determine if the patterns are linearly separable. The flow of program execution has a branch point here which depends on separability. If the patterns presented to the HK algorithm are linearly separable then the final weight vector $\mathbf{w}$ is written to an output file for later use and the training phase of the overall algorithm is complete. If the patterns are not linearly separable then the algorithm goes to the outer product expansion step.

Outerproduct Expansion In the outerproduct expansion step the pattern vectors undergo a second order expansion. This expansion is done using the components of the pattern vectors as they exist at that point in the algorithm. After the first iteration through the main loop of the overall algorithm at least some of the terms of the pattern vectors will contain the effects of the previous expansion (having been linearly transformed by the KL expansion). In this way, higher order correlations can be constructed. Or if the problem is of low order but requires a high degree polynomial decision surface, then that polynomial may be constructed in several iterations through the outerproduct expansion. It should be noted that this iterated OP expansion, as it is currently implemented, is not capable of generating any arbitrary polynomial; for example, it cannot generate powers of only one vector component.

Reiteration After the pattern vectors are expanded they are sent on to the KL expansion for the beginning of the next iteration of the algorithm.

Exit Loop The loop as described above has only one normal exit point and that is from the H–K algorithm. The criterion for exiting the loop is linear separability of the two classes of expanded pattern vectors.

In the case of overlapping classes of data points, the algorithm will attempt to separate the individual pattern vectors in the overlapping region of the sample data set. The algorithm has no way of distinguishing between correlation type separation problems for which we wish it to find a way to separate data which overlap when represented as pattern vectors (parity problems, for example) and overlapping category classification problems for which we would like to get a nice minimum error classification surface of the sort produced by the Bayesian classifier.

The complexity of the separation which the algorithm can make is limited by computer memory and the time required for the calculations. In the event that all available computer memory is allocated and the algorithm calls for more, the algorithm will abort without success. The user may set some upper time limit for the execution of the algorithm in order to prevent endless cycling.

Reduction to One Layer Once linear separability has been obtained, we are in a position to reduce the multilayer network created in the above steps to a single layer functional link network. This is done by passing a symbolic valued vector of the same dimensionality as the training vectors through all of the transformations required to achieve linear separability. This may best be explained by use of an example.

The standard example problem which illustrates the problems of nonlinear separation is the venerable xor problem, shown in Figure 9. This is the simplest version of the parity problem studied by Minsky and Papert [12]. A backpropagation solution is presented by Rumelhart and McClelland [19] and a functional link solution by Pao [14]. Figure 10 shows the output of the program which implements the multilayer algorithm as described above. Figure 11 illustrates the network and the various transformations involved in the multilayer xor solution.

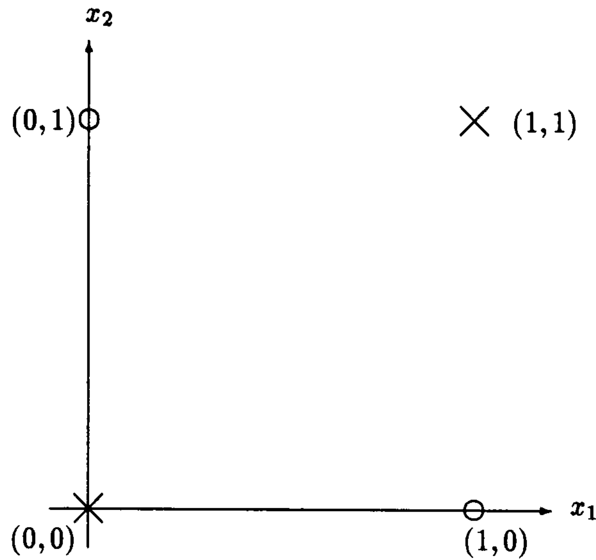


Figure 9: The XOR Problem

A two dimensional row vector $\mathbf{x}$ is introduced at the bottom of Figure 11. This vector passes through the translate and scale operation as follows. First $\mathbf{x}$ is translated

$$\mathbf{x}_t = \mathbf{x} - \mathbf{t} \tag{29}$$

$$\begin{bmatrix} x_{1,t} & x_{2,t} \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} - \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} = \begin{bmatrix} x_1 - 0.5 & x_2 - 0.5 \end{bmatrix} \tag{30}$$

and then scaled.

$$\mathbf{x}_{t,s} = \frac{1}{s}\mathbf{x}_t \tag{31}$$

$$\begin{bmatrix} x_{1,t,s} & x_{2,t,s} \end{bmatrix} = \frac{1}{0.5}\begin{bmatrix} x_{1,t} & x_{2,t} \end{bmatrix} = \begin{bmatrix} 2.0x_1 - 1.0 & 2.0x_2 - 1.0 \end{bmatrix} \tag{32}$$

The translated and scaled vector $\mathbf{x}_{t,s}$ is then transformed by the first matrix generated by the K–L expansion.

$$\mathbf{x}_{t,s,V_1} = \mathbf{x}_{t,s}\mathbf{V}_1 \tag{33}$$

$$\begin{bmatrix} 2.0x_1 - 1.0 & 2.0x_2 - 1.0 \end{bmatrix}\begin{bmatrix} 0.0 & 1.0 \\ 1.0 & 0.0 \end{bmatrix} = \begin{bmatrix} 2.0x_2 - 1.0 & 2.0x_1 - 1.0 \end{bmatrix} \tag{34}$$

```
1.000000                                        /* value of theta */
2                                         /* dimensionality of X */
0.500000  0.500000                          /* translation vector */
0.500000                                          /* scale factor */
 t                                    /* a matrix transformation */
2 2                                          /* matrix dimensions */
  0.000000    1.000000                               /* matrix V1 */
  1.000000    0.000000
 t                                    /* a matrix transformation */
3 3                                          /* matrix dimensions */
  0.000000    1.000000    0.000000                  /* matrix  V2 */
  0.000000    0.000000    1.000000
  1.000000    0.000000    0.000000
 w                                           /* weight vector w */
4                                         /* dimensionality of w */
  0.000000    1.001245    0.000000    0.000000  /* vector w */
```

Figure 10: Program Output for XOR Problem

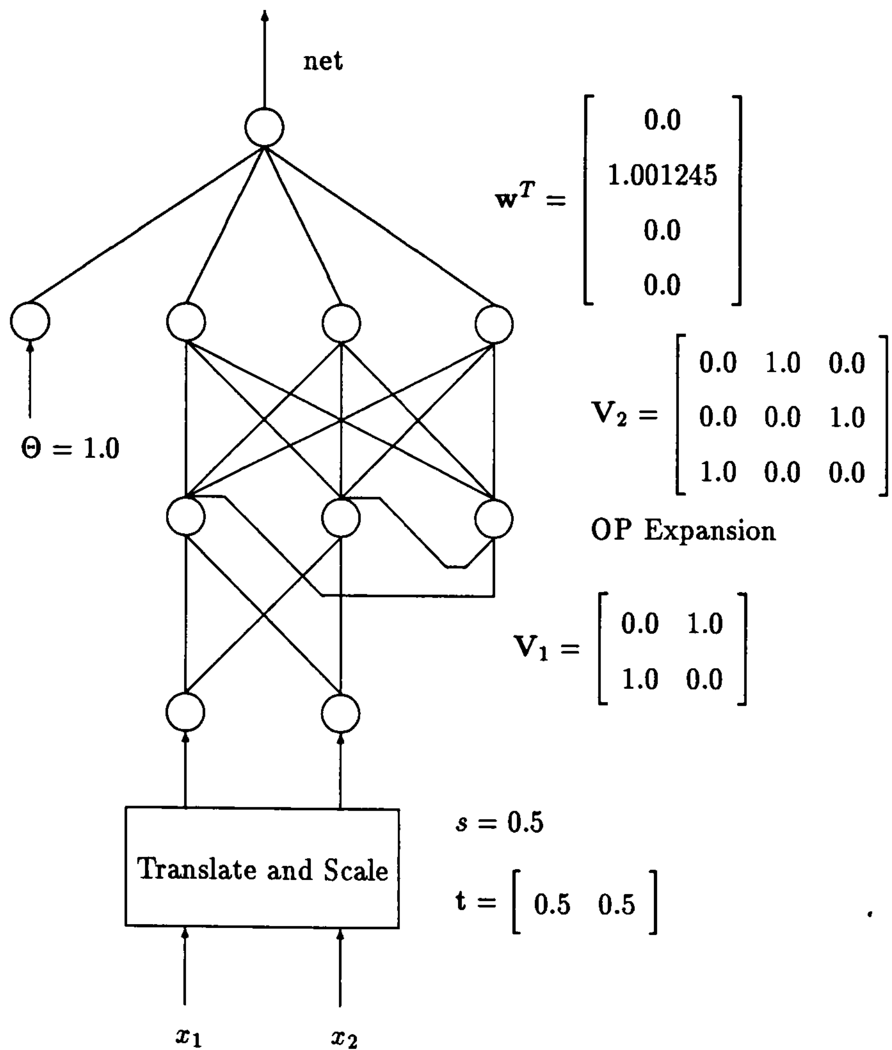


Figure 11: Multilayer Network Solution to XOR Problem

The transformed vector $\mathbf{x}_{t,s,V_1}$ is then expanded using the OP expansion to give $\mathbf{x}_{t,s,V_1,OP}$.

$$\begin{bmatrix} x_{1,t,s,V_1,OP} & x_{2,t,s,V_1,OP} & x_{3,t,s,V_1,OP} \end{bmatrix} = \begin{bmatrix} x_{1,t,s,V_1} & x_{2,t,s,V_1} & x_{1,t,s,V_1}x_{2,t,s,V_1} \end{bmatrix} \quad (35)$$

$$\mathbf{x}_{t,s,V_1,OP} = \begin{bmatrix} 2.0x_2 - 1.0 & 2.0x_1 - 1.0 & 4.0x_1x_2 - 2.0x_1 - 2.0x_2 + 1.0 \end{bmatrix} \quad (36)$$

The expanded vector $\mathbf{x}_{t,s,V_1,OP}$ is then transformed by the second matrix generated by the K–L expansion.

$$\mathbf{x}_{t,s,V_1,OP,V_2} = \mathbf{x}_{t,s,V_1,OP}\mathbf{V}_2 \quad (37)$$

$$\begin{bmatrix} 2.0x_2 - 1.0 & 2.0x_1 - 1.0 & 4.0x_1x_2 - 2.0x_1 - 2.0x_2 + 1.0 \end{bmatrix} \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 1.0 & 0.0 & 0.0 \end{bmatrix} =$$

$$\begin{bmatrix} 4.0x_1x_2 - 2.0x_1 - 2.0x_2 + 1.0 & 2.0x_2 - 1.0 & 2.0x_1 - 1.0 \end{bmatrix} \quad (38)$$

The resultant vector $\mathbf{x}_{t,s,V_1,OP,V_2}$ is augmented by an additional component of value 1.0 and then multiplied by the weight vector $\mathbf{w}$. The result is a scalar value *net*.

$$net = \mathbf{x}_{t,s,V_1,OP,V_2}\mathbf{w}^T \quad (39)$$

$$\begin{bmatrix} 1.0 & 4.0x_1x_2 - 2.0x_1 - 2.0x_2 + 1.0 & 2.0x_2 - 1.0 & 2.0x_1 - 1.0 \end{bmatrix} \begin{bmatrix} 0.0 \\ 1.001245 \\ 0.0 \\ 0.0 \end{bmatrix} =$$

$$4.00498x_1x_2 - 2.00249x_1 - 2.0249x_2 + 1.001245 = net \quad (40)$$

Thus the entire network can be expressed as a single scalar valued equation which is a polynomial in the components of the original input vector $\mathbf{x}$.

This scalar equation may then be represented as a single layer functional link network as shown in Figure 12.

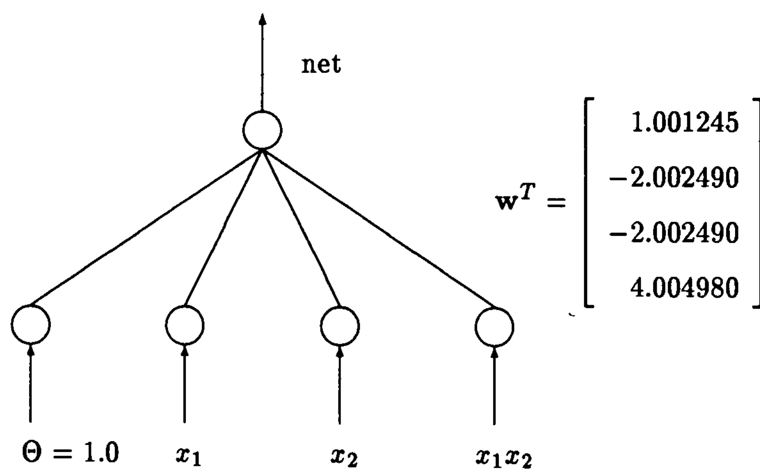


Figure 12: Equivalent Single Layer Functional Link Solution to XOR Problem

4 Results and Conclusions

We present below the general characteristics of the algorithm and a summary of results from solving a series of simple problems and a more realistic problem using nuclear power plant simulator data from the Watts Bar Nuclear Power Station [1,2].

4.1 General Characteristics of the Algorithm

Separation The objective of this algorithm is to achieve separation between two data sets. Accordingly, the algorithm runs until it achieves separation, it uses all available memory, or the user aborts the algorithm. This characteristic of the algorithm is good if complete separation is what is desired; for example, a parity problem in which it is desired to separate the various data points which overlap in vector space. However, there are many classification problems in which the classes overlap and it is desired to find a decision surface which minimizes the error of classification. This algorithm as it stands is unable to distinguish between the two types of classification problem.

It should be noted that while there is a convergence proof for the functional expansion model [21], there is no convergence proof for the outerproduct expansion model. However, the outerproduct expansion treated in [21] and [14] must all take place within a single layer, hence limiting the number of expansion terms. In the proposed algorithm, outer-product expansion takes place in as many layers as may be necessary to achieve separation; therefore, the inability to find a convergence proof for single layer outerproduct model does not rule out the possibility of a convergence proof for the multilayer OP expansion.

A Geometric Explanation of the Algorithm If we consider each data point or pattern vector as specifying a point in n dimensional space, the classification problem becomes that of finding an n dimensional hyperplane which has all of the points of Class 1 on one side and all of the points of Class 2 on the other side. In many problems of interest the points are located such that no hyperplane can separate them correctly. In such cases one way of achieving separability is to map the points into a space of higher dimension using a nonlinear mapping. The OP expansion does this by appending to the original vectors additional components made up of nonlinear combinations of the original

points. It is then possible that a hyperplane may be found in the new higher dimension space which will separate the enhanced points properly. If no such hyperplane exists the enhanced vectors may be further enhanced by mapping them into another space of even higher dimensionality by using nonlinear combinations of of the components of the enhanced vectors. This process may be repeated as many times as required in order to find a separating hyperplane.

As an example, consider the xor problem discussed above in the description of the algorithm. The pattern vectors for the xor problem are represented in 2–dimension space in Figure 9. The multilayer network solution is shown in Figure 11. From this figure we note that only one outerproduct expansion has taken place. By reference to Figure 12 we see that only one additional component is necessary to produce a higher dimension space in which separation by a plane is possible. That plane of separation is shown in Figure 13. Note that there are three dimensions in this new space and that the data points are plotted as 3 dimension points. Because the point $(1,1)$ has been expanded to the point $(1,1,1)$ and no longer lies in the plane of x_1 and x_2, it is possible to separate the two classes with a plane in 3 dimension space. The equivalent 2 dimensional nonlinear separation is shown in Figure 14.

Memory Requirements The execution of this algorithm can require the creation of matrices which are large compared to the size of the problem. This is not a concern for simple problems such as the xor problem, as shown in Figure 10, where the largest matrix transformation is 3x3; however, for problems of higher dimensionality the matrices can grow quickly. For example, the parity 3 multilayer solution has as its largest matrix a 21x7 matrix, and the parity 4 problem has as its largest matrix a 55x15 matrix. As a more complete example, Figure 15 shows the dimensions of the matrices necessary to solve the loop problem which is illustrated in Figure 16. For the solution to the loop problem the original 2 dimension vectors are expanded to 120 dimensions before being finally reduced to 26 dimension vectors. A matrix of size 120x26 is necessary to make this transformation.

