ABSTRACT

POTTER, MATTHEW JAMES. Improving ANN Generalization via Self-Organized Flocking in conjunction with Multitasked Backpropagation. (Under the direction of Mark White.)

The purpose of this research has been to develop methods of improving the generalization capabilities of artificial neural networks. Tools for examining the influence of individual training set patterns on the learning abilities of individual neurons are put forth and utilized in the implementation of new network learning algorithms. Algorithms are based largely on the supervised training algorithm: backpropagation, and all experiments use the standard backpropagation algorithm for comparison of results. The focus of the new learning algorithms revolve around the addition of two main components. The first addition is that of an unsupervised learning algorithm called flocking. Flocking attempts to provide network hyperplane divisions that are more evenly influenced by examples. The second addition is that of a multi-tasking approach called convergence training: initially, information provided by a clustering algorithm is used to create subtasks that represent divisions between clusters. These subtasks are then trained in unison in order to promote hyperplane sharing within the problem space. Generalization was improved in most cases and the solutions produced by the new learning algorithms are demonstrated to be very robust against different random weight initializations. This research is not only a search for better generalizing ANN learning algorithms, but also a search for better understanding when dealing with the complexities involved in ANN generalization.

Improving ANN Generalization via Self-Organized Flocking in conjunction with Multitasked Backpropagation

by

Matthew James Potter

A thesis submitted to the Graduate Faculty of
North Carolina State University
In partial fulfillment of the
Requirements for the Degree of
Master of Science

Computer Engineering

Raleigh

2003

APPROVED BY:

(Chair of Advisory Committee)

DEDICATION

To my father

For endowing me with a love for all things science fiction

To my mother

For keeping me levelheaded and inspiring me to accomplish my goals

To Leslie

For her love, patience, and support during this creative process

BIOGRAPHY

Matthew Potter was born on February 7, 1978 and raised in the small town of Kernersville North Carolina. He received his Bachelor of Science degree in Computer Engineering from North Carolina State University in 2001. That same year he began working toward his Masters Degree also at NCSU. He plans to graduate in May 2003 at which point the real world will be obliged to welcome its newest member.

ACKNOWLEDGEMENTS

None of this research would have been possible without the many contributions of Mark White. Most of the ideas in the following paper are his and without the many hours of coffee house discussions and paper napkin scribbles this thesis would never have come to pass. It has been an honor and a pleasure working with him on this endeavor. A very special thanks is also in order to Jonathan Palmer and Trevor Cox, whose immense contributions were the jumping off point for the exploration of this material. A final thanks goes out to Dr. Krim and Dr. Selgrade for agreeing to be members of my advisory committee and for their commitment to overseeing the thesis process. For the considerable contributions of all these individuals I am enormously grateful.

TABLE OF CONTENTS

List of Figures……..……..vi

List of Symbols & Abbreviations..……..…… viii

1. Introduction……1

2. Core Concepts and Foundations……..……..…….. 3
2.1 ANN Fundamentals.…….... 3
2.2 Deriving the Backprop Algorithm.…….. 8
2.3 Network Generalization.…….15
2.4 Object-Oriented Backpropagation with JOONE.…… 17
2.5 Example Influence…….20
2.6 Flocking..……...…….23
2.7 Multi-task Training…..……..……..…28

3. Intermediate Learning Algorithms……..……..…….. 31
3.1 Split Flocking……..……..……..……..……31
3.2 Convergence Training……..……..……..…… 34

4. Problem Statement……..…….. 37

5. Bringing It All Together (Primary Algorithm One)…….……..40
5.1 Methods of Approach…..…….…….…….…… 40
5.2 Results & Analysis…..…….... 44

6. Adding Multiple Hidden Layers (Primary Algorithm Two)..…….... 61
6.1 Methods of Approach…..…….…….…….…… 61
6.2 Results & Analysis…….... 64

7. Ad-hoc Split Flocking (Supplemental Algorithm)…….. 76
7.1 Methods of Approach…..…….…….…….…… 78
7.2 Results & Analysis…..……..…….. 79

8. Ideas for Future Research…….....……..……..82

9. Conclusions…….……..……..84

10. List of References…..……..……..87

11. Appendices…..……..……..88
11.1 Appendix 1: A Flocking Demonstration……..……..…89
11.2 Appendix 2: Implementation Details……..…..98

LIST OF FIGURES

Figure 2.1 The Neuron Model……...……4
Figure 2.2 Neuron Output Response……5
Figure 2.3 Typical Single Hidden Layer Network Architecture……...6
Figure 2.4 The JOONE Network Model……... 18
Figure 2.5 Object-Oriented Derivatives……..... 19
Figure 2.6 Sigmoid Function……21
Figure 2.7 Sigmoid Derivative……21
Figure 4.1 The Ideal Data Set…….... 37
Figure 4.2 The Actual Data Set……38
Figure 4.3 The Training Set……39
Figure 4.4 The Test Set……39
Figure 5.1 Network Architecture of Control ANN……..… 45
Figure 5.2 Control Network Hyperplanes……46
Figure 5.3 Control Network SSE…….. 47
Figure 5.4 Network Architecture for duration of Phase 1……48
Figure 5.5 Phase 1 post split flocking……..……49
Figure 5.6 Phase 1 post convergence training……..……49
Figure 5.7 Phase 2 split flocking neuron 1……...……50
Figure 5.8 Phase 2 hyperplanes post split flocking……..……50
Figure 5.9 Phase 2 convergence training……...…..…… 51
Figure 5.10 Phase 2 hyperplanes post convergence training…..…… 51
Figure 5.11 Phase 3 split flocking neuron 3……....52
Figure 5.12 Phase 3 hyperplanes post split flocking……...…52
Figure 5.13 Phase 3 convergence training…….. 53
Figure 5.14 Phase 3 hyperplanes post convergence training…..…… 53
Figure 5.15 Phase 4 split flocking neuron 11……..53
Figure 5.16 Phase 4 hyperplanes post split flocking…..……..53
Figure 5.17 Phase 4 convergence training…….…….. 54
Figure 5.18 Phase 4 hyperplanes post convergence training……..…… 54
Figure 5.19 Final Hidden Layer Output Responses……...……54
Figure 5.20 Subtasks created by split flocking…..…… 55
Figure 5.21 Network Architecture during final training stage...……56
Figure 5.22 Sum Squared Error during final training stage...……56
Figure 5.23 Control ANN Hyperplanes (From 10 runs)……57
Figure 5.24 Primary ANN Hyperplanes (From 10 runs)……58
Figure 5.25 Control ANN Sum Squared Error (From 10 runs)…….…….… 59
Figure 5.26 Primary ANN Sum Squared Error (From 10 runs)…….……59
Figure 6.1 Neuron 0 for duration of Phase 1……65
Figure 6.2 Phase 2 split flocking neuron 1……...……65
Figure 6.3 Phase 2 convergence training neuron 1……..…… 65
Figure 6.4 Phase 2 final convergence training…….…… 65
Figure 6.5 Phase 3 split flocking neuron 0……..…66
Figure 6.6 Phase 3 convergence training neuron 0……..…… 66

