

To the Graduate Council:

I am submitting herewith a thesis written by John Joseph Gerber entitled "A Dynamically Allocating Piecewise Linear 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 Computer Science.

Jens Gregor

Jens Gregor, Major Professor

We have read this thesis
and recommend its acceptance:

Michael G. Thomson

Bruce Hor Mann

Accepted for the Council:

Cwminkel

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, provided that accurate acknowledgment 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 for thesis financial gain shall not be allowed without my written permission.

Signature



Date

11/29/94

**A Dynamically Allocating
Piecewise Linear
Neural Network**

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

John Joseph Gerber

December 1994

DEDICATION

While there are many people I would like to thank for their support, there are three people who have made this thesis possible. To these very special individuals, I would like to dedicate my thesis. While I hope they know who they are and how grateful I am to them, there is nothing like seeing your name in print. My support team is of course my parents, David and Judith Gerber, and my best friend, Emily Paine. I know at times they probably had their doubts whether this thesis would ever be written. I know I did. They believed in me and their belief made it all possible. While they might not understand much of what follows, I do hope they know how much they have helped me to achieve this and many other goals in my life.

ACKNOWLEDGMENTS

I am indebted to my advisor and major professor, Dr. Jens Gregor. It has been an honor to work with him. His guidance and patience have made this thesis possible. I would also like to thank the other committee members, Dr. Bruce MacLennan and Dr. Michael Thomason, for their comments and assistance in developing this thesis. Finally, I would like to thank Greg Cole, a nice guy and a great boss. While his encouragement and understanding in relation to the time requirements of thesis work would be enough to gain my gratitude, by allowing me the use of the facilities at work he has made this thesis possible.

ABSTRACT

The research reported within this thesis focuses on applying the methods of piecewise linear classification to artificial neural networks. To some extent, artificial neural networks utilize a black box strategy, in that the classification is done from an input/output viewpoint. Organization and training are handled by the network producing results that, while quite possibly correct for the training set, are difficult to predict and understand. Several choices must be made in order to use feed forward back propagation neural networks effectively. One of the key choice is network topology, in respect to the number of layers and the number of units in each layer. Recent work within the artificial neural network community has been to develop a network that can dynamically allocate units during training and create its own structure. This thesis describes a neural network that configures itself by dynamically allocating units based on the techniques of piecewise linear classification.

Piecewise linear classification methods are based on the idea that if nonlinearly separable regions could be separated into subregions consisting of linearly separable patterns, then each subregion could be learned by a linear discriminant function. The dynamically allocating piecewise linear (DAPL) neural network uses the techniques developed in piecewise linear classification for deciding when to divide nonlinearly separable regions. Division of the resulting subregions continues until only subregions consisting of linearly separable patterns are generated. A single layered artificial neural subnetwork is generated each time a region needs to be split into subregions. Through adjustments to the training set, a single sided decision surface is generated within the subnetwork for linearly separating the region. The resulting set of subnetworks produces hyperplanes that linearly separate the training space. On top of the subnetworks, an overseer network is built combining the hyperplanes into a decision surface. The resulting network not only handles the problem of dynamically configuring itself, its units function in developing a more predictable decision surface.

When creating the DAPL network approach, emphasis was placed on developing a self

configuring network that would use less of a black box strategy while still being able to generalize. The resulting network compares favorably in timing and correctness to existing self configuring networks.

Contents

1	Overview	1
1.1	Introduction	1
1.2	Pattern Recognition Systems	3
1.3	Classifier	6
1.4	Neural Networks	8
1.4.1	Units	9
1.4.2	Links	11
1.4.3	Layers	12
1.5	Discussion of the Title	15
2	Background and Methodology	16
2.1	Piecewise Linear Classifiers	16
2.1.1	Nilsson's Committee Machine	17
2.1.2	Mizoguchi et al.'s Classifier	19
2.1.3	Lee and Richards' Classifier	20
2.2	Dynamically Configuring Neural Networks	20
2.2.1	Hirose et al.'s Network	21
2.2.2	Fahlman and Lebiere's Cascade Correlation Network	22
2.2.3	Romaniuk and Hall's Divide and Conquer Neural Network	24
2.3	Pre and Post Processing	26
2.4	Summary	27

3	The DAPL Methodology	28
3.1	Introduction	28
3.2	Basic Algorithm	29
3.3	Division of Classes	31
3.4	The Overseer Network	32
3.5	Variations from Standard Feed Forward Training	33
3.6	Noise Handling and Generalization	35
3.7	Correction of the Single Sided Decision Surfaces	38
4	Implementation of the DAPL Algorithm	42
4.1	System and Tools	42
4.2	Code for Creating the DAPL Network	44
4.3	Trace of an Example	54
5	Results	58
5.1	Introduction	58
5.1.1	Sample Data	58
5.1.2	Initial Setup	59
5.2	Results	62
5.2.1	Comparison to Hirose et al.	63
5.2.2	Overseer Network	64
5.2.3	Decision Surfaces	64
6	Summary and Conclusions	67
6.1	Summary	67
6.2	Continuation of Work	68
	Bibliography	71
	Appendices	75

A	User Manual for Running the DAPL Algorithm	76
A.1	Introduction	76
A.2	Getting Started	76
B	A Flow Diagram of the DAPL Code	83
C	Parameter Settings within the DAPL Network	85
C.1	Learning Rules	85
C.2	Activation Functions	85
C.3	Output Functions	87
	Vita	88

List of Tables

5.1	Time: Training vs. Construction	61
5.2	Linear Breaks: Timing vs. Accuracy	63
5.3	Timing and Accuracy: DAPL vs. Hirose et al.	64
5.4	Training Time: Subnetworks vs. Overseer Network	66
C.1	Learning Rules	85
C.2	Activation Functions	86
C.3	Output Functions	87

List of Figures

1.1	Data Set Division	3
1.2	Pattern Recognition System	4
1.3	Zero Error Rate	5
1.4	Classifier	7
1.5	Linear Machine	9
1.6	Neural Network	10
1.7	Single Layered Neural Network	13
1.8	Perceptron Net Layers	14
2.1	Piecewise Linear Classifier	17
2.2	Nilsson's Committee Machine Approach	18
2.3	Mizoguchi et al.'s Approach	19
2.4	Lee and Richards' Approach	21
2.5	Hirose et al.'s Approach	22
2.6	Fahlman and Lebiere's Cascade Correlation Approach	23
2.7	Romaniuk and Hall's Divide and Conquer Approach	24
3.1	Multiclass Problem	31
3.2	Development of a DAPL network	34
3.3	The Use of Strikes	35
3.4	Correcting Single Sided Decision Surfaces	39
3.5	The Adjustment Formula for Single Sided Decision Surface	41

4.1	Stuttgart Neural Network Simulator	43
4.2	The Core Program Flow	45
4.3	DAPL Network Development on the UT Problem	54
4.4	Analyze Decision Surfaces of the UT Problem	56
4.5	DAPL Network Development with Breaks on the UT Problem	57
4.6	Analyze Decision Surfaces of the UT Problem with Breaks	57
5.1	The Two Class U and T Shaped Class Regions	59
5.2	Three Class Banana Shaped Sample Class Regions	60
5.3	The Eight Class Regions	60
5.4	Exactness Specification	63
5.5	The Resulting Decision Surfaces for 3 Classification Problems	65
A.1	Help Menu	77
A.2	Main Menu	77
A.3	Configuration Menu	78
A.4	Display Menu	79
A.5	Edit Menu	79
A.6	Load/Save Menu	80
A.7	Load Graph Menu	80
A.8	Class Menu	80
A.9	SNNS Operations Menu	81
A.10	Matlab Operations Menu	81
A.11	Displaying Results	82
B.1	The Whole Program Flow	84

Chapter 1

Overview

This chapter provides an overview of the thesis and the concepts that will be used throughout the thesis. A brief discussion of the concepts and terminology of pattern recognition systems, classifiers, and artificial neural networks follows.

1.1 Introduction

The purpose of this thesis is to describe a new approach for dynamically creating an artificial neural network (henceforth referred to as a neural network), using methods developed for piecewise linear classification. The resulting dynamically allocating piecewise linear neural network (henceforth referred to as the DAPL network), combines the adaptability of a neural network, the benefits of a dynamically configuring classifier, and the predictability of a piecewise linear classifier.

Following is a brief outline of the chapters presented in this thesis. Chapter 1 discusses an overview of this thesis and the concepts behind it. The foundations of pattern recognition systems and classifiers are given, followed by the basic terminology and concepts of neural networks. Finally, the terminology is related to the thesis's title for a basic understanding of the DAPL network.

Chapter 2 explores some of the current work in piecewise linear classification and dynamically configuring neural networks that influenced the development of the DAPL algorithm. The piecewise linear classifiers of Nilsson, Mizoguchi et al., and Lee and Richards are dis-

cussed along with the neural networks developed by Hirose et al., Fahlman and Libiere, and Romaniuk and Hall. Concluding the chapter is a brief discussion of some work relating to pre and post processing of the training sets and the networks, respectively.

The DAPL algorithm is built upon the approaches mentioned in Chapter 2. Chapter 3 discusses how Chapter 2's approaches were developed, adapted, and merged into the DAPL algorithm. At a high level, the ideas implemented in the DAPL algorithm are discussed.

Many procedures comprise the complete DAPL algorithm, that in addition to generating the DAPL neural network also provides a rich environment in which to create training sets and analyze resulting networks. Chapter 4 discusses the tools used by the DAPL algorithm along with pseudo-code of the basic procedures that play a prominent role in the generation of the network. In this chapter, the ideas discussed in Chapter 3 are given form.

Chapter 5 reports on the results of several pattern classification experiments. The effects of different parameters on the network are demonstrated. Timing considerations, classification correctness, and the inner workings of the network are the main focus of this chapter.

Finally, Chapter 6 provides a summary and conclusion to the thesis. An overview of the goals and the success in accomplishing them are reviewed. The goals that are yet to be fulfilled along with future work are also discussed.

Three appendices are provided for a better understanding of the DAPL algorithm capabilities. Appendix A gives a short user manual for the interactive operation of the DAPL algorithm. The menus are shown with an explanation to the various options. Appendix B provides a flow diagram of the complete program. Appendix C consists of tables showing the different learning, activation, and output functions available for use within the DAPL algorithm.

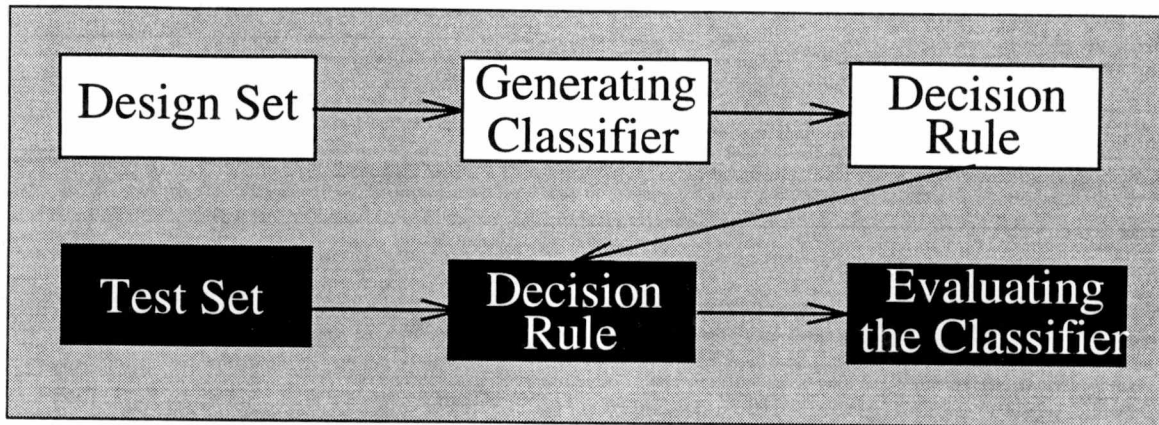


Figure 1.1: Examples of dividing the data set into the design and test set.

1.2 Pattern Recognition Systems

Pattern Recognition (PR) is the science that concerns the description or classification (recognition) of measurements (features) [Sch92]. In many applications, an object needs to be assigned to one of several classes. The pattern features (which may be obtained through a variety of audio, visual, and other sensors) become the measurements upon which classification is determined.

Patterns from various classes are grouped into data sets for training and evaluation. Several methods for deciding how to divide and handle the data sets are discussed in [Han86]. In this thesis, the data sets will be divided into two subsets: the design and the test sets. This division is shown in Figure 1.1. The design set, often referred to as the training set, is used for the training and generation of the classifier. A carefully chosen training set can produce a near optimal network. The test set is used on the resulting classifier to evaluate the classifier's performance. Division of the data set helps keep the classifier from over-training on specific data and producing results that do not accurately reflect the classifier's capabilities.

Figure 1.2 displays the general structure of a pattern recognition system. The stages consisting of the sensor/transducer, processing and enhancements, and the feature/primitives extraction algorithm will be considered outside the scope of this thesis. It will be assumed

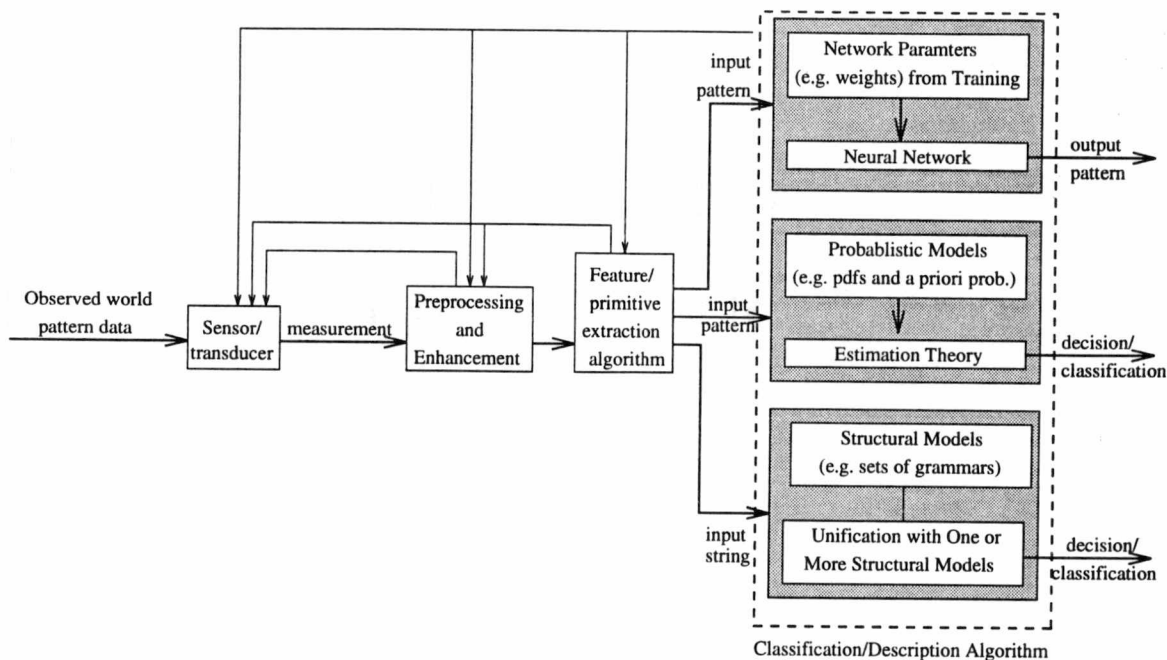


Figure 1.2: Typical pattern recognition system structure [Sch92].

that the DAPL algorithm will start with a data set already divided into the training and the test set. Once a data set is selected, a classifier will need to be developed. Three approaches for developing a classifier exist: the structural, the decision theoretic, and the neural network approach.

In the structural approach, patterns are typically represented as strings, trees, or graphs consisting of sub-patterns, or pattern primitives. Defining the sub-patterns depends largely on intuition and knowledge of the patterns under study. The sub-patterns are applied to the principle of ‘analysis by synthesis’ to detect known structures. One model per pattern class is used to specify possible ways of configuring the sub-patterns. Probabilities may be incorporated into the models to reflect the likelihood of noise. This approach offers the benefit of dealing with pattern data that could not be easily described numerically.

In the decision theoretic approach, patterns are composed of measurable features that are points in a multidimensional feature space. Classification strategies are developed that form the classifiers. Discriminant functions implement these strategies by dividing the feature space into disjoint regions representing the classes. Correct classification of the

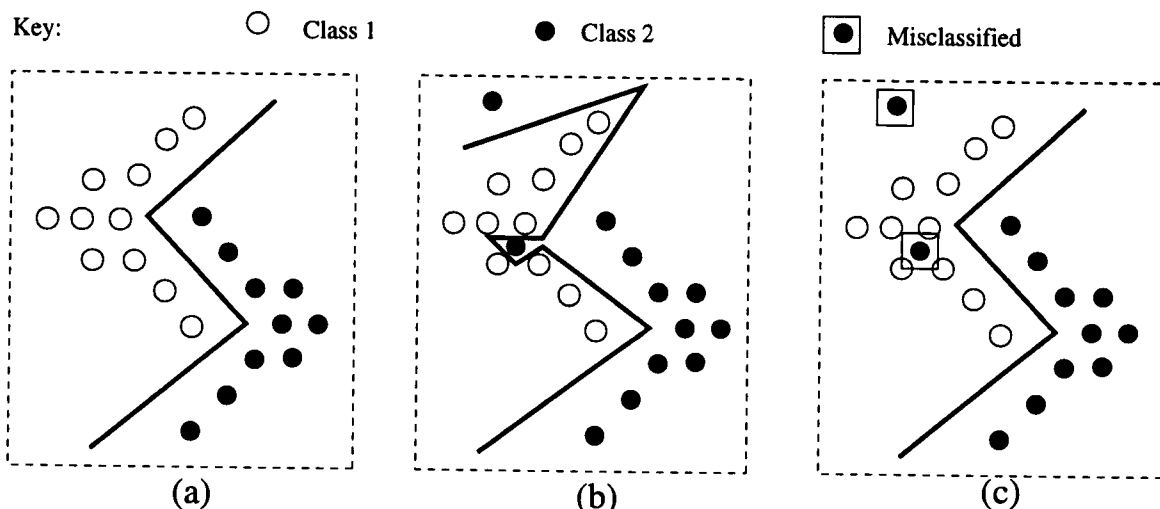


Figure 1.3: Piecewise linear classification with different acceptable error rates. In part (a), perfect data with perfect classification is demonstrated. In part (b), the affects are shown on the decision surface of noisy data coupled with a zero error rate. The desired decision surface is represented in part (c), with misclassification occurring.

patterns is the goal. Noise often makes the goal of perfect classification with a zero error rate impossible and sometimes undesirable. A zero error rate can indicate adaptation to particularities of one data set. Figure 1.3 demonstrates such a condition using a piecewise linear classification strategy. While attempts are made by the classifier to minimize the error, an acceptable error rate must be decided upon.

The neural network approach uses a network to act as its classifier. The network is composed of many cells (units) connected by links. Patterns are stored across these links through the use of weights, similar to the human brain, thus the use of the word “artificial” to distinguish these networks from the biological neural networks. It is the weights that are iteratively adjusted to produce correct output. Within the decision theoretic approach, a similar classification method exists where linear discriminant functions are developed through iteratively adjusting the function’s parameters (weights).

This thesis is interested in taking advantage of the similarities existing between methods in the decision theoretic and the neural network approaches. Within the decision theoretic approach, the piecewise linear classification strategy which uses linear discriminant function

will be examined. Within the neural network approach, feed forward neural networks will be examined. To combine methods existing in both approaches a better understanding of the classifier with particular attention to linear discriminant functions, needs to be explored along with the components and concepts of neural networks.

1.3 Classifier

Classification encompasses the theories necessary for making decisions concerning how to assign a given pattern with certain features to one of several possible classes. If d features are represented in d -dimensional Euclidian space E^d then the pattern can be represented as vector $X = (x_1, x_2, \dots, x_d)^t$. E^d will be called the pattern or feature space.