One may ask why is it necessary to have so many nodes in the intermediate layers of the multilayer network when, as an example, the backpropagation network normally

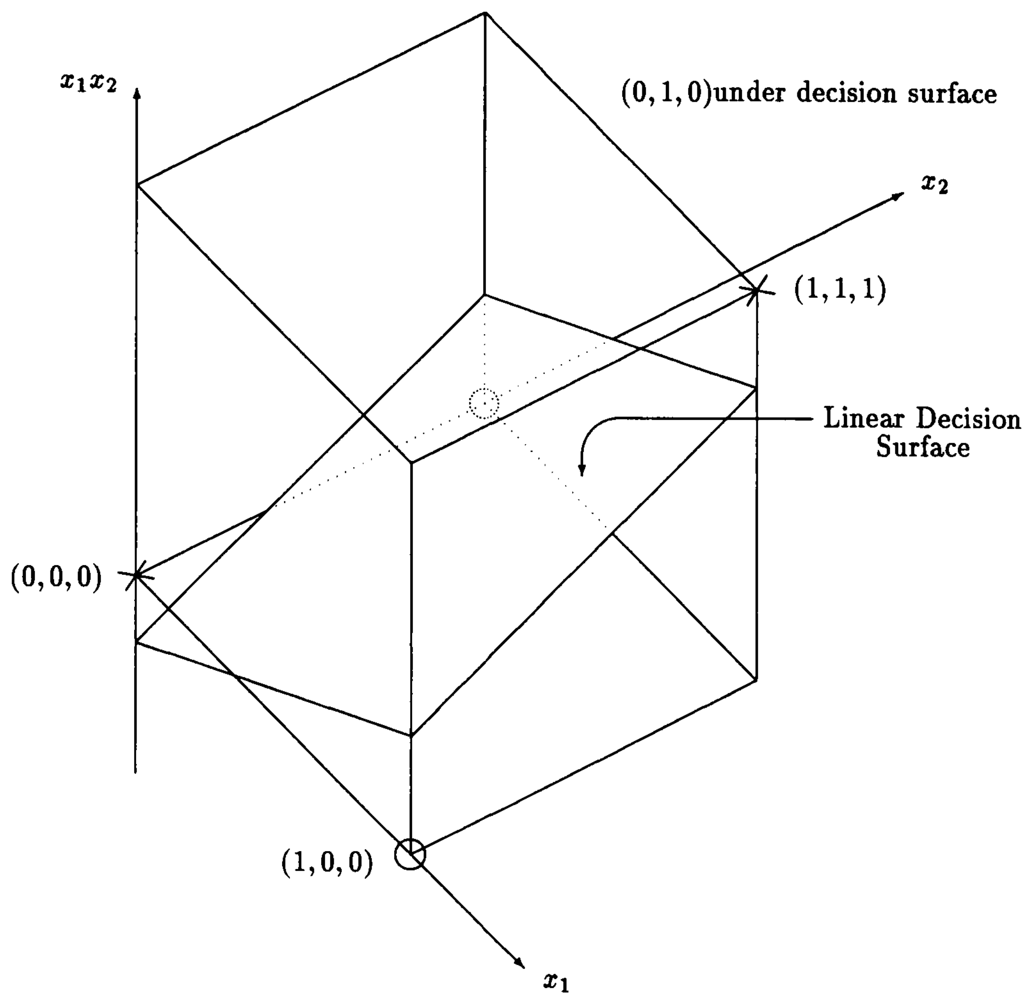


Figure 13: XOR Problem Mapped Onto 3 Dimensions

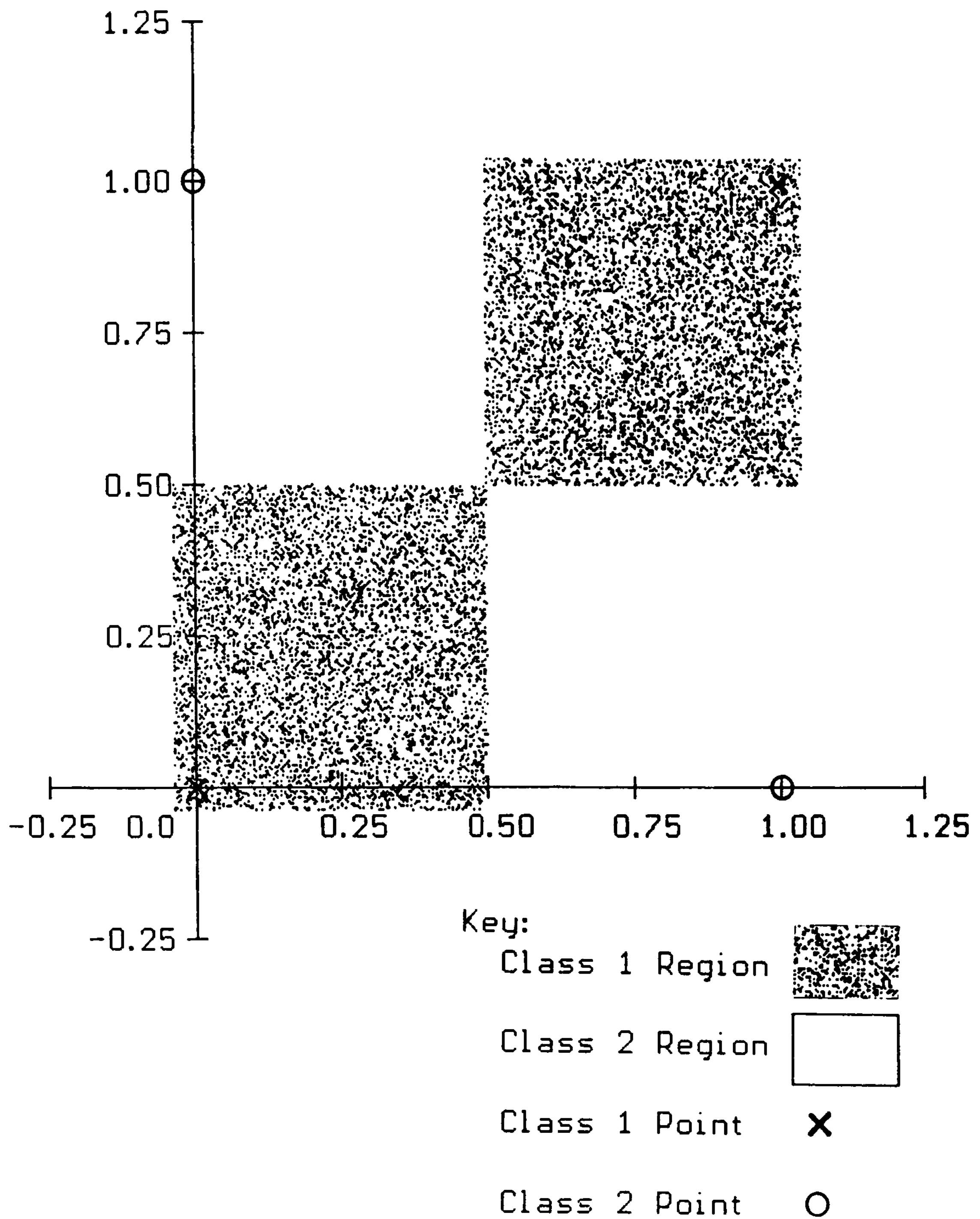


Figure 14: XOR Problem Showing Class Regions

First KL Transformation Matrix	2 x 2
Second KL Transformation Matrix	3 x 3
Third KL Transformation Matrix	6 x 6
Fourth KL Transformation Matrix	21 x 15
Fifth KL Transformation Matrix	120 x 26
Final Weight Vector	27

Figure 15: Size of Matrices for Multilayer Solution of Loop Problem

has fewer nodes in the intermediate (hidden) layers than in the input layer. First, the outerproduct introduces new dimensions (or nodes) at every intermediate layer. Second, the KL expansion is not able to pick through the new dimensions to find only those which will contribute to the desired separation because it does not know to which class each pattern vector belongs.

The KL expansion retains those dimensions which contribute to separability in general without regard to the class membership of the individual pattern vectors. This is bad in that the KL expansion is not able to achieve the dimensionality reductions which a more knowledgeable method might make; however, it is good in that the KL expansion requires no training or gradient descent and does not suffer from the shortcomings thereof. Also, if the present high–order network algorithm is modified to be a multiclass separation algorithm the nonspecific nature of the KL expansion would be advantageous.

Execution Time To a great extent the execution time of this algorithm is determined by two operations: the computation of the eigenvalues and eigenvectors of the autocorrelation matrix in the KL expansion, and the computation of the inverse of the autocorrelation matrix in the KH algorithm.

The eigenvector/eigenvalue method used in the implementation of the KL expansion is the Jacobi method, taken directly from Press *et al.* [15]. This is a reasonably efficient, very reliable method for computing eigenvalues and eigenvectors. The Jacobi method has

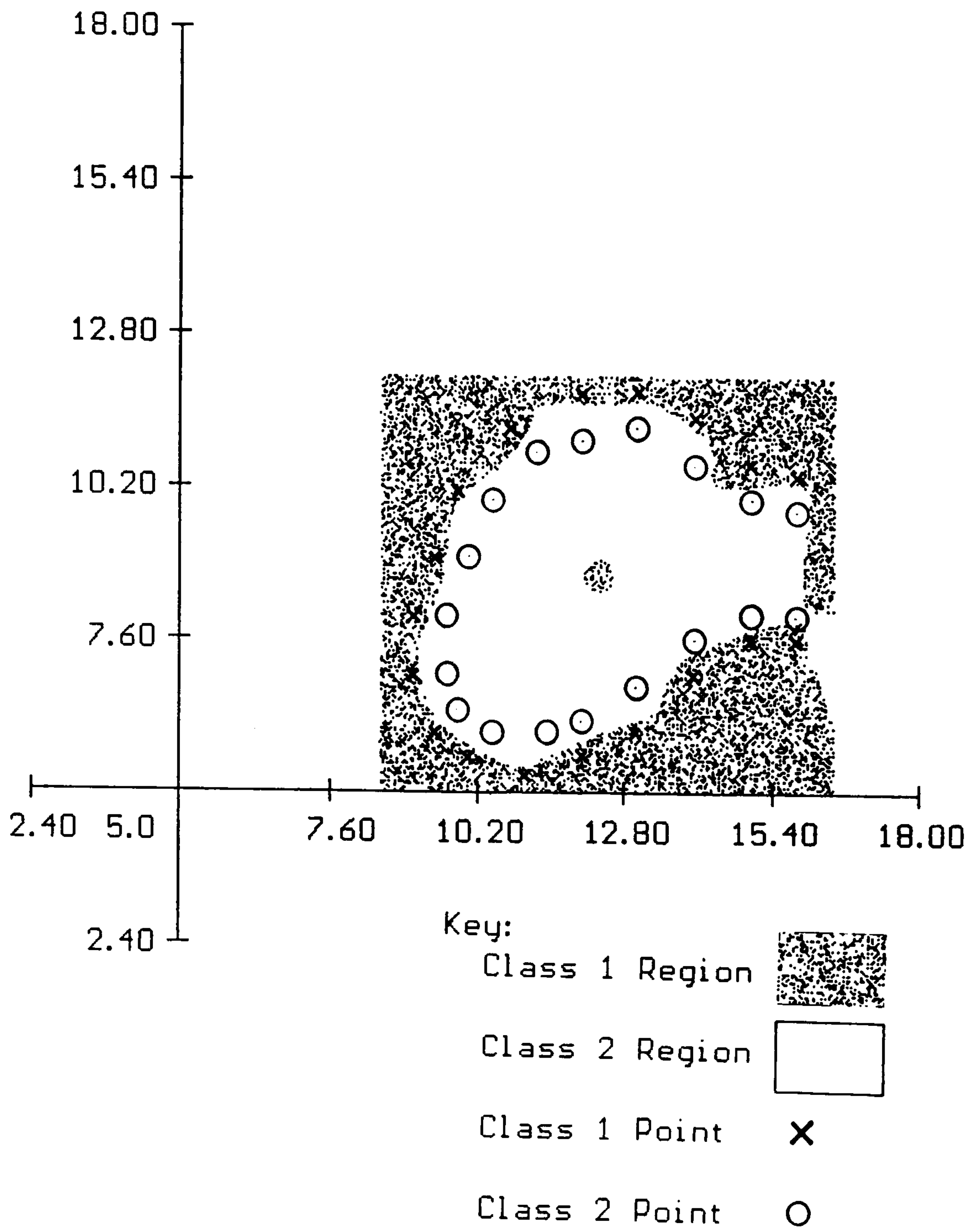


Figure 16: Loop Problem Showing Class Regions

an operation count of $O(n^3)$ where nxn is the size of the matrix.

The matrix inversion method used in the HK algorithm is the LU decomposition followed by backsubstitution, again taken directly from Press *et al.* [15]. This method of computing the inverse of a matrix has an operation count of $O(n^3)$. Since the off diagonal elements are all zero or very small the diagonal contains the largest elements, and the inversion routine is spared the extra work of the row interchanges involved in pivoting.

A much faster method of matrix inversion has been tried which is based on the theoretical promise that after the pattern vectors have been transformed by the KL expansion their autocorrelation matrix would be diagonal. This is not always the case in practice; in some cases, off diagonal components are very small, but nonzero. This is thought to be the result of roundoff errors in the various operations. Ignoring roundoff errors by setting off diagonal elements to 0.0 and inverting the matrix by inverting diagonal elements works well for easy problems, but when employed on the loop problem, Figure 16, it introduces enough difference to prevent the algorithm from achieving linear separability, all other things being equal. This method is not considered reliable enough for general use; therefore, all results reported in this thesis are achieved using LU decomposition followed by backsubstitution.

4.2 Results from Theoretical Problems

Separation Problems We refer to separation problems as two class classification problems in which the points are separated primarily on the basis of their spatial relationship; that is, all of the points in a class are "close" to each other. These are typical nonlinear separation problems which might arise from some continuous physical process.

In order to test the algorithm on this type of problem we have created two arbitrary two dimensional nonlinear separation problems. For these problems we demonstrate the decision surfaces by using the results of the multilayer network to classify randomly generated points in the region of the problem. Figure 16 and Figure 17 show the training patterns as an "x" for class 1 or as an "o" for class 2. The class 1 region is white speckled with black dots and the class 2 region is white. The decision surface is the interface between the two regions.

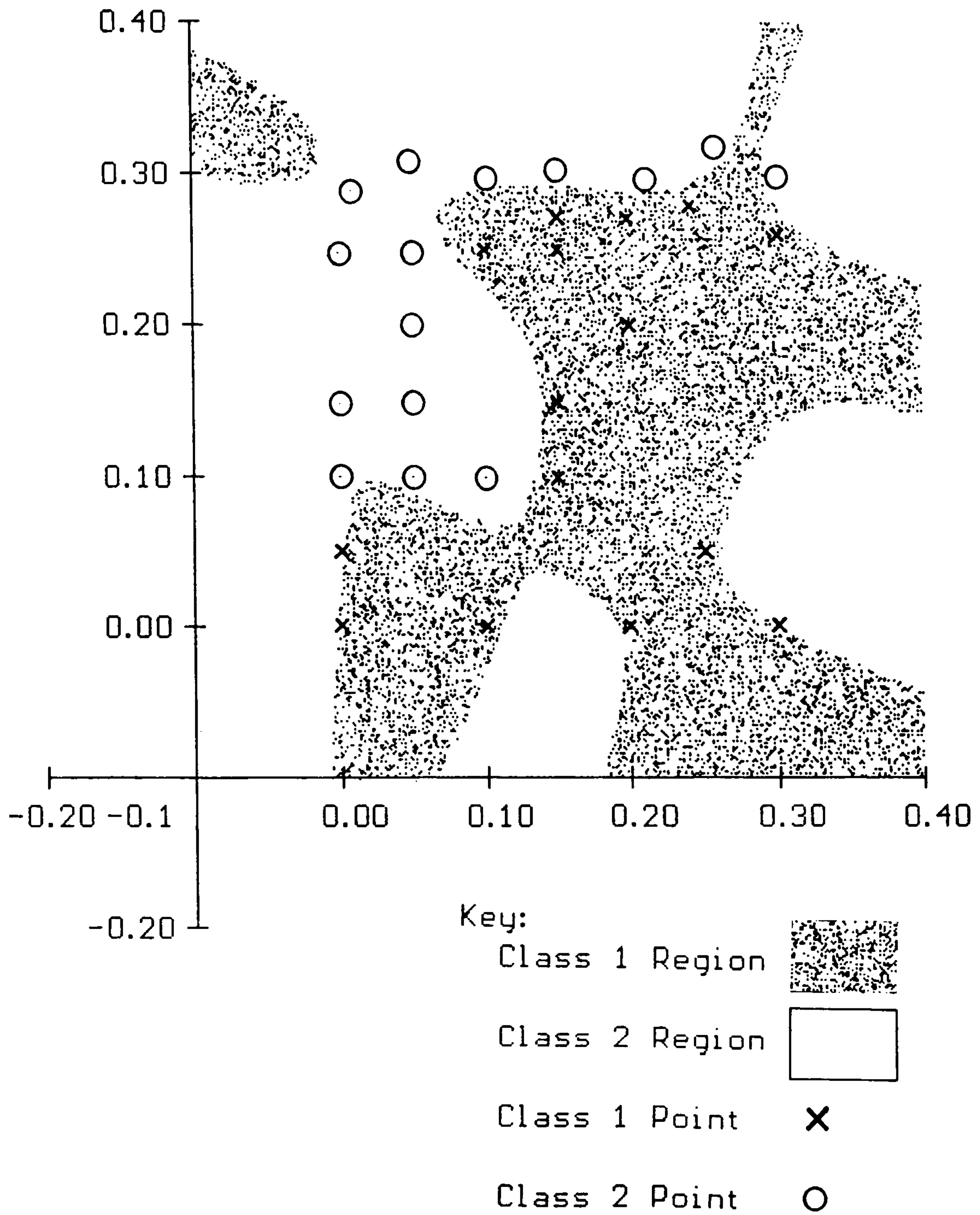


Figure 17: Test2-d Problem Showing Class Regions

In Figure 17 the separation between the classes is generally very smooth with only one small portion of class 1 going over into what the human observer might consider class 2 territory.

In Figure 16 the boundary between class 1 and class 2 is not so well behaved and there is a small piece of class 1 within the center of the class 2 region. The decision surface acts like a high degree polynomial, indicating that the algorithm has used successive outerproduct expansions to build a polynomial function.

The loop problem illustrates very well the problem of how many features to retain through the KL expansion. This decision is made by a threshold on the eigenvalues, as outlined above in Section 3.4. The result reported here is achieved using a threshold of 0.0001 $\sum_{i=1}^{n} \lambda_i$ where n is the number of eigenvalues. For this problem a fixed threshold of 0.0001 retains too many terms and results in a combinatorial explosion which exhausts all memory on an IBM AT computer.

Parity Problems Parity problems are a special case of a type of classification problem in which the classification of a pattern vector depends purely upon some relationship or correlation among two or more components of that pattern vector. The objective of parity problems is to separate binary numbers having even parity from binary numbers having odd parity, where parity is a count of the number of ones in the binary number. The simplest of the parity problems is the xor problem described above. It is not possible to determine the parity of a binary number without taking into account all of the digits.

As an example, for the parity 3 problem all three digit binary numbers are placed into class 1 or class 2 based on their parity. Equation 41 shows the equivalent single layer functional link network solution for the parity 3 problem. Note that the algorithm has used successive outerproduct expansions to build the required high–order correlation terms. Our solution agrees with that shown by Sobajic [21], who has derived a general solution to the parity n problem for the single layer functional link network.

$$net = -8.036x_1x_2x_3 + 4.018x_1x_2 + 4.018x_1x_3 + 4.018x_2x_3 - 2.009x_1 - 2.009x_2 - 2.009x_3 + 1.004 \quad (41)$$

The results for the xor problem shown above in Equation 40 are also in agreement with

Sobajic's results.

Parity problems through parity 4 have been solved on the IBM AT; the parity 5 problem has been attempted but cannot be completed due to lack of sufficient memory.

Correlation/Separation Problems Some problems resemble both parity problems and separation problems. The 12 point problem, taken from Sobajic [21] and shown in Figure 18, has a small xor problem near the origin and some well separated points further out the axes. The maze problem shown in Figure 19 is made up of four little xor problems put together. In both cases, the decision surfaces are altogether different from the solution to the simple xor problem shown in Figure 14. Our solution to the 12 point problem is altogether different from that obtained by Sobajic using backpropagation.