Figure 6.7 Phase 3 final convergence training…. 66
Figure 6.8 Phase 4 split flocking neuron 0……...…….66
Figure 6.9 Phase 4 convergence training neuron 3……..…… 67
Figure 6.10 Phase 4 final convergence training…….…… 67
Figure 6.11 Layout during final training stage…...…….67
Figure 6.12 Final training stage SSE graph……...…… 67
Figure 6.13 Hidden Layer 1 output response…….…….68
Figure 6.14 Hidden Layer 2 output responses……...…… 68
Figure 6.15 Hidden Layer 3 output responses…...…… 69
Figure 6.16 Hidden Layer 4 output responses…...…… 69
Figure 6.17 Hidden Layer 4 hyperplane representations…..……..…. 71
Figure 6.18 Layer 3 Neuron 2 Preflock Response…… 73
Figure 6.19 Layer 3 Neuron 2 Postflock Response…….73
Figure 6.20 Layer 4 Neuron 3 Preflock Response…….…….. 73
Figure 6.21 Layer 4 Neuron 3 Postflock Response…….73
Figure 6.22 Overlaid output responses of 10 trained Control Networks……...……. 74
Figure 6.23 Overlaid output responses of 10 trained Multi-hidden Layer Networks…. 75
Figure 6.24 Overlaid SSE graphs of 10 Multi-hidden Network training sessions.……75
Figure 7.1 unmagnified split flock training session…….…….. 77
Figure 7.2 Control Network Hyperplanes……80
Figure 7.3 Ad-hoc Network Hyperplanes…… 80
Figure 7.4 Control Network SSE…….…….. 81
Figure 7.5 Ad-hoc Network SSE…….…….. 81
Figure 11.1 Flocking Demo Topology……...……..89
Figure 11.2 Demo data set with overlay of initial hidden neuron hyperplane……...….90
Figure 11.3 Initial hidden neuron Output Response……...…. 91
Figure 11.4 Initial influence of each example……91
Figure 11.5 Initial hidden neuron Weights…….. 91
Figure 11.6 Initial example influence as a Function of Net Input…….. 91
Figure 11.7 hidden neuron hyperplane after backpropagation training…….. 92
Figure 11.8 Preflock hidden neuron Output Response……... 93
Figure 11.9 Preflock influence of each example……93
Figure 11.10 Preflock hidden neuron Weights…….. 93
Figure 11.11 Preflock example influence as a Function of Net Input……..…. 93
Figure 11.12 Hidden neuron hyperplane post backpropagation training…….……95
Figure 11.13 Sum Squared Error during backpropagation and flocking…….. 95
Figure 11.14 Final hidden neuron Output Response……96
Figure 11.15 Final influence of each example……...96
Figure 11.16 Final hidden neuron Weights……...96
Figure 11.17 Final example influence as a Function of Net Input……....96

LIST OF SYMBOLS & ABBREVIATIONS

i h o = Subscripts denoting a particular input, hidden, or output neuron respectively

p = Subscript denoting a particular example pattern*

i_i = Input $_i$

t_o = Target output for neuron $_o$

η = Learning rate

h_h = Output of hidden neuron $_h$

net_h = Net Input to hidden neuron $_h$

b_h = Bias of hidden neuron $_h$

w_{ih} = Weight connecting input $_i$ to hidden neuron $_h$

o_o = Output of output neuron $_o$

net_o = Net Input to output neuron $_o$

b_o = Bias of output neuron $_o$

w_{ho} = Weight connecting hidden neuron $_h$ to output neuron $_o$

E_o = Error of output neuron $_o$

Ψ_{ph} = Flocking Error for pattern $_p$ and hidden neuron $_o$

EI_{ph} = Example Influence of pattern $_p$ on net_h

Nef_h = Effective number of examples influencing neuron $_h$

$mean_h$ = Weighted mean net_h magnitude

avg_h = Weighted mean net_h

α = Constant defined in the flocking equations

β = Constant defined in the flocking equations

H = Total number of hidden layer neurons

O = Total number of output layer Neurons

I = Total number of inputs

ANN: Artificial Neural Network

SSE: Sum Squared Error

JOONE: Java Object Oriented Neural Engine

* Many of the equations used in this paper are assumed to be for a particular pattern, if the subscript $_p$ is missing then it is safe to assume that the equation is the same for all example patterns.

1. Introduction

Artificial neural networks (or ANNs for short) are very simplistic models of neurons in the brain. A multitude of different ANNs exist, and each have unique learning algorithms associated with them. All ANNs also have certain things in common, such as neuron and synapse components. A brief history of Artificial Neural networks and a summary of the typical components used to construct them will be covered in the section 2.1. For the most part this paper assumes that the reader already has a background in the basics of ANN construction and is familiar with the details of the most widely known ANN learning algorithm: backpropagation. A brief account of the algorithm and mathematics associated with it is provided however, and can be found in section 2.2. Some of the key components of backpropagation will also be reviewed in section 2.4 for the purpose of clearly explaining the implementation of the methods used in this paper.

The research conducted is an attempt to address two main concerns associated with the study of ANNs. The first of these two concerns is simply put, to gain a better understanding of the mysterious inner-workings associated with Artificial Neural Networks and backpropagation learning. Though the gradient-descent based algorithm has proven itself time and time again in practice, backpropagation actually has a very low degree of comprehensibility when it comes to trying to understand and predict exactly how and why a solution will be arrived at. More often than not, the exact same network architecture and learning algorithm can arrive at vastly different solutions to the same problem depending on the initial weights and other learning parameters. All of these solutions may "solve" the training problem perfectly, but some may end up providing much better general solutions on

unseen data. What have the networks that perform better on unseen data learned that the previous networks did not learn? What components of these networks are responsible for the differences in learning? It is the unraveling of this "black-box" that will be the first of the two main goals of this paper.

The second goal relates directly to the first, for once we have gained some insight into the "black-box", how can we utilize this information in order to train networks which more consistently generalize better on unseen data? To this end, the tools developed to gain insight into the network will be used to create a smarter more robust ANN learning algorithm.

Both the objectives of this paper are very ambitious ones and will undoubtedly result in far more questions than answers. A small step forward in a daunting task is none-the-less a step forward, and I only hope that this research can be regarded as one such step. With that in mind the next section provides the background and basis for all the tools that will be utilized in this discovery process, and the remainder of the paper describes the specifics of the experiments and learning algorithms used in them.