Consider c to be the number of classes. For each class there exists a set of points in E^d represented by the decision regions $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_c$. A pattern X is mapped into class i if its a member of $\mathcal{R}_i$. The decision surface consists of a set of continuous, single-valued, scalar functions (discriminant functions), with c members $g_1(x), g_2(x), \dots, g_c(x)$ such that for all patterns X in $\mathcal{R}_i$, $g_i(X) > g_j(X)$ for $i, j = 1, \dots, c$ and $j \neq i$. In $\mathcal{R}_i$, the i th discriminant function has the largest value. The decision surface between two regions $\mathcal{R}_i$ and $\mathcal{R}_j$ exists where

$$g_i(X) - g_j(X) = 0 \quad (1.1)$$

With this terminology defined, the classifier can be defined as a device that determines the decision surface. Applying the terminology and concepts to the classifier represented in Figure 1.2 produces Figure 1.4.

The selection of the discriminant functions depends on how much a priori knowledge is available. With complete a priori knowledge, the discriminant functions can be calculated precisely. Otherwise guesses need to be made based on what is known. The discriminant function often needs to be adjusted using the training set, a process known as training. When a classifier can be determined through training, the classifier is called a trainable

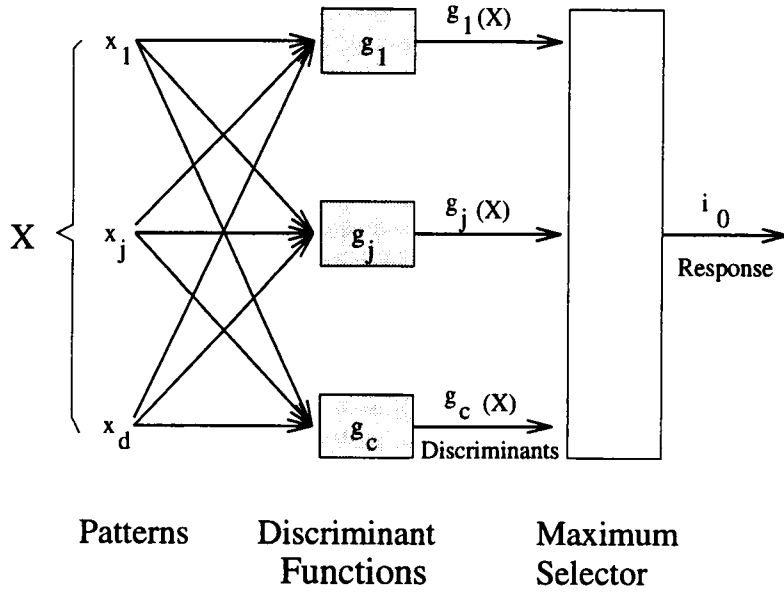


Figure 1.4: Basic classifier model [Nil65].

pattern classifier. This thesis will examine the case where a priori information does not exist and training is the major phase.

The neural network training approach is very similar to the error-correction-based pattern recognition techniques employed in developing linear discriminant functions. For this reason, the focus of this thesis in respect to the decision theoretic approach will be limited to methods that train linear discriminant functions using iterative decent procedures. A linear discriminant function has the form:

$$g_j(X) = w_{1j}x_1 + \dots + w_{dj}x_d + w_{d+1j} = \sum_{i=1}^d w_{ij}x_j + w_{d+1j} \quad (1.2)$$

Another way of writing this would be:

$$Y = \begin{pmatrix} X \\ 1 \end{pmatrix} \quad (1.3)$$

with the weight vector equal to:

$$W_j = \begin{pmatrix} w_{1j} \\ w_{2j} \\ \dots \\ w_{dj} \\ w_{d+1j} \end{pmatrix} \quad (1.4)$$

such that:

$$g_j(X) = W_j^t Y \quad (1.5)$$

The iterative decent procedure determines W_j by using the formula:

$$W_j^{n+1} = W_j^n - \alpha_n \cdot \frac{\Delta J(W_j^n)}{\Delta W_j^n} \quad (1.6)$$

with a stopping criteria of

$$\| W_j^{n+1} - W_j^n \| < \epsilon \quad (1.7)$$

or

$$n = n_{max} \quad (1.8)$$

where:

ϵ	error tolerance
n_{max}	maximum number of iterations
α_n	controls the adjustment at each iteration
$J(W_j^n)$	a measure of the classification error

A linear machine is a classifier that implements weights and summing in its linear discriminant functions. Figure 1.5 depicts a linear machine.

1.4 Neural Networks

The neural network approach offers the benefits of high computation rates provided by massive parallelism, and a high degree of robustness or fault tolerance. These networks are composed of units connected by directed, weighted links. What follows is a deeper description with accompanying terminology of units, links, combining layers, learning, and update modes. This section will give a basic understanding of neural network's terminology. Neural networks exist in many forms, and absolutes are generally avoided in defining terminology. In this thesis, feed forward neural networks are the only network architecture being considered. For a better understanding of neural networks, the reader is referred to [RM88].

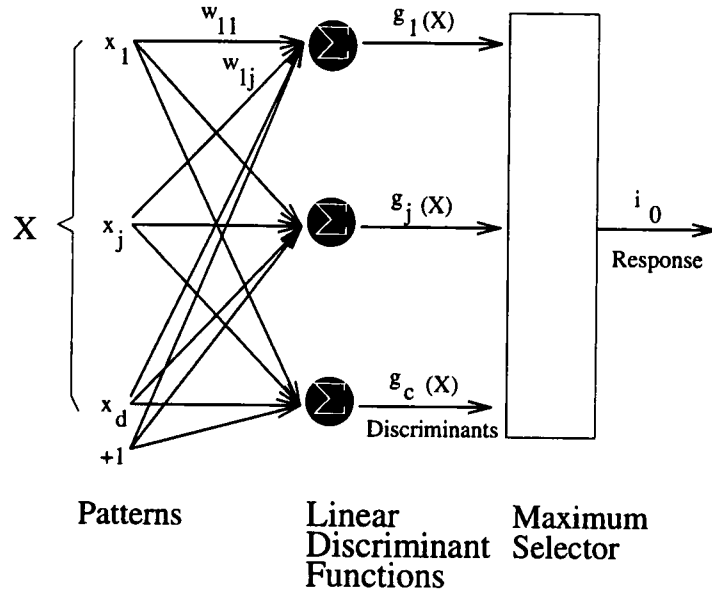


Figure 1.5: Basic Linear Machine.

1.4.1 Units

Neural networks are composed of units connected by directed, weighted links. The functionality and topological position of the unit determine which of the three types of unit it is. Input units have activation values equal to the input pattern and consist of only output connections. Often when discussing the number of layers in a network, the input layer is not counted since no calculations are performed at this layer. Output units have their outputs values equal to the network's outputs and lack output connections. Hidden units comprise the remaining units and have both input and output connections. Figure 1.6 displays a network with the different types of units.

The nodes often perform two functions: an activation and an output function. The output value of a unit (o_i) is multiplied by the weights connecting the unit to the next layer's units (w_{ij}). The units in the next layer sum these values from the units they are connected with ($\sum_{i=1}^m w_{ij} \cdot o_i$). This sum is used with the unit's current activation value and bias producing a new activation value. The bias of the unit can be represented in two ways. It can be an additional input feature with a value of one ($x_{d+1} = 1$) that is multiplied

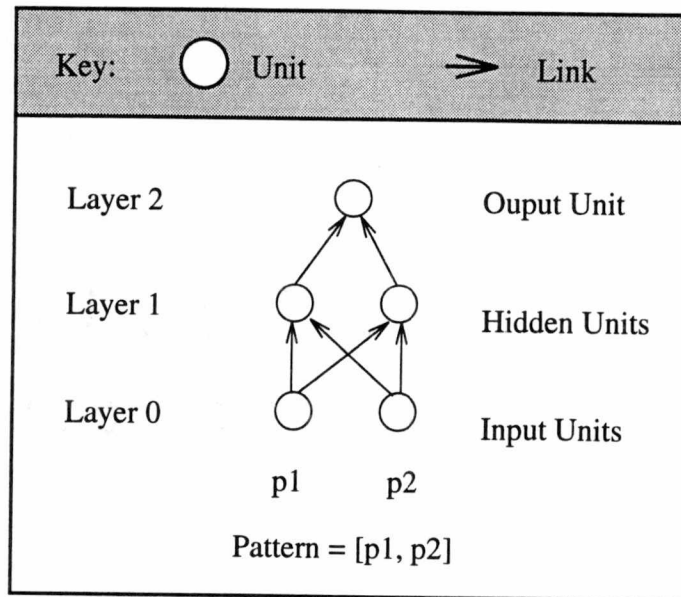


Figure 1.6: A fully connected feed forward three layered neural network.

by weights connecting it to all units (except input units). In this case, the bias is adjusted through adjustments to the weights. It can also be viewed as values stored at each unit. In this case, the bias value is adjusted at each unit. The general format of an activation function is:

$$a_i(t+1) = f_{act}(net_i(t), a(t), \theta_i) \quad (1.9)$$

where: $net_i(t)$ represents the net input in unit i at time t
 $a_i(t)$ activation of unit i at time t
 θ_i bias of unit i

The net input is computed with:

$$net_j(t) = \sum_{i=1}^d w_{ij} o_i(t) \quad (1.10)$$

where: $o_i(t)$ output of unit i at time t
 i index for unit i in the net
 j index for unit j in the net
 w_{ij} weight of the link from unit i to unit j

An example of an often used activation function would be:

$$a_j(t+1) = \frac{1}{1 + e^{(\theta_j - net_j(t))}} \quad (1.11)$$

Other activation functions are listed in Appendix C.

The output function computes the current output given the current activation, allowing processing of the activation value before an output occurs. The general format is:

$$o_j(t) = f_{out}(a_j(t)) \quad (1.12)$$

Often the identity function ($o_j(t) = a_j(t)$) is used for the output function.

The number of units must be enough to form a decision surface complex enough to classify the training set below a certain error rate. There are various ways of making an educated guess. This educated guess must consider the numbers of layers in the network and the type of pattern values in the training set. Lippmann [Lip87] states:

The number of nodes in the second layer must be greater than one when decision regions are disconnected or meshed and cannot be formed from one convex area. The number of second layer nodes required in the worst case is equal to the number of disconnected regions in the input distributions. The number of nodes in the first layer must typically be sufficient to provide three or more edges for each convex area generated for every second layer node. There should thus be more than three times as many nodes in the second as in the first layer.

1.4.2 Links

Units are connected by weighted directed links. The connections start at the source unit and end at the target unit with weights assigned to each connection. Recursive connections, where the source and target are identical, are possible but will not be considered in this thesis. Often a hierarchical bottom up approach is used where the inputs into units come only from units in the preceding layers. A network using such an architecture is called a feed network. A fully connected network is where every unit in one layer has connections to all units in the succeeding layer.

The weights assigned to each link affect the activation values, and thus the output, of all the connected units in the succeeding layer. The output value from a source unit is

multiplied by the weight of the link that connects it from the target unit. Negative weights will decrease the activity of the unit, acting as inhibitory weights, while positive weights will be excitatory.

1.4.3 Layers

Units can be connected into layers with multiple layers possible. Input is received through the input layer that functions only to pass the values of the pattern to the next layer via the links. The network outputs are generated from the output layers. All other layers are called hidden layers. Referring back to Figure 1.6, the layer organization is shown.

The single layer perceptron [Ros62] generated much interest when initially developed due to its ability to learn simple patterns. A single layer network works as a linear classifier, dividing the feature space into regions by using hyperplanes. Compare Figure 1.7 to Figure 1.5 that represents a linear classifier. These two figures demonstrate the similarity between a single layer network and a linear classifier. While it has been proven that convergence will occur for linearly separable classes [Ros62], no such convergence will occur for non-linearly separable classes. Minsky and Pappert proved this fact by examining the disjoint distributions for the two class exclusive OR problem [MP69].

Combining a typical single layer network's activation, output, and net functions defined by (1.11), the identity function, and (1.10), respectively, would produce a function that could be used as the discriminant function:

$$g_j(t+1) = \frac{1}{1 + e^{\left(\theta_j - \sum_i^d w_{ij}x_i(t)\right)}} \quad (1.13)$$

The single layer network will take on the same form as a classifier with no maximum selector if a single layer network is created where there are c classes and c output nodes. A pattern X_m belonging to class j has an expected output for $g_j(X_m) = 1$, while the expected output for $g_k(X_m) = 0$ for $j, k = 1 \dots c$ and $j \neq k$. This is depicted in Figure 1.7.

Multilayer perceptrons were developed to overcome many of the limitations of the single layer perceptrons. As is shown in Figure 1.8, the single layer perceptron network can

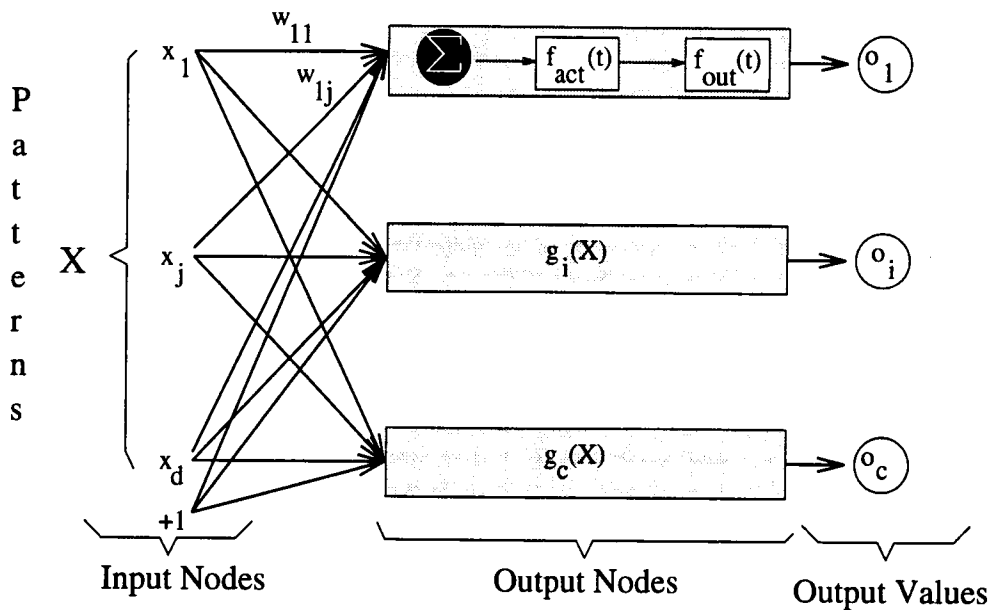


Figure 1.7: A single layer neural network.

form half-plane decision regions, a two layer perceptron network can form convex open or closed regions, and a three layer perceptron can form arbitrary complex regions. Kolmogorov proved a theorem that provided some indication to the capabilities of multilayered perceptrons and is described in [Lor76]. The theorem stated that a three layered perceptron network with $N(2N + 1)$ units using linear summation and nonlinear but continuously increasing functions of one variable, can compute any continuous function of N variables. The problem with multilayer perceptrons has been the lack of effective training algorithms [Lip87]. It was not until a way of back-propagating the error to the weights in multiple layered networks by replacing the simple perceptron rule with a differential function that interest was renewed in feed forward neural networks. An effective method for deciding how many layers and units should exist in a network has yet to be derived.

Even if a three layered network can be used to classify all the patterns, it is questionable whether this is desirable. There has been much debate concerning if higher layered networks. Viewing neural networks in terms of mathematical logic provides an argument for higher layers [RM88]. In this argument, thinking of a complex Boolean expression limited to representation in two or fewer hidden layers would be equivalent to representing such an

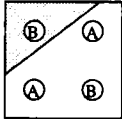
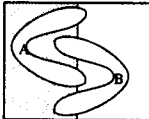
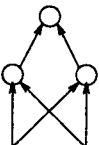
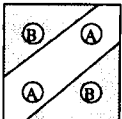
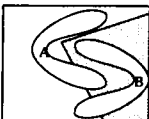
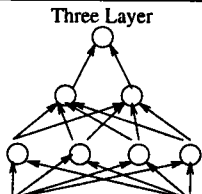
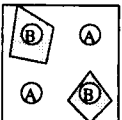
Structure	Types of Decision Regions	Exclusive Or Problem	Classes with Meshed Regions	Most General Region Shapes
Single Layer 	Half Plane Bounded By Hyperplane			
Two Layer 	Convex Open Or Closed Regions			
Three Layer 	Arbitrary (Complexity Limited By Number of Nodes)			

Figure 1.8: The different types of decision regions formed by single and multiple layered perceptrons [Lip87].

expression in conjunctive normal form. In viewing networks in this way, it can be seen that higher layers would require possibly fewer units and have a higher probability of using common features allowing it to generalize better

Such an argument suffers in real world applications. Sietsma and Dow [SD91] experimented with narrow networks with many layers and broad networks with few layers. They found that narrow networks suffer from being more difficult to train than broad networks. This is especially true when noise is introduced into the network. One reason for these results might be found in the decision surfaces created by multilayered networks. By using the nonlinear sigmoid function within a single layer, the decision regions are typically bounded by straight fuzzy edged line segments. Running the results of one sigmoid unit through another sigmoid unit, which is what occurs in multilayer networks, allows intricate curves to be created. Using highly layered networks allows the use of curved dividing hyper-surfaces to partition the space using truly curved boundaries. These highly multilayered networks

might produce curved boundaries that because of their complexity are difficult to adjust.

1.5 Discussion of the Title

The title of the paper "A Dynamically Allocating Piecewise Linear Neural Network" illustrates the functionality of the developed algorithm. Dynamically allocating describes the fact that the resulting network is organized by the algorithm. While the user may provide initial information to be incorporated in the development of the network, no information is required. Nodes are created as they are determined to be needed. The layers also are constructed by the algorithm. Piecewise linear describes the methodology for determining when the units need to be created. Attempts are made to divide a d -dimensional feature space into linearly separable subregions. A unit is created that act as a linear discriminant function, dividing a region into two subregions. Creation of units and the division of subregions continue until the feature space is divided within a certain error rate or a maximum number of iterations is reached. Neural networks are used to determine the weights for each of the discrimininant functions. In additional, an overseer network is created to determine a decision surface that separates the subregions.

Chapter 2

Background and Methodology

This chapter discusses related work in the piecewise linear classification and dynamically configuring neural network fields. Six approaches are discussed, each of which helped in the development of the DAPL algorithm. A brief discussion follows of work relating to pre and post processing of the training set and network respectively.

2.1 Piecewise Linear Classifiers

Piecewise linear classifiers are classifiers where the resulting decision surfaces separating two classes consist of sets of segments of hyperplanes [SW81]. A linear classifier is first applied to the training set. The sample space of a given class is then split into two subsample spaces if the error rate is too high. Figure 2.1 displays one way a piecewise linear classifier might operate. A piecewise linear classifier is generated and evaluated for each of the subclasses. The splitting and evaluation will continue until the performance reaches a certain level, no further improvement is made, or the maximum number of iterations is reached. Piecewise linear classifiers may have a poor generalization ability if they adapt to the particularities of the training set. This problem along with the problem of how to split a given (sub)sample space, generating linear discriminant functions, and when to terminate the splitting are important issues that must be addressed in any discussion of piecewise linear classification methods.

Two approaches exist for construction piecewise linear classifiers. The number of weight

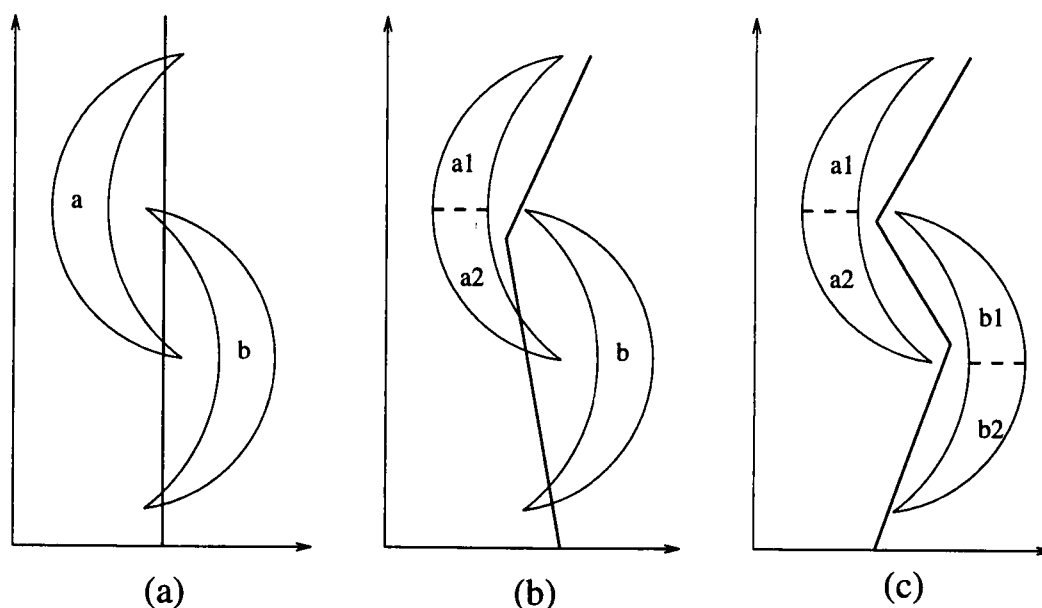


Figure 2.1: In part (a), a linear classifier is applied. Errors occur, so the sample space of class *a* is split and the resulting classifier is shown in part (b). Errors still occur, so the sample space of class *b* is split and the resulting classifier is shown. in part (c).

vectors can be fixed, that is the iterative method, or the number of weight vectors can be increased until correct classification of the training set are achieved [Pal90a]. In this thesis a piecewise linear classification method will be used in the constructing a self configuring neural network. Problems associated with determining the number of required units within a neural network have already been discussed. Only methodologies that allow the weight vectors to be increased until correct classification occurs will be considered.