4.3 Results from Nuclear Power Plant Simulator Data

In order to test the algorithm on a more realistic problem we obtained data from the Watts Bar Nuclear Power Station plant simulator[1,2]. This data consists of seven accident scenarios. The accidents analyzed are:

1. Total loss of offsite power (ED1)
2. Main feedwater line break (F23)
3. Main Steam line break (MS1)
4. Control rod ejection (RD6)
5. Hot leg loss of coolant accident (TH1)
6. Cold leg loss of coolant accident (TH2)
7. Steam generator tube leak (TH5)

The five plant variables used in these tests are:

1. Steam flow
2. Steam pressure
3. Pressurizer pressure
4. Cold leg temperature
5. Coolant flow

The state of each variable is recorded at approximately one half second intervals and one

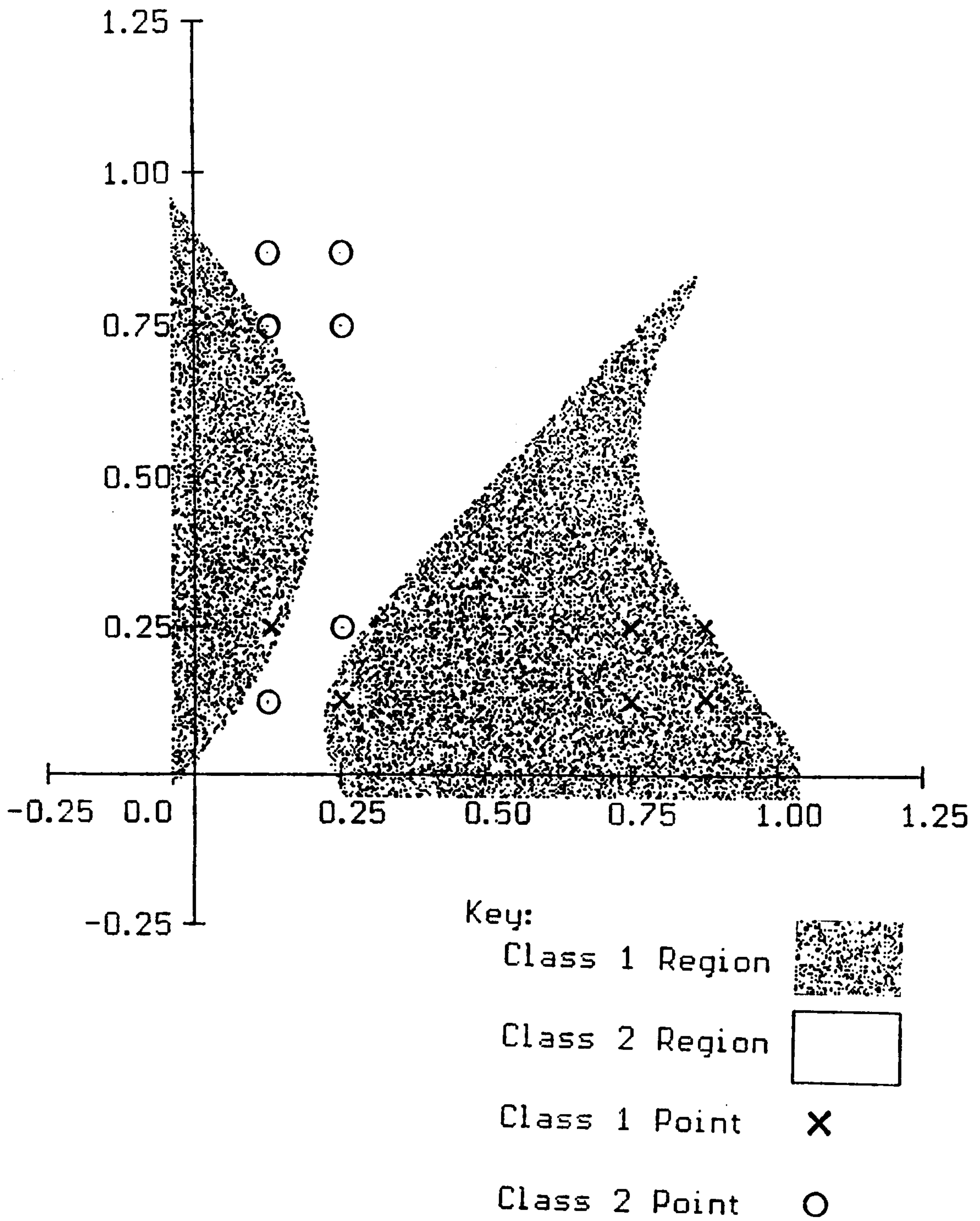


Figure 18: 12 Point Problem Showing Class Regions

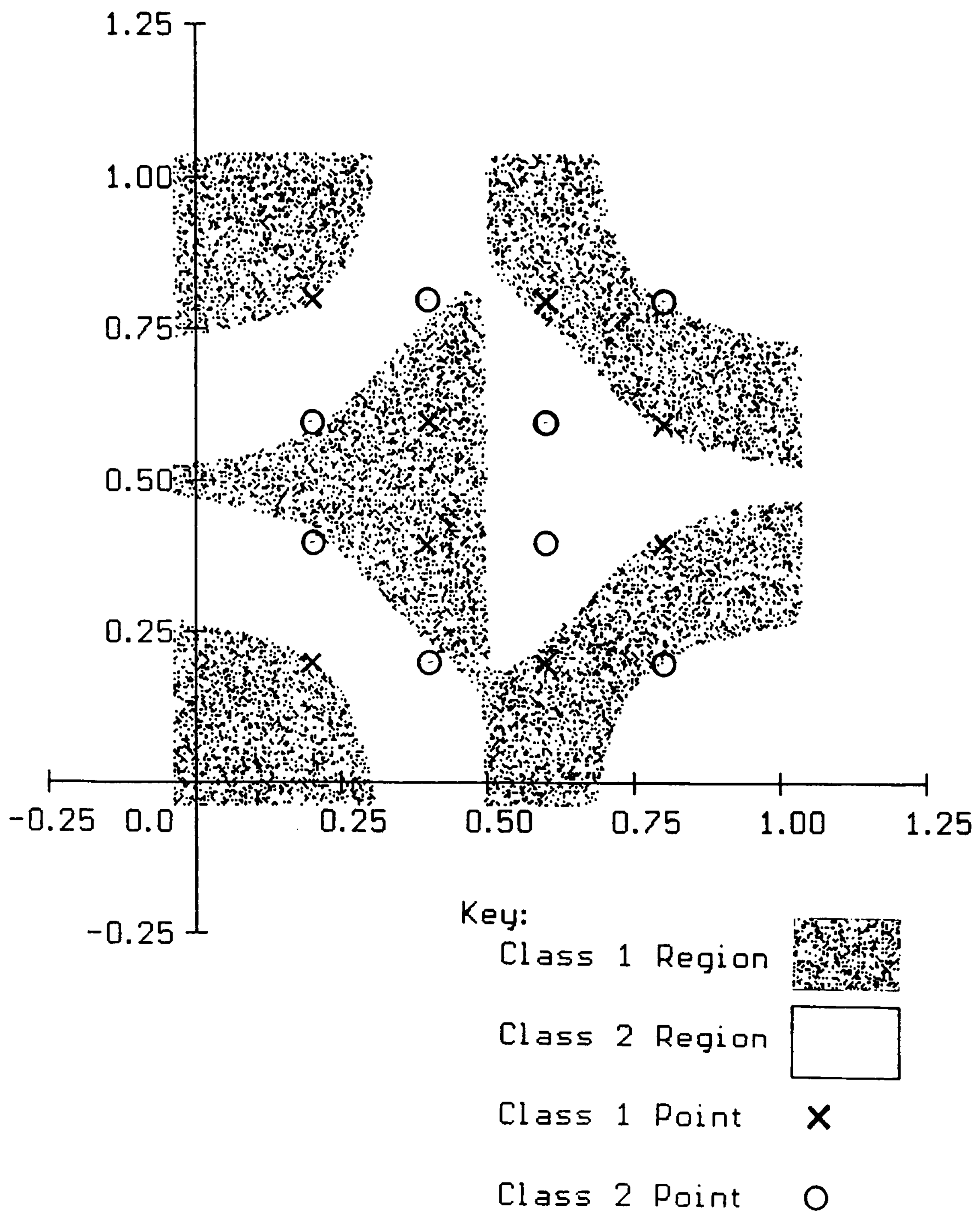


Figure 19: Maze Problem Showing Class Regions

pattern vector consists of a time value and values for each of the five above variables. In the original data the accident conditions were preceded by a period of steady state operation. These steady state data have been removed and only the data from the beginning of the transient condition have been considered. Data representing each of two accident scenarios, (MS1) and (TH5), are shown in Appendix A. These accident scenarios and the variables are described in detail in [3].

The Testing Program The high-order neural network is a two class classifier; therefore, each of the seven classes was paired with each of the remaining six classes for a total of 21 combinations of classes. Each of the 21 combinations was presented to the classifier. In every case the classes were linearly separable. There is, therefore, nothing from the algorithmic point of view to distinguish between the various class combinations; they are all linearly separable. We arbitrarily choose two classes, (MS1) and (TH5), for detailed discussion with the understanding that they embody all of the characteristics herein considered essential.

Classification Results The MS1 and TH5 classes were presented to the classifier, and the output is shown in Figure 20. A variable threshold was used in the KL expansion. It may be noted from the output that only one matrix transformation occurred, that resulting from the initial application of the KL expansion to the translated and scaled data. That transformation is followed by the weight vector which is a result of the application of the HK algorithm to the transformed data. This indicates that no outerproduct expansion was performed; neither was one necessary for class separability. We may then conclude that the classes are linearly separable.

We may then find the equivalent single layer functional link network for this classification problem in the same way as indicated above for the XOR problem. That network may be expressed as a function of the components of the pattern vector $\mathbf{x}$ as follows:

$$net = 0.1098x_1 - 0.2060x_2 + 0.6131x_3 + 0.5201x_4 + 0.0426x_5 + 0.3654x_6 - 0.871 \quad (42)$$

```
1.000000                                        /* value of theta */
6                                     /* dimensionality of X */
                                        /* translation vector */
0.497500  0.320486  0.765329  0.229336  0.782045  0.732492
0.137737                                       /* scale factor */
 t                                /* a matrix transformation */
6 4                                       /* matrix dimensions */
 -0.225792    0.932314    0.196252    0.142478  /* matrix V1 */
  0.908403    0.142740    0.377332   -0.084864
  0.011947   -0.076673    0.278262    0.746508
 -0.082708   -0.271134    0.268361    0.489062
  0.023212   -0.017972   -0.030609    0.107689
 -0.341026   -0.175214    0.817754   -0.405518
 w                                         /* weight vector w */
5                                     /* dimensionality of w */
                                                /* vector w */
  0.000003   -0.371483   -0.180591    0.552206    0.602808
```

Figure 20: Program Output for MS1 vs. TH5 Problem

4.4 Conclusions

Conclusions from Theoretical Problems The high–order network is able to solve examples of correlation type problems as well as nonlinear separation problems without knowing or being told in advance anything about the problem other than the assumption of separability which is built into the algorithm. This is believed to be due to the complementary actions of the OP expansion and the KL expansion. The OP expansion creates a group of new expansion terms at each layer without regard to nature of the separation desired; and the KL expansion selects those terms which contribute strongly to the separability of the data, again without regard for the specific separation desired. Without any knowledge of the desired separation the algorithm inevitably produces a lot of information which is unnecessary. It is anticipated that if it is possible to extend this algorithm to multiclass classification problems that information, now irrelevant, will be useful.

The choice of a KL threshold is important for the solution of high dimension problems or of any problem which requires several iterations of the OP expansion/KL expansion process. More investigation will be required in order to determine good values for the KL threshold.

Conclusions from Nuclear Power Plant Simulator Problem The High–order network successfully classified the seven accident scenarios. It was believed that the scenarios would be nonlinearly separable or overlapping and that this problem would provide an opportunity to display the OP expansion portion of the algorithm. However, as revealed by the classification process, the scenarios are all linearly separable from each other. The network did configure itself to solve the problem, and that is all that we can ask of it. This portion of the testing of the algorithm must be considered successful; however, further testing is certainly required in order to display the full utility of the self configuring high–order neural network for practical problems.

General Conclusions The concept of an adaptive topology is demonstrated by the process whereby the high–order neural network algorithm is able to make decisions about

whether or not to add an additional layer of processing and, if so, how many neurons should be in that additional layer. This concept is combined with a final layer having the standard concept of adaptive weights to create an algorithm which has the potential for considerable development. The elements of this algorithm are all linear with the exception of the OP expansion; yet, the algorithm is capable of learning nonlinear decision surfaces.

Minsky and Papert [12] say that they doubt that it is possible to create a set of general purpose predicates (or functions) which will work for any separation problem. It is, however, a different matter to design a general purpose *algorithm for the selection of specialized functions* which would work for any separation problem. The high–order neural network reported herein represents significant progress toward the creation of such a general purpose algorithm.

4.5 Possible Further Study

During the course of the investigations reported in this thesis a number of ideas have presented themselves which have not yet been investigated in any detail. They are presented below.

Extension to Multicategory Classification The network as presently developed is capable of only two category classifications. It may be feasible to extend the algorithm to multicategory classifications by modifying the final layer to include multiple HK algorithm tests for linear separability.

Use of High–Order Neurons of Degree 3 and Above The network as presently developed uses degree 2 OP expansion. The utility of higher order expansion terms has not been investigated.

Use of other Functional Expansions The use of trigonometric and polynomial expansion terms were investigated briefly but were not pursued in order to concentrate on the more biologically plausible outerproduct expansion. It may be possible to use an ancillary algorithm to choose at each expansion layer which of several expansions to use. For

example, the first expansion could be the OP expansion; the second expansion could be the trigonometric expansion.

Choice of Threshold in KL Expansion The choice of threshold in the KL expansion affects the number of neurons in the next layer and determines whether small, but potentially useful, features will be allowed to survive. An intelligent, adaptive, choice of threshold would allow useful features to survive without clogging the network with irrelevant information.

Replacement of KL Expansion with a More Powerful Algorithm The KL expansion is a linear algorithm and has the advantage of not confusing the issue of introduction of nonlinearity through expansion terms; however, it is computationally expensive. The possibility of replacing it with a nonlinear data compression algorithm might prove fruitful.

Extension to Nonseparable Classes The network as presently developed cannot deal with overlapping classes in any way other than by learning complex decision surfaces which separate the training pattern set. If it is known that the pattern classes overlap, it might be possible to limit the number of OP expansions and for the final classification substitute a delta rule classifier for the HK algorithm.

List of References

References

[1] *T.V.A. Watts Bar Malfunction cause and Effects Report.* Watts Bar Nuclear Power Station, Soddy–Daisy, Tennessee, 1989.

[2] *Watts Bar Control Room Instrumentation Report, Rev. C.* Watts Bar Nuclear Power Station, Soddy–Daisy, Tennessee, 1989.

[3] Eric Bruce Bartlet. *Nuclear Power Plant Status Diagnostics Using Simulated Condensation: an Auto–Adaptive Computer Learning Technique.* PhD thesis, The University of Tennessee, Knoxville, 1990.

[4] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York, 1973.

[5] Keinosuke Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, New York, 1972.

[6] C. Lee Giles and Tom Maxwell. Learning, invariance, and generalization in high order neural networks. *Applied Optics*, 26(23):4972 – 4978, December 1987.

[7] Donald O. Hebb. *The Organization of Behavior.* Wiley, New York, 1949. Reprinted in: *Neurocomputing*, James A. Anderson and Edward Rosenfeld (ed.), The MIT Press, Cambridge, Massachusetts, 1988, pp. 92 - 114.

[8] Y-C. Ho and R. L. Kashyap. An algorithm for linear inequalities and its applications. *IEEE Transactions on Electronic Computers*, EC–14(5):683–688, 1965. Reprinted in: *Pattern Recognition*, J. Sklansky (ed.), Dowden, Hutchinson & Ross, Stroudsburg, PA, 1973, pp. 49–54.

[9] Teuvo Kohonen, Erkki Oja, and Pekka Lehtio. Storage and processing in distributed associative memory systems. In Geoffrey E. Hinton and James A. Anderson, editors, *Parallel Models of Associative Memory*, chapter 4, pages 105 – 143, Lawrence Erlbaum Associates, Inc., 1981.

[10] Bart Kosko. *Neural Networks and Fuzzy Systems.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[11] Tom Maxwell, C. Lee Guiles, and Y. C. Lee. Generalization in neural networks: the contiguity problem. In Maurine Caudil and Charles Butler, editors, *Proceedings of the IEEE First International Conference on Neural Networks*, pages II 41 – II 46, IEEE, June 1987.

[12] Marvin Minsky and Semour Papert. *Perceptrons.* The MIT Press, Cambridge, Massachusetts, 1988.

[13] Nils J. Nilsson. *Learning Machines.* McGraw-Hill Book Company, New York, 1965.

[14] Yoh-Han Pao. *Adaptive Pattern Recognition and Neural Networks.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

[15] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, 1988.

[16] S. Qian, Y.C. Lee, R.D.Jones, C.W. Barnes, and K. Lee. Functional approximation with an orthogonal basis net. In *Proceedings of the International Joint Conference on Neural Networks*, pages III 605 – III 619, IEEE and INNS, June 1990.

[17] F. Rosenblat. The perceptron: a probabilistic model for information storage in the brain. *Psychological Review*, 65:386 – 408, 1958. Reprinted in: *Neurocomputing*, James A. Anderson and Edward Rosenfeld (ed.), The MIT Press, Cambridge, Massachusetts, 1988, pp. 92 - 114.

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(9):533–536, October 1986.

[19] David E. Rumelhart and James L. McLelland. *Parallel Distributed Processing.* The MIT Press, Cambridge, Massachusetts, 1986.

[20] Jack Sklansky and Gustav N. Wassel. *Pattern Classifiers and Trainable Machines.* Springer-Verlag, New York, 1981.

[21] Dejan J. Sobajic. *Artificial Neural Networks for Transient Stability Assesment of Electric Power Systems.* PhD thesis, Case Western Reserve University, 1988.

[22] Donald F. Specht. *Generation of polynomial discriminate functions for pattern recognition.* PhD thesis, Stanford University, 1966.

[23] Donald F. Specht. Probabilistic neural networks and the polynomial adaline as complementary techniques for classification. *IEEE Transactions on Neural Networks*, 1(1):111 – 121, 1990.

[24] Julius T. Tou and Rafael C. Gonzales. *Pattern Recognition Principles.* Addison–Wesley Publishing Company, Reading, Massachusetts, 1974.

[25] Satosi Watanabe. Karhunen-Loève expansion and factor analysis theoretical remarks and applications. In *Trans. Fourth Prague Conf. Inform. Theory, Statist. Decision Functions, and Random Processes*, pages 635–660, 1965. Reprinted in: *Pattern Recognition*, J. Sklansky (ed.), Dowden, Hutchinson & Ross, Stroudsburg, PA, 1973, pp. 146–171.

[26] Bernard Widrow, editor. *DARPA Neural Network Study.* AFCEA International Press, Fairfax, VA, 1988.

Appendices

Appendix A
Simulated Power Plant Data

Data representing each of two nuclear power plant accident scenarios are presented below. For each scenario, the data consist of 200 pattern vectors, made up of five plant variables and time. Together these vectors trace the path of the plant through a state space and, therefore, represent the accident scenario.