2. Core Concepts and Foundations

The background information in this section provides much of the base knowledge for the learning algorithms that will be the primary focus of this paper. The following sections supply algorithm details as well as some of the basic theoretical concepts behind backpropagation, example influence, flocking, and multi-task training; all of which are key components of the network learning algorithms to be presented in later chapters.

2.1 ANN Fundamentals

ANNs are very simplified models of the interconnected neurons found in the brain. To say that they are simplified models is quite an understatement. Wet nets, the neural networks found in the brain, are composed of billions of neurons that undergo remarkably complex electro-chemical reactions and can have tens of thousands of interconnections within just a single neuron. Artificial neural networks on the other hand, are composed of relatively few neurons with relatively few interconnections that undergo relatively simple mathematical transformations. If one were to try and model the exact behavior of but a single neuron in the brain, the model of this neuron would likely become so complex that the primary goal of using the neuron to simulate learning would become overshadowed by the sheer processing power that would be required to simulate the neuron model.

The neuron model used in most artificial networks is therefore a simple one based on just one primary aspect of how we believe learning takes place in the brain. This primary aspect has to do with how neurons communicate through the use of electrical impulses traveling through synapses. The synapses have the ability to inhibit or amplify these

electrical signals and when the total magnitude of the electrical impulses on the input synapses of a particular neuron builds up and passes a certain threshold, the neuron fires an impulse that then becomes an input signal to all the neurons connected to its output synapses. Artificial networks model this behavior by designating weights to the connections (synapses) in the network and non-linear thresholding functions to the neurons in the network. The most common neuron model (and the one used in the research of this paper) is illustrated in the following figure:

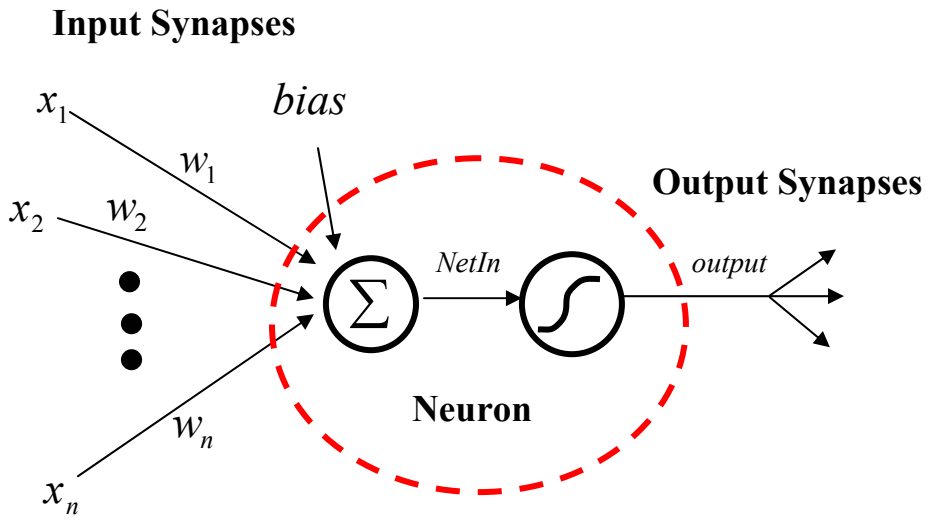

Figure 2.1 The Neuron Model

In this model the input signals x_1 to x_n are modified by the weights w_1 to w_n respectively on their way through the neuron's input synapses. The modified input signals are then summed together along with a bias input to create the *NetIn* signal for the neuron. Finally the *NetIn* signal is used as the input into the neuron's non-linear function. For the neurons used in this paper the non-linear function used is the logistic sigmoid function.

The following figure depicts the logistic function and the graph of its output response.

$$output = \frac{1}{1 + e^{-NetIn}}$$

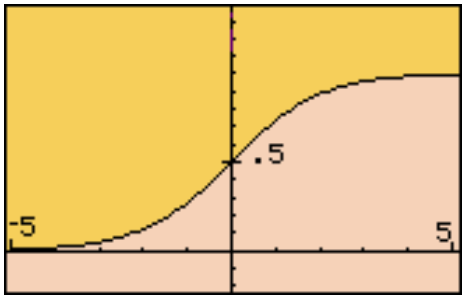

Figure 2.2 Neuron Output Response

The idea of modeling the behavior of neurons in the brain has been around for almost fifty years, but the advancement of key learning algorithms and the ability to carry out the large amount of processing required for most applications has only occured more recently. The backpropagation algorithm (the central learning algorithm used in this research), though independently invented much earlier, did not gain wide spread awareness until the publishing of Learning Representations by Back-propagating Errors by Rummelhart, Hinton, & Williams in 1986 [1]. Today ANNs are being applied to a variety of real world problems, enjoying success in areas such as modeling, classification, pattern recognition, and multivariate data analysis [2]. However, they are probably most well known for their classification abilities because ANNs can be used to effectively perform speech, image, and character recognition.

Earlier we discussed the actual model used for neurons within most ANNs, but how are these neurons usually fitted together within a typical network? The answer to this question describes the topology of a given network, and many different topologies exist.

Here is an example topology of a typical three layer feed-forward ANN:

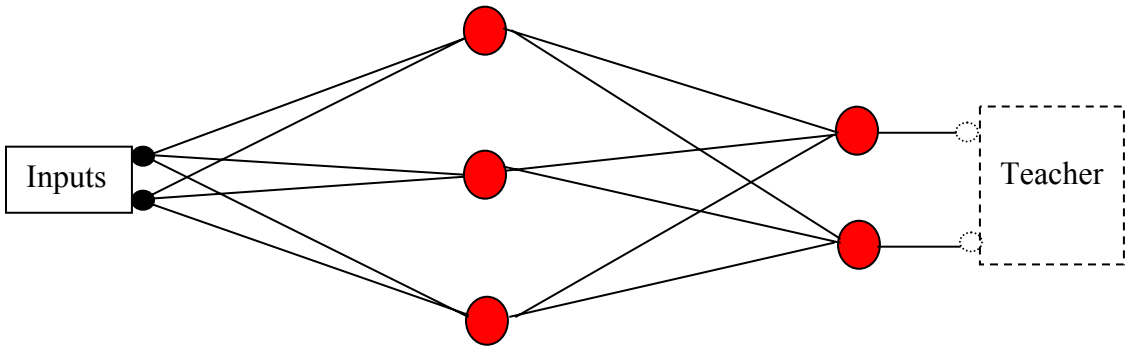

Figure 2.3 Typical Single Hidden Layer Network Architecture

Notice that what is usually designated as the input layer is really just the inputs themselves. The next two layers are referred to as the hidden and output layers respectively, and the synapses associated with each of theses layers are the connections depicted entering each layer from left to right. Also remember that each of the neurons in these two layers has an associated bias input (which is not depicted in the Figure). Other network topologies that exist are usually modifications of this design. Such modifications include adding more hidden layers, removing the hidden layer, adding connections that "jump" across layers, adding recurrent (feedback) connections, or arranging a layer's neurons on a two dimensional plane (the idea being to that location of a neuron in that plan is used as a parameter in the learning algorithm).