2.1.1 Nilsson's Committee Machine

Nilsson [Nil65] described a piecewise linear classifier that used a committee machine for organization of the resulting hyperplanes. The committee machine is a network of perceptrons (linear discriminant functions) where each perceptron is called a committee member. Within the network, there can be any number of committee members each getting one vote. Figure 2.2 displays the structure of a committee machine with n members. In the

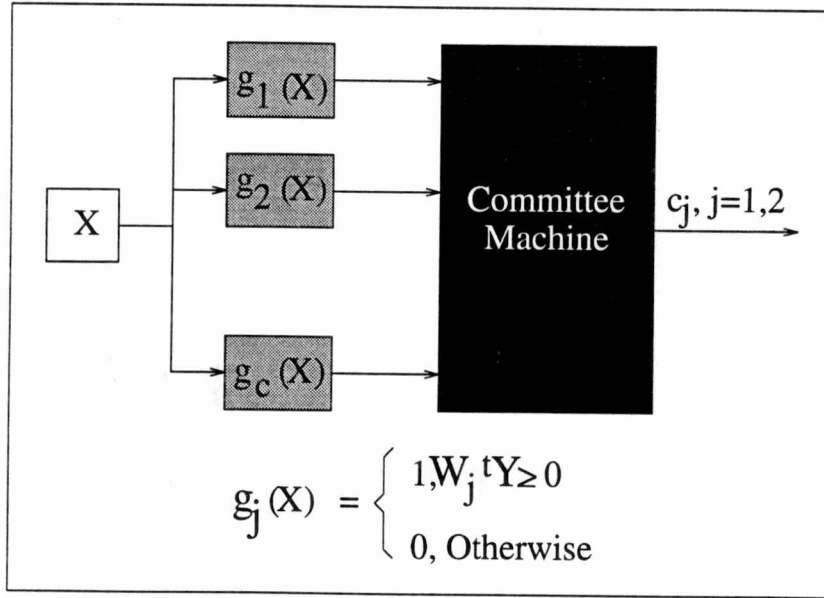


Figure 2.2: The committee machine for a two-class problem with y denoting the augmented feature vector [Pal90a].

figure,

$$g_j(X) = \begin{cases} 1 & \text{if } W_j^t Y \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.1)$$

where Y is defined by (1.3) and W is defined by (1.4).

The discriminant function $g_j(X)$ is acting as the perceptron function with $W_j^t Y \geq 0$. The vector W_j^t consists of connected weights to the perceptron j , that are being multiplied by the input pattern. Two types of committees discussed by Nilsson, because they are the best known (and easy to implement for training), are the majority and veto committees.

For the two class problem, using majority logic, a sample pattern X is assigned to class c_1 if

$$\sum_{j=1}^c g_j(X) \geq \frac{c+1}{2} \quad (2.2)$$

If more than half the committee members give a vote of one, than the network produces as output of one, otherwise a zero output is produced.

For the two class problem, using veto logic results in a sample X being assigned to class

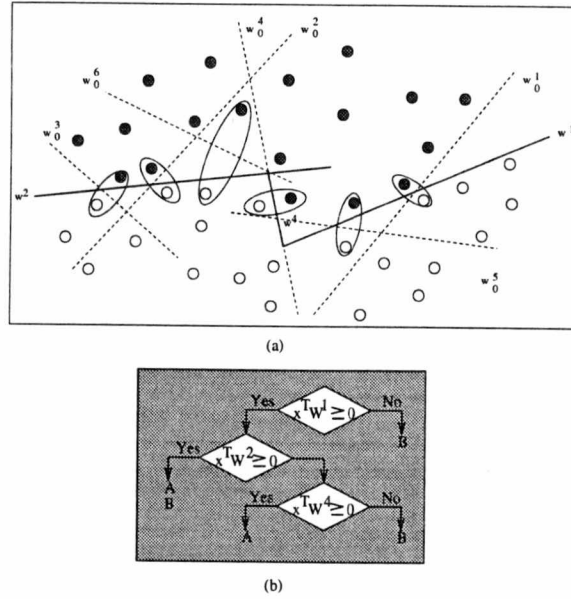


Figure 2.3: Part (a) demonstrates the division of the training space by three linear discriminant functions. In part (b), the resulting tree design is shown [MSK80].

c_1 if

$$\sum_{j=1}^c g_j(X) \geq c$$

If one member of the committee outputs a zero, the network will output a zero.

2.1.2 Mizoguchi et al.'s Classifier

With the Mizoguchi et al.'s [MSK80] piecewise linear classifier, linear discriminant functions are combined in a tree design. Figure 2.3 demonstrates Mizoguchi et al.'s method. Initially, a linear discriminant function is used to divided the feature space into two regions. If error occurs, indicating that the two classes were not linearly separable, each region containing samples from both classes is further divided into subregions. This continues until an acceptable error rate is reached. In the example in Figure 2.3, three linear discriminant functions are needed, generating three weight vectors.

An important concept to understand in viewing the Mizoguchi et al.'s method is that it is based on subdividing a two class set of patterns such that a maximum number of patterns are correctly classified. This approach handles the problem of solving for the set of linear

inequalities by using a gradient projection and the nearest neighbor method. Mizoguchi et al. make it a point to state how the initial weight vector must be chosen with great care. This points out a problem with the Mizoguchi et al.'s classifier, mainly deciding how to initialize the classifier's values.

2.1.3 Lee and Richards' Classifier

Lee and Richards [LR84] use the idea of a single sided decision hyperplane in separating patterns. The single sided decision hyperplane operates by dividing the subspace such that the maximum number of patterns is correctly classified while only one class exists on the positive side of the plane. Figure 2.4 shows this algorithm. Initially, a hyperplane is generated, represented in the Figure 2.4 by $\underline{a}_1$. The positive side of this plane faces class ω_1 . Note that on the positive side there are only patterns belonging to ω_1 . Errors occur, so training continues using the patterns located on the negative side. The patterns existing on the positive side are correctly classified. The hyperplane represented by $\underline{a}_2$ is generated with its positive side in the direction of ω_2 . Errors still exist, so the final hyperplane is generated represented by $\underline{a}_3$. On both sides of this hyperplane, patterns from only one class exist. Therefore, no errors occur and training stops.

The algorithm can be considered a committee machine that uses seniority logic, where seniority logic allows some members to hold more influence than other members. In Figure 2.4, the committee member $h_{i-1}(X)$ has more influence than $h_i(X)$. Only when $h_{i-1}(X)$ does not produce a value of 1, will further committee members be visited. The single sided decision hyperplane simplifies the mechanism required for finding the weight vector while avoiding the difficulties Mizoguchi et al.'s algorithm suffered in initialization.

2.2 Dynamically Configuring Neural Networks

Determining an efficient architecture for neural network can be a time-consuming task that involves deciding upon the number of layers, the number of hidden cells, the learning rule, the learning parameters, and the training set. To spare the user from having to make

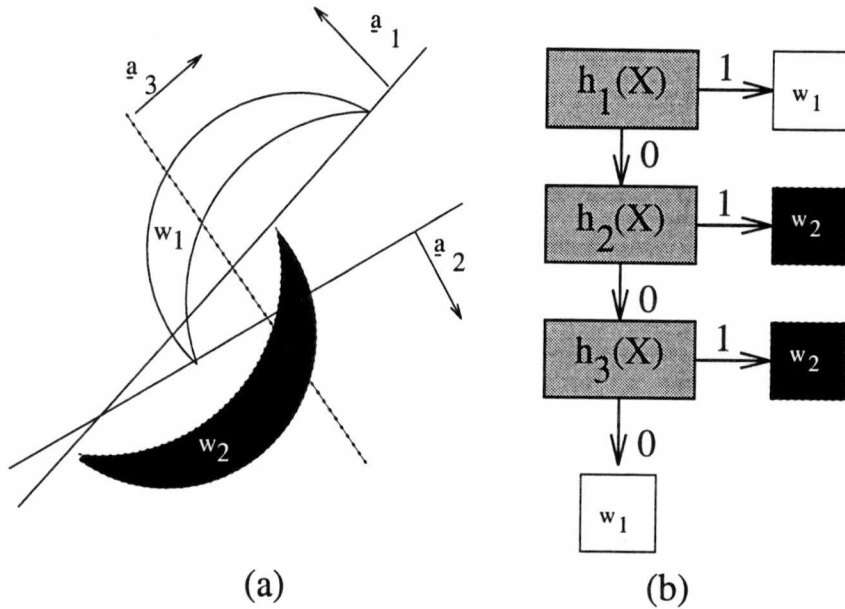


Figure 2.4: Demonstration of the Lee and Richards' Classifier. Part (a) demonstrates how the two banana problem might be divided by three single sided decision hyperplanes. Part (b) shows the resulting classifier, that oversees how the hyperplanes must be combined for correct classification.

all these decisions and to reduce the network complexity, dynamically configuring neural networks were developed [RH93]. Several approaches exist for deriving feasible network architectures such as the Romaniuk and Hall's Divide & Conquer network, an algorithm reported by Hirose et al., and Fahlman and Lebiere's Cascade Correlation learning architecture. These approaches are discussed in the following subsections.

2.2.1 Hirose et al.'s Network

Hirose et al. [HYH91] derived a back propagation network that varies the number of hidden units. All hidden units are added to a single hidden layer. Figure 2.5 shows a flow diagram of the program. It starts off like most back propagation algorithm, but it differs in that if a local minimum is hit a new hidden unit is added. A local minimum is determined to exist if no decrease in the error occurs by more than 1 percent after one hundred weight corrections. This continues until convergence occurs. Once convergence occurs, the algorithm attempts to decrease network complexity by removing nodes. It does

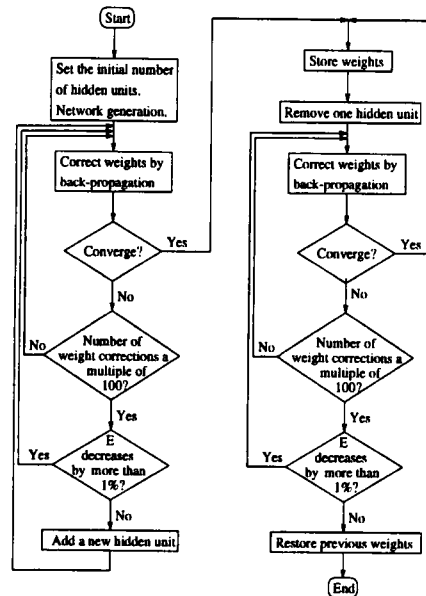


Figure 2.5: Flow diagram of the algorithm [HYH91].

this by removing the last node added and checking if convergence occurs. If it does, the network will continue removing nodes until convergence does not occur. The network that existed before the last node was removed is the final network.

Hirose et al. developed a self configuring network whose main purpose is to derive a way to escape local minimums. It does this successfully, allowing the user to start this algorithm in any initial configuration. The final network is the same as a network generated by a back propagation algorithm. It does not provide units that act as feature detectors or help in understanding the purpose of each unit.

2.2.2 Fahlman and Lebiere's Cascade Correlation Network

Fahlman and Lebiere's [FL90] Cascade Correlation (CC) method creates a network of a different design than the standard back propagation method. Errors do not need to be back propagated through multiple layers. It can be thought as a meta-algorithm since it allows various learning algorithms to be embedded in its network creation process.

The CC algorithm starts with a minimal network consisting of the input and output units fully connected. Figure 2.6 shows the development of a CC network. Training is done on this

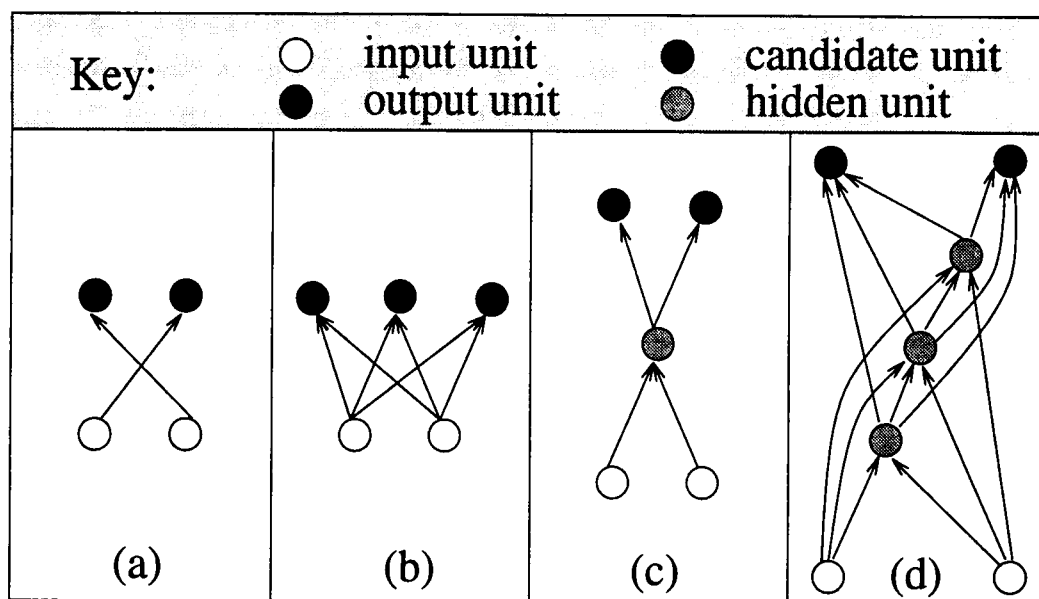


Figure 2.6: Demonstration of a developing Cascade Correlation network. In part (a), the initial network is trained and the error is determined. In part (b), the connection candidates are connected to the inputs and trained. One candidate is selected for the hidden unit and is attached to the output units in part (c). Part (d) shows a possible completed cascade correlation network.

initial network until the error fails to reduce further, or all patterns are trained correctly. If the former condition is true, candidate units are produced that are fully connect to all existing input and hidden units. The candidate units are trained in an attempt to maximize the correlation between its activation value and the residual error of the net. Training halts when correlation scores do not improve for any of the candidates. The candidate with the maximum correlation is chosen, its weights are frozen so not to change any further in future training, and it becomes the next hidden unit. All other candidate units are discarded. The hidden unit is than fully connected to the output units, and training begins on the whole net to determine the network error. The process of generating candidates and choosing a candidate to become the next hidden unit continues until the error rate falls below a minimal value or a maximum number of hidden units is generated.

Freezing the weights allows the generated hidden units to act as feature detectors in the network. The training set is never modified. Parallelization can be implemented in the

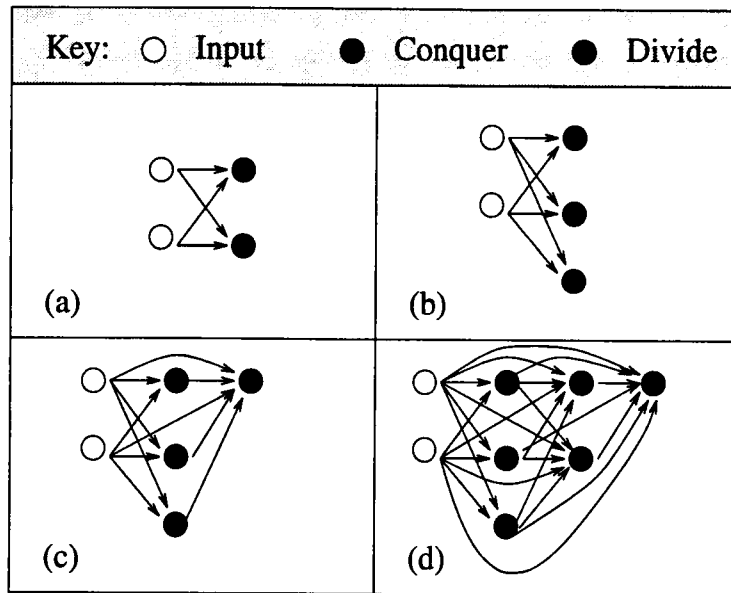


Figure 2.7: Demonstration of the creation of a Divide and Conquer network. In this figure, a two input and one output network needs to be created. Part (a) demonstrates the network after the conquer phase has completed and the divide phase is starting. In part (b), a new cell is added during the divide phase, demonstrating multiple nodes on a hidden unit. Part (c) is that start of a new conquer phase after further error reduction was not possible in the network from part (b). Part (d) is a demonstration of a possible final D&C network.

training of the candidate units, that can be arbitrary in number. The error is never back propagated so various learning functions, including the perceptron learning method, can be used within the network.

2.2.3 Romaniuk and Hall's Divide and Conquer Neural Network

Romaniuk and Hall's [RH93] Divide and Conquer (D&C) method (D&C) multiple nodes to be added to multiple layers. It is divided into two phases: the divide phase and the conquer phase. Figure 2.7 displays the creation of a D&C network.

A key concept in understanding the D&C algorithm is that it trains one output at a time. Subnetworks are developed for correct classification of each output. The training of the subnetworks can be done in parallel. There is also a sharing mode in an attempt to reduce the number of nodes, but parallel training is then lost. In the sharing mode, training on one output at a time is still done, but the next output will have as its input

units connected to all previous units in the network, including the previous output units.

The program begins in the conquer phase. A new cell is introduced into a new layer. This cell will be connected to all the cells existing on previous layers. Training is carried out using a training set consisting of all patterns for the desired single output. Training is completed by either complete classification of patterns, in which case learning is done for this output and the conquer unit becomes the output unit, or no further decrease in the error rate occurs in which case it enters the pre-divide phase.

In the pre-divide phase, a new unit is introduced on the same layer as the conquer cell and it is fully connected to all units from all the previous layers. If no correct classifications occurred in the conquer phase, a random sample is removed from the training set, the conquer phase unit is dropped, and training is done again. This process is repeated until at least one pattern is correctly classified, upon which a non-trained divide unit is created fully connected to all units on the previous layers (this is not necessary if the conquer phase correctly classified a pattern), and the divide phase is entered.

The correctly classified patterns are removed from the training set. The D&C algorithm uses the concept of neighbors, that are calculated before training begins, to help with adjusting the training set. Neighbors of the remaining incorrectly classified patterns are added to the set; one neighbor from each class, ensuring that close training examples, from different classes, are used during training. Training occurs only on the current divide unit's weights, while all other units' weights are frozen. If no patterns from the training set are correctly classified, a random pattern from the training set is removed. Training and removing of random patterns continue until at least one pattern is correctly classified. The process of removing correctly classified patterns and adding the remaining patterns' neighbors continue while the training set is nonempty and improvement in the error rate is occurring. The conquer phase is then re-entered.