The six variables are identified as:

1. Time (t)
2. Steam flow (sf)
3. Steam pressure (sp)
4. Pressurizer pressure (pp)
5. Cold leg temperature (clt)
6. Coolant flow (cf)

1 Main Steam Line Break (MS1)

t	sf	sp	pp	clt	cf
.000000	.000500	.802200	.394700	.790700	.854600
.005000	.000500	.801700	.393800	.790600	.854800
.010000	.000500	.801400	.392600	.790400	.855000
.015000	.000500	.801000	.391600	.790300	.855300
.020000	.000500	.800700	.390500	.790200	.855500
.025000	.000500	.800300	.389600	.790100	.855700
.030000	.000500	.799800	.388600	.789900	.855900
.035000	.000500	.799400	.387500	.789800	.856100
.040000	.000500	.798900	.386600	.789600	.856400
.045000	.000500	.798600	.385500	.789500	.856500
.050000	.000500	.798100	.384500	.789400	.856800
.055000	.000500	.797600	.383600	.789200	.857000
.060000	.000500	.797300	.382500	.789100	.857300
.065000	.000500	.796900	.381500	.789000	.857500
.070000	.000500	.796500	.380400	.788800	.857700
.075000	.000500	.796200	.379300	.788600	.858000
.080000	.000500	.795800	.378300	.788500	.858300
.085000	.000500	.795400	.377200	.788300	.858500
.090000	.000500	.795100	.376300	.788200	.858700
.095000	.000500	.794800	.375100	.788000	.859000
.100000	.000500	.794400	.374100	.787900	.859300
.105000	.000500	.794100	.373000	.787700	.859500
.110000	.000500	.793700	.371900	.787600	.859800
.115000	.000500	.793400	.370700	.787400	.860000

.120000	.000500	.793200	.369700	.787200	.860200
.125000	.000500	.792900	.368600	.787100	.860500
.130000	.000500	.792600	.367500	.786900	.860700
.135000	.000500	.792200	.366500	.786800	.861000
.140000	.000500	.791800	.365200	.786600	.861200
.145000	.000500	.791500	.364200	.786500	.861500
.150000	.000500	.791200	.363100	.786300	.861700
.155000	.000500	.790900	.361900	.786100	.861900
.160000	.000500	.790500	.360800	.786000	.862200
.165000	.000500	.790200	.359700	.785800	.862500
.170000	.000500	.789900	.358700	.785700	.862700
.175000	.000500	.789700	.357500	.785500	.862900
.180000	.000500	.789400	.356400	.785300	.863200
.185000	.000500	.789100	.355200	.785100	.863400
.190000	.000500	.788700	.354200	.785000	.863700
.195000	.000500	.788400	.353200	.784800	.864000
.200000	.000500	.788100	.352000	.784600	.864100
.205000	.000500	.787800	.350800	.784500	.864400
.210000	.000500	.787400	.349700	.784300	.864700
.215000	.000500	.787200	.348700	.784200	.864900
.220000	.000500	.786900	.347500	.784000	.865200
.225000	.000500	.786600	.346400	.783900	.865400
.230000	.000500	.786400	.345200	.783700	.865700
.235000	.000500	.786000	.344100	.783500	.865900
.240000	.000500	.785700	.342900	.783400	.866200
.245000	.000500	.785400	.341800	.783200	.866500
.250000	.000500	.785100	.340700	.783000	.866700
.255000	.000500	.784700	.339600	.782900	.867000
.260000	.000500	.784400	.338400	.782700	.867200
.265000	.000500	.784100	.337300	.782500	.867500
.270000	.000500	.783800	.336100	.782400	.867700
.275000	.000500	.783500	.335000	.782200	.868000
.280000	.000500	.783200	.333900	.782000	.868300
.285000	.000500	.783000	.332700	.781900	.868500
.290000	.000500	.782700	.331700	.781700	.868700
.295000	.000500	.782400	.330500	.781500	.869000
.300000	.000500	.782100	.329400	.781400	.869200
.305000	.000500	.781800	.328200	.781200	.869500
.310000	.000500	.781500	.327100	.781100	.869800
.315000	.000500	.781100	.325900	.780900	.870000
.320000	.000500	.780800	.324800	.780700	.870200
.325000	.000500	.780600	.323600	.780600	.870500
.330000	.000500	.780400	.322500	.780400	.870800
.335000	.000500	.780200	.321300	.780200	.871000
.340000	.000500	.779900	.320200	.780100	.871300

.345000	.000500	.779600	.319200	.779900	.871600
.350000	.000500	.779300	.318000	.779700	.871800
.355000	.000500	.779000	.316900	.779600	.872100
.360000	.000500	.778600	.315700	.779400	.872300
.365000	.000500	.778300	.314600	.779200	.872600
.370000	.000500	.778100	.313400	.779100	.872900
.375000	.000500	.777800	.312300	.778900	.873100
.380000	.000500	.777500	.311200	.778700	.873400
.385000	.000500	.777300	.310000	.778600	.873600
.390000	.000500	.777100	.308800	.778400	.873800
.395000	.000500	.776800	.307700	.778200	.874100
.400000	.000500	.776500	.306600	.778100	.874400
.405000	.000500	.776300	.305500	.777900	.874600
.410000	.000500	.776000	.304400	.777800	.874900
.415000	.000500	.775800	.303200	.777600	.875200
.420000	.000500	.775500	.302200	.777400	.875400
.425000	.000500	.775300	.300900	.777300	.875600
.430000	.000500	.775100	.299900	.777100	.875900
.435000	.000500	.774900	.298600	.776900	.876200
.440000	.000500	.774700	.297600	.776800	.876400
.445000	.000500	.774400	.296400	.776600	.876700
.450000	.000500	.774200	.295300	.776400	.876900
.455000	.000400	.773900	.294200	.776300	.877200
.460000	.000400	.773600	.293100	.776100	.877400
.465000	.000400	.773300	.291900	.776000	.877700
.470000	.000400	.773000	.290800	.775800	.878000
.475000	.000400	.772700	.289800	.775600	.878200
.480000	.000400	.772500	.288600	.775500	.878400
.485000	.000400	.772400	.287500	.775300	.878700
.490000	.000400	.772200	.286300	.775100	.879000
.495000	.000400	.771900	.285300	.775000	.879200
.500000	.000400	.771600	.284100	.774800	.879500
.505000	.000400	.771400	.283000	.774600	.879700
.510000	.000400	.771100	.281900	.774500	.880000
.515000	.000400	.770800	.280800	.774300	.880200
.520000	.000400	.770600	.279700	.774200	.880500
.525000	.000400	.770300	.278600	.774000	.880800
.530000	.000400	.770000	.277600	.773900	.881000
.535000	.000400	.769900	.276400	.773700	.881300
.540000	.000400	.769700	.275400	.773600	.881500
.545000	.000400	.769500	.274200	.773400	.881800
.550000	.000400	.769200	.273200	.773200	.882000
.555000	.000400	.768900	.272000	.773100	.882300
.560000	.000400	.768600	.270900	.772900	.882600
.565000	.000400	.768300	.269900	.772800	.882800

.570000	.000400	.768100	.268800	.772500	.883100
.575000	.000400	.767800	.267700	.772500	.883300
.580000	.000400	.767500	.266600	.772200	.883500
.585000	.000400	.767300	.265700	.772100	.883800
.590000	.000400	.767100	.264500	.772000	.884100
.595000	.000400	.767000	.263500	.771800	.884300
.600000	.000400	.766700	.262400	.771700	.884600
.605000	.000400	.766400	.261300	.771500	.884800
.610000	.000400	.766100	.260200	.771300	.885000
.615000	.000400	.765900	.259200	.771200	.885300
.620000	.000400	.765600	.258100	.771000	.885600
.625000	.000400	.765300	.257000	.770900	.885800
.630000	.000400	.765100	.256100	.770700	.886100
.635000	.000400	.764900	.254900	.770500	.886300
.640000	.000400	.764800	.253900	.770400	.886600
.645000	.000400	.764600	.252800	.770200	.886800
.650000	.000400	.764300	.251800	.770000	.887100
.655000	.000400	.764000	.250700	.769900	.887300
.660000	.000400	.763700	.249700	.769700	.887600
.665000	.000400	.763400	.248600	.769600	.887800
.670000	.000400	.763100	.247500	.769400	.888100
.675000	.000400	.762900	.246600	.769300	.888400
.680000	.000400	.762600	.245500	.769100	.888600
.685000	.000400	.762300	.244500	.769000	.888900
.690000	.000400	.762200	.243400	.768800	.889100
.695000	.000400	.762000	.242200	.768600	.889300
.700000	.000400	.761800	.241100	.768500	.889600
.705000	.000400	.761600	.240000	.768300	.889900
.710000	.000400	.761300	.238800	.768200	.890100
.714999	.000400	.761000	.237800	.768000	.890400
.719999	.000400	.760700	.236900	.767800	.890600
.724999	.000400	.760500	.235600	.767700	.890900
.729999	.000400	.760200	.234600	.767500	.891100
.734999	.000400	.760000	.233500	.767400	.891400
.739999	.000400	.759900	.232400	.767300	.891600
.744999	.000400	.759700	.231400	.767100	.891800
.749999	.000400	.759500	.230300	.767000	.892100
.754999	.000400	.759200	.229200	.766700	.892400
.759999	.000400	.758900	.228100	.766600	.892600
.764999	.000400	.758600	.227200	.766500	.892800
.769999	.000400	.758300	.226100	.766300	.893100
.774999	.000400	.758000	.225100	.766200	.893300
.779999	.000400	.757700	.224000	.766000	.893600
.784999	.000400	.757400	.222900	.765900	.893800
.789999	.000400	.757200	.221900	.765700	.894100

.794999	.000400	.757100	.220900	.765600	.894300
.799999	.000400	.757000	.219800	.765400	.894500
.804999	.000400	.756700	.218800	.765300	.894800
.809999	.000400	.756400	.217900	.765100	.895100
.814999	.000400	.756200	.216700	.765000	.895200
.819999	.000400	.755900	.215700	.764800	.895500
.824999	.000400	.755600	.214500	.764700	.895700
.829999	.000400	.755300	.213500	.764500	.896000
.834999	.000400	.755100	.212500	.764400	.896300
.839999	.000400	.755000	.211200	.764200	.896500
.844999	.000400	.754800	.210100	.764100	.896800
.849999	.000400	.754700	.209000	.763900	.897000
.854999	.000400	.754400	.208000	.763700	.897200
.859999	.000400	.754100	.206900	.763600	.897500
.864999	.000400	.753800	.205900	.763400	.897700
.869999	.000400	.753500	.204800	.763300	.897900
.874999	.000400	.753200	.203900	.763100	.898200
.879999	.000400	.752900	.202800	.763000	.898400
.884999	.000400	.752600	.201800	.762800	.898700
.889999	.000400	.752400	.200900	.762700	.898900
.894999	.000400	.752300	.199700	.762500	.899200
.899999	.000400	.752100	.198800	.762400	.899300
.904999	.000400	.752000	.197700	.762300	.899600
.909999	.000400	.751700	.196700	.762100	.899800
.914999	.000400	.751500	.195700	.762000	.900100
.919999	.000400	.751200	.194700	.761800	.900300
.924999	.000400	.750900	.193700	.761600	.900600
.929999	.000400	.750600	.192700	.761500	.900800
.934999	.000500	.750400	.191600	.761400	.901100
.939999	.000500	.750200	.190700	.761200	.901400
.944999	.000500	.750100	.189700	.761100	.901500
.949999	.000500	.750000	.188700	.760900	.901800
.954999	.000400	.749800	.187700	.760800	.902000
.959999	.000400	.749500	.186700	.760600	.902300
.964999	.000400	.749200	.185700	.760500	.902500
.969999	.000400	.748900	.184700	.760300	.902800
.974999	.000400	.748600	.183800	.760200	.903000
.979999	.000400	.748300	.182700	.760000	.903200
.984999	.000400	.748000	.181900	.759800	.903500
.989999	.000400	.747700	.180800	.759800	.903700
.994999	.000400	.747500	.179900	.759600	.903900

2 Steam Generator Tube Leak (TH5)

t	sf	sp	pp	clt	cf
.000000	.730700	.780700	.281000	.799600	.611200
.005000	.730700	.780700	.279900	.799500	.611100
.010000	.730700	.780700	.278800	.799400	.611100
.015000	.730600	.780600	.277700	.799300	.610900
.020000	.730600	.780600	.276600	.799200	.610800
.025000	.730700	.780500	.275500	.799100	.610800
.030000	.730600	.780500	.274400	.799000	.610700
.035000	.730600	.780400	.273300	.798900	.610600
.040000	.730600	.780400	.272200	.798800	.610500
.045000	.730600	.780300	.271100	.798700	.610400
.050000	.730600	.780300	.270000	.798600	.610300
.055000	.730600	.780300	.268900	.798400	.610200
.060000	.730500	.780200	.267800	.798400	.610100
.065000	.730500	.780100	.266700	.798300	.610100
.070000	.730500	.780100	.265600	.798200	.610000
.075000	.730500	.780000	.264500	.798100	.609800
.080000	.730500	.780000	.263500	.798000	.609800
.085000	.730500	.780000	.262400	.797900	.609700
.090000	.730500	.779900	.261300	.797700	.609600
.095000	.730500	.779800	.260200	.797600	.609500
.100000	.730500	.779800	.259100	.797600	.609400
.105000	.730500	.779700	.258000	.797500	.609300
.110000	.730500	.779600	.256900	.797400	.609200
.115000	.730500	.779600	.255800	.797300	.609100
.120000	.730500	.779600	.254700	.797200	.609000
.125000	.730500	.779500	.253600	.797100	.608900
.130000	.730500	.779400	.252500	.797000	.608800
.135000	.730500	.779400	.251400	.796900	.608700
.140000	.730500	.779400	.250300	.796800	.608600
.145000	.730500	.779300	.249300	.796700	.608500
.150000	.730400	.779200	.248200	.796600	.608400
.155000	.730500	.779200	.247100	.796500	.608300
.160000	.730500	.779000	.246000	.796400	.608200
.165000	.730400	.779000	.244900	.796300	.608000
.170000	.730400	.779000	.243800	.796200	.608000
.175000	.730400	.778900	.242700	.796100	.607900
.180000	.730400	.778800	.241700	.796000	.607700
.185000	.730400	.778800	.240600	.795900	.607600
.190000	.730400	.778700	.239500	.795800	.607500
.195000	.730400	.778600	.238400	.795800	.607400
.200000	.730400	.778600	.237300	.795700	.607300
.205000	.730400	.778500	.236200	.795500	.607200

.210000	.730400	.778400	.235100	.795500	.607100
.215000	.730400	.778300	.234100	.795400	.607000
.220000	.730400	.778300	.233000	.795300	.606800
.225000	.730400	.778200	.231900	.795200	.606700
.230000	.730400	.778200	.230800	.795100	.606600
.235000	.730400	.778100	.229800	.795000	.606500
.240000	.730400	.778100	.228700	.794900	.606400
.245000	.730300	.778000	.227600	.794800	.606200
.250000	.730300	.777900	.226500	.794700	.606100
.255000	.730400	.777800	.225500	.794700	.606000
.260000	.730300	.777800	.224400	.794600	.605900
.265000	.730300	.777700	.223300	.794500	.605800
.270000	.730300	.777600	.222200	.794400	.605700
.275000	.730300	.777500	.221200	.794300	.605500
.280000	.730300	.777500	.220100	.794200	.605400
.285000	.730300	.777400	.219000	.794100	.605300
.290000	.730300	.777300	.218000	.794000	.605100
.295000	.730300	.777200	.216900	.793900	.605100
.300000	.730300	.777100	.215800	.793900	.604900
.305000	.730300	.777100	.214700	.793700	.604800
.310000	.690200	.763800	.213700	.793700	.604700
.315000	.680100	.761700	.213600	.793700	.604600
.320000	.690100	.766900	.213600	.793200	.604500
.325000	.693900	.769700	.213600	.792500	.604800
.330000	.694800	.771000	.213700	.792100	.605300
.335000	.694900	.771800	.213700	.791800	.605900
.340000	.695400	.772600	.213500	.791700	.606600
.345000	.695500	.773300	.212900	.791600	.607400
.350000	.695900	.773700	.212000	.791400	.608000
.355000	.695600	.774300	.211000	.791200	.608500
.360000	.695200	.774800	.210000	.791100	.608700
.365000	.694600	.775100	.209000	.791100	.608900
.370000	.694500	.775100	.208000	.791000	.608900
.375000	.694200	.775200	.207000	.790900	.608900
.380000	.694000	.775300	.206000	.790800	.608700
.385000	.693800	.775400	.205000	.790800	.608600
.390000	.693800	.775400	.203900	.790700	.608400
.395000	.693700	.775400	.202800	.790600	.608200
.400000	.693600	.775400	.201800	.790500	.607900
.405000	.693400	.775300	.200700	.790400	.607600
.410000	.693300	.775300	.199600	.790400	.607400
.415000	.693300	.775300	.198500	.790300	.607200
.420000	.693300	.775200	.197500	.790200	.607000
.425000	.693200	.775000	.196500	.790100	.606700
.430000	.693100	.775000	.195400	.790000	.606500

.435000	.693200	.774900	.194400	.789900	.606400
.440000	.693100	.774900	.193400	.789800	.606200
.445000	.693100	.774800	.192400	.789700	.606100
.450000	.693100	.774700	.191400	.789600	.605900
.455000	.693100	.774700	.190400	.789500	.605800
.460000	.693100	.774600	.189400	.789400	.605700
.465000	.693000	.774500	.188400	.789400	.605600
.470000	.693000	.774500	.187400	.789300	.605400
.475000	.693000	.774400	.186500	.789100	.605400
.480000	.693000	.774300	.185500	.789100	.605200
.485000	.692900	.774300	.184500	.789000	.605100
.490000	.692900	.774200	.183500	.788900	.605000
.495000	.692900	.774200	.182500	.788800	.604900
.500000	.692800	.774100	.181500	.788700	.604800
.505000	.693100	.773600	.180500	.788700	.604700
.510000	.693100	.773300	.179600	.788500	.604600
.515000	.692400	.773800	.178700	.788500	.604500
.520000	.692000	.774200	.177600	.788400	.604400
.525000	.692400	.773800	.176600	.788300	.604300
.530000	.692700	.773400	.175600	.788100	.604300
.535000	.692900	.773100	.174600	.788100	.604200
.540000	.692800	.773200	.173600	.788100	.604100
.545000	.692600	.773300	.172500	.788000	.603900
.550000	.692600	.773300	.171600	.787900	.603800
.555000	.692600	.773200	.170600	.787800	.603700
.560000	.692600	.773100	.169600	.787700	.603500
.565000	.692600	.773000	.168600	.787600	.603400
.570000	.692500	.773000	.167700	.787500	.603300
.575000	.692500	.773000	.166700	.787500	.603200
.580000	.692500	.772900	.165700	.787300	.603100
.585000	.692500	.772900	.164800	.787300	.603000
.590000	.692400	.772800	.163800	.787100	.602900
.595000	.692400	.772800	.162800	.787100	.602800
.600000	.692400	.772800	.161800	.787000	.602700
.605000	.692400	.772700	.160900	.786900	.602600
.610000	.692400	.772700	.159900	.786800	.602500
.615000	.692400	.772600	.159000	.786700	.602400
.620000	.692300	.772600	.158000	.786600	.602300
.625000	.692300	.772500	.157100	.786500	.602200
.630000	.692300	.772500	.156100	.786400	.602200
.635000	.692300	.772400	.155100	.786300	.602100
.640000	.692300	.772400	.154200	.786200	.601900
.645000	.692300	.772300	.153200	.786100	.601900
.650000	.692200	.772300	.152200	.786100	.601800
.655000	.692200	.772200	.151300	.786000	.601700