All ANNs usually have an associated learning algorithm that is used to set the weights of the network to achieve some sort of desired response. These associated learning algorithms usually fall into one of two distinct categories, either the ANN learns by means of

supervised training or by means of unsupervised training. ANNs that undergo unsupervised training organize themselves based solely on the inputs they are given. No help from the outside world is given to the network to help it evaluate how well it is doing. The networks that use this type of training do not use the "Teacher" component depicted in Figure 2.3. The flocking algorithm (described in section 2.4) is an example of an unsupervised learning algorithm.

ANNs that undergo supervised training on the other hand, do use the "Teacher" component depicted in Figure 2.3. In supervised training the teacher is used to analyze the outputs of the network and feedback some sort of error that the learning algorithm then uses to update the weights. There are large variety of supervised learning algorithms, but perhaps the most well known is that of the previously mentioned backpropagation algorithm, which uses the process of gradient descent to traverse the problem's associated error surface in search of a global minima. The particular details of this algorithm will be discussed in section 2.2, which is devoted to deriving the backprop algorithm.

Another important distinction to mention when discussing neural nets is the mode of operation that the network is in. There are two basic categories here as well that will be referred to as training mode and recall mode. While the network is in training mode, the weights are being continually updated as a result of either supervised or unsupervised training. While the network is in recall mode however, all network weights are locked (unchanging). Recall mode has no associated learning algorithm and serves only to collect the network output response for a given set of inputs.

For further information regarding the variety of models, topologies, and learning algorithms that exist within the field of artificial neural networks, please refer to the reference section listed at the end of this paper. Or perform a library or web search online; a simple search using the words "neural nets" will return a multitude of information.

2.2 Deriving the Backprop Algorithm

This section is devoted to describing the details of the backpropagation algorithm. To understand the derivation, a background in calculus fundamentals (specifically in regards to the gradient operator and using the chain rule to take partial derivatives) is required.

Backpropagation is really nothing more than an implementation of the well-known gradient descent algorithm. This algorithm is used to traverse the error surface created by a defined error function that measures the magnitude of the differences between actual network outputs for a particular example pattern and the pattern's respective target outputs. By calculating the gradient of the error produced by this function, the error surface can be traversed in a search for a minimum error. Implementation of the error function is the responsibility of the "Teacher" module discussed in the previous section. Following is the basic gradient descent equation governing a single weight change made in the backprop algorithm:

$$w_{t+1} = w_t - \eta \frac{\partial E}{\partial w}$$

This equation states that each weight w in the network is updated for $epoch_{t+1}$ according to its previous value at $epoch_t$ modified by the gradient of the error function with respect to that particular weight. The parameter η is merely a learning rate parameter that controls the rate of change within the network.

The reason that backpropagation is so computationally intensive is because of the calculations involved in determining the derivative of the error function with respect to the individual weights (i.e. the gradient of the error surface). The remainder of this section is devoted to the derivation of the equations used to calculate this gradient within a typical single hidden layer feed-forward ANN.

Lets start by taking a look at the error function for a particular output:

$$\mathrm{E}_o = \frac{1}{2}\left(o_o - t_o\right)^2$$

This equation states that the error associated with a particular output is proportional the magnitude of the difference between the output and the target output of a particular neuron for a particular example pattern. In order to determine the derivative of this error function with respect to a particular weight we are going to need the equations that relate how these weights are used to determine the network outputs. To this end we will next examine the equations governing the feed-forward operation of our network.

When the input signals first enter the network they are modified by the weights of the hidden layer synapses associated with a particular hidden layer neuron and the results are summed together along with the neuron's bias to produce the net input into that particular

hidden layer neuron. Following is the equation governing the net Input into a particular hidden layer neuron:

$$net_h = \sum_{i=1}^{I} \left(w_{ih} i_i \right) + b_h$$

Next the net Input is modified by the neuron's transfer function in order to produce a neuron output signal. Following is the equation governing the transfer function of a hidden layer neuron:

$$h_h = \frac{1}{1 + e^{-net_h}}$$

Next the output signals of the hidden layer neurons are all modified by the associated weights of the synapses of each particular output layer neuron. All the modified signals are then added up along with the bias to produce a net input into each output layer neuron. Following is the equation governing the net Input into a particular output layer neuron:

$$net_o = \sum_{h=1}^{H} \left(w_{ho} h_h \right) + b_o$$

Finally the actual output signals of the network are produce by running each output layer neuron's net input through the output layer neuron's associated transfer function. Following is the equation governing the transfer function of a particular output layer neuron:

$$o_o = \frac{1}{1 + e^{-net_o}}$$

Now we have all the base information needed to derive the equations governing the gradient of the error surface with respect to the individual weights of the network. By studying the relationships of all the previous equations it should be easy to see that this goal can be accomplished by repeated application of the chain rule. Following are the primary equations governing the calculation of the gradient with respect to the different weights and biases using the chain rule:

$$\left(\frac{\partial \mathrm{E}}{\partial w_{ho}}\right)=\left(\frac{\partial \mathrm{E}_o}{\partial o_o}\right)\left(\frac{\partial o_o}{\partial net_o}\right)\left(\frac{\partial net_o}{\partial w_{ho}}\right)$$

$$\left(\frac{\partial \mathrm{E}}{\partial b_o}\right)=\left(\frac{\partial \mathrm{E}_o}{\partial o_o}\right)\left(\frac{\partial o_o}{\partial net_o}\right)\left(\frac{\partial net_o}{\partial b_o}\right)$$

$$\left(\frac{\partial \mathrm{E}}{\partial w_{ih}}\right)=\left(\sum_{o=1}^{O}\left(\frac{\partial \mathrm{E}_o}{\partial o_o}\right)\left(\frac{\partial o_o}{\partial net_o}\right)\left(\frac{\partial net_o}{\partial h_h}\right)\right)\left(\frac{\partial h_h}{\partial net_h}\right)\left(\frac{\partial net_h}{\partial w_{ih}}\right)$$

$$\left(\frac{\partial \mathrm{E}}{\partial b_h}\right)=\left(\sum_{o=1}^{O}\left(\frac{\partial \mathrm{E}_o}{\partial o_o}\right)\left(\frac{\partial o_o}{\partial net_o}\right)\left(\frac{\partial net_o}{\partial h_h}\right)\right)\left(\frac{\partial h_h}{\partial net_h}\right)\left(\frac{\partial net_h}{\partial b_h}\right)$$