2.3 Pre and Post Processing

Any preprocessing involves analysis of the training set in an attempt to adjust the training set in some way. Recall that a carefully chosen training set can provide a near optimal network. An example of preprocessing was given with the D&C algorithm's use of neighbors. The neighbors are determined before the algorithm begins and help produce an understanding of the relationships between patterns of different classes. Another example of preprocessing would be the Principle Components Analysis (PCA) [DP91]. It attempts to calculate the variation of the data point in each dimensionality of a pattern space using statistical techniques. Dimensions determined to have less variations can be dropped, thus reducing the dimensions and complexity of the training set. There is also the Canonical Discriminant Analysis (CDA) [DP91] method, that is another statistical technique. It attempts to compress high dimensional space into lower dimension by clustering the patterns and finding from such clusters the directions where points are closely clustered. Other non-statistical methods analyze the training set in an attempt to find and remove those patterns that will be counter productive to training (such as noise patterns). While preprocessing might make training simpler, it can also reduce generalization capability. Some algorithms will add noisy patterns in an attempt to help generalization [SD91]. No preprocessing is done in the DAPL algorithm.

Post-processing usually consists of pruning the network. In pruning, those units that are determined, through examination of the resulting network, not to be necessary are removed. Examination of the solution network consists of analyzing the outputs of the hidden units across all the training set inputs [SD91]. This can be done after training is completed or during training. In addition, pruning can improve generalization and help detect the important connections (other connections are removed). The remaining connections act as better feature detectors [RH93]. Two stages of pruning are of particular concern: removing units that do not contribute to the solution and removing units that give information not required at the next layer [SD91]. No post-processing is done in the DAPL algorithm.

2.4 Summary

In this chapter, several existing methodologies have been investigated. Nilsson's committee machine demonstrated one method of organizing the resulting hyperplanes using majority and veto logic into one decision surface. Mizoguchi et al.'s algorithm demonstrated another method for organizing the hyperplanes using a tree structure. Lee and Richards build upon Nilsson's committee machine and introduce the idea of a single sided decision surface for determining the separating hyperplanes. In respect to neural networks, Hirose et al. demonstrated a simple way of constructing a network by adding units when the network error cannot be further reduced. Only provisions for one hidden layer are specified in this algorithm. Fahlman and Lebiere's CC network introduced the idea of freezing weights and using the units as feature detectors. Each unit added is connected to all previous units. Romaniuk and Hall's D&C network also freezes weights and trains on one output at a time to help the units act as feature detectors. In the D&C method, multiple nodes and layers can be added. All six papers provide a rich source of approaches towards unit creation.

Chapter 3

The DAPL Methodology

Creation of the DAPL algorithm was motivated by a desire to develop a network that would allow a better understanding of how the decision surface was being derived, be capable of generalization, and be capable of dynamically allocating units and creating the network topology. The resulting network's timing and complexity were constantly taken into consideration. The DAPL algorithm creates a network that performs well in all of these areas. In this chapter, the methods applied in the development of the DAPL algorithm are discussed.

3.1 Introduction

Romaniuk and Hall's Divide & Conquer (D&C) and Fahlman and Lebiere's Cascade Correlation (CC) algorithms offer different ways to configure the network through the use of different network topologies and different handling of the training set. The training process (adjustment of the weights), however, remains the same in both algorithms. While different learning rules, activation functions, and output functions may be selected, both offer the same selection.

In the CC and the D&C algorithms, during training only the weights connecting units from the previous layers to the current unit being trained are modified. Other weights are frozen allowing units to act more like feature detectors. Even though both algorithms develop multilayered networks, with the weights being frozen training occurs only on single

layer subnetworks. Back propagation of error through multilayers is never required. This simplifies the algorithms and allows for more of a selection of learning rules. The only requirement is that the learning rules need to be capable of adapting to working in a single layer architecture.

Within single layer architectures, learning rules act as linear classifiers producing hyperplanes that the higher layers use to create the final decision surface. There are established methods in pattern recognition for defining a decision surface based on a set of hyperplanes created by linear discriminant functions. These methods are part of the piecewise linear classification approach.

The piecewise linear classifiers proposed by Nilsson, Mizoguchi et al., and Lee and Richards are methods that operate by increasing the number of weight vectors until correct classification of the training set is achieved. Lee and Richards' method uses Nilsson's committee machine with seniority logic. In implementing seniority logic, the concept of single sided decision hyperplanes was introduced.

When developing the DAPL algorithm, attention was paid to the use of linear classifiers (single layered networks) within the CC and D&C algorithms. Instead of organizing the developing hyperplanes in the same manor as the D&C and CC algorithm, a piecewise linear classifier method was used. The approach that the DAPL algorithm most closely patterned itself after was the approach of Lee and Richards with the use of single sided decision surfaces. Dynamic configuration of the network was managed by the method's capability to increase the number of weight vectors. Weight adjustments were accomplished through a neural network training method.

3.2 Basic Algorithm

The DAPL algorithm incorporates the training approach describe by Lee and Richards [LR84]. If a pattern is classified according to:

$$P_1 = \{Y \in P^i | W^t \cdot Y \geq 0\} \quad (3.1)$$

$$P_2 = \{Y \in P^i | W^t \cdot Y < 0\} \quad (3.2)$$

where P^i is the training set. P_1 consists of the patterns that are classified on the positive side of the single sided decision surface and P_2 consists of the patterns that are classified on the negative side. Training attempts to optimize the number of members of P_1 . To reduce the training set, the following formula is used:

$$P^{i+1} = P^i - \{Y \in P^i | W_i^t \cdot Y \geq 0\} = P^i - P_1 \quad (3.3)$$

where P^{i+1} is the training set with the correctly classified patterns of P_1 removed.

The basic training procedure for a two class problem is as follows:

Program Training

```

const maxw = maximum # of weight vectors for each committee
begin
  i = 0
  repeat
    i++
    initialize( $W_i$ ,  $P^i$ )
    train( $W_i$ ,  $P^i$ )
     $P^{i+1} = P^i - \{Y \in P^i | W_i^t \cdot Y \geq 0\}$ 
  until ( $P^i \subset$  class 1) or ( $P^{i+1} \subset$  class 2) or ( $i > \text{maxw}$ )
end

```

Lee and Richards' *initialize* procedure will be replaced with a procedure that will create a subnetwork consisting of a new unit fully connected to the inputs and assign random values as the initial connections' weights. The training procedure *train* will be a neural network training procedure that trains a subnetwork producing a single sided hyperplane. When training is completed the weights are frozen. Finally, a procedure will join all the subnetworks together and train an overseer network whose purpose is to join the hyperplanes of the previous layer and create a decision surface that will correctly classify the training set.

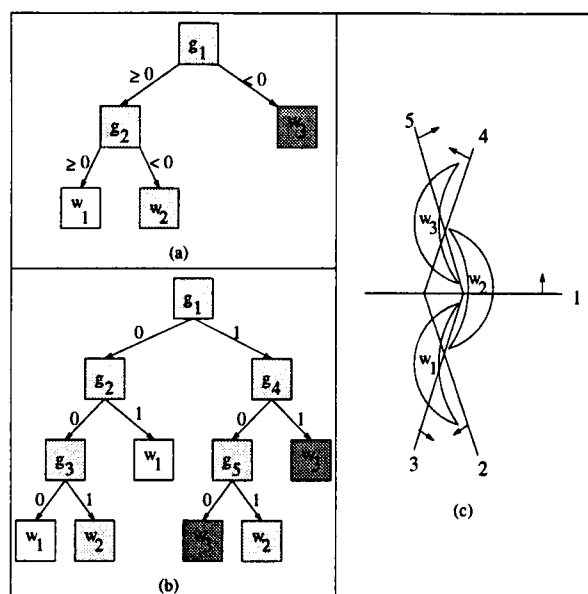


Figure 3.1: Illustration of how to use a two class classifier for a c class problem. In part (a), the Palm method is shown. In part (b), the “us/them” method is shown. The “us/them” method is the approach used by the DAPL algorithm. Part (c) shows a possible division of the problem space.

3.3 Division of Classes

Nilsson, Lee and Richards, and Mizoguchi et al.’s piecewise linear classification algorithms are all described and developed for a two class problem. Palm [Pal90b] describes a method by which c classes could be classified. In this method, several classes are merged together and classification occurs by classifying a pattern through different class groups until it is assigned to one of c classes. An example, where $c = 3$, is shown in Figure 3.1. Classes ω_1 and ω_2 are merged into one class group while class ω_3 is left alone. Classification of ω_2 will entail first classifying ω_2 into the class group, and then further classifying occurring until the ω_2 class is correctly classified.

To some extent, this approach is implemented in the DAPL algorithm. The grouping is done in more of a “us/them” approach, where “us” represents the class to classify and “them” represents all the remaining classes that need classification. With the use of the single sided decision surface, patterns on the positive side of the hyperplane belonging to

the “us” class can be classified directly into a class while patterns belonging to the “them” class will need further subdivision. Patterns on the negative side are a mixture of “us” and “them” patterns. Further division of the region is needed to determine class membership. The division of classes can be implemented in parallel. The single sided decision surface is continuously creating groups of “them” patterns on its positive side. Each of these groups could be given to another processor for further subdivision. An example of both methods is given in Figure 3.1.

3.4 The Overseer Network

Lee and Richards implemented single sided decision surfaces using a committee machine with seniority logic. With the “us/them” approach, the “them” group of classes is treated as if it was a single class. After it is separated or classified by the hyperplane, the region is then further divided. The process continues in this manner until correct classification occurs. Relating Figure 3.1 with Figure 2.2 demonstrates that the committee machine with seniority logic is no longer capable of handling the organization of the resulting hyperplanes.

The Mizoguchi et al.’s tree structure could be implemented on top of the subnetworks that generate the hyperplanes. The problem with adapting such a solution would be that a complete neural network would not be generated by the DAPL algorithm. The resulting network would consist of subnetworks that would send its results to a selector, and that selector would produce the class membership. While this is a valid classifier, it is not a valid neural network. In addition, a tree structure does not allow for further training. The addition of a network on top of the subnetworks would allow the network to train further, determining the weights between each hyperplane and output unit. Different hyperplanes will be more critical for correct classification of certain units. Adding weights to such hyperplanes help the network’s capability for generalization.

In the DAPL network, Nilsson’s committee machine was replaced with another overseer network. The complexity of the overseer network will depend on the complexity of the

decision surface and the minimum error rate. For development of the overseer network, many approaches were considered, including continuing to apply the piecewise linear classification methods to layer after layer until correct classification occurred. While this method of higher dimensional piecewise linear division was theoretically intriguing, in practice little was gained in predictability or understanding of the network's units while training suffered. The most efficient method was produced by combining ideas from Hirose et al., Romaniuk and Hall, and Fahlman and Lebiere.

The piecewise linear classification division of the network produces output that greatly simplifies training for the overseer network. The piecewise linear classification subnetworks are fully connected to the output units. Training occurs on the resulting network. When the network fails to train below a set minimum error, the simplicity of the Hirose et al.'s algorithm proved attractive. A unit is added to a new layer created between the subnetworks and the output layer. Connections are added from the subnetworks to the newly added hidden unit and from the hidden unit to the outputs. By Lippman's [Lip87] paper (Figure 1.8), the three layered network should be sufficient (i.e. no further layers will need to be added) to train the network for any problem. Rather than lose the training that has already occurred between the subnetworks and the output units, the connections between these two layers are left in place. The additional units and layer serve as higher dimensional feature detectors. Training and addition of units continue until the network is fully trained. Figure 3.2 demonstrates a possible creation of a DAPL network. The resulting network is similar in appearance to the Romaniuk and Hall's D&C network, except connection to the input layer only exist between the input units and the piecewise linear subnetworks.

3.5 Variations from Standard Feed Forward Training

The DAPL network can be separated into two major parts: the piecewise linear subnetworks and the network whose function is to oversee the joining of the subnetworks' hyperplanes to produce the minimum network error. The subnetworks and the overseer

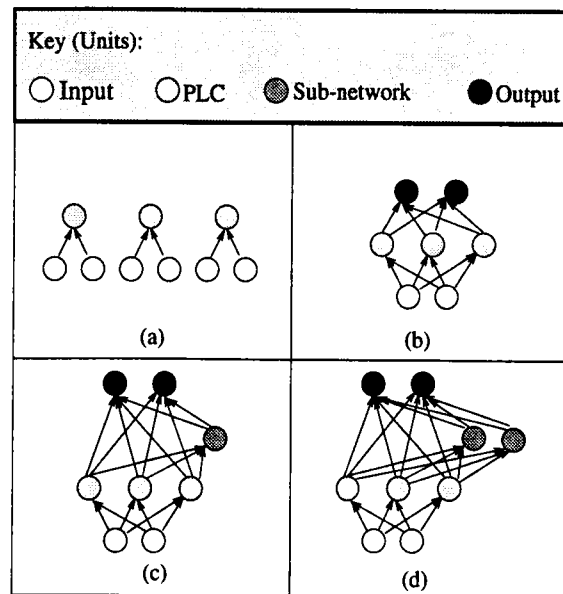


Figure 3.2: Illustration of how the DAPL network develops for a two input, two output network. In part (a), the subnetworks are created dividing the training space by hyperplanes. In part (b), the subnetworks are joined to the output and training occurs. In part (c) and (d), the network failed to drop below the minimum error value resulting in hidden units being added.

network have different requirements and are trained slightly differently. The subnetworks need to be trained quickly and there is no need to watch the subnetwork's development, so monitoring options are disabled. More interest exists in the overseer network's organization and how the organization is derived. For this reason, the network is saved at several spots as it trains.

The overseer network can be saved at different error rates. As the network's error drops below each specified error rate, the network configuration is saved. The user can later examine the network at various stages of its development. The concept of strikes is also implemented. Strikes allow for the network's progress to be monitored over a period of time. It is not unusual to train the network and after some number of cycles, check for improvement halting if there has been no improvement. A problem could develop when the error surface has local minimums. Sometimes, to escape local minimum, the network needs to increase the network error. While introduction of further units, as Hirose et al. suggest,

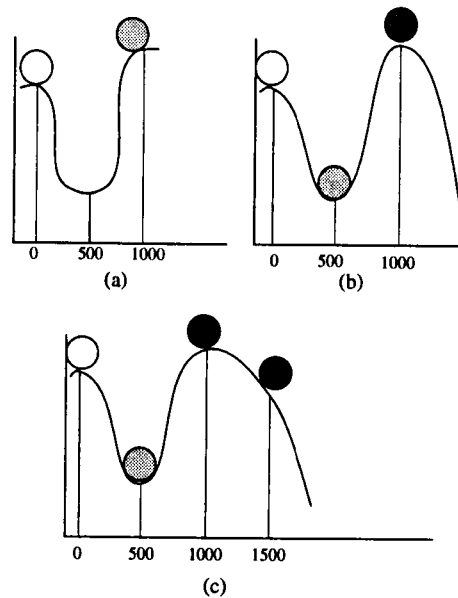


Figure 3.3: Demonstrates how local minimums can cause trouble and how the use of strikes can help avoid local minimum problems. In part (a), a large cycle value of 1000 is kept and a local minimum causes the network to halt execution. In part (b), the cycle number is reduced to 500 and strikes are used. Part (c) demonstrates the importance of keeping the old minimum error rate saved and not relying on the last error rate.

will solve the local minimum problem, such a solution is time consuming. For this reason, the use of strikes were introduced. With the use of strikes, the number of cycles is reduced. Training is done and if the network error fails to improve, the number of allowable strikes is decremented. Training continues until the minimum error is reached or all the strikes are used up. Figure 3.3 demonstrates the use of strikes with a decision surface containing local minimums.

3.6 Noise Handling and Generalization

The DAPL algorithm takes into account that noise will occur. Training for such noise will produce inaccurate decision surfaces that will be more complicated, take longer to train, and probably handle the test data set poorly due to over training on the training set. Generalization within the DAPL algorithm's first layer is implemented using two approaches: the "us/them" pool and the acceptable noise approaches. The overseer network also assists

in the network's generalization capabilities. For a more detailed discussion, refer to section 3.4.

In a c class problem, when the "us/them" approach is applied a hyperplane is generated separating the region into two subregions. The hyperplane's positive side consists of all of patterns from one class. When the positive side consists of the "them" class, which is a combination of real classes, further separation is necessary. When the positive side consists of patterns from the "us" class, classification on the positive side's subregion is completed. Further classification will need to be done on the negative side's subregion, unless it consists of patterns all of one class. The "us/them" pool approach attempts to provide the highest amount of correct classification possible before time runs out or a set number of hyperplanes is used. A pool is implemented that keeps regions containing patterns from multiple classes. A hyperplane is determined that divides a region into two subregions. The resulting subregions (unless the subregion consists of only one class) are added to the pool. The processor is continuously clearing its memory, fetching a region from the pool, determining a dividing hyperplane, and adding the resulting subregions to the pool. The pool distributes regions based on the number of patterns existing in each region. The regions with the greatest number of patterns are distributed first. In this way, regions are processed from the largest regions, where failure to classify correctly will result in the largest error rates for the network, to the smallest regions, where little error will result. Since the largest regions are handled first, if time runs out or the maximum number of hyperplanes is reached, the smallest regions (that produce the smallest errors) are what remains to be done.

The acceptable noise approach is founded on the idea that noise will constitute a certain percentage of the training set. This percentage represents the acceptable misclassification that may occur. It is easiest to visualize this approach with a two class problem. A single sided decision hyperplane is generated with misclassifications occurring on the positive side. If P_1 represents the set of patterns from the class that should be on the positive side of

the hyperplane with c_1 members and P_3 represents the set of patterns misclassified on the positive side of the hyperplane with c_3 members, then the hyperplane is accepted if

$$c_3/(c_1 + c_3) \geq \lambda \quad (3.4)$$

where λ is the acceptable noise percentage. By accepting the decision surface, P_3 will be misclassified while P_1 will be correctly classified. When more than two classes occur, this idea will still be true when classification patterns from the “us” class are meant to be on the positive side of the hyperplane. When members from the “them” class are meant to be classified on the positive side of the decision surface, these patterns represent patterns that have not been misclassified. They do not yet represent correctly classified patterns. Since the acceptable noise percentage represents the most misclassifications that may occur with the selection of the decision surface, often this percentage will not be reached for each hyperplane generated. For this reason, there seems little chance that over generalization will occur. The acceptable noise percentage is set by the user, and if fear of over generalization is high this variable can be set appropriately.

The acceptable noise approach can be further developed when pools of regions are implemented, as with the “us/them” pool approach. Before classification on a region occurs, the region is checked to determine how much difference exists in the region in respect to classes. If one class within the region makes up the majority of the patterns with c_1 members and the remaining classes have c_3 members, acceptance of the majority class without further processing will result in c_3 misclassifications. Acceptance can be based on definition (3.4).

The DAPL algorithm was developed on a non-parallel machine. There is overhead associated with the “us/them” pool approach that on a non-parallel machine cannot be countered by parallelizing. Whether the approach should be used depends on what is desired. Chapter 5 reports on the training differences between both approaches. Implementing pools allows features such as limiting the number of subnetworks, limiting the training time of the subnetworks, and how regions are selected for training. These options will often counter any additional training time associated with the use of pools.

3.7 Correction of the Single Sided Decision Surfaces

Single sided decision surfaces are a very important component of the DAPL network. Implementing this concept in neural networks or linear discriminant functions is accomplished through the process of initially guessing at the weights followed by iterative adjustment of the weights. Lee and Richards' method involves initial calculations to determine the hyperplane's location determined by the weights of the linear discriminant functions. The initial calculations are followed by adjustments to the weights in an attempt to maximize the number of patterns on the positive side of the hyperplane. As discussed previously, a characteristic of the neural network approach is that it is an adjustment approach. The DAPL algorithm will use neural networks to handle the initial guess and iterative adjustments of the weights.

In the DAPL approach, the network trains and determines its own decision hyperplanes. Extensive training is not done because the hyperplane's position is expected to change. The hyperplane initially generated minimizes network error. Figure 3.4 gives an example of this. The network error will differ from the single sided decision surface error. The single sided decision surface, is concerned with the number of patterns from P_2 on the positive side of the hyperplane (i.e. misclassifications). Once misclassified patterns (P_3) are determined, the DAPL algorithm attempts to adjust the training set so that the misclassified patterns are correctly classified (i.e. classified on the negative side).

While initial attempts were made to ensure correct classification through repeating the pattern in the training set, this method proved inadequate. The more times a pattern occurs in the training set, the more adjustment to the weights based on that pattern are made through one cycle of learning. Problems occur in two areas. First, there is a lack of control in the degree of adjustments to the error surface. Secondly, addition of patterns to the training set increases the training time.