.660000	.692200	.772200	.150300	.785900	.601600
.665000	.692200	.772100	.149400	.785800	.601500
.670000	.692200	.772100	.148400	.785700	.601400
.675000	.692100	.772100	.147500	.785600	.601300
.680000	.692100	.772000	.146500	.785500	.601200
.685000	.692100	.772000	.145500	.785400	.601100
.690000	.692100	.771900	.144600	.785300	.601000
.695000	.692100	.771900	.143600	.785200	.600900
.700000	.692100	.771800	.142700	.785100	.600800
.705000	.692100	.771700	.141700	.785000	.600700
.710000	.692000	.771700	.140800	.785000	.600600
.714999	.692000	.771700	.139800	.784900	.600500
.719999	.692000	.771600	.138900	.784700	.600400
.724999	.692000	.771500	.137900	.784700	.600300
.729999	.692000	.771500	.137000	.784600	.600200
.734999	.692000	.771500	.136000	.784500	.600100
.739999	.691900	.771400	.135100	.784400	.600000
.744999	.691900	.771400	.134100	.784300	.599900
.749999	.691900	.771300	.133200	.784200	.599800
.754999	.691900	.771300	.132200	.784100	.599700
.759999	.691900	.771200	.131300	.784000	.599600
.764999	.691900	.771100	.130300	.784000	.599500
.769999	.691900	.771100	.129400	.783900	.599400
.774999	.691900	.771000	.128400	.783800	.599300
.779999	.691800	.771000	.127500	.783700	.599200
.784999	.691900	.770900	.126600	.783600	.599100
.789999	.691900	.770900	.125600	.783500	.599000
.794999	.659700	.760300	.124700	.783500	.598900
.799999	.650700	.758300	.124500	.783500	.598700
.804999	.652900	.760600	.124200	.782900	.598700
.809999	.654700	.762800	.124300	.782500	.598900
.814999	.655400	.764400	.124400	.782100	.599200
.819999	.656000	.765500	.124400	.781800	.599800
.824999	.656000	.766400	.124200	.781500	.600400
.829999	.656300	.767200	.123500	.781400	.601000
.834999	.656000	.767800	.122400	.781300	.601500
.839999	.655800	.768200	.121100	.781300	.601700
.844999	.655100	.768400	.119700	.781300	.601700
.849999	.654600	.768600	.118400	.781200	.601300
.854999	.654000	.768500	.117100	.781100	.600600
.859999	.641800	.764400	.116000	.781100	.599900
.864999	.412300	.695600	.115300	.781200	.599300
.869999	.334600	.672900	.111300	.782700	.598400
.874999	.361200	.691600	.102600	.782500	.593300
.879999	.394900	.712500	.091000	.780100	.582000

.884999	.403600	.718500	.080200	.776900	.567000
.889999	.392200	.715800	.070200	.775800	.551100
.894999	.373200	.707200	.060200	.777000	.535000
.899999	.351600	.695200	.049800	.779000	.518800
.904999	.327600	.683000	.038700	.780800	.502100
.909999	.301700	.671200	.028800	.781700	.485000
.914999	.275100	.660200	.020400	.782000	.468300
.919999	.249100	.650400	.013300	.782100	.453200
.924999	.225400	.642000	.008000	.782200	.440200
.929999	.202100	.633900	.004300	.782200	.429000
.934999	.182900	.626900	.001400	.781700	.419600
.939999	.167800	.621900	.000000	.780800	.411700
.944999	.156100	.619000	.000000	.779700	.405100
.949999	.147600	.617700	.000000	.778800	.399800
.954999	.142000	.617800	.000000	.778100	.395700
.959999	.138500	.618500	.000000	.777800	.392600
.964999	.135800	.619300	.000000	.777500	.390400
.969999	.133800	.620200	.000000	.777300	.389100
.974999	.132300	.621400	.000000	.777200	.388300
.979999	.131400	.622800	.000000	.777200	.388000
.984999	.131700	.624200	.000000	.777100	.387900
.989999	.132400	.625500	.000000	.777100	.388000
.994999	.133700	.626700	.000000	.777000	.388300

Appendix B
Source Code for High–Order Neural Network Program

This is the source code for the program which executes the High–Order Neural Network as described in the body of this thesis.

The program is made up of the following files:

1. fl1.c
2. hk1.c
3. kl1.c
4. nr1.c
5. exp_op.c
6. jacobi.c
7. eigsrt.c
8. inverse.c
9. ludcmp.c
10. lubksb.c
11. nrerror.c
12. ivector.c
13. vector.c
14. freevector.c
15. matrix.c
16. freematrix.c

Files 6 through 16 are from the Numerical Recipes C Diskette (Numerical Recipes Software, P.O. Box 243, Cambridge, MA 02238) and are not reproduced here for copyright reasons. File fl1.c contains the function main().

This program compiles with the Borland C++ compiler version 2.0 running in C mode under the MS–DOS operating system. With minor changes it has compiled with Microsoft C version 5.1 running under MS–DOS and with the VAX C compiler running under VMS. Typically the program may be compiled using the following command line, which will vary from compiler to compiler:

```
cc fl1.c hk1.c kl1.c nr1.c exp_op.c jacobi.c eigsrt.c inverse.c
   ludcmp.c lubksb.c nrerror.c ivector.c vector.c freevector.c
   matrix.c freematrix.c
```

1 File fl1.c

```
/******************************************************************************/
/* filename fl1.c                                                            */
/*                                                                            */
/*              A Self Configuring High-Order Neural Network                  */
/*                                                                            */
/*                         Ronald Brett Michaels                              */
/*                                                                            */
/*                 The University of Tennessee, Knoxville                     */
/*              Department of Engineering Science and Mechanics               */
/*                                                                            */
/*                                 1991                                       */
/*                                                                            */
/******************************************************************************/
/******************************************************************************/
/* input data file fl1.dat ordered as follows                                 */
/* dimensionality of points                                                   */
/* number of points in class 1                                                */
/* number of points in class 2                                                */
/* points in class 1                                                          */
/* points in class 2                                                          */
/******************************************************************************/
/******************************************************************************/
/* output data file fl1.tx ordered as follows                                 */
/* THETA                                                                      */
/* dimensionality of translation vector                                       */
/* elements of translation vector                                             */
/* scaling factor                                                             */
/* 't' indicates a matrix transformation                                      */
/* number of rows in matrix                                                   */
/* number of cols in matrix                                                   */
/* transformation matrix                                                      */
/* possibly more transformation matrices                                      */
/* 'w' indicates a weight vector                                              */
/* dimensionality of weight vector                                            */
/* weight vector                                                              */
/* nothing else can come after weight vector                                  */
/******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include <float.h>

#define THETA 1.0

   /* function prototypes */
int hk(float **,int,int,int,FILE *);
void kl(float ***,int *,int,int,FILE *);
```

```
void exp_t(float ***,int *,int,int);
void exp_op(float ***,int *,int,int);
void get_parms(FILE *,int *,int *,int *);
void get_patterns(FILE *,float **,int,int,int);
void copy_matrix(float **,float **,int,int);
void translate(float **,int,int,int,FILE *);
void scale(float **,int,int,int,FILE *);
void fprint_matrix(FILE *,float **,int,int);          /* print to file */
void print_matrix(float **,int,int);                  /* print to stdout */

   /* matrix inversion and handling prototypes */
float *vector(int nl,int nh);
void nrerror(char *error_text);
void free_vector(float *v,int nl,int nh);
float **matrix(int nrl,int nrh,int ncl,int nch);
void free_matrix(float **,int,int,int,int);

void fp_status_test(char error_text[]);

/*****************************************************************************/
/* main                                                                      */
/*****************************************************************************/

int main()
{
   float **original_matrix;    /* pointer to training pattern matrix X */
   float **transformed_matrix; /* pointer to training pattern matrix X */
                               /* as expanded and reduced by exp and kl */
   int d_original;             /* dimensionality of input patterns */
   int d_transformed;          /* dimensionality of transformed patterns */
                               /* note that d_trans includes augmentation */
   int n_pattern1;             /* number of training patterns in class 1 */
   int n_pattern2;             /* number of training patterns in class 2  */

   FILE *fl1;                  /* pointer to input file fl1.dat */
   FILE *fl1_out;              /* pointer to output file fl1.tx */

      /* mask float on, print exceptions */
   _control87 (MCW_EM, MCW_EM);
      /* open data input file */
   if((fl1=fopen("fl1.dat","r"))==NULL){
      nrerror("cannot open data input file");
   }
      /* open data output file */
   if((fl1_out=fopen("fl1.tx","w"))==NULL){
      nrerror("cannot open data output file");
   }
      /* get dimensionality and number of patterns */
   get_parms(fl1,&d_original,&n_pattern1,&n_pattern2);
   d_transformed = d_original +1; /* d_trans is augmented dimension */
      /* allocate memory to hold patterns */
```

```
original_matrix = matrix(1,n_pattern1+n_pattern2,1,d_original+1);
transformed_matrix = matrix(1,n_pattern1+n_pattern2,1,d_original+1);
   /* get pattern values and place into matrix */
get_patterns(fl1,original_matrix,d_original,n_pattern1,n_pattern2);
   /* note that all patterns are augmented and class 2 patterns */
   /* are multiplied by -1 */
printf("\noriginal patterns");
print_matrix(original_matrix,n_pattern1+n_pattern2,d_original+1);

fp_status_test("floating point exception test point: main() 1");

fprintf(fl1_out," %f \n",THETA);
   /* translate pattern vectors so that mean is 0.0 */
   /* translate() prints to file mean_vec[] */
translate(original_matrix,d_original,n_pattern1,n_pattern2,fl1_out);
printf("\ntranslated patterns");
print_matrix(original_matrix,n_pattern1+n_pattern2,d_original+1);
scale(original_matrix,d_original,n_pattern1,n_pattern2,fl1_out);
printf("\nscaled patterns");
print_matrix(original_matrix,n_pattern1+n_pattern2,d_original+1);
fp_status_test("floating point exception test point: main() 2");
copy_matrix(
   original_matrix,transformed_matrix,n_pattern1+n_pattern2,d_original+1);

   /* eliminate vector terms which do not contribute to separation */
kl(&transformed_matrix,&d_transformed,n_pattern1,n_pattern2,fl1_out);
printf("\nreduced matrix after kl()");
print_matrix(transformed_matrix,n_pattern1+n_pattern2,d_transformed);

   /* call Ho-Kashyap to determine separability */
   /* while not seperable */
while(!hk(transformed_matrix,d_transformed,n_pattern1,n_pattern2,fl1_out)){
fp_status_test("floating point exception test point: main() 3");

      /* perform functional expansion of pattern vectors */
   exp_op(&transformed_matrix,&d_transformed,n_pattern1,n_pattern2);
   printf("\nexpanded matrix");
   print_matrix(transformed_matrix,n_pattern1+n_pattern2,d_transformed);
   fp_status_test("floating point exception test point: main() 4");
   /* eliminate vector terms which do not contribute to separation */
   kl(&transformed_matrix,&d_transformed,n_pattern1,n_pattern2,fl1_out);
   printf("\nreduced matrix after kl()");
   print_matrix(transformed_matrix,n_pattern1+n_pattern2,d_transformed);
   fp_status_test("floating point exception test point: main() 5");
}
printf("\nseparation complete\n");
free_matrix(original_matrix,1,n_pattern1+n_pattern2,1,d_original+1);
free_matrix(transformed_matrix,1,n_pattern1+n_pattern2,1,d_transformed);
fp_status_test("floating point exception test point: main() 6");
fclose(fl1);
return 0;
```

```
}

/******************************************************************************/
/* get_parms                                                                  */
/* this function gets dimensionality and number of data points                */
/******************************************************************************/

void get_parms(
   FILE *hk1,        /* file pointer to data file */
   int *d_original,     /* pointer to dimensionality of input data */
   int *n_pattern1, /* pointer to number of patterns in class 1 */
   int *n_pattern2  /* pointer to number of patterns in class 2 */
)
{
   if(fscanf(hk1,"%d",d_original)==EOF){           /* get dimensionality */
      nrerror("problem with input data file");  /* of pattern vectors */
   }
   if(fscanf(hk1,"%d",n_pattern1)==EOF){  /* get number of pattern vectors */
      nrerror("problem with input data file");             /* in class 1 */
   }
   if(fscanf(hk1,"%d",n_pattern2)==EOF){  /* get number of pattern vectors */
      nrerror("problem with input data file");             /* in class 2 */
   }
}

/******************************************************************************/
/* get_patterns                                                               */
/* this function gets pattern values from data file and places them into     */
/* the matrix pointed to by original_matrix                                   */
/* note that all patterns are augmented with 1 and class 2 patterns          */
/* are multiplied by -1                                                       */
/******************************************************************************/

void get_patterns(
   FILE *hk1,
   float **mat,  /* pointer to matrix to receive data */
   int dim,      /* dimension of data */
   int n_1,      /* number of patterns in class 1 */
   int n_2       /* number of patterns in class 2 */
)
{
   int i,j;

      /* read in class 1 patterns */
   for(i=1;i<=n_1;i++){
      mat[i][1] = THETA;   /* augment each pattern */
      for(j=2;j<=dim+1;j++){
         if(fscanf(hk1,"%f",&(mat[i][j]))==NULL){
            nrerror("error reading data file in get_patterns()");
         }
      }
```

```
   }

      /* read in class 2 patterns */
   for(;i<=n_1+n_2;i++){   /* continue with class 2 */
      mat[i][1] = -THETA;   /* augment each pattern */
      for(j=2;j<=dim+1;j++){
         if(fscanf(hk1,"%f",&(mat[i][j]))==NULL){
            nrerror("error reading data file in get_patterns()");
         }
            /* multiply class 2 patterns by -1 */
         mat[i][j] = -mat[i][j];
      }
   }
}

/****************************************************************************/
/* copy_matrix                                                              */
/* this function copies pattern values from from_matrix and places them    */
/* into the to_matrix                                                       */
/* note that there is no test for comformability                            */
/****************************************************************************/

void copy_matrix(
   float **from_matrix,  /* pointer to matrix to send data */
   float **to_matrix,    /* pointer to matrix to receive data */
   int n_row,            /* number of rows */
   int n_col             /* number of columns */
)
{
   int i,j;

   for(i=1;i<=n_row;i++){
      for(j=1;j<=n_col;j++){
         to_matrix[i][j] = from_matrix[i][j];
      }
   }
}

/****************************************************************************/
/* translate                                                                */
/* this function translates the patterns so that the mean of all pattern   */
/* vectors of both classes is 0.0                                           */
/****************************************************************************/

void translate(
   float **mat,  /* pointer to matrix to receive data */
   int dim,      /* dimension of data */
   int n_1,      /* number of patterns in class 1 */
   int n_2,      /* number of patterns in class 2 */
   FILE *fl1_out /* file pointer to output file */
)
```

```
{
   int i,j;
   float *mean_vec;
   mean_vec = vector(1,dim+1);          /* allocate memory for mean vector */

   for(j=1;j<=dim+1;j++){               /* zero mean_vec */
      mean_vec[j] = 0.0;
   }
   for(i=1;i<=n_1;i++){                 /* class 1 patterns */
      for(j=2;j<=dim+1;j++){            /* skip past augmentation */
         mean_vec[j] += mat[i][j];
      }
   }
   for(;i<=n_1+n_2;i++){                /* continue with class 2 patterns */
      for(j=2;j<=dim+1;j++){            /* skip past augmentation */
         mean_vec[j] -= mat[i][j];      /* multiply class 2 patterns by -1 */
      }
   }
   for(j=2;j<=dim+1;j++){
      mean_vec[j] = mean_vec[j]/(n_1+n_2);
   }
   for(i=1;i<=n_1;i++){                 /* class 1 patterns */
      for(j=2;j<=dim+1;j++){            /* skip past augmentation */
         mat[i][j] -= mean_vec[j];
      }
   }
   for(;i<=n_1+n_2;i++){                /* continue with class 2 patterns */
      for(j=2;j<=dim+1;j++){            /* skip past augmentation */
   mat[i][j] += mean_vec[j];
      }
   }
   fprintf(fl1_out," %d \n",dim+1);
   for(j=1;j<=dim+1;j++){
      fprintf(fl1_out," %f ",mean_vec[j]);
   }
   fprintf(fl1_out," \n ");
   free_vector(mean_vec,1,dim+1);       /* free memory for mean vector */
}

/*****************************************************************************/
/* scale                                                                     */
/* this function scales the patterns so that the mean of the absolute        */
/* values of all pattern components is 1.0                                   */
/*****************************************************************************/

void scale(
   float **mat,  /* pointer to matrix to receive data */
   int dim,      /* dimension of data */
   int n_1,      /* number of patterns in class 1 */
   int n_2,      /* number of patterns in class 2 */
   FILE *fl1_out /* file pointer to output file */
```

```
)
{
   int i,j;
   float mean;

   mean = 0.0;
   for(i=1;i<=n_1+n_2;i++){            /* class 1 patterns */
      for(j=2;j<=dim+1;j++){           /* skip past augmentation */
         mean += fabs(mat[i][j]);
      }
   }
   mean = mean/((n_1+n_2)*dim);


   for(i=1;i<=n_1+n_2;i++){
      for(j=2;j<=dim+1;j++){           /* skip past augmentation */
         mat[i][j] = mat[i][j]/mean;
      }
   }
   fprintf(fl1_out,"\n %f \n ",mean);
}

/*****************************************************************************/
/* fp_status_test                                                            */
/* this function prints the nature of any floating point exceptions which    */
/* may have occured during program execution                                 */
/* this function inspired by nrerror() from Numerical Recipes in C           */
/*****************************************************************************/

void fp_status_test(char error_text[])
{
   unsigned int status;     /* 80x87 status word */
   int error_flag;          /* 0 if no error, 1 if error */
   status = _status87();    /* get status word */

   if(status==0x1||status==0x2||status==0x4||status==0x8
      ||status==0x10){
      error_flag = 1;
      fprintf(stderr,"\nFunctional Link run time error");
      fprintf(stderr,"\n%s",error_text);
   }
   else error_flag = 0;

   switch (status) {
      case 0x1:
         fprintf(stderr,"\ninvalid floating point operation");
      case 0x2:
         fprintf(stderr,"\ndenormalized operator");
      case 0x4:
         fprintf(stderr,"\nzero devide");
      case 0x8:
```

```
         fprintf(stderr,"\noverflow");
      case 0x10:
         fprintf(stderr,"\nunderflow");
/*       case 0x20:
         fprintf(stderr,"\ninexact result");                    */
   }
   if(error_flag==1){
      fprintf(stderr,"\ninvalid floating point operation");
      fprintf(stderr,"\nexiting to system\n");
      exit(1);
   }
}
```