Now all that is left is to calculate the partial derivative components used in the gradient equations above and then to simply make the necessary substitutions. Lets start with the derivative of the error function with respect to the output:

$$\left(\frac{\partial \mathrm{E}}{\partial o_o}\right) = \left(o_o - t_o\right)$$

Now for the derivatives of all the different net Inputs with respect to the different various weights and biases of the network:

$$\left(\frac{\partial net_o}{\partial w_{ho}}\right) = h_h \qquad \left(\frac{\partial net_o}{\partial h_h}\right) = w_{ho} \qquad \left(\frac{\partial net_o}{\partial b_o}\right) = 1$$

$$\left(\frac{\partial net_h}{\partial w_{ih}}\right) = i_i \qquad \left(\frac{\partial net_h}{\partial i_i}\right) = w_{ih} \qquad \left(\frac{\partial net_h}{\partial b_h}\right) = 1$$

For the remaining partial derivative components (those corresponding to the sigmoid transfer functions) we will exploit an inherent property of the sigmoid function in order to dramatically improve the computational requirements of the backpropagation algorithm.

Instead of proving the derivation completely, here is a brief overview using the equation for the sigmoid function of an output layer neuron:

Start with the equation for the actual derivative of the output with respect to the net input (Note the computations that would be required)

$$\left(\frac{\partial o_o}{\partial net_o}\right) = \frac{1}{4\left(\cosh\left(\frac{net_o}{2}\right)\right)^2}$$

Now if we take the original sigmoid function and simply solve it in terms of the net input, we get the following:

$$net_o = \ln\left(\frac{-o_o}{o_o - 1}\right)$$

Then we simply plug this result into our partial derivative equation and do some fancy reduction (not shown here):

$$\left(\frac{\partial o_o}{\partial net_o}\right) = \frac{1}{4\left(\cosh\left(\frac{\ln\left(\frac{-o_o}{o_o-1}\right)}{2}\right)\right)^2} = o_o(1-o_o)$$

So our final partial derivative components turn out to be the following (Note the reduction in the computational effort required to compute the derivative):

$$\left(\frac{\partial h_h}{\partial net_h}\right) = h_h(1-h_h) \qquad \left(\frac{\partial o_o}{\partial net_o}\right) = o_o(1-o_o)$$

Plugging the equations governing the different various partial derivative components back into the gradient equations gives us the following:

$$\left(\frac{\partial \mathrm{E}}{\partial w_{ho}}\right) = (o_o - t_o)(o_o(1-o_o))(h_h)$$

$$\left(\frac{\partial \mathrm{E}}{\partial b_o}\right) = (o_o - t_o)(o_o(1-o_o))(1)$$

$$\left(\frac{\partial \mathrm{E}}{\partial w_{ih}}\right) = \left(\sum_{o=1}^{O}(o_o - t_o)(o_o(1-o_o))(w_{ho})\right)\ (h_h(1-h_h))(i_i)$$

$$\left(\frac{\partial \mathrm{E}}{\partial b_h}\right) = \left(\sum_{o=1}^{O}(o_o - t_o)(o_o(1-o_o))(w_{ho})\right)\ (h_h(1-h_h))(1)$$

To then actually obtain the final equations governing backpropagation all that is left is to plug the above equations into the gradient descent equation described at the beginning of this section.

2.3 Network Generalization

The true test of any ANN lies not in its ability to learn what it is taught, but in its ability to apply what it has learned on new data. If the ANN makes good predictions based on never before seen data, it is said to generalize well. If it makes poor predictions it is said to generalize poorly. One common practice used when training ANNs is to simultaneously train on a training set while testing on a test set. The idea being that the error of the test set will drop along with the error of the training set, until at some point the network begins to "memorize" the training data and then the error of the test set will begin to rise while the error of the training set continues to fall. It is at this point that training of the network is usually halted because further training would negatively affect generalization. The error on the test set gives us a good stopping criterion and a decent method for improved generalization. However, as far as understanding the generalization process, this method is of little use. Methods for characterizing and understanding generalization should therefore be explored.

One such characterization is simply to note that examples in close proximity to each other in the input space are likely to have similar target outputs. Therefore if the decision regions formed by the hyperplanes in a network are fewer and larger as opposed to superfluous and smaller, then it is probably safe to assume that the output of the network will be more general because of it. This is synonymous with having a few big generalized categories as opposed to many smaller ones. It would also seem a safe assumption to say that if training set patterns are located toward the center of decision regions (as opposed to being located on or around the hyperplane boundaries), then similar patterns in the test set are more

likely to be categorized correctly. This is simply because the test set patterns will be more likely to fall into the correct decision region. These are key observations that our learning algorithm should try to exploit if it is to construct a network that generalizes well. It is also worth noting that basic backpropagation does not seek to do this, and that the size of each decision region is often more a function of the number of neurons in the network.

Another such characterization that extends directly from this idea is the concept of "hyperplane-sharing". That is, if a neuron's weights are such that the hyperplane (formed by the weights) is effectively used in multiple decision regions, then this hyperplane's function is a more general one, and it seems likely that this sharing would improve generalization. This assumption is based not only on the idea that hyperplane sharing will result in fewer overall hyperplanes, but also on the idea that it is exploiting inherent symmetry or similarities across parts of the problem space. When a network is learning something in one area of the input space, typically the changes made to learn in this area will affect the output in other locations of the input space. If these affects are positive, then symmetry is being exploited, and it is useful for generalization. However, if the opposite is true, then this new learning has interfered with the learning at other locations and a less general solution will result simply because the given mapping will have less inherent regularity within it. When memorization occurs, this interference is actually overcompensated for which causes "unlearning" to occur on the test set. So the right balance between "hyperplane-sharing" and removing interference must be found in order to maximize generalization and learning while preventing memorization.

The approach taken by this paper to improve generalization in ANNs will seek to exploit the ideas just discussed. Also, as a consequence of this endeavor, it is hoped that regardless of the results, some of the mystery surrounding the nature of generalization, and its place in the relationship between inputs and outputs will become slightly less mysterious. One thing to keep in mind is that a network's ability to generalize is not simply related to how well it has learned what it is taught. In fact ANNs have the least variance at the beginning of training before they have learned anything. This is because in the beginning all possible inputs map to approximately the same point in the output space. Therefore improving a network's ability to generalize may often times interfere with its ability to learn a training set.