What needed to be developed was a training algorithm whose goals were slightly different from most training methods. Instead of trying to traverse the error surface in an attempt to

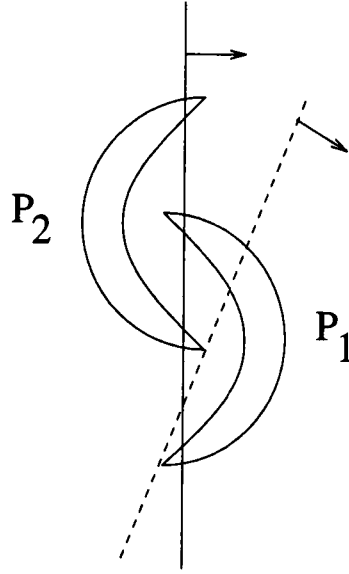


Figure 3.4: Demonstration of a single sided decision surface. The solid line is the initial guess, with misclassification occurring. Members from P_2 appear on the positive side of the hyperplane (i.e. misclassifications occur). Adjustments are made to the outputs, training is done again, and the process of adjustments and training continue until only patterns from P_1 appear on the hyperplane's positive side. The dotted line represents an adjusted single sided decision surface.

minimize error using some gradient decent method, the error surface needed to be adjusted to correctly classify P_3 . Once correct classification of P_3 was obtained, the network error could be minimized. The error surface is adjusted through modifying the misclassified patterns' expected outputs in the training sets. Through adjustment of the outputs, greater error can be created for the misclassification of a pattern. The neural network will than attempt to minimize this error producing the desired classification.

In this thesis, the generate-and-test method was used in adjusting the outputs [RK91]. This is equivalent to making an initial guess, seeing if the solution is obtained and if not adjusting the initial guess. This proved the simplest method to implement. The adjustment formula is defined as follows:

$$A_i = E_i \cdot (1.0 - \Omega_i) \cdot \left(1 - \frac{\sum_{j=1}^m E_j}{\sum_{k=1}^n E_k}\right) \quad (3.5)$$

$$1 \leq i \leq n \leq m \quad (3.6)$$

$$0 \leq \Omega_i \leq 1 \quad (3.7)$$

where:

- i the ith misclassified pattern
- Ω_i Adjusted output for pattern i
- A_i Adjustment to Ω_i
- E_i Error for pattern i
- m number of patterns classified with output of 1
- n number of patterns misclassified with output of 1

The adjustment factors are shown in Figure 3.5. The adjustment to Ω_i is based on E_i , whose value is reflected in the proximity to the hyperplane. $(1.0 - \Omega_i)$ is a scaling factor for E_i . $\sum_{j=1}^m E_j$ is the sum of the errors of all patterns classified on the positive side of the hyperplane. $\sum_{k=1}^n E_k$ is the sum of errors for the misclassified patterns. $\frac{\sum_{j=1}^m E_j}{\sum_{k=1}^n E_k}$ allow adjustments to be made based on ratio the misclassified patterns' errors are to the errors generated from the whole positive side of the decision surface.

Training is done, with further adjustments made until correct classification is made, as defined by the single sided decision surface. When this will occur with no patterns being classified in P_1 , the algorithm switches the class it attempts to classify on the negative side of the hyperplane. The whole process begins again. A single sided decision surface that classifies at least one pattern can always be generated, but not always for a particular class. For example, a class that is surrounded by another class could not appear on the positive side of the decision surface. If instead an attempt is made to classify members from the surrounding class on the positive side of the decision surface, this could be accomplished. The initial guess might make adjustments too large. If after attempting to classify both classes on the negative side of the decision surface, a solution is not found the process begins again with a smaller adjustment. This continues until a solution is found.

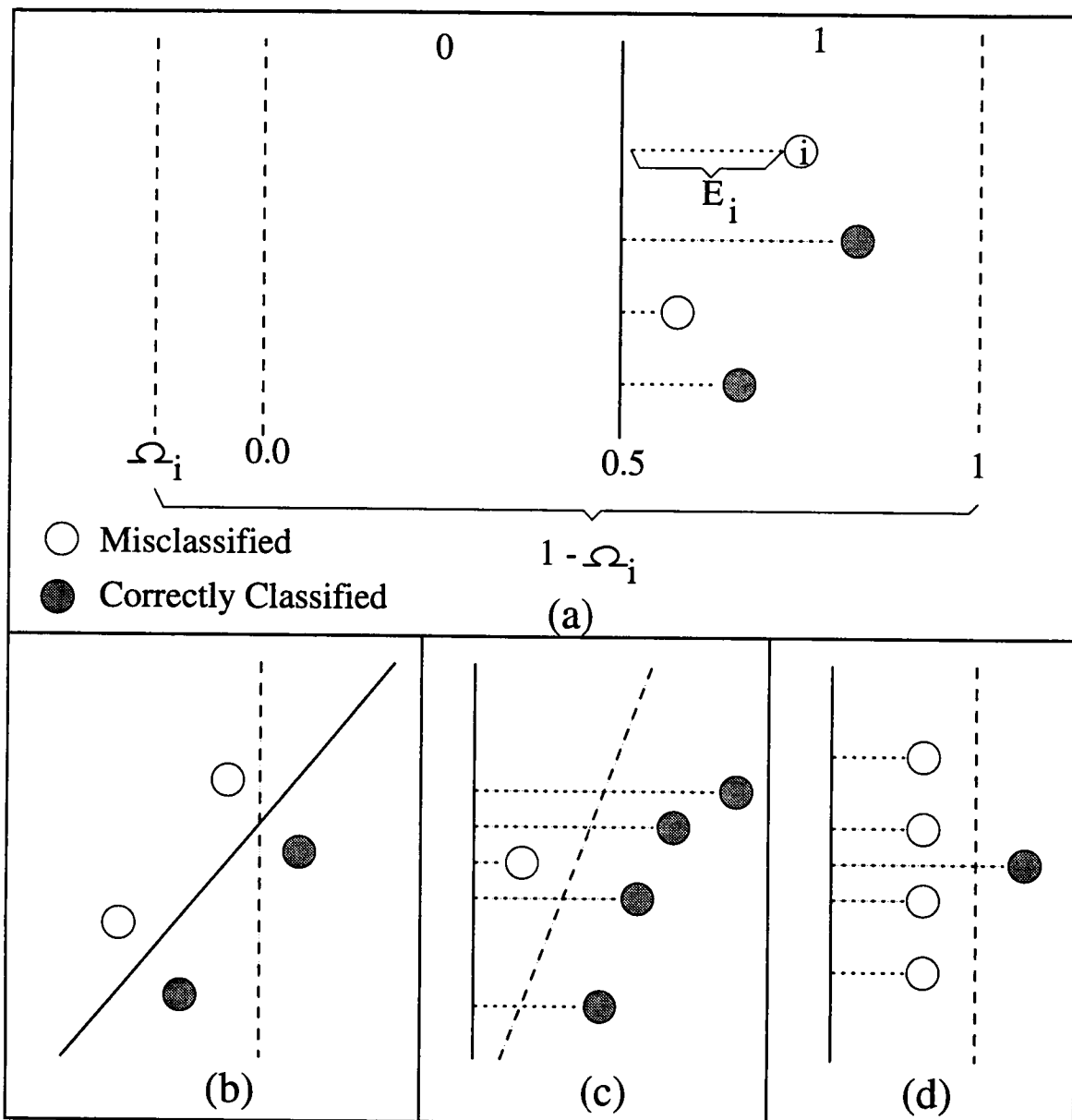


Figure 3.5: Displays the reasoning behind the adjustment formula. Part (a) displays the variables. Part (b) displays the desired modification of the hyperplane (solid line) versus a possible adjustment if no attention is paid to the misclassified patterns' proximity to the hyperplane. Part (c) displays the condition where $\sum_{j=1}^m E_j$ will greatly exceed $\sum_{k=1}^n E_k$. Part (d) displays the condition where $\sum_{j=1}^m E_j$ will just slightly exceed $\sum_{k=1}^n E_k$.

Chapter 4

Implementation of the DAPL Algorithm

The purpose of this chapter is to provide some of the details concerning implementation of the DAPL algorithm. The DAPL algorithm constitutes more than just an algorithm for the creation of the DAPL neural network. While the algorithm does create the DAPL network, it also allows for interactive creation and visualization of lower dimensional data sets. The code interacts heavily with existing tools, like SNNS and Matlab, to provide a rich working environment. What follows is a description of the system upon which the DAPL algorithm was developed, the tools it is integrated with, and a description at a high level of the procedures that create the DAPL network. For a brief description and overview of the complete DAPL algorithm the reader is referred to Appendix A for a description of executing the program, and Appendix B for the general code flow of the complete algorithm.

4.1 System and Tools

The DAPL algorithm was developed and tested on a Sparc 10 workstation. While the topic of parallelization was considered in the development of the program, there was no parallel capability on the workstation. The Sparc 10 operating system was SunOS 4.1.3u1 with the X11R5 windowing system. Within this environment, a rich source of tools is available such as the Stuttgart Neural Network Simulator (SNNS), Perl, and Matlab.

The Stuttgart Neural Network Simulator (SNNS) is a neural network simulator devel-

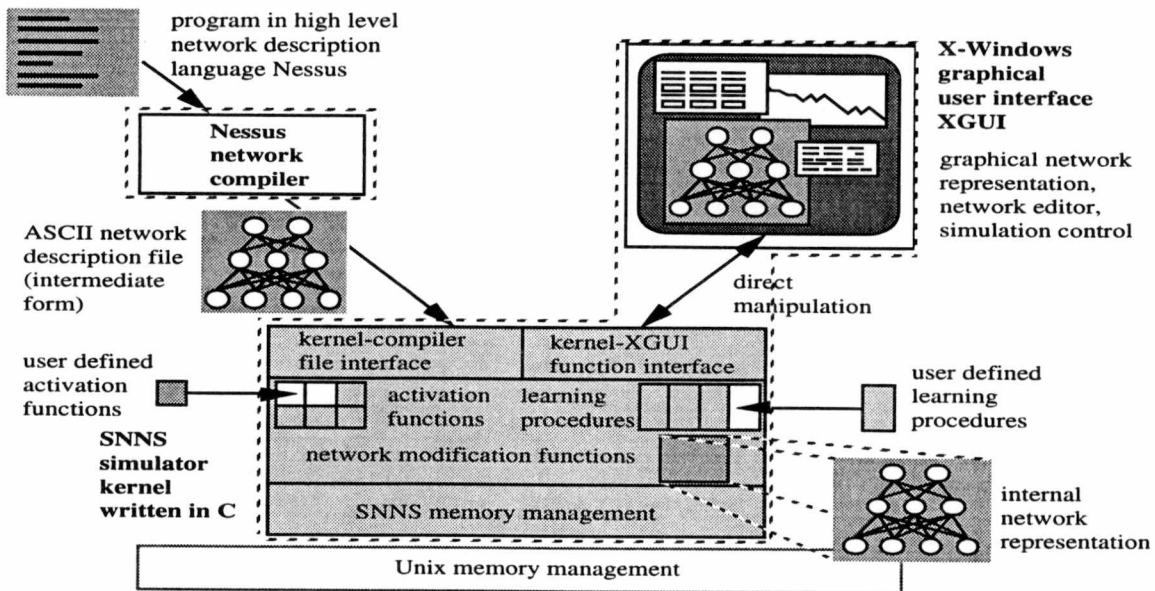


Figure 4.1: The components of the Stuttgart Neural Network Simulator [ZMV⁺94].

oped at the Institute for Parallel and Distributed High Performance Systems at the Universität Stuttgart. When work began on this thesis, many neural network simulators were investigated, such as the SESAME, GENESIS and XODUS, PlaNet, and the Rochester Connectionist Simulator. SNNS offered two modes of operations: a batch mode and an interactive mode with 3D graphics and a good blend of analyzing tools. In addition, the software was available free of charge.

Figure 4.1 shows the components of the SNNS. The two components of the SNNS consist of the kernel and the graphical user interface. Through the use of the Perl language, files are easily created describing the network. The network files are sent to the kernel for training and production of a result file. The result file is analyzed and appropriate action based on the training results is taken. These actions may consist of deciding to train the network further, adding a unit, completing work on the subnetworks and beginning work on the overseer network, etc. The graphical user interface provided users with the capability to examine the completed networks or the intermediate networks that were saved in the creation the DAPL network.

Matlab is an interactive software package that integrates numerical analysis, matrix computation, and graphics. While it is often thought of as an interactive software package, commands can be sent to Matlab for execution. The graphical capabilities are integrated into the DAPL algorithm, which sends commands to Matlab for displaying resulting network's decision surfaces when possible. The feature space is separated into uniform distributed points. These points are combined to form a test set that is sent to the `snnbat` program. The `snnbat` program sends the points through the DAPL network and creates a result file containing the actual output for each point. The actual outputs are used to indicate the classification certainty and six separate Matlab files are produced for visualization of the decision surface.

Perl is a language that is optimized for manipulating text, files, and processes. The C language cannot easily do many of the things a shell language can do, and visa versa. Perl is a language that bridges the gap. With the heavy interaction between SNNS, the DAPL algorithm, and Matlab, there was a strong need for easy manipulation of text files.

4.2 Code for Creating the DAPL Network

Figure 4.2 gives a flow diagram of the core procedures used in the creation of the DAPL network. Many more procedures exist within the DAPL algorithm, and the user is referred to Appendix B for a listing and organization of those procedures. The shaded boxes represent the procedures *classdiff* and *whichClass*. These procedures are implemented only if the "us/them" pool approach of generalization will be implemented. What follows is a discussion of each of the procedures of 4.2. At a high level, pseudo-code is given for the more complicated and important procedures.

Whether or not the "us/them" pool approach of generalization is implemented will affect the code, creating two approaches towards classification. Using the "us/them" approach, a hyperplane is used to split a region into two subregions. What happens to the two subregions after the split is where the methods differ. The "us/them" pool approach uses

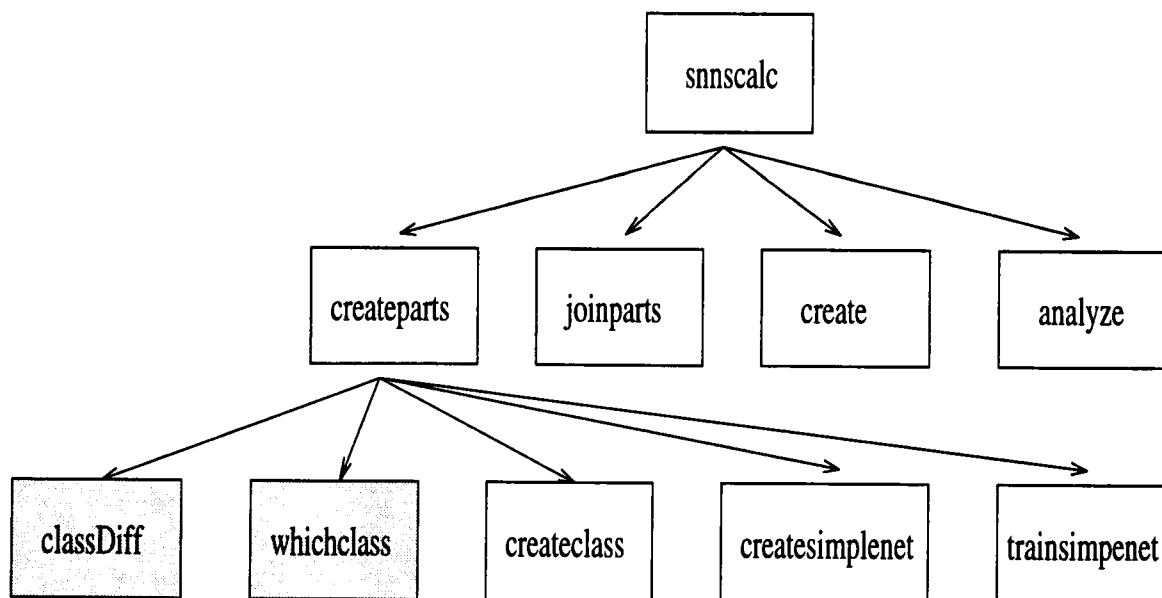


Figure 4.2: The flow of the major procedures used in the creation and training of the network.

pools. Both subregions are placed in the pool. The next regions split will be the region from the pool with the most number of patterns. In the non-pool “us/them” approach (the plain approach), classification continues on the “us” class until full classification of that class occurs. As the single sided decision surfaces separate the “us” class from the “them” class, the “them” subregions are placed into a ‘todo’ array. Once the “us” class is classified, each subregion within the ‘todo’ array is visited, one at time, where upon the “us/them” classification process begins again.

When describing the code, certain conventions were adapted for representing variable structures. A ‘\$’ indicates a scalar variable while a ‘@’ indicates a set or an array of some form. Certain variables are used throughout the code. These variables are listed below:

- @Waiters = pool of regions waiting to be processed
- @Members = patterns belonging to the “us” class
- @NonMembers = patterns belonging to the “them” class
- \$MemberPat = the desired output for the class represented in @Members
- \$WhoMust = who must be trained. Either ‘members’ or ‘nonmembers’
- @NonPatt = the desired output for each pattern in @NonMembers
- @ToBeDone = the patterns that end up on the positive side of the decision region

Some procedures, while playing a role in the DAPL network development, are either not very important or serve only one function. For these procedures, a full description is not warranted. The *analyze* procedure is used to create output files that can be viewed by the user using Matlab, providing a better understanding of the decision surface. A pattern file is created of uniformly distributed points over the feature space, along with a configuration file. These files are used to test how the current network would classify the distributed points and to what degree of certainty it makes these classifications. The results are used in displaying the decision surface. The less certain the network is, the closer the point lies toward the decision surface.

The *createclass* function reads into an array the patterns from the original pattern file. This array, consisting of patterns inputs and desired outputs, can be thought of as the full training region.

The procedure *whichclass* takes as its input a region. It separates the region into two classes (@Members and @NonMembers) in the spirit of the "us/them" approach discussed earlier. Procedure *classdiff* needs to be run before the region is passed to this procedure.

Another region handling procedure is the procedure *classdif* that takes a region and returns true if the points within the region belong to different classes. It takes into account the acceptable noise setting in deciding if a region does contain patterns from different classes. No further processing of regions consisting of patterns in the same class is necessary.

Finally, the procedure *createsimplenet* creates a fully connected feed forward network with inputs the same as specified in training set inputs and outputs equal to one. The weights, bias, and activation values are randomly set. The network is saved for later use by *trainsimplenet*.

The *snnscal* procedure is the procedure that controls the creation of the DAPL network. It calls the procedures that will create the subnetworks (*createparts*), join the subnetworks (*joinparts*), and then trains the whole network with calls to procedures to add

units to the overseer network (*create*) when it judges it to be necessary. Networks are saved throughout the process and files are created to allow the user the opportunity to view the development of the network (*analyze*). Multiple error rates are implemented so that the network can be save at different times in its development. When the current error rate fails to be met, improvement is looked for in a lower network error. The net error must decrease by a certain percentage of the last minimum error reached, requiring less improvement as the net gets closer to its minimum error. Strikes are implemented to avoid local minimum.

Procedure snnscale

\$numnets = number of subnetworks created for the first layer.
 \$strikes = number of chances the network configuration gets with no improvement.
 \$node_number = number of nodes currently in the second layer.
 \$network_done = boolean for telling when the network is finished training.
 \$config_done = boolean for telling when the particular configuration is done.
 @min_error = array of minimum error values
 \$min_spot = current minimum error
 \$Improvement = percentage of improvement necessary to count as real improvement
 \$stored_error = the stored error rate used in checking for improvement
 \$Allowed_layers = tells if overseer network is allowed a layer of its own

begin

```

    $numnets = call createparts
    call joinparts($numnets) procedure
    $network_done = false; $node_number = 0

    while not $network_done do
        $strikes = 3
        if $node_number > 0 then call create($node_number)
        $config_done = false; $min_spot = 0; $stored_error = a large value

        while not $config_done do
            create SNNS configuration specifying the network, pattern, log, result files
            call snnsbat(configuration file)
            call analyze
            if the network error  $\leq$  @min_error(min_spot) then
                save the network
                if $min_spot = last element of @min_error then
                    $config_done = true; $network_done = true
    
```

```

        else $min_spot++; $strike = 3;
    else
        if ($stored_error - network error) < ($Improvement * $stored_error) then
            $strikes = 3; $stored_error = network error; save network
        else
            $strike = $strike - 1
            if $strike < 0 then $config_done = true
        endif
    endif
enddo

    if $node_number > maximum nodes allowed then $network_done = true
    if $Allowed_layers = 0 then $network_done = true
enddo
end.