2 File hk1.c

```
/******************************************************************************/
/* filename hk1.c                                                             */
/*                                                                            */
/*              A Self Configuring High-Order Neural Network                  */
/*                                                                            */
/*                        Ronald Brett Michaels                               */
/*                                                                            */
/*                 The University of Tennessee, Knoxville                     */
/*             Department of Engineering Science and Mechanics                */
/*                                                                            */
/*                                1991                                        */
/*                                                                            */
/******************************************************************************/
/******************************************************************************/
/* this is an implementation of the Ho-Kashyap algorithm as described in     */
/* Pattern Recognition Principles by Tou and Gonzales ch. 5.3.3              */
/******************************************************************************/

#include<stdio.h>
#include <conio.h>
#include<stdlib.h>
#include<math.h>
#include<malloc.h>

/* hk parameters */
#define MAX_LOOPS 20000          /* max number of iterations permitted */
#define C 1.0                    /* correction factor */
#define TOLERANCE 0.00001        /* do not set tolerance to 0.0 */

   /* function prototypes */
int hk(float **,int,int,int,FILE *);
float **transpose(float **,float **,int,int,int);
float **matrix_product(float **,int,int,float **,int,int,float **,int,int);
float *vector_subtract(float *,float*,float*,int);
float *vector_add(float *,float*,float*,int);
float *error_add(float *,float *,int);
float *matrix_vector_product(float **,int,int,float *,int,float *,int);
float *scalar_vector_product(float,float *,float *,int);
int test_positive(float *,int);
int test_negative(float *,int);
float **matrix(int,int,int,int);
void free_matrix(float **,int,int,int,int);
float *vector(int,int);
void free_vector(float *,int,int);
void inverse(float **,float **,int);
void print_matrix(float **,int,int);               /* print to stdout */
void print_vector(float *,int);                    /* print to stdout */

/******************************************************************************/
```

```
/* hk                                                                        */
/* this is an implementation of the Ho-Kashyap algorithm as described in    */
/* Pattern Recognition Principles by Tou and Gonzales ch. 5.3.3              */
/* function returns 1 if classes are separable, 0 if not separable           */
/*****************************************************************************/

int hk(
   float **pattern_matrix, /* pointer to training pattern matrix X */
   int n_input,            /* dimensionality of input patterns */
   int n_pattern1,         /* number of training patterns in class 1 */
   int n_pattern2,         /* number of training patterns in class 2  */
   FILE *fl1_out           /* pointer to output file */
)
{
   float **pat_mat_t;      /* pointer to transpose of pattern_matrix X_t */
   float **pat_mat_sq;     /* pointer to XtransposeX */
   float **pat_mat_sq_inv; /* pointer to XtransposeX inverse */
   float **gen_inv;        /* pointer to generalised inverse */
   float *b_vec;           /* pointer to b vector */
   float *w_vec;           /* pointer to weight vector */
   float *e_vec;           /* pointer to error vector */
   float *response_vec;    /* pointer to Xw vector */
   float *error_sum_vec;   /* pointer to sum of error vec and abs error vec */

   int loops;              /* counter for number of iterations */
   int i;                  /* counter in for loop */

      /* allocate memory to hold Xtranspose */
   pat_mat_t = matrix(1,n_input,1,n_pattern1+n_pattern2);

      /* allocate memory to hold pat_mat_sq and pat_mat_sq_inv */
   pat_mat_sq = matrix(1,n_input+1,1,n_input);
   pat_mat_sq_inv = matrix(1,n_input,1,n_input);

      /* allocate memory to hold generalised inverse */
   gen_inv = matrix(1,n_input,1,n_pattern1+n_pattern2);

      /* allocate memory to hold b vector */
   b_vec = vector(1,n_pattern1+n_pattern2);

      /* initialisation to 1.0 */
   for(i=1;i<=n_pattern1+n_pattern2;i++)
      b_vec[i] = 1.0;

      /* allocate memory to hold weight vector */
   w_vec = vector(1,n_input);

      /* allocate memory to hold error vector */
   e_vec = vector(1,n_pattern1+n_pattern2);

      /* allocate memory to hold error sum vector */
```

```
error_sum_vec = vector(1,n_pattern1+n_pattern2);

   /* allocate memory to hold Xw or response vector */
response_vec = vector(1,n_pattern1+n_pattern2);

   /* transpose the pattern matrix */
transpose(pattern_matrix,pat_mat_t,n_input,n_pattern1,n_pattern2);

matrix_product(pat_mat_t,n_input,n_pattern1+n_pattern2,
   pattern_matrix,n_pattern1+n_pattern2,n_input,
   pat_mat_sq,n_input,n_input);

inverse(pat_mat_sq,pat_mat_sq_inv,n_input);

matrix_product(pat_mat_sq_inv,n_input,n_input,
   pat_mat_t,n_input,n_pattern1+n_pattern2,
   gen_inv,n_input,n_pattern1+n_pattern2);

   /* calculate initial value for weights */
matrix_vector_product(gen_inv,n_input,n_pattern1+n_pattern2,
   b_vec,n_pattern1+n_pattern2,
   w_vec,n_input);

   /* calculate error vector */
vector_subtract(
   matrix_vector_product(pattern_matrix,n_pattern1+n_pattern2,n_input,
w_vec,n_input,response_vec,n_pattern1+n_pattern2),
b_vec,e_vec,n_pattern1+n_pattern2);

loops = 0;

while(test_positive(e_vec,n_pattern1+n_pattern2)){
   if((loops++)==MAX_LOOPS){
      /* free memory space */
      free_matrix(pat_mat_t,1,n_input,1,n_pattern1+n_pattern2);
      free_matrix(pat_mat_sq,1,n_input,1,n_input);
      free_matrix(pat_mat_sq_inv,1,n_input,1,n_input);
      free_matrix(gen_inv,1,n_input,1,n_pattern1+n_pattern2);
      free_vector(b_vec,1,n_pattern1+n_pattern2);
      free_vector(w_vec,1,n_input);
      free_vector(e_vec,1,n_pattern1+n_pattern2);
      free_vector(error_sum_vec,1,n_pattern1+n_pattern2);
      free_vector(response_vec,1,n_pattern1+n_pattern2);
      printf("\nho-kashyap looped out");
      return 0;     /* classes are not easily separable */
   }                /* assume that classes are not separable even though */
                    /* with further training they might be separable     */

      /* calculate next iteration of b vector */
   error_add(e_vec,error_sum_vec,n_pattern1+n_pattern2);
   scalar_vector_product(
```

```
      C,error_sum_vec,error_sum_vec,n_pattern1+n_pattern2);

   vector_add(b_vec,error_sum_vec,b_vec,n_pattern1+n_pattern2);

      /* calculate next iteration of weight vector */
   matrix_vector_product(gen_inv,n_input,n_pattern1+n_pattern2,
      b_vec,n_pattern1+n_pattern2,w_vec,n_input);

      /* calculate error vector */
   matrix_vector_product(pattern_matrix,n_pattern1+n_pattern2,n_input,
   w_vec,n_input,
   response_vec,n_pattern1+n_pattern2);
   vector_subtract(response_vec,b_vec,e_vec,n_pattern1+n_pattern2);
 }
   /* free memory space */
 free_matrix(pat_mat_t,1,n_input,1,n_pattern1+n_pattern2);
 free_matrix(pat_mat_sq,1,n_input,1,n_input);
 free_matrix(pat_mat_sq_inv,1,n_input,1,n_input);
 free_matrix(gen_inv,1,n_input,1,n_pattern1+n_pattern2);
 free_vector(b_vec,1,n_pattern1+n_pattern2);
 free_vector(w_vec,1,n_input);
 free_vector(error_sum_vec,1,n_pattern1+n_pattern2);
 free_vector(response_vec,1,n_pattern1+n_pattern2);

   /* if no components positive and at least one negative */
 if(test_negative(e_vec,n_pattern1+n_pattern2)){
 printf("\nho-kashyap error vector");
 print_vector(e_vec,n_pattern1+n_pattern2);
   free_vector(e_vec,1,n_pattern1+n_pattern2);
   return 0;     /* classes are non separable */
 }
   /* if all components are 0 */
 else{
   printf("\nho-kashyap weight vector");
   print_vector(w_vec,n_input);
      /* output to file for the classification program */
   fprintf(fl1_out," %c \n",'w');  /* w for weights */
   fprintf(fl1_out," %d \n",n_input);  /* write number of element */
   for(i=1;i<=n_input;i++){      /* write weight vector */
      fprintf(fl1_out," %10.6f ",w_vec[i]);
   }
   fprintf(fl1_out,"\n");

   free_vector(e_vec,1,n_pattern1+n_pattern2);
   return 1;      /* classes are separable */
 }
}

/*****************************************************************************/
/* error_add                                                                 */
/* this function adds a vector with its absolute value                       */
```

```
/* and returns a pointer to the result vector                              */
/****************************************************************************/

float *error_add(
   float *vec,              /* vector to be added */
   float *result,           /* result vector */
   int n                    /* number of elements in each vector */
)
{
   int i;

   for(i=1;i<=n;i++){
      result[i] = vec[i] + fabs(vec[i]);
   }
   return (result);
}

/****************************************************************************/
/* test_positive                                                            */
/* this function inspects each element of a vector and returns true if any */
/* element is positive or false if no element is positive                  */
/****************************************************************************/

int test_positive(
   float *vec,
   int n
)
{
   int i;

   for(i=1;i<=n;i++){
      if(vec[i]>TOLERANCE)
         return 1;                  /* true */
   }
   return 0;                        /* false */
}

/****************************************************************************/
/* test_negative                                                            */
/* this function inspects each element of a vector and returns true if any */
/* element is negative or false if no element is negative                  */
/****************************************************************************/

int test_negative(
   float *vec,
   int n
)
{
   int i;

   for(i=1;i<=n;i++){
```

```
         /* it appears that tolerance on negative test */
         /* must be larger than on positive test */
      if(vec[i]<-TOLERANCE*100.0)
         return 1;                   /* true */
   }
   return 0;                         /* false */
}
```

3 File kl1.c

```
/*****************************************************************************/
/* filename kl1.c                                                            */
/*                                                                           */
/*                A Self Configuring High-Order Neural Network               */
/*                                                                           */
/*                          Ronald Brett Michaels                            */
/*                                                                           */
/*                    The University of Tennessee, Knoxville                 */
/*              Department of Engineering Science and Mechanics              */
/*                                                                           */
/*                                  1991                                     */
/*                                                                           */
/*****************************************************************************/
/*****************************************************************************/
/* this is an implementation of the Karhunen-Loeve algorithm as described   */
/* in Pattern Recognition Principles by Tou and Gonzales ch. 7.6.2          */
/* file nr1.c required                                                       */
/*****************************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<malloc.h>
#include<conio.h>

#define THRESHOLD (0.0001)    /* multiply by sum of eigenvalues */

   /* function prototypes */
void kl(float ***,int *,int,int,FILE *);
void autocorrelation(float **,int,int,int,float **,int,int);
float **reduce(float **,int,int,float **,int,int,float **,int,int);
void nrerror(char *error_text);
void jacobi(float **,int,float *,float **,int *);
float *vector(int nl,int nh);
void free_vector(float *v,int nl,int nh);
float **matrix(int,int,int,int);
void free_matrix(float **,int,int,int,int);
void print_matrix(float **,int,int);              /* print to stdout */
void print_vector(float *,int);                   /* print to stdout */
void fprint_matrix(FILE *,float **,int,int);
void eigsrt(float *,float **,int);

/*****************************************************************************/
/* kl                                                                        */
/*****************************************************************************/

void kl(
   float ***mat,      /* pointer to pointer to training pattern matrix X */
   int *dim,          /* pointer to dimension of augmented, transformed data */
```

```
   int n_1,          /* number of patterns in class 1 */
   int n_2,          /* number of patterns in class 2 */
   FILE *fl1_out     /* file pointer to output file */
)
{

   float **r;      /* pointer to autocorrelation matrix */
   float **c;      /* matrix for transformed pattern vectors */
   float *d;       /* returns the eigenvalues */
   float **v;      /* returns normalized eigenvectors in its columns */
   float threshold; /* calculated eigenvalue threshold */
   int nrot;       /* returns number of Jacobi rotations required */
   int col_new;    /* the dimensionality of reduced pattern vectors */
   int i,j,k;      /* counters */

      /* allocate memory to hold autocorrelation matrix */
   r = matrix(1,(*dim)-1,1,(*dim)-1);
      /* create autocorrelation matrix */
   autocorrelation(*mat,n_1,n_2,(*dim)-1,
      r,(*dim)-1,(*dim)-1);
   printf("\nautocorrelation matrix");
   print_matrix(r,((*dim)-1),((*dim)-1));
      /* allocate memory for eigenvector matrix */
   v = matrix(1,(*dim)-1,1,(*dim)-1);
   d = vector(1,(*dim)-1);        /* allocate memory for vector */
      /* call eigenvector and eigen value functions */
   jacobi(r,(*dim)-1,d,v,&nrot);
   free_matrix(r,1,(*dim)-1,1,(*dim)-1);
   printf("\nunsorted eigenvector matrix");
   print_matrix(v,((*dim)-1),((*dim)-1));
   printf("\neigenvalue vector");
   print_vector(d,((*dim)-1));
   threshold = 0.0;
   for(j=1;j<=(*dim)-1;j++){
      threshold += d[j];
   }
/*   threshold = threshold*THRESHOLD;*/
   threshold = THRESHOLD;
   eigsrt(d,v,(*dim)-1);
   printf("\neigenvalue vector");
   print_vector(d,((*dim)-1));
   printf("\nthreshold = %f\n",threshold);
   getch();
      /* select eigenvectors corresponding to eigenvalues */
      /* larger than THRESHOLD */
   col_new = (*dim)-1;
   for(j=1;j<=col_new;j++){
      if(d[j]<threshold){
         for(k=j;k<col_new;k++){
            d[k] = d[k+1];
            for(i=1;i<=(*dim)-1;i++){
```

```
                v[i][k] = v[i][k+1]; /* we have to go back and check v[i][k] */
             }
          }
          col_new--;
          j--;  /* go back and check the new vector we moved into v[i][k] */
       }
    }
    free_vector(d,1,(*dim)-1);
       /* allocate c to hold the reduced pattern matrix */
    c = matrix(1,n_1+n_2,1,col_new+1);
    reduce(*mat,n_1+n_2,*dim,
       v,(*dim)-1,col_new,
       c,n_1+n_2,col_new);
    fprintf(fl1_out," %c \n",'t');  /* t for transformation */
    fprintf(fl1_out," %d %d \n",(*dim)-1,col_new);  /* write rows and cols */
    for(i=1;i<=((*dim)-1);i++){       /* write transformation matrix */
       for(j=1;j<=col_new;j++){
          fprintf(fl1_out," %10.6f ",v[i][j]);
       }
       fprintf(fl1_out,"\n");
    }
    fprintf(fl1_out,"\n");

       /* free old pattern matrix */
    free_matrix(*mat,1,n_1+n_2,1,(*dim));

    free_matrix(v,1,(*dim)-1,1,(*dim)-1);
       /* return new values */
    *mat = c;
    *dim = col_new+1;
}

/******************************************************************************/
/* autocorrelation                                                            */
/* this function finds the autocorrelation matrix of a pattern matrix         */
/* reference Tou and Gonzalez pg. 276                                         */
/* note that this function is wired for 2 equally probable classes            */
/******************************************************************************/

void autocorrelation(
   float **mat,          /* matrix containing patterns to be autocorrelated */
   int n_pat1,
   int n_pat2,
   int n_inpt,
   float **r,            /* autocorrelation matrix */
   int n_row,
   int n_col
)
{
   float factor;   /* class probability / number in class */
   int i;
```

```
    int j;
    int k;

    if(n_inpt!=n_row)
       nrerror("matrices not comformable in autocorrelate()");
    if(n_inpt!=n_col)
       nrerror("matrices not comformable in autocorrelate()");
    for(j=1;j<=n_inpt;j++){   /* zero autocorrelation matrix */
       for(k=1;k<=n_inpt;k++){
          r[j][k] = 0.0;
       }
    }
    factor = 0.5/n_pat1;   /* assume class probability of 0.5 */
    for(i=1;i<=n_pat1;i++){
       for(j=1;j<=n_inpt;j++){
          for(k=1;k<=n_inpt;k++){
             r[j][k] += mat[i][j+1]*mat[i][k+1]*factor;
          }
       }
    }
    factor = 0.5/n_pat2;   /* assume class probability of 0.5 */
    for(;i<=n_pat1+n_pat2;i++){
       for(j=1;j<=n_inpt;j++){
          for(k=1;k<=n_inpt;k++){
             r[j][k] += mat[i][j+1]*mat[i][k+1]*factor;
          }
       }
    }
}

/******************************************************************************/
/* reduce                                                                     */
/* this function computes the product of two matrices and returns             */
/* a pointer to the result matrix  the first column is augmentation           */
/* note that this function provides check for conformability                  */
/******************************************************************************/

float **reduce(
   float **matrix_1,
   int n_row_1,
   int n_col_1,
   float **matrix_2,
   int n_row_2,
   int n_col_2,
   float **result,
   int n_row_r,
   int n_col_r
)
{
   float sum;
   int i,j,k;
```

```
   /* test matrices for conformability */
   if(n_col_1-1!=n_row_2)
      nrerror("n_col_1-1!=n_row_2 nonconformable matrices in reduce()");
   if(n_row_1!=n_row_r)
      nrerror("n_row_1!=n_row_r nonconformable matrices in reduce()");
   if(n_col_2!=n_col_r)
      nrerror("n_col_2!=n_col_r nonconformable matrices in reduce()");

   for(i=1;i<=n_row_1;i++){
      for(j=1;j<=n_col_2;j++){
         sum = 0.0;
         for(k=1;k<=n_col_1-1;k++){
            sum += matrix_1[i][k+1] * matrix_2[k][j];
         }
         result[i][j+1] = sum;   /* put sum into result matrix */
      }
      result[i][1] = matrix_1[i][1];   /* transfer augmentation value */
   }
   return (result);
}
```