2.4 Object-Oriented Backpropagation with JOONE

The background information discussed in this section is intended to relate the Object Oriented philosophy to backpropagation used throughout this paper. The philosophy is central not only in the coding of the experiments, but also to the very train of thought surrounding many ideas that will be discussed. We begin by taking a typical weight update equation and discussing how it is interwoven into an object oriented based schema.

When one first learns the mathematical equations necessary to perform backpropagation (like the ones reviewed in section 2.2), he or she is usually presented them in a very concrete fashion. For example, one might be presented the following equation for a single Δw weight change in the hidden layer:

$$\Delta w_{ih} = \eta\left(\frac{\partial \mathrm{E}}{\partial w_{ih}}\right) = \eta\left(\sum_{o=1}^{O}(o_o - t_o)o_o(1-o_o)w_{ho}\right) \quad h_h(1-h_h) \quad i_i$$

While this is no doubt correct, it might leave one hard-pressed to then determine the equation used to change a weight in a second or third let alone a fourth hidden layer. Many approaches to modeling multi-hidden layer networks exist, but the approach used in this paper comes from a software package called JOONE.

JOONE stands for Java Object Oriented Neural Engine, and is appropriately named. Using this model, networks are constructed using two basic network objects: Layers and Synapses, which are assembled together, like legos. The following diagram demonstrates the role of these network components in a typical ANN:

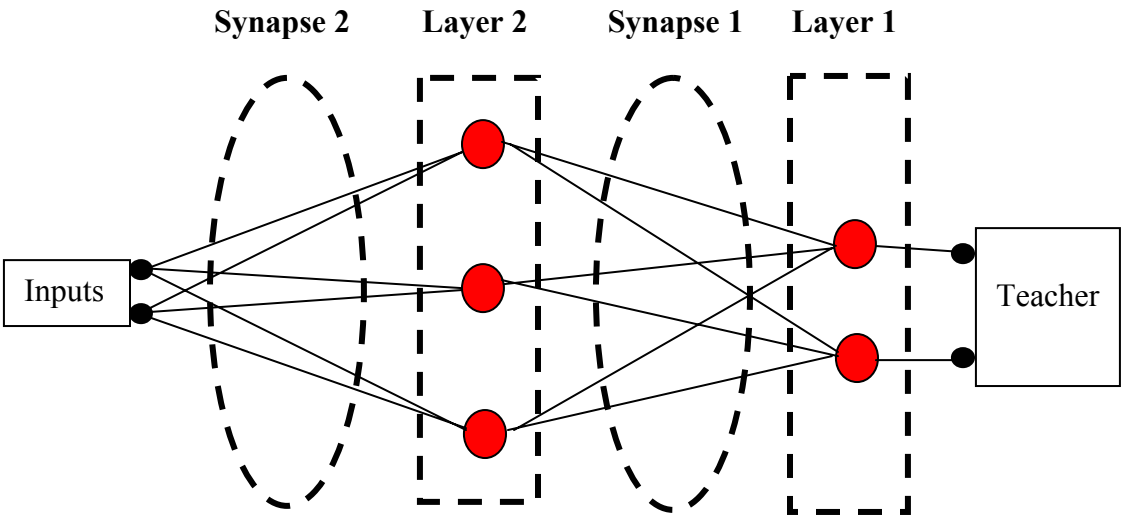

Figure 2.4 The JOONE Network Model

In the JOONE model, each network component (Layer or Synapse) computes weight changes in exactly the same way; i.e. all Layers use the same equations and all Synapses use

the same equations. In this way, extremely complex ANN structures can be created with relative ease. To demonstrate how this is accomplished, the following figure shows the calculations that each JOONE network component is responsible for:

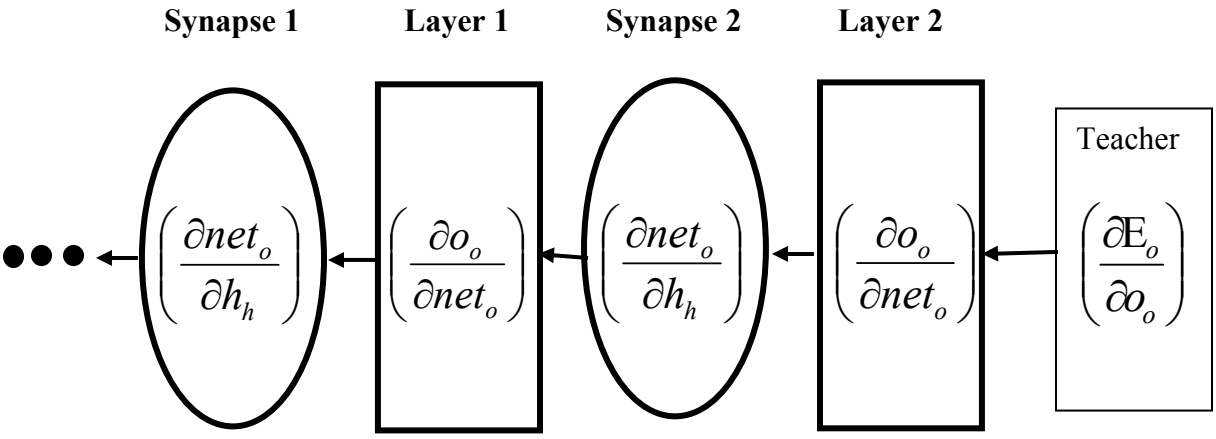

Figure 2.5 Object-Oriented Derivatives

In the above Figure o_o represents the current Layer's output from neuron o and h_h represents the previous layer's output from neuron h. Considering that all w_{ih}'s reside within Synapse 1 and also considering that the Δw_{ih} equation mentioned at the beginning of this section can be rewritten as the following:

$$\Delta w_{ih} = \eta\left(\frac{\partial E_o}{\partial w_{ih}}\right) = \eta\left(\sum_{o=1}^{O}\left(\frac{\partial E_o}{\partial o_o}\right)\left(\frac{\partial o_o}{\partial net_o}\right)\left(\frac{\partial net_o}{\partial h_h}\right)\right)\left(\frac{\partial h_h}{\partial net_h}\right)\left(\frac{\partial net_h}{\partial w_{ih}}\right)$$

It is not hard to imagine how the equation for Δw_{ih} could then be generalized into the following form (using *previousDerivative* as simply the backpropagated derivative of the network into Synapse 1):

$$\Delta w_{ih} = \eta\left(\frac{\partial net_h}{\partial w_{ih}}\right)(previousDerivative)$$

If every layer and synapse does its part all Δw's in the network can be calculated in a similar fashion, and all layers and synapses effectively retain the same equations.

This object-oriented philosophy will play a major role in the learning algorithms described later. In addition to that, the philosophy gives a much more generalized breakdown of the means by which example influence, flocking, and multi-task training calculations are carried out. The following three background sections are devoted to describing just these three concepts, but for the sake of understanding, the concepts will be described first with respect to how they are applied to the case of a single hidden layer network. Then they will be discussed in the context of the object-oriented philosophy for arbitrarily connected and/or arbitrarily sized ANNs.