```

The joinparts procedure will combine the subnetworks into a single network. The input units remain unchanged. The output units from the subnetworks become a layer of hidden units, with the connections to the input units remaining unchanged. All weights, biases, and activation values are transferred to the new network and are frozen. The output units are added corresponding to the number of outputs specified in the original training set. Full connections between the hidden units and the outputs units are made with random values being assigned to the weights, activation, and bias values.

Procedure joinparts(numnets)

\$numnets = the number of subnetworks to join

begin

calculate the number of connections and number units

initiate activation, biases, and weights for the fully connected network

foreach of the subnetworks (\$numnets) do

extract the bias and activation for the subnetwork's output unit

store bias and activation into the new network

extract weight on the connections

store weights into the new networks

enddo

```
    save new network
end.
```

The createparts procedure is the driver of the subnetwork creation process. It determines when to create another subnetwork and when training is completed. This procedure is the main procedure affected by the “us/them” pool generalization approach. The beginning part of the procedure could be the same in the “us/them” pool and plain approaches.

Function createparts

```
@region = array of patterns from one region
$net_number = the number of subnetwork created
$Max_Breaks = the maximum number of separating hyperplanes allowed
$Bailout = boolean to tell when to start wrapping up
$found_region = boolean specifying whether a region to split has been found
$done_subnets = boolean indicating if done training subnetwork

begin
    $Bailout = false
    @region = call createclass
    call whichclass(@region)
    if either @Members or @NonMembers are empty then $done_subnets = true
    while $done_subnets = false
```

The “us/them” pool method would continue as follows:

```
    call createsimplenet($net_number)
    if (($Max_breaks - $net_number - 1) ≤
        number of regions in @Waiters) then $Bailout = true
    trainsimplenet($Bailout)
    if @Members and @NonMembers contain patterns then
        join @Members and @NonMembers into one region and add it to @Waiters
    if (@ToBeDone contains patterns) and
        (call classDiff(@ToBeDone) = true) then
        add @ToBeDone region to @Waiters
    if @Waiters has members then
        @ToBeDone = pop region with most number of patterns from @Waiters
        call whichclass(@ToBeDone); $done_subnets = false
    else
        $done_subnets = true
    endif
```

The plain method would continue as follows:

```
call createsimplenet($net_number)
trainsimplenet(0)
if @Members or @NonMembers contain no patterns then
    $done_subnets = true
    if (@ToBeDone contains patterns) and
        (call classDiff(@ToBeDone) = true) then
        $done_subnets = false
        @Members = all patterns of one class from @ToBeDone
        @NonMembers = the rest of the patterns in @ToBeDone
        @ToBeDone is emptied
    endif
endif
```

Both approaches will end the same.

```
    $net_number++
endwhile
return $net_number++
end.
```

The trainsimplenet procedure implements the single sided decision region. It is one of the more critical procedures to understand in the development of the DAPL network. It implements training method for the single sided decision surface. Unlike training done in *snnscal*, snap shots of the network are not saved during training. Training is on a three unit, two connection network, so training is accomplished relatively quickly.

The variable \$Bailout is only used when the “us/them” pool approach is being implemented. When this approach is not implemented, \$Bailout can be given a value of false. For the single sided decision surface, the key is to find the units that are misclassified and decrease their outputs in the training set. Correct pattern classification on the positive side of the hyperplane is critical, not minimizing the network error. Patterns are expected to be misclassified on the negative side of the hyperplane. Eliminating misclassification on the positive side of the hyperplane must be handled through readjustment of the training set followed by retraining. The number of training cycles is decreased with the idea that

retraining will occur again. If \$Bailout is true the procedure does not decrease the training cycle and no re-training on misclassified patterns is done. This has the effect of causing this procedure to acts as a linear classifier dividing the two regions to minimize error.

Procedure trainsimplenet(\$Bailout)

\$Bailout = tells if the network is wrapping things up
 @misclassified = patterns that were incorrectly classified on positive side
 \$repeat = the number of times to repeat the pattern
 \$times = the number of times through the loop
 \$Exactness = how close to correct classification the hyperplane needs to be

begin

\$not_done = true; \$times = 0
 while \$not_done = true do

The procedure needs to initially set up a training set where \$WhoMust indicates the class whose pattern must be all correctly classified. With two classes ω_1 and ω_2 , only ω_1 's members should occur on the positive side of the hyperplane. On the negative side of the hyperplane, ω_1 and ω_2 can occur. \$WhoMust represents the class that contains patterns that must all be correctly classified on the negative side of the hyperplane. Misclassifications will consist of members from the \$WhoMust class that are classified on the positive side of the hyperplane. Any misclassified patterns must have their outputs adjusted in the training set in order to have greater an affect on the weights (as discussed in Chapter 3).

if \$WhoMust = 'members' then (\$zero,\$one) = (1,0)
 else (\$zero,\$one) = (0,1)
 open the pattern file for output
 foreach pattern in @Members
 write to the pattern file the patterns input values
 and \$one as the desired output
 for each @NonMembers
 write to the pattern file the patterns input values
 and \$zero as the desired output
 if @misclassified is not empty then
 \$times++
 if \$Whomust = 'members' then
 \$tmpvariable = number of patterns in @NonMembers

```

else $tmpvariable = number of patterns in @Members
foreach pattern in @misclassified do
  $error = the patterns actual output verses the desired output
  if $error > 0.5 then
    $adjustment = ($error - 0.5) + (-1 * $prev_adjustments)
    $adjustment = $adjustment * $panicratio * -1
    $adjustment = $adjustment * (1 + int($repeat_thru / 2.0))
    $repeat = 1; $error = 1 - $error
    if $error < 0.269 then $repeat = $repeat + $tmpvariable
    else $repeat = $repeat + int($tmpvariable * log( $\frac{1}{\text{\$error}}$  - 1))
    $repeat = $repeat * $times
    for ($tmpvariable = 1; $tmpvariable ≤ $repeat; $tmpvariable++) do
      output misclassified pattern with desired output 0
    endif
  endif
endfor
endif

empty @misclassified
if ($Bailout = false) then $cycles = normal training cycles / 5
else $cycles = normal training cycles
create SNNS configuration file with $cycles cycles

```

Training occurs at this point. The training procedure is the same method discussed for the **snnscal** procedure in relation to the use of multiple improvement and strike features. If the variable `$Bailout = false`, lower training cycles will be used. The final difference is that the subnetwork is not saved at each stage of its development nor is the developing network tested to determine classification of uniformly distributed points in the testing space.

```

When training is done, the final result file is examined
for each pattern in the result file
  if the difference between the desired and actual is < 0.5 then
    push pattern onto @correct
  else push pattern onto @incorrect
endfor

```

`$WhoMust` represents classes that must be correctly classified. Patterns misclassified need to have their outputs adjusted in the training set. Patterns correctly classified belonging to the negative side of the hyperplane consist of a mixture of both classes. Further classification will need to be done on this region. The non-`$WhoMust` class patterns can be classified

on the either side of the hyperplane. The algorithm does attempt to maximize the number of members on the positive side. Patterns classified on the negative side of the hyperplane belong to a region that needs to be further divided. Patterns on the positive hyperplane are classified with only patterns from the same “us/them” class. If these patterns belong to the “us” class, no more division needs to be made. Patterns belonging to the “them” class, a class composed of multiple classes, may need to be further divided.

```

if $WhoMust = 'members' then
    $right = 0; $wrong = 0; @MustTrain is empty
    foreach pattern in @Members do
        if pattern  $\subset$  @correct then
            $right++; push pattern onto @MustTrain
        else $wrong++; push pattern onto @misclassified
        $zerosum = sum all the errors of the misclassified
    endfor
    if ((($right / ($right + $wrong) + 0.0001)  $\geq$  $Exactness) then
        $not_done = false; @Members = @MustTrain
        $right = 0; $wrong = 0; @MustTrain is empty
        foreach pattern in @NonMembers do
            if pattern  $\subset$  @correct then
                push pattern onto @ToBeDone
            else push pattern onto @MustTrain
        endfor
        if ((($right / ($right + $wrong) + 0.0001)  $\geq$  $Exactness) then
            @NonMembers = @MustTrain; $WhoMust = 'nonmembers'
            @misclassified is emptied
            if @NonMembers or @Members contain all the initial patterns then
                $not_done = true
                $onesum = sum all the errors of the patterns being
                    classified as 1
            endif
            $panicratio = ($onesum - $zerosum) / $zerosum
        endif
    endif
endif

```

The same is done with ‘nonmembers’ but with @Members substituted by @NonMembers. The only other difference lies in that if while cycling through @Members patterns, if a pattern $\subset$ @correct, the pattern is not put onto @ToBeDone. The pattern is correctly classified and forgotten.

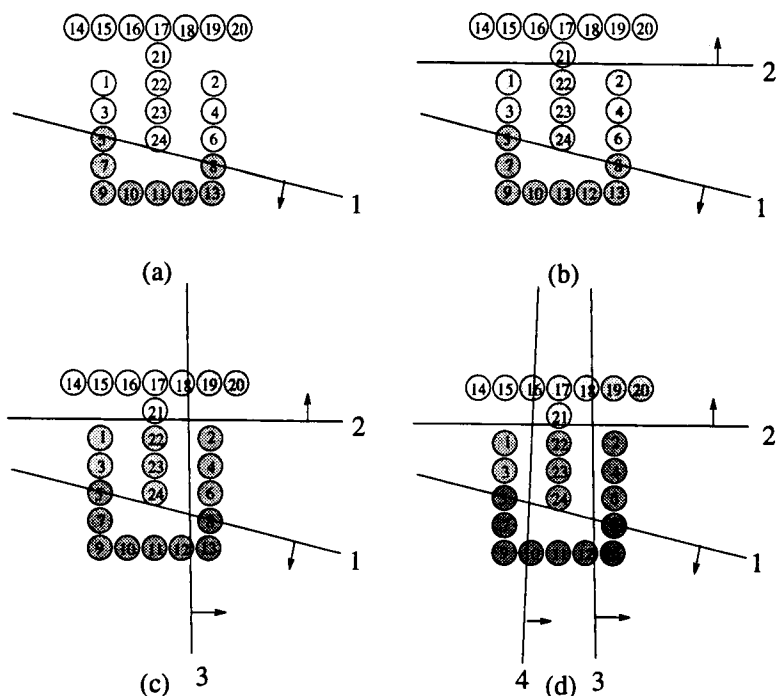


Figure 4.3: The division of the training set by the DAPL algorithm. The “us/them” classification can be seen as only members from the ‘T’ class are classified on the positive side in parts (b) and (d), while members of the ‘U’ class are classified in parts (a) and (c). Breaks and the \$Bailout variable were not used.

```

    if ($Bailout = true) then
        if (patterns exist in @misclassified) then $not_done = false
    enddo
end.

```

4.3 Trace of an Example

As a demonstration of the execution of the algorithm on a two class problem set, examine the classification problem depicted in Figure 4.3. For this trace, the “us/them” pool method will be traced. Within the region, class ω_1 will have patterns $p_1, \dots, p_{13}$ that constitutes the set P_1 . Within the region, class ω_2 will have patterns $p_{14}, \dots, p_{24}$ that constitutes the set P_2 .

The procedure *snnscal* is called, which calls procedure *createparts*. Procedure *createparts* calls *creatclass*, which reads the training set on file into one region that it passes

back. *Whichclass* is called, it selects the pattern set with the fewest members, P_2 , and sets up @Members = P_2 , @NonMembers = P_1 , and \$WhoMust = 'members'. P_2 becomes the pattern set that must be classified correctly.

A simple network of two inputs and one output is created by *createsimplenet*. *Trainsimplenet* is called. The training set is created with P_2 being assigned a desired output of 0 and P_1 has a desired output of 1. The network is trained. Initially patterns p_{23} and p_{24} are misclassified on the positive side of the hyperplane. The training set is readjusted and training occurs again. This time correct classification, based on the single sided decision region definition, occurs. The hyperplane depicted in Figure 4.3 part (a) splits the region.

@Members now has the patterns $p_{14}, \dots, p_{24}$. @NonMembers has $p_1, \dots, p_4$, and p_6 . @ToBeDone has $p_5, p_7, \dots, p_{13}$. @Members and @NonMembers are joined together as one region and added to @Waiters. *ClassDiff* is called on @ToBeDone and returns false, so nothing is added to @Waiters. @ToBeDone gets the only region that exists on @Waiters, the patterns left from P_1 : $p_1, \dots, p_4$, and p_6 , and the patterns of P_2 : $p_{14}, \dots, p_{24}$. *Whichclass* select the class with the fewest elements, P_1 . P_1 is now @Members, and @NonMembers is P_2 .

The process continues in this manor, creating three more subnetworks. *Createparts* halts when no more subregions exist in @Waiters. *Snnscal* calls *joinparts*, which joins the subnetworks together. *Create* never is called because resulting network is handled by the output units. Figure 4.4 depicts the results of the *analyze* procedure. This problem was run through the DAPL algorithm for \$MaxBreaks equal through 1, 2, 3, and 4 to create the four graphs. The variable \$Bailout was given a constant value of 0.

If the variable \$Bailout is allowed to be used, this affects how *trainsimplenet* trains the network. Because of the simplicity of the problem, @Waiters never grows beyond one member, which is removed and placed onto @ToBeDone at the end of *trainsimplenet*. *Createparts* checks to see if the program is running out of subnetworks, and if so sets \$Bailout to true. When \$Bailout is true in *trainsimplenet*, instead of doing a single sided decision

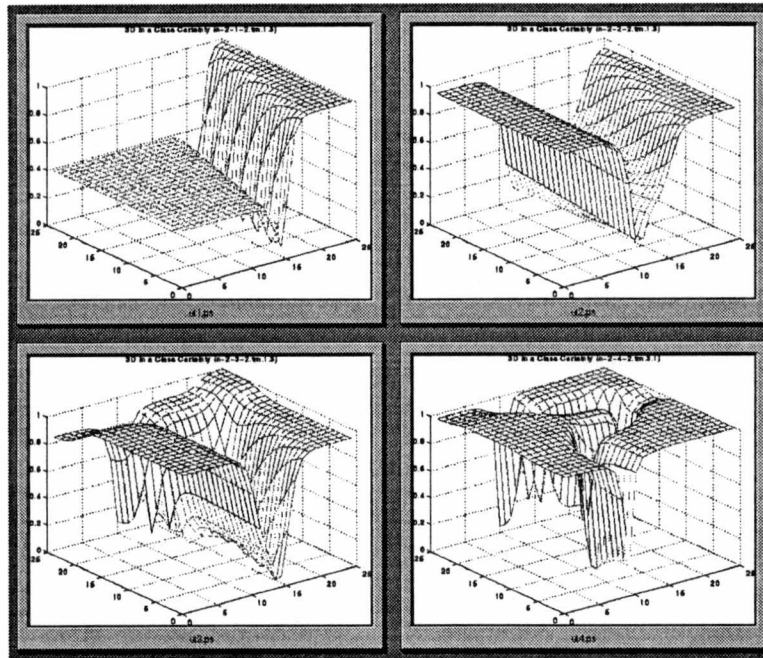


Figure 4.4: Depicts the certainty of the network in classifying uniformly distributed points in the sample space. Breaks and the \$Bailout variable were not used.

separation, a decision surface that minimizes the error for that region is generated. The effects are demonstrated in Figure 4.5 and 4.6.

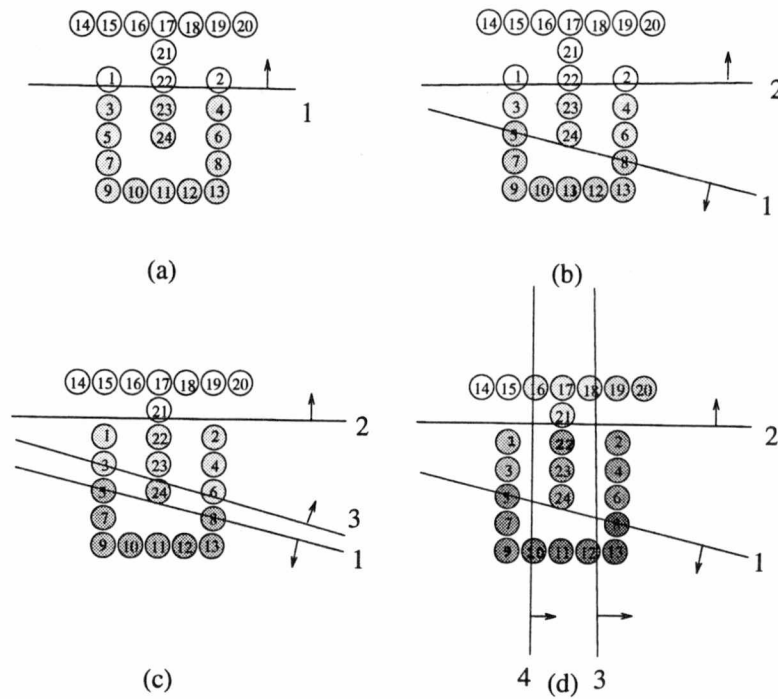


Figure 4.5: Depicts the certainty of the network in classifying uniformly distributed points in the sample space. Breaks were used limiting the number of lines to 1 in part (a), 2 in part (b), 3 in part (c), and 4 in part (d). The variable \$Bailout, causes an attempt to minimize error when the last decision surfaces are generated.

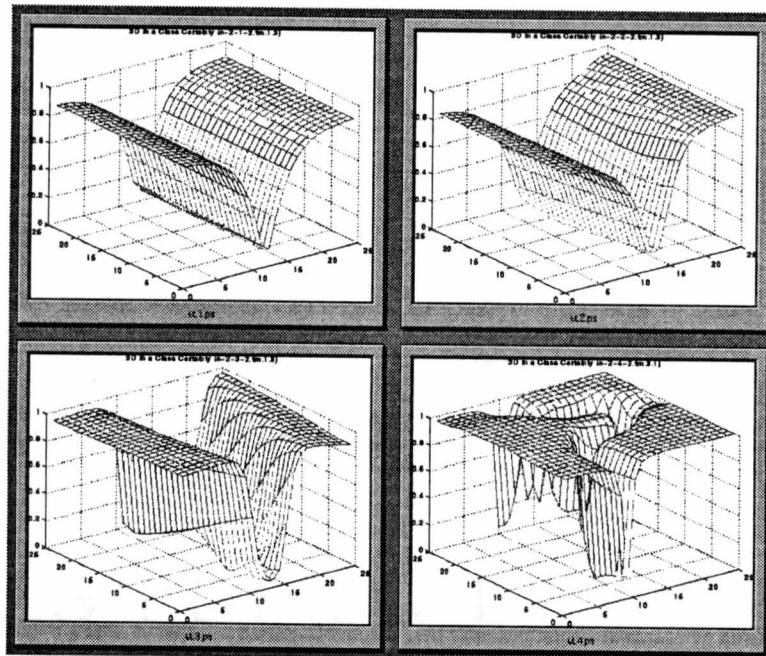


Figure 4.6: Depicts the certainty of the network in classifying uniformly distributed points in the sample space. Breaks were implemented with the variable \$Bailout allowed to used.

Chapter 5

Results

5.1 Introduction

One of the strengths of the DAPL algorithm, is its adaptability. It frees the user from the responsibility of configuring the network, while still providing the opportunity to make choices concerning the network's development. The user can specify such options as the various functions (learning, update, activation, and output function), the maximum linear separations, the training time, the acceptable noise level, if the network should have an overseer network and how many units this network is allowed. This chapter demonstrates the results of varying some of these network options along with comparisons to the more standard dynamically organizing network developed by Hirose et al.