4 File nr1.c

```
/******************************************************************************/
/* filename nr1.c                                                             */
/*                                                                            */
/*              A Self Configuring High-Order Neural Network                  */
/*                                                                            */
/*                         Ronald Brett Michaels                              */
/*                                                                            */
/*                  The University of Tennessee, Knoxville                    */
/*             Department of Engineering Science and Mechanics                */
/*                                                                            */
/*                                 1991                                       */
/*                                                                            */
/******************************************************************************/
/******************************************************************************/
/* this file contains:                                                        */
/* print_matrix(float **mat,int n);                                           */
/* print_vector(float *,int);                                                 */
/* matrix_product                                                             */
/* matrix_vector_product                                                      */
/* transpose                                                                  */
/* vector_subtract                                                            */
/* vector_add                                                                 */
/* scalar_vector_product                                                      */
/******************************************************************************/

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<alloc.h>
#include<math.h>

   /* prototypes */
void print_matrix(float **,int,int);
void print_vector(float *,int);
float **matrix_product(float **,int,int,float **,int,int,float **,int,int);
float *matrix_vector_product(float **,int,int,float *,int,float *,int);
float **transpose(float **,float **,int,int,int);
float *vector_subtract(float *,float*,float*,int);
float *vector_add(float *,float*,float*,int);
float *scalar_vector_product(float,float *,float *,int);

/******************************************************************************/
/* print_matrix                                                               */
/* this function prints the contents of a matrix to the screen                */
/* this is meant to be a diagnostic function                                  */
/******************************************************************************/

void print_matrix(
   float **mat,      /* pointer to matrix holding data */
```

```
   int n_row,        /* number of rows */
   int n_col         /* number of cols */
)
{
   int i,j;
   printf("\n");
   for(i=1;i<=n_row;i++){
      for(j=1;j<=n_col;j++){
         printf("%f  ",mat[i][j]);
      }
      printf("\n");
   }
}

/****************************************************************************/
/* print_vector                                                             */
/* this function prints vector to screen                                    */
/****************************************************************************/

void print_vector(
   float *vec,             /* pointer to vec to hold values */
   int n                   /* dimensionality of matrix */
)
{
   int p;

   for(p=1;p<=n;p++){
      printf("\n");
      printf("%f ",vec[p]);
   }
   printf("\n");
}


/****************************************************************************/
/* matrix_product                                                           */
/* this function computes the product of two matrices and returns           */
/* a pointer to the result matrix                                           */
/* note that this function provides check for conformability                */
/****************************************************************************/

float **matrix_product(
   float **matrix_1,
   int n_row_1,
   int n_col_1,
   float **matrix_2,
   int n_row_2,
   int n_col_2,
   float **result,
   int n_row_r,
   int n_col_r
```

```
)
{
   float sum;
   int i,j,k;

   /* test matrices for conformability */
   if(n_col_1!=n_row_2)
      nrerror("n_col_1!=n_row_2 nonconformable matrices in matrix_product()");
   if(n_row_1!=n_row_r)
      nrerror("n_row_1!=n_row_r nonconformable matrices in matrix_product()");
   if(n_col_2!=n_col_r)
      nrerror("n_col_1!=n_col_r nonconformable matrices in matrix_product()");

   for(i=1;i<=n_row_1;i++){
      for(j=1;j<=n_col_2;j++){
         sum = 0.0;
         for(k=1;k<=n_col_1;k++){
            sum += matrix_1[i][k] * matrix_2[k][j];
         }
         result[i][j] = sum;   /* put sum into result matrix */
      }
   }
   return (result);
}

/****************************************************************************/
/* matrix_vector_product                                                    */
/* this function computes the product of a matrix and a vector and returns */
/* a pointer to the result vector                                           */
/* note that this function provides check for conformability               */
/****************************************************************************/

float *matrix_vector_product(
   float **mat,                 /* matrix */
   int n_row_1,                 /* number of rows in left matrix */
   int n_col_1,                 /* number of cols in left matrix */
   float *vec,                  /* vector */
   int n_row_2,                 /* number of elements in vector */
   float *result,               /* result vector */
   int n_row_3                  /* number of elements in result vector */
)
{
   float sum;
   int i,j;

      /* test for conformability */
   if(n_col_1!=n_row_2)
      nrerror(
       "n_col_1!=n_row_2 nonconformable matrices in matrix_vector_product()");
   if(n_row_1!=n_row_3)
      nrerror(
```

```
        "n_row_1!=n_row_3 nonconformable matrices in matrix_vector_product()");

   for(i=1;i<=n_row_1;i++){
      sum = 0.0;
      for(j=1;j<=n_col_1;j++){
   sum += mat[i][j] * vec[j];
      }
      result[i] = sum;   /* put sum into result vector */
   }
   return (result);
}

/****************************************************************************/
/* transpose                                                                */
/* this function transposes a matrix pointed to by the first argument and   */
/* places the result in the matrix pointed to by the second argument        */
/* a pointer to the transposed matrix is returned                           */
/* no test for conformability of matrices is performed                      */
/****************************************************************************/

float **transpose(
   float **mat,  /* original matrix */
   float **mat_t,/* transposed matrix */
   int dim,      /* dimension of data */
   int n_1,      /* number of patterns in class 1 */
   int n_2       /* number of patterns in class 2 */
)
{
   int i,j;

   for(i=1;i<=n_1+n_2;i++){
      for(j=1;j<=dim;j++){
         mat_t[j][i] = mat[i][j];
      }
   }
   return mat_t;
}
/****************************************************************************/
/* vector_subtract                                                          */
/* this function computes the difference of two vectors  and returns        */
/* a pointer to the result vector                                           */
/* note that calling function must provide result vector and                */
/* check for conformability                                                 */
/****************************************************************************/

float *vector_subtract(
   float *vector_1,        /* left vector or minuend */
   float *vector_2,        /* right vector or subtrahend */
   float *result,          /* result vector */
   int n_row               /* number of elements in each vector */
)
```

```
{
   int i;

   for(i=1;i<=n_row;i++){
      result[i] = vector_1[i] - vector_2[i];
   }
   return (result);
}

/****************************************************************************/
/* vector_add                                                               */
/* this function computes the sum of two vectors  and returns               */
/* a pointer to the result vector                                           */
/* note that calling function must provide result vector and                */
/* check for conformability                                                 */
/****************************************************************************/

float *vector_add(
   float *vector_1,              /* left vector */
   float *vector_2,              /* right vector or addend */
   float *result,                /* result vector */
   int n_row                     /* number of elements in each vector */
)
{
   int i;

   for(i=1;i<=n_row;i++){
      result[i] = vector_1[i] + vector_2[i];
   }
   return (result);
}
/****************************************************************************/
/* scalar_vector_product                                                    */
/* this function multiplies a scalar times a vector and returns a pointer   */
/* to the result                                                            */
/****************************************************************************/

float *scalar_vector_product(
   float scalar,
   float *vec,
   float *result,
   int n_row
)
{
   int i;

   for(i=1;i<=n_row;i++){
      result[i] = vec[i] * scalar;
   }
   return (result);
}
```

5 File exp_op.c

```
/*****************************************************************************/
/* filename exp_op.c                                                         */
/*                                                                           */
/*             A Self Configuring High-Order Neural Network                  */
/*                                                                           */
/*                        Ronald Brett Michaels                              */
/*                                                                           */
/*                The University of Tennessee, Knoxville                     */
/*             Department of Engineering Science and Mechanics               */
/*                                                                           */
/*                                1991                                       */
/*                                                                           */
/*****************************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<malloc.h>

float **matrix(int,int,int,int);
void free_matrix(float **,int,int,int,int);
void exp_op(float ***,int *,int,int);

/*****************************************************************************/
/* expand                                                                    */
/* this function implements the outer product expansion                      */
/* for example parity3:                                                      */
/* given x1 x2 x3 expand to x1 x2 x3 x1x2 x1x3 x2x3                          */
/* do not include terms with 2 or more equal indices ref. Pao pg. 201        */
/*****************************************************************************/

void exp_op(
   float ***mat, /* pointer to pointer to training pattern matrix X */
   int *dim,     /* pointer to dimension of data */
   int n_1,      /* number of patterns in class 1 */
   int n_2       /* number of patterns in class 2 */
)
{
   int i,j,k,m;
   int dim_new;       /* dimension of expanded data */
   float **mat_new;  /* pointer to temporary training pattern matrix */

   dim_new = *dim;
      /* calculate number of dimensions in expanded vector */
   for(j=1;j<(*dim)-1;j++){
      dim_new += j;
   }
      /* allocate memory to hold input pattern values */
   mat_new = matrix(1,n_1+n_2,1,dim_new);
```

```
      /* expand class 1 patterns */
   for(i=1;i<=n_1;i++){
      m = (*dim)+1;           /* count additonal pattern vector dimensions */
      for(j=1;j<=*dim;j++){
         mat_new[i][j] = (*mat)[i][j];  /* transfer original data */
         for (k=j+2;k<=*dim;k++){     /* calculate expanded values */
            mat_new[i][m] = (*mat)[i][j+1] * (*mat)[i][k];
            m++;
         }
      }
   }

      /* expand class 2 patterns */
   for(;i<=n_1+n_2;i++){
      m = (*dim)+1;           /* count additonal pattern vector dimensions */
      for(j=1;j<=*dim;j++){
         mat_new[i][j] = (*mat)[i][j];  /* transfer original data */
         for (k=j+2;k<=*dim;k++){     /* calculate expanded values */
            mat_new[i][m] = -(-(*mat)[i][j+1] * -(*mat)[i][k]);
            m++;
         }
      }
   }
   free_matrix(*mat,1,n_1+n_2,1,*dim);
   *mat = mat_new;
   *dim = dim_new;
}
```

Appendix C
Source Code for Classifier Program

This is the source code for the classifier program which classifies pattern vectors using the output of the High–Order Neural Network program shown in Appendix B.

The entire program is made up of the following files:

1. class.c
2. bgi.c
3. class_op.c
4. class_nr.c
5. nrerror.c
6. vector.c
7. freevector.c
8. matrix.c
9. freematrix.c

Files 5 through 9 are from the Numerical Recipes C Diskette (Numerical Recipes Software, P.O. Box 243, Cambridge, MA 02238) and are not reproduced here for copyright reasons. File class.c contains the function main().

This program compiles with the Borland C++ compiler version 2.0 running in C mode under the MS–DOS operating system and requires the use of the Borland graphics library.

The program may be compiled using the following command line.

```
bcc class.c bgi.c class_op.c class_nr.c nrerror.c vector.c
   freevector.c matrix.c freematrix.c
```

1 File class.c

```
/*****************************************************************************/
/* filename class.c                                                          */
/*                                                                           */
/*                A Self Configuring High-Order Neural Network               */
/*                                                                           */
/*                           Ronald Brett Michaels                           */
/*                                                                           */
/*                   The University of Tennessee, Knoxville                  */
/*               Department of Engineering Science and Mechanics             */
/*                                                                           */
/*                                   1991                                    */
/*                                                                           */
/*****************************************************************************/
/*****************************************************************************/
/* data input file fl1.tx ordered as follows                                 */
/* THETA                                                                     */
/* dimensionality of translation vector                                      */
/* elements of translation vector                                            */
/* 't' indicates a matrix transformation                                     */
/* number of rows in matrix                                                  */
/* number of cols in matrix                                                  */
/* transformation matrix                                                     */
/* possibly more transformation matrices                                     */
/* 'w' indicates a weight vector                                             */
/* dimensionality of weight vector                                           */
/* weight vector                                                             */
/* nothing else can come after weight vector                                 */
/*****************************************************************************/
/*****************************************************************************/
/* this program reads the output file from the fl1 program and implements    */
/* the multilayer form of the high-order network to make classifications     */
/* of pattern vectors which are randomly generated over a region of          */
/* interest.  these patterns are then plotted on the screen either as white */
/* dots on black or black dots on black.  the result is a speckled region    */
/* which represents class 1 and a black region which represents class 2      */
/* the resulting screen display may then be printed in reverse colors        */
/* using the MS-DOS screen printing utility                                  */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include <float.h>
#include <ctype.h>
#include <graphics.h>
#include <conio.h>
#include <stdarg.h>
#include <dos.h>
#include <time.h>
```

```
   /* function prototypes */
void class_op(float ***,int *,int,int);
void class_t(float ***,int *,int,int);
int get_vector(FILE *,struct vector **);
int get_pattern(FILE *,struct vector **,float);
int get_random_pattern(struct vector **,float);
void scale_pattern(struct vector *,float);
int get_matrix(FILE *,struct matrix **);
void plot_pattern(struct vector *,int);
void nrerror(char *error_text);
float *vector(int nl,int nh);
void free_vector(float *v,int nl,int nh);
float **matrix(int nrl,int nrh,int ncl,int nch);
void free_matrix(float **,int,int,int,int);
void get_parms(FILE *,int *,int *,int *);
void get_patterns(FILE *,float **,int,int,int);
void print_matrix(float **,int,int);
void print_vector(struct vector *);
void vector_subtract(struct vector *,struct vector *,struct vector *);
void vector_matrix_product(struct vector *,struct matrix *,struct vector *);
float dot_product(struct vector *,struct vector *);
int threshold(float);
void free_struct_vector(struct vector *);
void exp_op(struct vector *);
int gr_setup(void);
int gr_close(void);
void axes(void);

typedef struct matrix{
   float **mat;
   int rows;
   int cols;
   struct matrix *next;
};

typedef struct vector{
   float *vec;
   int dim;
};

/*******************************************************************************/
/* main                                                                        */
/*******************************************************************************/

int main()
{
   float theta;                /* threshold value used in separation */
   float summation;            /* value of pattern x weights */
   int result;                 /* is pattern in class 1 or class 2 */
   char flag;                  /* holds 't'or 'w' */
```

```
FILE *fl1_in;                 /* pointer to input file fl1.tx */
struct matrix *transform;     /* pointer to linked list of matrices */
struct matrix *temp;          /* pointer to linked list of matrices */
struct vector *translate;     /* translation vector */
struct vector *weight;        /* H-K weight vector */
struct vector *pattern;       /* test pattern vector as transformed */
struct vector *orig_pat;      /* test pattern vector */
float factor;                 /* scale factor for patterns */

transform = temp = NULL;      /* initialize pointers */
translate = weight = pattern = NULL;
   /* open data input files */
if((fl1_in=fopen("fl1.tx","r"))==NULL){
   nrerror("cannot open data input file fl1.tx");
}
   /* get THETA */
if(fscanf(fl1_in,"%f",&theta)==EOF){
   nrerror("problem with input data file");
}
   /* get translation vector */
if(get_vector(fl1_in,&translate)==EOF){
   nrerror("problem with input data file");
}
   /* get scale factor */
if(fscanf(fl1_in,"%f",&factor)==EOF){
   nrerror("problem with input data file");
}
   /* get transformation matrices and weight vector */
do{
   flag = ' ';
   while(!isgraph(flag)){   /* eat whitespace */
      if(fscanf(fl1_in,"%c",&flag)==EOF){   /* get 't' or 'w' */
         nrerror("problem with input data file");
      }
   }
   switch (flag) {
      case 't':
            /* allocate array and read contents from file */
         if(transform==NULL){
            if(get_matrix(fl1_in,&transform)==EOF){
               nrerror("problem with input data file");
            }

            temp = transform;  /* set temp pointer to this matrix */
         }
         else{
            if(get_matrix(fl1_in,&(temp->next))==EOF){
               nrerror("problem with input data file");
            }
            temp = temp->next;  /* set temp pointer to this matrix */
```

```
            }
            break;
         case 'w':
               /* allocate array and read contents from file */
            if(get_vector(fl1_in,&weight)==EOF){
               nrerror("problem with input data file");
            }
            break;
         default:
            nrerror("expected 't' or 'w' in file fl1.tx");
      }
   } while (flag=='t');   /* last time through should be 'w' */
   fclose(fl1_in);
   randomize();
   gr_setup();  /* activate graphic mode */
   getch();     /* pause after axes plotted */
      /* get patterns and classify them */
   while(!kbhit()){                          /* strike any key to stop */
      get_random_pattern(&orig_pat,theta);
      if((pattern = (struct vector *)malloc(sizeof(struct vector)))==NULL){
         nrerror("cannot allocate memory for struct vector");
      }
      pattern->dim = 1;  /* zero size vector */
      pattern->vec = NULL;
         /* translate pattern */
      vector_subtract(orig_pat,translate,pattern);
         /* scale pattern */
      scale_pattern(pattern,factor);
      temp = transform;
      vector_matrix_product(pattern,temp,pattern);
      temp = temp->next;
      while(temp!=NULL){
         exp_op(pattern);
         vector_matrix_product(pattern,temp,pattern);
         temp = temp->next;
      }
      summation = dot_product(pattern,weight);
      result = threshold(summation);
      plot_pattern(orig_pat,result);
      free_struct_vector(pattern);
      free_struct_vector(orig_pat);
   }
   axes();        /* paint axes on screen */
   getch();      /* pause after axes plotted */
   gr_close();   /* close graphics screen */
   /* allocated memory dies automatically */
   /* files close automatically */
   return 0;
}

/******************************************************************************/
```

```
/* get_vector                                                              */
/* this function allocates space for a vector array and gets the components */
/* from the file pointed to by FILE *                                      */
/***************************************************************************/

int get_vector(
   FILE *fl1_in,
   struct vector **a_vec
)
{
   int i;

   if(((*a_vec) = (struct vector *)malloc(sizeof(struct vector)))==NULL){
      nrerror("cannot allocate memory for struct vector");
   }
   if(fscanf(fl1_in,"%d",&((*a_vec)->dim))==EOF){   /* get dimensionality */
      return EOF;
   }
      /* allocate a vector */
   (*a_vec)->vec = vector(1,(*a_vec)->dim);
   for(i=1;i<=(*a_vec)->dim;i++){   /* get translation vector */
      if(fscanf(fl1_in,"%f",&((*a_vec)->vec[i]))==EOF){
         return EOF;
      }
   }
   return 1;
}

/***************************************************************************/
/* get_pattern                                                             */
/* this function allocates space for a vector array and gets the components */
/* from the file pointed to by FILE *                                      */
/* note that vectors are augmented with threshold                          */
/***************************************************************************/

int get_pattern(
   FILE *fl1_pat,
   struct vector *(*a_vec),
   float theta
)
{
   int i;

   if(((*a_vec) = (struct vector *)malloc(sizeof(struct vector)))==NULL){
      nrerror("cannot allocate memory for struct vector");
   }
   if(fscanf(fl1_pat,"%d",&((*a_vec)->dim))==EOF){   /* get dimensionality */
      return EOF;
   }
   ((*a_vec)->dim)++;   /* increment by one for threshold theta */
      /* allocate a vector */
```

```
   (*a_vec)->vec = vector(1,((*a_vec)->dim));
   (*a_vec)->vec[1] = theta;   /* augment pattern vector */
   for(i=2;i<=(*a_vec)->dim;i++){   /* get  vector */
      if(fscanf(fl1_pat,"%f",&((*a_vec)->vec[i]))==EOF){
         return EOF;
      }
   }
   return 1;
}

/*****************************************************************************/
/* get_matrix                                                                */
/* this function allocates space for a matrix array and gets the components */
/* from the file pointed to by FILE *                                        */
/*****************************************************************************/

int get_matrix(
   FILE *fl1_in,
   struct matrix **a_mat
)
{
   int i,j;

   if(((*a_mat) = (struct matrix *)malloc(sizeof(struct matrix)))==NULL){
      nrerror("cannot allocate memory for struct matrix");
   }
      /* get row and col values for matrix */
   if(fscanf(fl1_in,"%d%d",&(((*a_mat))->rows),&(((*a_mat))->cols))==EOF){
      return EOF;
   }
      /* allocate a matrix */
   (*a_mat)->mat = matrix(1,(*a_mat)->rows,1,(*a_mat)->cols);

      /* get matrix values */
   for(i=1;i<=((*a_mat))->rows;i++){
      for(j=1;j<=((*a_mat))->cols;j++){
         if(fscanf(fl1_in,"%f",&(((*a_mat))->mat[i][j]))==EOF){
            return EOF;
         }
      }
   }
   ((*a_mat))->next = NULL;         /* null pointer to next matrix */
   return 1;
}

/*****************************************************************************/
/* free_struct_vector                                                        */
/* this function frees space of a vector array                               */
/*****************************************************************************/

void free_struct_vector(
```

```
   struct vector *a_vec
)
{
   free_vector(a_vec->vec,1,a_vec->dim);
   free(a_vec);
}

/******************************************************************************/
/* scale_pattern                                                              */
/* this function scales the pattern and places it back in the pattern holder*/
/******************************************************************************/

void scale_pattern(
   struct vector *a_vec,
   float factor
)
{
   int i;

   for(i=2;i<=a_vec->dim;i++){
      a_vec->vec[i] /=factor;
   }
}
```