2.5 Example Influence

Consider what would happen if during the middle of training every output for every example pattern of a particular network were suddenly given the exact same error of one for all remaining epochs. The end result would be that the derivative of the error with respect to each neuron's output would cease to have any relevance with respect to the weight changes.

Instead the weight changes would be based on the derivative of the network outputs themselves with respect to the individual weights. This means that the magnitude of the weight Change will be directly proportional to whatever the current derivative of the sigmoid function happens to be. In order to consider the implications of this, look at the following two figures of the sigmoid function and its derivative:

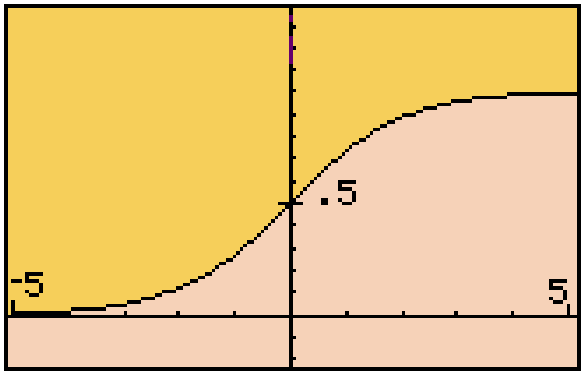

Figure 2.6 Sigmoid Function

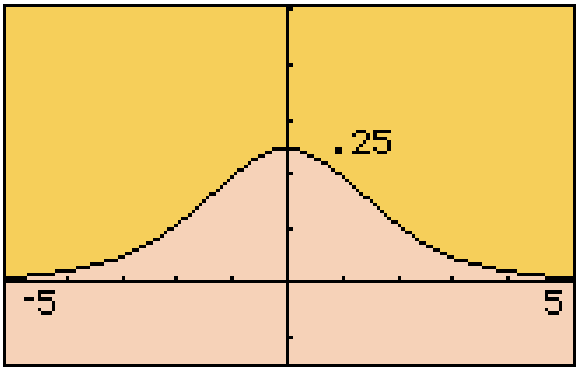

Figure 2.7 Sigmoid Derivative

Since the derivative of the output is clearly largest when the net input into a neuron is equal to zero, those examples that produce a net input close to zero are going to have the greatest affect on the next weight change. This idea is the basis for example influence.

When actually computing example influence in practice, it is not quite as easy as simply making the error for all outputs for all patterns equal to one, though this is the first and most important step. The catch is that, if trying to compute example influence for anything other than a single layer, the derivatives have to be backpropagated through some synapses, which contain positive and negative derivatives (unlike the sigmoid derivative).

These positive and negative values will effectively cancel each other out in the summation associated with the previous layer, even though a large negative derivative is every bit as influential as a large positive derivative. So in order to compensate for this and get a true measure of example influence, we must take the absolute value of the derivative of the net Inputs with respect to the previous layer outputs. With that said, here is the equation for the example influence of a particular pattern on a particular hidden layer neuron in a simple single hidden layer ANN:

$$EI_h = \left(\sum_{o=1}^{O} \left(\frac{\partial o_o}{\partial net_o} \right) \left| \frac{\partial net_o}{\partial h_h} \right| \right) \left(\frac{\partial h_h}{\partial net_h} \right)$$

$$EI_h = \left(\sum_{o=1}^{O} \left(o_o (1 - o_o) \right) \left| w_{ho} \right| \right) \left(h_h (1 - h_h) \right)$$

It must be noted that this is the raw example influence, in practice the example influences of all patterns are normalized to between 0 and 1 for each neuron in the layer. Another useful and informative parameter is Nef_h, which is the "Effective number of examples used to calculate the weight changes for a particular neuron". Nef_h is calculated in the following way:

$$Nef_h = \frac{\sum_{h=1}^{H} EI_h}{\max EI_h}$$

In the above equation $\max EI_h$ simply refers the largest example influence. At the start of training all neurons have an output of approximately .5 and as a result all the output derivatives are effectively maxed out at .25. Therefore at the start of training Nef_h will always be approximately equal to the total number of examples in the training set.

So now that we have the ability to measure the amount of influence that different example patterns have on a neuron as well as a measure of the effective number of examples in the training set currently influencing the neuron's weights, what can be done with this information to improve generalization? One interesting procedure that utilizes this information is known as flocking, which is described in the following section.

2.6 Flocking

Flocking, like backpropagation, is a gradient descent based algorithm, but unlike backpropagation, it is a self-organizing one. It is essentially a mechanism by which weights in a synapse are changed based on how a particular weight contributes to the net input becoming closer to a neuron's flocking mean net input. This change is weighted however, according to the influence of the current example pattern that is being used for training. The flocking mean net input itself is a weighted average (also weighted according to example influence) of all net inputs for all patterns. The trick that makes flocking work is that when the mean is calculated, the absolute value of the net input is used instead of the actual value, but when the derivative is calculated, if the net input for a particular pattern is negative (i.e. neuron output is < .5) then the inverse of the mean is used for flocking. This causes example patterns on one side of the neuron's hyperplane to flock towards one mean, and example

patterns on the other side of the hyperplane to flock towards a different mean. Since the weights are changing in such a way as to promote similar net inputs, the hyperplane is encouraged to settle into input space locations where it can be equally influenced by many examples and hence promote hyperplane sharing among multiple decision regions. This is exactly the hyperplane-sharing property we were looking for in order to improve generalization.

One large problem to overcome with this process however is the fact that it does reduce varience so well. In most instances it will be impossible for the network to ever reach a state where all the net inputs are exactly equal to the flocking mean. That is unless all the weights are at or close to zero, which results in all the net inputs being at or close to zero. As a result of this basic property, weights are as a general rule gradually reduced over time as the gradient descent flocking process is continued.