5.1.1 Sample Data

Three easy to visualize problems were chosen to demonstrate and contrast how the DAPL algorithm performs differently with different options. For the first problem set, the classification problem introduced in the last chapter will be used. This is a simple two class problem where the classes form regions in the shape of a 'U' and a 'T'. Figure 5.1 displays this problem set. This problem set will be referred to as the "UT" problem. Often in piecewise linear classification papers, two facing banana regions are classified. To add a little variation on this concept, and to demonstrate the DAPL algorithm capability to

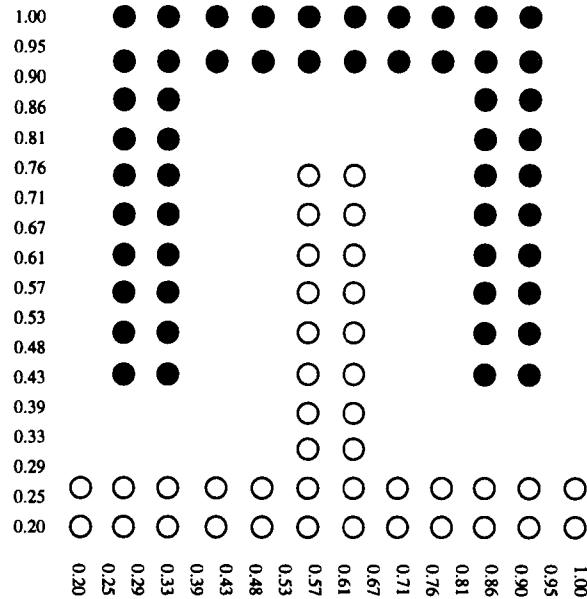


Figure 5.1: Plotted points for the two class U and T shaped region classification problem.

classify c class problems, where $c > 2$, a three banana region was selected for the second problem set. This problem set is demonstrated in Figure 5.2. The problem will be referred to as the “3 banana” problem. Finally to add more regions, that are adjoined to a high degree of other class regions, the classification set depicted in Figure 5.3 was selected. This problem will be referred to as the “8 class” problem.

The three problems vary in complexity while they still remain in lower dimensional space, helping with visualization representation.

5.1.2 Initial Setup

The DAPL algorithm has the SNNS program (more accurately *snnsbat*) heavily integrated into its operation. The tables report the learning time and the execution time of the program. The learning time means the time spent training the network as it is developed. Much of the algorithm’s time is spent getting at the results produced by *snnsbat* and building files to run on *snnsbat*. What is of interest is the actual time spent training the network, not exchanging information. Table 5.1 demonstrates the actual difference between the time to execute the program and the time it spends training. Initially, the

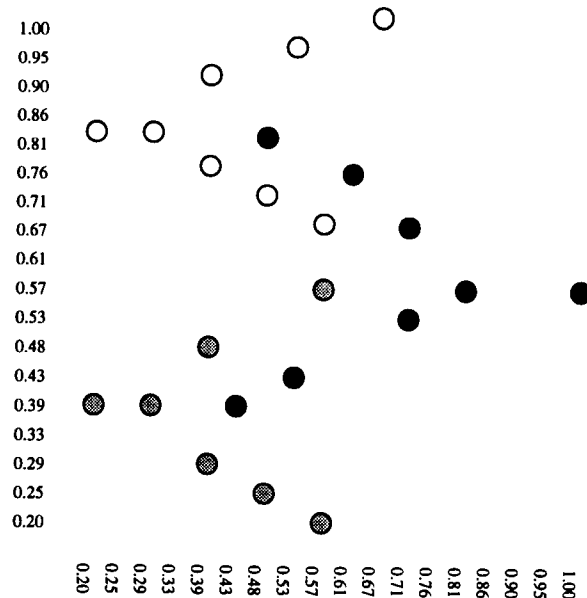


Figure 5.2: Plotted points for a three class banana shaped region classification problem.

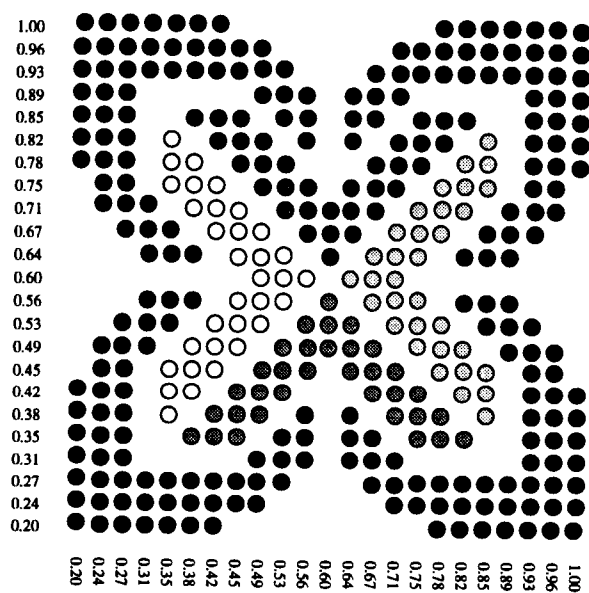


Figure 5.3: Plotted points for the eight class classification problem.

Table 5.1: Training time comparison the of the network total time verses training time.

	UT	3 Banana	8 Class
Training Time	35.16	84.68	7678.06
Total Time	52.3	133.5	7878.0

construction time takes up a high percentage of the training time. As the problems become more complex and the learning time increases, the construction time, as a percentage of the training time, decreases. When timing comparisons were done with the DAPL algorithm not implementing the “us/them” pool approach, there was no noticeable difference in time.

The networks were trained to a very low network error rate of 0.01. This was done to demonstrate the differences in the training time between the various options. The training time for the subnetworks is often very low. For example, in Table 5.1, the training time for the 8 class problem is 7676.06. The subnetworks’ training takes only a small part of that time (90.18). Often the network will have achieved one hundred percent correct classification in a fraction of the training time necessary to decrease the network error to 0.01.

For the task of deciding how correct a network classifies, two methods were investigated: the winner takes all (WTA) and the 40-20-40 method. The WTA method will classify a pattern as correct if its output matches closer than all other patterns the desired output. The 40-20-40 method will classify a pattern correct if its outputs are within 40 percent of what they need to be, incorrect if they are below 60 percent, and undefined otherwise. Both methods, because of the amount of training done on networks, will very often produce very similar results. The 40-20-40 method will be used in the tables with a mention to the WTA method when it produces substantially different results.

The number of output units was selected to allow one output unit to represent one and only one class, just like a classifier. The output representation, where the class’s output is equal to the class’s number in binary (e.g. ω_1 outputs 01, ω_2 outputs 10, ω_3 outputs 11, etc.), was considered and tested. The results indicated that the overseer network was

doing the additional task of translating from the output scheme of having one output unit representing each class, to this other output representation. Since this method increased the training time, the output representation of one output unit per class was selected.

To contrast the resulting training time of the DAPL algorithm, the Hirose et al.'s algorithm was implemented interfacing with the SNNS program.

5.2 Results

The results that are reported in following tables consist of values representing the time and the correctness of the resulting networks. The learning time refers to the time spent running the SNNS program. The SNNS program trains the network. The user time indicates the complete time, from the start the DAPL program until finish. One of the goals in creating the DAPL algorithm was to create an algorithm that could train a network in competitive time to existing dynamically allocating neural networks. A second goal was that the resulting network would produce a decision surface that was piecewise linearly separable.

The number of separations can be specified when running the DAPL algorithm with the "us/them" pool code in place. The DAPL algorithm compares the number of regions in its pool to the number of separations it has available. When the numbers are equal, the DAPL algorithm will stop separating the remaining regions using single sided decision surfaces. Instead, linear classification is attempted that minimizes network error within the region. This minimizes the network error over the whole training set when full classification can not be done. Table 5.2 demonstrates some timing results. With these tests, the overseer hidden layer was set to 0. The error and classification results will of course differ depending on the problem. Table 5.2 demonstrates the option and how timing, network, and classification results can be affected. For example, with 30 breaks classification with 90.53 percent correct might be acceptable verses the added time of full classification of a 2-33-2-8 network. The training time for a 2-33-2-8 network is slightly deceiving. The network achieved 100 percent

Table 5.2: Time vs. accuracy demonstration when limiting the linear line breaks for the 8 class problem.

Linear Break: Timing vs Accuracy						
Problem	Breaks	Correct	Inc.	Error	Learn	User
8 Class	10	42.06	11.98	180.5	934.6	143.33
8 Class	20	74.37	0	80.41	1023.7	1101.7
8 Class	30	90.53	0	34.42	1436.65	1533
8 Class	33-2	100	0	0.01	3426.21	3563.5

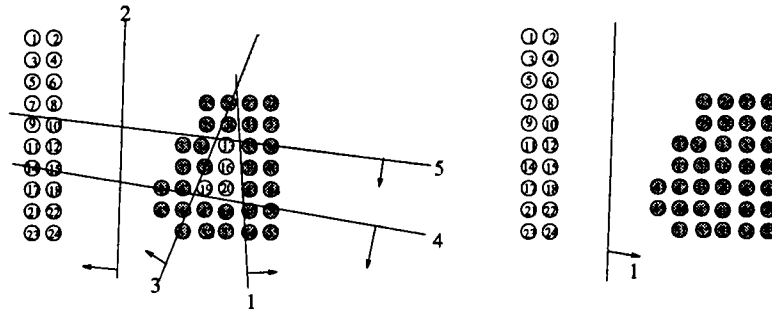


Figure 5.4: The figures depict two different exactness settings within the DAPL algorithm. If exactness is to be 100 percent, the figure on the left is the result. With the figure on the right, the exactness was lowered to 80 percent.

correct classification at learn time 2331.79. The remaining time was spent reducing the network error.

Another way of adjusting the generalization capability of the DAPL network is by specifying the exactness of the network. Figure 5.4 demonstrates the results of different exactness settings for the acceptable noise generalization approach. This test is highly conditional on the training set, and so no time comparisons were executed.

5.2.1 Comparison to Hirose et al.

The Hirose et al.'s algorithm was implemented interfacing with SNNS in the same manner the DAPL algorithm does. This was to allow a more accurate comparison between the two systems. Table 5.3 demonstrates the results. The algorithm was started with no initial configuration information to base its network topology on, so it began as a two input, one output network. The training time listed in the tables includes the algorithm's backtracking

Table 5.3: Performance of the DAPL network verse Hirose et al.

DAPL vs. Hirose et al.'s Networks										
Function	DAPL					Hirose et al.				
	Corr.	Inc.	Error	Learn	User	Corr.	Inc.	Error	Learn	User
UT	100	0	0.01	35.12	51.4	100	0	0.01	87.65	119.3
3 Banana	100	0	0.01	69.24	100.2	100	0	0.01	281.63	342.8

time where it deletes nodes to ensure best configuration. Discounting the time spent backtracking, the learn time is 65.28 for the UT problem, and 229.59 for the banana problem.

5.2.2 Overseer Network

The overseer network can contribute highly to the overall training time of the network's development. Table 5.4 demonstrates the difference in training time between the subnetworks and the overseer network. The first value in the table represents the first training attempt of the overseer network made on the subnetworks. An efficient overseer network is necessary for low training times. An experiment was done to compare the overseer network based on the Hirose et al. method, and a network closer to resembling the Cascade Correlation network and the Divide and Conquer network. To test if the outputs from previously introduced units in the overseer network could help speed up classification, an overseer network similar in construction to the cascade correlation network was developed. It is not trained in the same manor as the cascade correlation algorithm, but every newly introduced unit on the overseer layer is connected to previously introduced units along with the subnetwork units. The results seem to indicate that the overseer network performs the same with in either configuration.

5.2.3 Decision Surfaces

Figure 5.5 demonstrates the resulting decision surfaces for the three classification problems discussed in this chapter. These decision surfaces were generated using the DAPL network with the Hirose et al.'s overseer network.

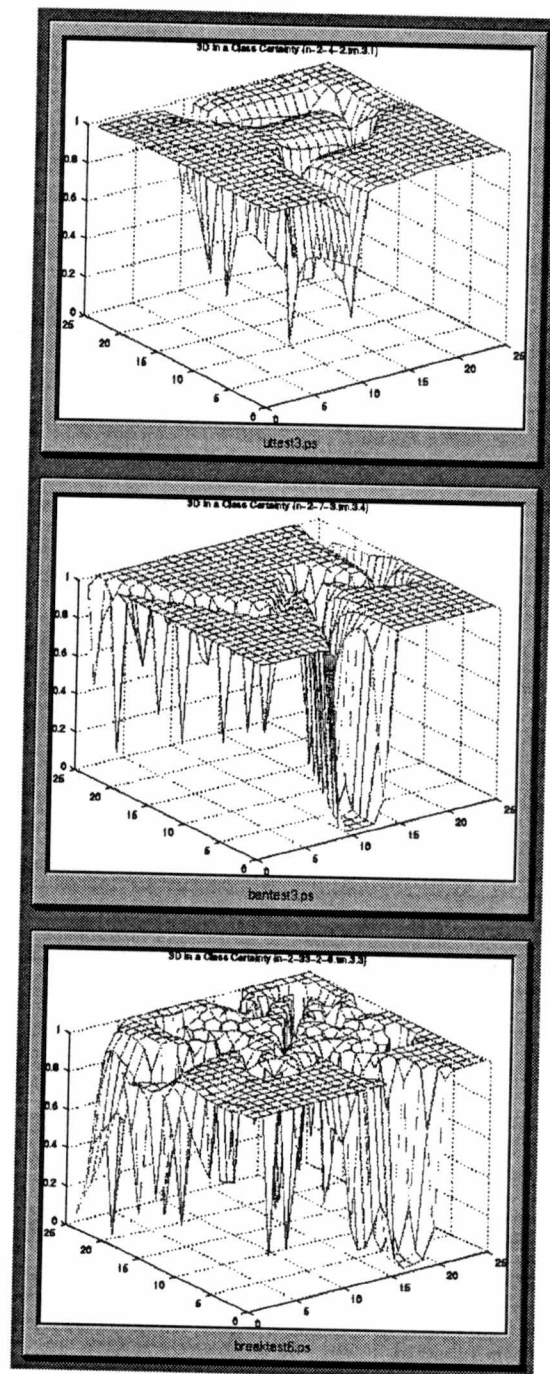


Figure 5.5: The decision surfaces produced for the UT, 3 Banana, and 8 Class classification problem.

Table 5.4: A comparison of the subnetwork and overseer network training time.

Subnetworks vs. Overseer Network Training Time			
Training Time			
	After 1st Training	100% Correct Class	Error Rate < 0.01
UT	22.42	22.42	35.12
3 Banana	49.69	53.31	69.24
8 Class	1098.67	2317.66	3426.21

The DAPL algorithm has been demonstrated to perform comparably with such dynamically configuring neural networks as Hirose et al.'s algorithm. This result is most encouraging. The training time seems to be mostly spent on the overseer network. Experiments with minor variations to the overseer network produced little difference in the results or the training time.

Chapter 6

Summary and Conclusions

6.1 Summary

An approach has been presented for applying the methods of piecewise linear classification to neural networks. It has been shown how single layered networks can be viewed as linear classifiers. Training procedures that create networks on multiple layers have been demonstrated to be linear classifiers if their training procedures consist of training only one layer at a time. Networks that attempt to create units that act as feature detectors often behave in this manner by freezing their weights.

The motivation for combining neural networks and piecewise linear classifiers was to gain the benefits from both fields. Linear classifiers are quick and easy to implement. Piecewise linear classifiers have been worked with and developed for many years, while dynamically allocating neural networks are relatively new. Neural networks offer the benefits of parallelism, robustness, and fault tolerance. Piecewise linear systems offer the benefits of fairly predictable decision surfaces. Learning procedures for piecewise linear classifiers could be used to configure and train the network removing the responsibility concerning deciding the network's topology from the user.

Before these benefits could be realized, the problems of how to split a given (sub)space, generating linear discriminant functions, and when to terminate the splitting of the (sub)space had to be solved. Single sided decision surfaces were implemented using neural networks for the division of the (sub)space. The linear discriminant functions were gen-

erated through the use of the vanilla back propagation method for adjusting the weights. The training procedure developed by Lee and Richards was applied for determining when to terminate the splitting of the (sub)space.

Parallelization of the DAPL network is outside the scope this thesis, although it would prove an interesting project. The generalization capability of the network was maintained through the use of pools for collecting the regions yet to be classified. Pools allowed regions with a greater number of patterns to be classified before smaller regions. If time runs out, or the number of decision surfaces reaches the maximum number allowed, having the larger regions classified first should result in a greater number of patterns being correctly classified. An exactness value could be specified to help the network to ignore noise. The overseer network that was developed assists the generalization of the network further by assigning weights to the subnetworks' inputs.

The resulting decision region is fairly predictable with the units acting like feature detectors. The training space is separated into subregions with the output of the subnetworks specifying which subregion each pattern belongs to. The weights between the subnetworks and the output units can also be understood. Weights connecting hyperplanes that are instrumental in separating the training set for an output which represents a class, will have large absolute values. Weights in the other hyperplanes will be small.

The DAPL algorithm was compared to the Hirose et al.'s algorithm with favorable results in both time and classification accuracy. The DAPL algorithm offers many options in the construction of the DAPL networks. The timing and correctness of these options were explored.

6.2 Continuation of Work

One of the problems encountered during classification was the generation of redundant single sided decision surfaces. This is not a problem particular to the DAPL algorithm. Lee and Richards reported the same problem when working with single sided decision sur-

faces [LR84]. This is compounded by the adaptation of the algorithm to work with the c-class problem set. Each region operates with no knowledge about the hyperplanes generated from the other regions. While attempts were made to minimize these redundant occurrences, further development of techniques in a post processing of the network might produce interesting results. Pruning could be used to reduce the number of units in the network, and in so doing decrease the complexity of the network. It should be noted, however, that these redundant decision surfaces can be beneficial in terms of the network capability to generalize. The first instance of a decision surface often will help generalize while the latter occurrences of decision surfaces become more specific. Initial test results in removing these redundant decision surfaces have resulted in networks that have a more difficult time classifying. Redundancy can also be considered robustness. Robustness is one of the benefits of neural networks that is lost when units are made to act as feature detectors. Removal or loss of a unit can result in loss of the network capability to be able to recognize that feature.

An efficient method for adjusting the decision surface of the subnetworks still needs to be developed for the DAPL algorithm. The generate-and-test method used by the DAPL algorithm with complicated decision surfaces will produce high training times. The methods currently being explored to help traverse the decision surface of neural networks seem to hold the most promise for a more efficient method for adjusting outputs. Not all outputs need to be adjusted. Adjustment of some outputs will result in the decision surfaces being adjusted for correct classification of other patterns. Further adjustment will result in the decision surface becoming over adjusted. Genetic algorithms have demonstrated interesting results with the creation of test sets composed of different pattern adjustments.

Parallelization would benefit the network greatly. Division of the problem space into regions by the single sided decision surfaces would allow each generated subregion to be handled by different processors. If the approach of genetic algorithms were applied to finding output adjustment sets for the decision surface, each set could be explored with

separate processors. Even in post-processing, the effects of removing certain connections could be tested independently by different processors.

The DAPL algorithm is an attempt to bridge the knowledge between two fields in pattern recognition. Although many questions are left unanswered, the initial results are very promising.

Bibliography

Bibliography

- [DP91] S. Dennis and S. Phillips. "Analysis Tools for Neural Networks". Technical Report 207, The University of Queensland, Brisbane, Australia, May 1991.
- [FL90] S. Fahlman and C. Lebiere. "The Cascade Correlation Learning Architecture". In D. Touretzky, editor, *Advance in Neural Information Processing Systems 2*, pages 524–532. Morgan Kaufman, San Mateo, CA, 1990.
- [Han86] D. J. Hand. "Recent Advances in Error Estimation". *Pattern Recognition Letters*, 4:335–346, 1986.
- [HYH91] Y. Hirose, K. Yamashita, and S. Hijya. "Back-Propagation Algorithm Which Varies the Number of Hidden Units". *Neural Networks*, 4:61–66, August 1991.
- [Lip87] R. P. Lippmann. "An Introduction to Computing with Neural Nets". *IEEE ASSP*, pages 4–22, April 1987.
- [Lor76] G. G. Lorentz. "The 13th Problem of Hilbert". In F. E. Browder, editor, *Mathematical Developments Arising from Hilbert Problems*. American Mathematical Society, Providence, RI, 1976.
- [LR84] T. Lee and J. A. Richards. "Piecewise Linear Classification Using Seniority Logic Committee Methods, with Application to Remote Sensing". *Pattern Recognition*, 17(4):453–464, 1984.
- [MP69] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969.