2 File bgi.c

```
/******************************************************************************/
/* filename bgi.c                                                             */
/*                                                                            */
/*              A Self Configuring High-Order Neural Network                  */
/*                                                                            */
/*                          Ronald Brett Michaels                             */
/*                                                                            */
/*                  The University of Tennessee, Knoxville                    */
/*             Department of Engineering Science and Mechanics                */
/*                                                                            */
/*                                 1991                                       */
/*                                                                            */
/******************************************************************************/
/******************************************************************************/
/* this file contains the graphics operations for class()                    */
/******************************************************************************/
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>
#include <dos.h>

#define MAX (0.4)  /* max scale reading on graph */
#define MIN (-0.1)   /* scale reading at axis crossing */

#define X_MIN_PAT (-0.1)
#define X_MAX_PAT (0.4)
#define Y_MIN_PAT (-0.1)
#define Y_MAX_PAT (0.4)

#define U(x) ((unsigned)x)
#define X(x) (x)        /* conversion macro for coordinates */
#define Y(y) (480-(y))  /* conversion macro for coordinates */

   /* prototypes */
int gr_setup(void);
int gr_close(void);
int  gprintf(int,int,char *fmt, ... );
void plot_pattern(struct vector *,int);
int get_random_pattern(struct vector **,float);
void nrerror(char *error_text);
float *vector(int nl,int nh);

typedef struct matrix{
   float **mat;
   int rows;
   int cols;
   struct matrix *next;
```

```
};
typedef struct vector{
   float *vec;
   int dim;
};

/*****************************************************************************/
/* gr_setup                                                                  */
/* this function sets up the graphic screen                                  */
/* this function modified from Borland help screen                           */
/*****************************************************************************/

int gr_setup(void)
{
   /* request auto detection */
   int gdriver = VGA;
   int  gmode = VGAHI;
   int errorcode;

   /* initialize graphics and local variables */
   initgraph(&gdriver, &gmode, "d:\borlandc\bgi");

   /* read result of initialization */
   errorcode = graphresult();
   if (errorcode != grOk){  /* an error occurred */
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1); /* terminate with an error code */
   }
   cleardevice();
   return 0;
}

/*****************************************************************************/
/* gr_close                                                                  */
/* this function closes the graphic screen                                   */
/* this function modified from Borland help screen                           */
/*****************************************************************************/

int gr_close(void){

   getch();
   /* clean up */
   closegraph();
   return 0;
}

/*****************************************************************************/
/* GPRINTF: Used like PRINTF except the output is sent to the               */
/* screen in graphics mode at the specified co-ordinate.                    */
```

```
/* this function adapted from Borland example bgidemo.c                       */
/******************************************************************************/

int gprintf(
   int xloc,   /* x coordinate */
   int yloc,   /* y coordinate */
   char *fmt, ... )
{
  va_list  argptr;          /* Argument list pointer   */
  char str[140];          /* Buffer to build sting into */
  int cnt;             /* Result of SPRINTF for return */

  va_start( argptr, fmt );    /* Initialize va_ functions   */

  cnt = vsprintf( str, fmt, argptr ); /* prints string to buffer */
  outtextxy(xloc,yloc, str );   /* Send string in graphics mode */
  yloc += textheight( "H" ) + 2;      /* Advance to next line         */

  va_end( argptr );         /* Close va_ functions     */

  return( cnt );         /* Return the conversion count   */
}

/******************************************************************************/
/* get_random_pattern                                                         */
/* this function allocates space for a vector array and gets random           */
/* components                                                                 */
/* note that vectors are augmented with threshold                             */
/******************************************************************************/

int get_random_pattern(
   struct vector **a_vec,
   float theta
)
{

   if(((*a_vec) = (struct vector *)malloc(sizeof(struct vector)))==NULL){
      nrerror("cannot allocate memory for struct vector");
   }
   (*a_vec)->dim = 3;  /* this function hardwired for 2-d problems */
      /* allocate a vector */
   (*a_vec)->vec = vector(1,((*a_vec)->dim));
   (*a_vec)->vec[1] = theta;   /* augment pattern vector */

   (*a_vec)->vec[2] =
      (((float)random((int)((X_MAX_PAT - X_MIN_PAT)*1000.0)))/
         1000.0)+X_MIN_PAT;
   (*a_vec)->vec[3] =
      (((float)random((int)((Y_MAX_PAT - Y_MIN_PAT)*1000.0)))/
         1000.0)+Y_MIN_PAT;
```

```
   return 1;
}

/*****************************************************************************/
/* plot_pattern                                                              */
/* this function plots the pattern vector and colors the pixel               */
/* black for class 0 and white for class 1                                   */
/* note that theta, the threshold, is not plotted                            */
/*****************************************************************************/

void plot_pattern(
   struct vector *pat,
   int class
)
{
   if(class==0){
      putpixel(X((int)(((pat->vec[2]-MIN)*(float)300)*(1.25/(MAX-MIN)))+175),
         Y((int)(((pat->vec[3]-MIN)*(float)300)*(1.25/(MAX-MIN)))+90),BLACK);
   }
   else{
      if(class==1){
         putpixel(X((int)(((pat->vec[2]-MIN)*(float)300)*
            (1.25/(MAX-MIN)))+175),
            Y((int)(((pat->vec[3]-MIN)*(float)300)*(1.25/(MAX-MIN)))+90),
            WHITE);
      }
   }
}

/*****************************************************************************/
/* axes                                                                      */
/* this function paints the axes onto the screen                             */
/*****************************************************************************/

void axes()
{
   int i;
   int style;
   int size;

   style = 2;
   size = 6;


      /* select the text style */
   settextstyle(style, HORIZ_DIR, size);

      /* output a line */
   setlinestyle(SOLID_LINE,0xffff,NORM_WIDTH);

      /* vertical axis */
```

```
    moveto(X(175),Y(15));
    lineto(X(175),Y(465));

       /* vertical scale */
    for(i=15;i<=465;i+=75){
       moveto(X(170),Y(i));
       lineto(X(180),Y(i));
    }
       /* vertical numbers */
    for(i=15;i<=465;i+=75){
       if(i==90)continue;   /* skip 0.0 */
       gprintf(X(120),Y(i+10),
          "%5.2f",((MAX-MIN)/1.25)*(float)(i-90)/(float)(300)+MIN);
    }
       /* horizontal axis */
    moveto(X(100),Y(90));
    lineto(X(550),Y(90));

       /* horizontal scale */
    for(i=100;i<=550;i+=75){
       moveto(X(i),Y(85));
       lineto(X(i),Y(95));
    }
       /* horizontal numbers */
    for(i=100;i<=550;i+=75){
       if(i==175)continue;   /* skip 0.0 */
       gprintf(X(i-20),Y(81),
          "%5.2f",((MAX-MIN)/1.25)*(float)(i-175)/(float)(300)+MIN);
    }
       /* put in zero or whatever MIN is */
    gprintf(X(130),Y(81),"%4.1f",MIN);

       /* templates for lines
    moveto(X(),Y());
    lineto(X(),Y());
       */
}
```

3 File class_op.c

```
/*****************************************************************************/
/* filename class_op.c                                                       */
/*                                                                           */
/*              A Self Configuring High-Order Neural Network                 */
/*                                                                           */
/*                        Ronald Brett Michaels                              */
/*                                                                           */
/*                 The University of Tennessee, Knoxville                    */
/*              Department of Engineering Science and Mechanics              */
/*                                                                           */
/*                                1991                                       */
/*                                                                           */
/*****************************************************************************/
/*****************************************************************************/
/* this file implements the outer product expansion                          */
/*****************************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<malloc.h>

float **matrix(int,int,int,int);
void free_matrix(float **,int,int,int,int);
void exp_op(struct vector *);
float *vector(int,int);
void nrerror(char *);
void free_vector(float *,int,int);

typedef struct vector{
   float *vec;
   int dim;
};

/*****************************************************************************/
/* exp_op                                                                    */
/* this function implements the outer product expansion                      */
/* for example parity3:                                                      */
/* given x1 x2 x3 expand to x1 x2 x3 x1x2 x1x3 x2x3                          */
/* do not include terms with 2 or more equal indices ref. Pao pg. 201        */
/*****************************************************************************/

void exp_op(
   struct vector *a_vec     /* pointer to pattern */
)
{
   int j,k,m;          /* loop counters */
   int dim_new;        /* dimension of expanded pattern vector */
   float *vec_new;     /* pointer to temporary pattern matrix */
```

```
    dim_new = a_vec->dim;
       /* calculate number of dimensions in expanded vector */
    for(j=1;j<a_vec->dim-1;j++){
       dim_new += j;
    }
       /* allocate memory to hold input pattern values */
    vec_new = vector(1,dim_new);

       /* expand pattern */
    for(j=1;j<=a_vec->dim;j++){
       vec_new[j] = a_vec->vec[j];  /* transfer original data */
    }
    m = (a_vec->dim)+1;     /* count additional pattern vector dimensions */
    for(j=1;j<=a_vec->dim;j++){
       for (k=j+2;k<=a_vec->dim;k++){     /* calculate expanded values */
          vec_new[m] = a_vec->vec[j+1] * a_vec->vec[k];
          m++;
       }
    }
    free_vector(a_vec->vec,1,a_vec->dim);
    a_vec->vec = vec_new;
    a_vec->dim = dim_new;
}
```

4 File class_nr.c

```
/******************************************************************************/
/* filename class_nr.c                                                        */
/*                                                                            */
/*              A Self Configuring High-Order Neural Network                  */
/*                                                                            */
/*                        Ronald Brett Michaels                               */
/*                                                                            */
/*                 The University of Tennessee, Knoxville                     */
/*             Department of Engineering Science and Mechanics                */
/*                                                                            */
/*                                1991                                        */
/*                                                                            */
/******************************************************************************/
/******************************************************************************/
/* filename class_nr.c                                                        */
/* this file contains:                                                        */
/* print_matrix(float **mat,int n);                                           */
/* print_vector(float *,int);                                                 */
/* matrix_vector_product                                                      */
/* vector_subtract                                                            */
/* vector_add                                                                 */
/* vector_matrix_product                                                      */
/* dot_product                                                                */
/* scalar_vector_product                                                      */
/* threshold                                                                  */
/******************************************************************************/

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<malloc.h>
#include<math.h>

#define THRESHOLD (0.0)

   /* prototypes */
void print_matrix(float **,int,int);
void print_vector(struct vector *);
float *matrix_vector_product(float **,int,int,float *,int,float *,int);
float dot_product(struct vector *,struct vector *);
void vector_matrix_product(struct vector *,struct matrix *,struct vector *);
void vector_subtract(struct vector *,struct vector *,struct vector *);
float *vector_add(float *,float*,float*,int);
float *scalar_vector_product(float,float *,float *,int);
int threshold(float);

typedef struct matrix{
   float **mat;
   int rows;
```

```
   int cols;
   struct matrix *next;
};

typedef struct vector{
   float *vec;
   int dim;
};

/*****************************************************************************/
/* print_matrix                                                              */
/* this function prints the contents of a matrix to the screen               */
/* this is meant to be a diagnostic function                                 */
/*****************************************************************************/

void print_matrix(
   float **mat,        /* pointer to matrix holding data */
   int n_row,          /* number of rows */
   int n_col           /* number of cols */
)
{
   int i,j,k;

   for(k=27;k<=n_col;k+=27){
      printf("\n");
      for(i=1;i<=n_row;i++){
         for(j=k-26;j<=k;j++){
            printf("%10.6f",mat[i][j]);
         }
         printf("\n");
      }
   }
   printf("\n");
   for(i=1;i<=n_row;i++){
      for(j=k-26;j<=n_col;j++){
         printf("%10.6f",mat[i][j]);
      }
      printf("\n");
   }
}

/*****************************************************************************/
/* print_vector                                                              */
/* this function prints vector to screen                                     */
/*****************************************************************************/

void print_vector(
   struct vector *a_vec          /* pointer to vec to hold values */
)
{
   int p;
```

```
   for(p=1;p<=a_vec->dim;p++){
      printf("\n");
      printf("%10.6f ",a_vec->vec[p]);
   }
   printf("\n");
}


/*****************************************************************************/
/* vector_matrix_product                                                     */
/* this function computes the product of a vector and a matrix               */
/* note that this function provides check for conformability                 */
/*****************************************************************************/

void vector_matrix_product(
   struct vector *a_vec,             /* vector */
   struct matrix *a_mat,             /* matrix */
   struct vector *vec_result         /* result vector */
)
{
   float sum;
   float *temp;
   int i,j;

      /* test for conformability */
   if(a_vec->dim!=a_mat->rows+1){
      nrerror("nonconformable matrices in vector_matrix_product()");
   }
   temp = vector(1,a_mat->cols+1);
   temp[1] = a_vec->vec[1];  /* transfer theta from pattern to expanded pat */
   for(i=1;i<=a_mat->cols;i++){
      sum = 0.0;
      for(j=1;j<=a_mat->rows;j++){
         sum += a_vec->vec[j+1] * a_mat->mat[j][i];
      }
      temp[i+1] = sum;   /* put sum into result vector */
   }
   free_vector(vec_result->vec,1,vec_result->dim);
   vec_result->dim = a_mat->cols+1;
   vec_result->vec = temp;
}

/*****************************************************************************/
/* dot_product                                                               */
/* this function computes the dot product of two vectors and returns         */
/* the result                                                                */
/* note that this function provides check for conformability                 */
/*****************************************************************************/

float dot_product(
```

```
   struct vector *a_vec,              /* vector */
   struct vector *b_vec               /* vector */
)
{
   float sum;
   int i;

      /* test for conformability */
   if(a_vec->dim!=b_vec->dim){
      nrerror("nonconformable vectors in dot_product()");
   }
   sum = 0.0;
   for(i=1;i<=a_vec->dim;i++){
      sum += a_vec->vec[i] * b_vec->vec[i];
   }
   return sum;
}


/******************************************************************************/
/* matrix_vector_product                                                      */
/* this function computes the product of a matrix and a vector and returns   */
/* a pointer to the result vector                                             */
/* note that this function provides check for conformability                  */
/******************************************************************************/

float *matrix_vector_product(
   float **mat,                     /* matrix */
   int n_row_1,                     /* number of rows in left matrix */
   int n_col_1,                     /* number of cols in left matrix */
   float *vec,                      /* vector */
   int n_row_2,                     /* number of elements in vector */
   float *result,                   /* result vector */
   int n_row_3                      /* number of elements in result vector */
)
{
   float sum;
   int i,j;

      /* test for conformability */
   if(n_col_1!=n_row_2)nrerror(
      "n_col_1!=n_row_2 nonconformable matrices in matrix_vector_product()");
   if(n_row_1!=n_row_3)nrerror(
      "n_row_1!=n_row_3 nonconformable matrices in matrix_vector_product()");

   for(i=1;i<=n_row_1;i++){
      sum = 0.0;
      for(j=1;j<=n_col_1;j++){
         sum += mat[i][j] * vec[j];
      }
      result[i] = sum;   /* put sum into result vector */
```

```
   }
   return (result);
}

/*****************************************************************************/
/* vector_subtract                                                           */
/* this function computes the difference of two vectors                      */
/* note that calling function must provide result vector                     */
/* this function checks for conformability                                   */
/*****************************************************************************/

void vector_subtract(
   struct vector *vec_1,            /* left vector or minuend */
   struct vector *vec_2,            /* right vector or subtrahend */
   struct vector *vec_result        /* result vector */
)
{
   int i;

   if(vec_1->dim!=vec_2->dim){
      nrerror("nonconforming vectors in vector subtract");
   }
   free_vector(vec_result->vec,1,vec_result->dim);
   vec_result->dim = vec_1->dim;
      /* allocate a vector */
   vec_result->vec = vector(1,vec_result->dim);
   for(i=1;i<=vec_1->dim;i++){
      vec_result->vec[i] = vec_1->vec[i] - vec_2->vec[i];
   }
}

/*****************************************************************************/
/* vector_add                                                                */
/* this function computes the sum of two vectors  and returns                */
/* a pointer to the result vector                                            */
/* note that calling function must provide result vector and                 */
/* check for conformability                                                  */
/*****************************************************************************/

float *vector_add(
   float *vector_1,             /* left vector */
   float *vector_2,             /* right vector or addend */
   float *result,               /* result vector */
   int n_row                    /* number of elements in each vector */
)
{
   int i;

   for(i=1;i<=n_row;i++){
      result[i] = vector_1[i] + vector_2[i];
   }
```

```
   return (result);
}
/*****************************************************************************/
/* scalar_vector_product                                                     */
/* this function multiplies a scalar times a vector and returns a pointer   */
/* to the result                                                             */
/*****************************************************************************/

float *scalar_vector_product(
   float scalar,
   float *vec,
   float *result,
   int n_row
)
{
   int i;

   for(i=1;i<=n_row;i++){
      result[i] = vec[i] * scalar;
   }
   return (result);
}

/*****************************************************************************/
/* threshold                                                                 */
/* this function thresholds the summation of weight x pattern               */
/* threshold is at THRESHOLD                                                 */
/*****************************************************************************/

int threshold(
   float value
)
{
   if(value<THRESHOLD)return 0;
   else return 1;
}
```

VITA

Ronald Brett Michaels was born in Knoxville, Tennessee on September 3, 1943. He is a graduate of Young High School. He served in the U.S. Navy as an enlisted man from 1963 to 1967. He received the B.S. degree in Engineering Physics from The University of Tennessee in 1970. He is a returned Peace Corps Volunteer, having served in Ghana. He has been employed as an engineer in the construction industry, primarily at overseas locations, for a number of years. Most recently, he was a Lecturer in Civil Engineering at Harare Polytechnic, Harare, Zimbabwe from 1985 through 1989.

He is currently a graduate student at The University of Tennessee in the Department of Engineering Science and Mechanics.