- [MSK80] R. Mizoguchi, M. Shimura, and O. Kakusho. "A New Algorithm for Constructing Piecewise Linear Discriminative Functions". *Proceedings of the 5th Intl Conf Pattern Recognition*, pages 666–670, March 1980.
- [Nil65] N. J. Nilsson. *Learning Machines*. McGraw-Hill, New York, NY, 1965.
- [Pal90a] H. C. Palm. "A New Piecewise Linear Classifier". *Proceedings of the 10th Intl Conf Pattern Recognition*, pages 742–744, May 1990.
- [Pal90b] H. C. Palm. "On Algorithms for Generating Computationally Simple Piecewise Linear Classifiers". PhD thesis, Norwegian Defence Research Establishment, Department of Computer Science, June 1990.
- [RH93] S. G. Romaniuk and L. O. Hall. "Divide and Conquer Neural Networks". *Neural Networks*, 6:1105–1116, August 1993.
- [RK91] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, New York, NY, 1991.
- [RM88] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing*, volume 1. MIT Press, Cambridge, MA, 1988.
- [Ros62] R. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, New York, NY, 1962.
- [Sch92] R. Schalkoff. *Pattern Recognition*. John Wiley and Sons, Inc., New York, NY, 1992.
- [SD91] J. Sietsma and R. J. F. Dow. "Creating Artificial Neural Networks That Generalize". *Neural Networks*, 4:67–79, August 1991.
- [SW81] J. Sklansky and G. N. Wassel. *Pattern Classifiers and Trainable Machines*. Springer-Verlag, New York, NY, 1981.

[ZMV⁺94] A. Zell, G. Mamier, M. Vogt, N. Mache, R. Hubner, K. Herrmann, T. Soyez, M. Schmalzl, T. Sommer, A. Hatzigeorgiou, S. Doring, and D. Posselt. *Stuttgart Neural Network Simulator User Manual*. University of Stuttgart, 3.2 edition, March 1994.

Appendices

Appendix A

User Manual for Running the DAPL Algorithm

A.1 Introduction

The DAPL algorithm provides a rich environment for the development and analysis of networks in addition to generating the DAPL neural network. In this appendix, a basic walk through the DAPL algorithm is provided for a better understanding of the capabilities of the algorithm.

A.2 Getting Started

The DAPL algorithm operates in two possible modes: an interactive and a batch mode. The batch mode was developed to allow large amounts of testing to be done without the need of an operator. This becomes particularly useful in running classification problems at night when the system's usage is low.

This appendix is meant to describe the interactive mode of the DAPL algorithm. If at anytime you need help in traversing the menus, press "h" and the screen in Figure A.1 will appear.

When the DAPL algorithm is started, the main menu appears. The main menu can be seen in Figure A.2. From here selection of the main options offered by the DAPL algorithm may be selected.

The DAPL algorithm removes the responsibility of selecting the network topology from the user, but also allows the freedom to configure the network in many different ways. Selecting option "1) Change Default Settings" from the main menu will bring up the configuration menu, which is displayed in Figure A.3. The first 16 options are concerned with generation of training sets. The DAPL algorithm was initially created to automatically generate its

Menu Help (perimenu version 2.2)

Use up/down arrows to place ">" in front of your selection.
- Press Return (or Enter) when ready to continue.

Available action-keys:

- h - Show this help screen.
- q - Quit entirely.
- u - Return to the previous menu or function.
- b - Move to the item at the beginning of the menu.
- e - Move to the item at the end of the menu.
- r - Refresh the screen.
- ! - Enter command shell via "/bin/csh".

[Press any key to continue]

Figure A.1: The help menu displayed after pressing "h" from anywhere.

NN Generate Program

- > 1) Change Default Settings
2) Create blank graph
3) Create a new graph
4) Display the graph
5) Edit the graph
6) Load/Save the graph
7) Handle classes
8) SNNS operations
9) Matlab displays
10) Exit this demo

Figure A.2: The DAPL algorithm main menu.

```

file: Configuration
-> 1) X Dimension (Width) : 10
   2) Y Dimension (Width) : 10
   3) Minimum # of Moves : 3
   4) Number of Classes : 100
   5) Border Width : 1
   6) Distribution of Points (RND/UNIFORM) : UNIFORM
   7) Percentage of Points Used : 1
   8) Max Num Leaders : 5
   9) Number of Leaders : 1
  10) Starting Location of Leader : EAST
  11) Number of Exact Followers : 1
  12) Location of Exact Followers : 15
  13) Number of Different Direction Followers: 1
  14) Locations of Different Directed Follows: NORTH
  15) Number of Misdirected Followers : 0
  16) Locations of Misdirected Followers : SOUTH
  17) Starting Number of Nodes : 1
  18) Minimum Number of Nodes Per Layer : 1
  19) Maximum Number of Nodes Per Layer : 100
  20) Maximum Number of Hidden Layers : 1
  21) Generate Results at Error Rates : 0.5,0.25,0.01
  22) Keep Graphs Error Rate Snapshots : yes
  23) Keep Graphs that Produced Failures : yes
  24) Keep Generating Solutions : no
  25) Prune the Graph : yes
  26) Build from Previous Failures : yes
  27) Tolerable Error Rate : 0.2999999999999999889
  28) Cycles : 2000
  29) Grid for Plotting and Testing : 25
  30) Needs to Make an Improvement of : 0.050000000000000002776
  31) Strikes (allows lesser cycles) : 2
  32) Maximum Number of Linear Separations : 100
  33) Start Wrapping Up Time : 180
  34) Must Stop Time : 240
  35) The exactness of separating lines : 1
  36) The Learning Function : Std_Backpropagation
  37) The Update Function : Topological_Order
  38) The Hidden Units Activation Function : Act_Logistic
  39) The Hidden Units Output Function : Out_Identity
  40) The Output Units Activation Function : Act_Logistic
  41) The Output Units Output Function : Out_Identity

(Ctrl) Which value would you like to change? (Press "h" for help)

```

Figure A.3: The configuration menu displayed after choosing option “1) Change Default Settings” from the main menu.

own training sets, so it could test the training sets itself. Options 17 through 41 relate to the how the DAPL network will be built.

The DAPL algorithm, in addition to the automatic generation of testing sets, can be used to manually enter testing sets. This is done visually, and thus can only be done for two dimensional space. This only applies to creating testing sets. The user can provide the algorithm with any pattern file to be used as the training set. Figure A.4 demonstrates a pattern that is displayed when option “4) Display the graph” is selected. This is very similar in appearance to “5) Edit the graph” shown in Figure A.5. Modification of the the classes’ membership or the decision surfaces for that training space can be done when editing the graph.

With editing capabilities there develops a need to be able to save and load the graphs that are created. The “6) Load/Save the graph” allow the user to accomplish this task as shown in Figure A.6 and A.7.

The ability to go beyond just editing graphs was desired. A class library was created with all the capabilities to load, edit, and manipulate classes. Figure A.8 demonstrates the “7) Handle classes” menu item.

Mapping of Current Graphs

[illegible]

```
(^H) h)elp q)uit u)p t)op b)egin e)nd r)efresh
```

Figure A.4: The display menu displayed after choosing option “4) Display the graph” from the main menu.

[illegible]

2222

".tmp" 25 lines, 650 characters

Figure A.5: The edit menu displayed after choosing option “5) Edit the graph” from the main menu.

Graph IO Options

- > 1) Load a graph
- 2) Save graph to file (currently: graph0)
- 3) Select Directory (currently: ./results)

Figure A.6: The load/save menu displayed after pressing "6) Load/Save the graph" from the main menu.

Directory Contents

```
-> 34) jjg1.pat      12/31/69 19:00 root
    35) jjg1.dat      12/31/69 19:00 root
    36) jjg1.dem      12/31/69 19:00 root
    37) ban2          12/31/69 19:00 root
    38) ban2.m         12/31/69 19:00 root
    39) ban2.pat       12/31/69 19:00 root
    40) ban2.dat       12/31/69 19:00 root
    41) ban2.dem       12/31/69 19:00 root
    42) exam.me        12/31/69 19:00 root
    43) ban3           12/31/69 19:00 root
    44) ban3.m         12/31/69 19:00 root
```

(39%) h)elp q)uit u)p t)op n)ext-pg p)rev-pg b)egin e)nd

Figure A.7: The load graph menu displayed after pressing "1) Load a graph" from the load/save menu.

Handle Class

- 1) Delete a class
- 2) Move a class
- 3) Load a class
- > 4) Display the graph
- 5) Select directory (currently: ./results)
- 6) Save the class (currently: class1)

Figure A.8: The class menu displayed after choosing option "7) Handle classes" from the main menu.

SNNS Operations

- >■ 1) Run SNNS evaluation programs
- 2) Examine resulting files
- 3) Results Directory (./results)
- 4) Data Directory (./results)
- 5) Pattern file (graph0)

Figure A.9: The SNNS operations menu displayed after choosing option “8) SNNS operations” from the main menu.

Matlab Operations

- >■ 1) Display pattern file
- 2) Data Directory (./results)
- 3) Pattern file (graph0)
- 4) Examine resulting files

Figure A.10: The Matlab display menu displayed after choosing option “9) Matlab displays” from the main menu.

Once a pattern file is created, either through the menus discussed above or manually, it can be used as the training set for the developing DAPL network. This is done through the “8) SNNS operations” option of the main menu. This option is shown in Figure A.9. The creation of the DAPL network is done by selecting menu item “1) Run SNNS evaluation programs”. Before doing this, make sure the desired pattern file has been loaded. This can be accomplished by selecting “5) Pattern file (graph0)”. The “2) Examine resulting files” allows you to examine the resulting text files. To examine the resulting Matlab files, the user needs to select the Matlab display selection from the main menu.

Figure A.10 shows the Matlab options resulting by selection of the “9) Matlab displays” from the main menu. From here the resulting files are viewed and postscript files are generated. Many of the files used throughout this thesis were generated in this manner. Before displaying the file, make sure a file is loaded. This is done by selecting “3) Pattern file (graph0)”. Two types of files can be displayed. The files with no extension, which are the pattern files generated by the DAPL algorithm previously discussed menus, or files with the extension ‘.res’. These files are the result files of tests on the DAPL network through its various stages of development. A directory listing for the selection of one of these files is shown in Figure A.11.

That should give you a basic understanding of the DAPL algorithm functionality.

Directory Contents

```
105) n-2-5-3.trn.1.2.cnf 12/31/69 19:00 root
106) n-2-5-3.trn.1.2.log 12/31/69 19:00 root
107) n-2-5-3.trn.1.2.res 12/31/69 19:00 root
108) n-2-5-3.trn.1.2.dat 12/31/69 19:00 root
109) n-2-5-3.trn.1.2.dem 12/31/69 19:00 root
110) n-2-5-3.trn.1.3      12/31/69 19:00 root
111) n-2-5-3.trn.1.3.pat 12/31/69 19:00 root
112) n-2-5-3.trn.1.3.cnf 12/31/69 19:00 root
113) n-2-5-3.trn.1.3.log 12/31/69 19:00 root
114) n-2-5-3.trn.1.3.res 12/31/69 19:00 root
115) n-2-5-3.trn.1.3.dat 12/31/69 19:00 root
116) n-2-5-3.trn.1.3.dem 12/31/69 19:00 root
117) n-2-5-3.trn.1.4      12/31/69 19:00 root
118) n-2-5-3.trn.1.4.pat 12/31/69 19:00 root
119) n-2-5-3.trn.1.4.cnf 12/31/69 19:00 root
120) n-2-5-3.trn.1.4.log 12/31/69 19:00 root
-> 121) n-2-5-3.trn.1.4.res 12/31/69 19:00 root
    122) n-2-5-3.trn.1.4.dat 12/31/69 19:00 root
    123) n-2-5-3.trn.1.4.dem 12/31/69 19:00 root
    124) n-2-5-3.trn.1.4.one 12/31/69 19:00 root
```

```
( 77%) h)elp q)uit u)p t)op n)ext-pg p)rev-pg b)egin e)nd r)efresh
```

Figure A.11: The selecting pattern menu displayed after choosing option “3) Pattern file” from the Matlab display menu.

Appendix B

A Flow Diagram of the DAPL Code

When trying to understand the flow of the program, it is best to divide the code into several logical sections: program settings, graph handling, neural network generation and training, and finally viewing results. Figure B.1 is provided so flow of the complete DAPL algorithm can be better understood. A full of description of the functions, for the sake brevity, will be left out of this thesis.

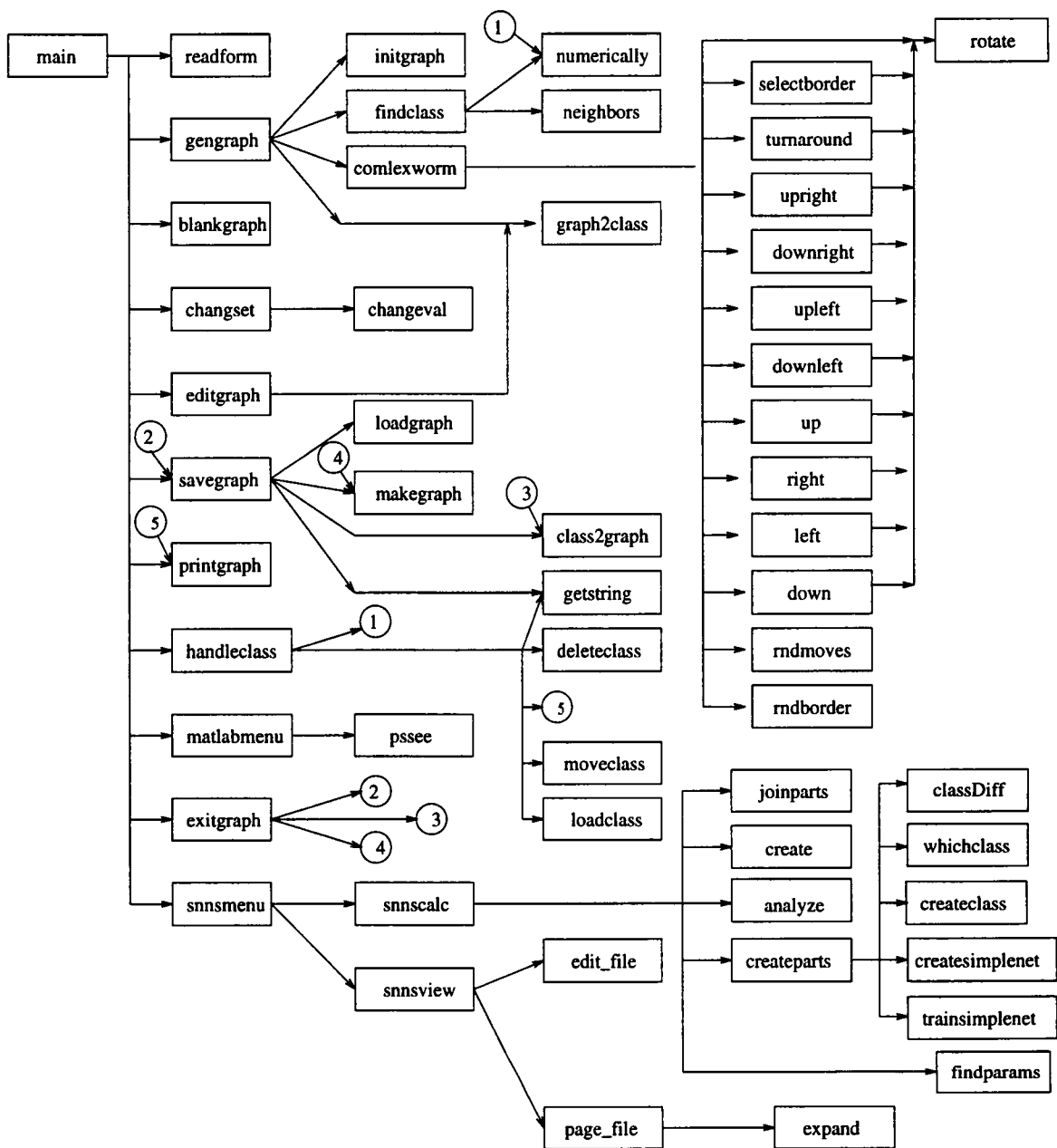


Figure B.1: The DAPL program flow.

Appendix C

Parameter Settings within the DAPL Network

C.1 Learning Rules

The learning rules appear in Table C.1.

C.2 Activation Functions

The activation functions appear in Table C.2.

Table C.1: Learning rules recognized by the DAPL network.

Std_Backpropagation	"vanilla" Backpropagation
BackpropBatch	Backpropagation for batch training
BackpropMomentum	Backpropagation with momentum term
BackpropWeightDecay	Backpropagation with weight decay
Quickprop	Quickprop (Scott Fahlman)
Rprop	Radial Basis Learning

Table C.2: Activation functions recognized by the DAPL network.

Function	Formula
Act_BAM	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ a_j(t-1) & \text{if } net_j(t) = 0 \\ -1 & \text{if } net_j(t) < 0 \end{cases}$
Act_BSB	$a_j(t) = net_j(t) * \theta$
Act_Elliott	$a_j(t) = \frac{net_j(t)}{1 + net_j(t) }$
Act_Identity	$a_j(t) = net_j(t)$
Act_IdentityPlusBias	$a_j(t) = net_j(t) + \theta$
Act_Logistic	$a_j(t) = \frac{1}{1 + e^{-(net_j(t) + \theta)}}$
Act_Logistic_notInhibit	like Logistic, but skip input from units named "Inhibit"
Act_Logistic_Tbl	like Logistic, but with table lookup instead of computation
Act_MinOutPlusWeight	$a_j(t) = \min_i(w_{ij} + o_i)$
Act_Perceptron	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) \geq \theta \\ 0 & \text{if } net_j(t) < \theta \end{cases}$
Act_Product	$a_j(t) = \prod_i w_{ij} o_i$
Act_RBF_Gaussian	$h(r^2, p) = h(q, p) = e^{-pq} \quad \text{where } q = \vec{x} - \vec{t} ^2$
Act_RBF_MultiQuadratic	$h(r^2, p) = h(q, p) = \sqrt{p+q} \quad \text{where } q = \vec{x} - \vec{t} ^2$
Act_RBF_ThinPlateSpline	$h(r^2, p) = h(q, p) = p^2 q \ln(p\sqrt{q})$
Act_Signum	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ -1 & \text{if } net_j(t) \leq 0 \end{cases}$
Act_Signum0	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ 0 & \text{if } net_j(t) = 0 \\ -1 & \text{if } net_j(t) < 0 \end{cases}$
Act_StepFunc	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ 0 & \text{if } net_j(t) \leq 0 \end{cases}$
Act_TanH	$a_j(t) = \tanh(net_j(t))$
Act_TanHPlusBias	$a_j(t) = \tanh(net_j(t) + \theta)$
Act_TanH_Xdiv2	$a_j(t) = \tanh(net_j(t)/2)$

Table C.3: Output functions recognized by the DAPL network.

Function	Formula
Out_Identity	$o_j(t) = a_j(t)$
Out_Clip_0_1	$o_j(t) = \begin{cases} 0 & \text{if } a_j(t) \leq 0 \\ a_j(t) & \text{if } 0 < a_j(t) < 1 \\ 1 & \text{if } a_j(t) \geq 1 \end{cases}$
Out_Clip_1_1	$o_j(t) = \begin{cases} -1 & \text{if } a_j(t) \leq -1 \\ a_j(t) & \text{if } -1 < a_j(t) < 1 \\ 1 & \text{if } a_j(t) \geq 1 \end{cases}$
Out_Threshold_0_5	$o_j(t) = \begin{cases} 0 & \text{if } a_j(t) \leq 0.5 \\ 1 & \text{if } a_j(t) > 0.5 \end{cases}$

C.3 Output Functions

The output functions appear in Table C.3.

VITA

John Joseph Gerber was born in Patterson, New Jersey on May 14, 1967. After graduating high school in 1985, he attended college at Millersville University in Pennsylvania. There he received his Bachelor of Science degree in Computer Science, graduating cum laude in 1989. At Millersville University, a school that has a high reputation in education, John gained his appreciation for the science of teaching. This appreciation for education would propel him to continue his studies, and cause him to change schools twice more in his search for a university which he could be proud to graduate from. In September of 1992, he entered The University of Tennessee, Knoxville and in December 1994 received a Master of Science degree in Computer Science.

He is presently employed at Research Services at the University of Tennessee, Knoxville, where he is continuing his education towards his doctorates.