With this problem in mind, small amounts of flocking exhibit some useful properties when applied to ANNs that have already been partially trained using standard backpropagation. Before going into any further detail, let us take a look at the actual flocking equations used in the learning algorithms of this research. Following is the formal definition of the equation used to calculate the flocking mean net input for a particular neuron $_h$ of a hidden layer (this equation uses the normalized example influences):

$$mean_h = \frac{\sum_{j=1}^{P} \left(EI_{jh} \right) \left| net_{jh} \right|}{Nef_{jh}}$$

Now that the flocking mean is defined, all the parameters necessary to define the flocking error equation have been formally addressed and we are ready to proceed with the derivation of the Δw equations that will be used to update weights in out gradient descent based flocking algorithm. The following definition of Ψ_{ph} is our flocking error equation. Note that it a simple sum squared error design similar to that of the error equation commonly used for backpropagation. The subscript p has been included on purpose to emphasize the fact that all parameters are relative to a specific example pattern except for the parameter $mean_h$, which is calculated only once for all patterns during each epoch.

$$\Psi_{ph} = \frac{1}{2}\left(EI_{ph}\right)\left(net_{ph} - mean_h\right)^2$$

Since our objective is to lower the over all flocking error, the logical thing to do is to start by taking the derivative of Ψ_{ph} with respect to net_{ph}. This derivative will eventually be used to find the derivative of the error with respect to individual weights. Remember however that a net_{ph} term resides in the $mean_h$ term, so before proceeding, the definition of $mean_h$ is inserted into the equation as follows:

$$\Psi_{ph} = \frac{1}{2}\left(EI_{ph}\right)\left(net_{ph} - \left(\frac{\sum_{j=1}^{P}\left(EI_{jh}\right)\ \left|net_{jh}\right|}{Nef_{jh}}\right)\right)^2$$

Now we can better see the task at hand, but this is still not a pretty form for taking a derivative. In order to make things cleaner let us next move all the terms associated with the current pattern out from behind the summation, and update the definition of the summation accordingly. Doing so gives us the following:

$$\Psi_{ph} = \frac{1}{2}\left(EI_{ph}\right)\left(net_{ph} - \left(\frac{\left(EI_{ph}\right)\ \left|net_{ph}\right|}{Nef_{ph}} \right) - \left(\frac{\sum_{j=1, j\neq p}^{P} \left(EI_{jh}\right)\ \left|net_{jh}\right|}{Nef_{jh}} \right) \right)^2$$

To make things cleaner still, we define the terms α_{ph} and β_{ph} which will represent individual constants that will have to be recalculated for each pattern for each neuron:

$$\alpha_{ph} = \frac{EI_{ph}}{Nef_{ph}} \qquad \beta_{ph} = \frac{\sum_{j=1, j\neq p}^{P} \left(EI_{jh}\right)\ \left|net_{jh}\right|}{Nef_{jh}}$$

After substitution of α_{ph} and β_{ph} our original flocking error equation takes on the following form:

$$\Psi_{ph} = \frac{1}{2}\left(EI_{ph}\right)\left(net_{ph} - \alpha_{ph}\left|net_{ph}\right| - \beta_{ph}\right)^2$$

Because we can consider α_{ph} and β_{ph} constants when taking the partial derivative with respect to net_{ph}, we are finally ready to take that derivative as follows:

$$\left(\frac{\partial \Psi_{ph}}{net_{ph}}\right) = \left(EI_{ph}\right)\left(net_{ph} - \alpha_{ph}\left|net_{ph}\right| - \beta_{ph}\right)\left(1 - \alpha_{ph}\right)$$

Now the final step is a bit tricky, but remember that if the net_{ph} is less than zero we want the net_{ph} to flock toward the negative of the mean. Consequently both the α_{ph} and β_{ph} terms flip sign in the equation above as a result of this. Taking this into account and with a little algebra we can then arrive at the final form of our equation for the derivative of flocking error with respect to net input:

$$\left(\frac{\partial \Psi_{ph}}{net_{ph}}\right) = \begin{cases} \left(EI_{ph}\right)\left(\left(1-\alpha_{ph}\right)net_{ph} - \beta_{ph}\right)\left(1-\alpha_{ph}\right) & if\left(net_{ph} \geq 0\right) \\ \left(EI_{ph}\right)\left(\left(1-\alpha_{ph}\right)net_{ph} + \beta_{ph}\right)\left(1-\alpha_{ph}\right) & f\left(net_{ph} < 0\right) \end{cases}$$

With these equations at our disposal, it is now a simple matter of multiplying by the derivative of net_{ph} with respect to the individual synapse weights, which amounts to simply multiplying by the previous layer's outputs (just as in basic backpropagation). For

completeness here is the definition of that weight change as it pertains to a single weight in the synapse associated with the hidden layer of a single hidden layer network:

$$\Delta w_{ih} = \eta \left(\frac{\partial \Psi_{ph}}{net_{ph}} \right) \left(\frac{\partial net_h}{\partial w_{ih}} \right)$$

All other weights in the synapse and corresponding hidden layer would be calculated in a similar fashion.

For a good demonstration and better understanding of the effects of flocking, the reader is referred to Appendix 1 which provides some supplemental research that examines the flocking equation in the context of a straightforward two input, single hidden neuron, single output neuron ANN using a simple but very revealing training set. Also a modification to the flocking algorithm is described in section 3.1 that allows for fast and effective movement of hyperplanes into positions that efficiently divide the input space into intuitive decision regions.

2.7 Multi-task Training

A given ANN almost always learns faster if it already has prior knowledge that relates in some way to the knowledge contained in the training set. Multi-task training is a very effective way to instill such knowledge into a network prior to training on the training set. The most standard form of multi-task training usually involves dividing up the task to be learned into smaller sub-tasks. Then each sub-task is trained for a certain number of epochs

(or to within a specified error limit). Following this certain weights in the network are reset to a low random value before the next sub-task is trained. One full sequence of training on each sub-task in this fashion is commonly referred to as a single meta-epoch. After a specified number of meta-epochs have completed the ANN is trained using the actual training set. The very attractive thing about multi-task training is that it almost always results in significantly faster training times than standard backpropagation training on an untrained network.

The reason this process works so well is because the sub-tasks are as a rule much simpler than the primary task (and therefore easier to learn), but at the same time these sub-tasks are providing valuable generalizable information about the primary task. If the sub-tasks were more complex than the primary task, or if they were completely unrelated to it, then it would be very unlikely that multi-task training would prove useful at all.

The real issue when it comes to multi-task training is how to come up with the sub-tasks. We already know what it is we want to learn, but how do we create sub-tasks that are simple yet capture as much of the information contained in the primary task as possible? Many approaches to this problem exist. A common example of multi-task training uses the AND and OR functions as sub-tasks to pre-train a neural network for the more complex XOR function. The reason this process works so well is the fact that the AND and OR functions each require a single (but different) hyperplane, and it just so happens that both these hyperplanes together provide the solution to the XOR function. As a general rule, tasks requiring fewer hyperplanes are easier to solve than those requiring more hyperplanes. So when we say that a sub-task is related to a primary task, this usually just means that the

solution to the sub-task contains one or more hyperplanes in common with the primary mapping task.

With the XOR solution, prior knowledge was needed in order to know that the AND and OR functions would provide good sub-tasks for multi-task training. One way of creating sub-tasks without having such prior knowledge is to simply use subsets of the example patterns contained in the primary task's training set as sub-tasks. One issue with this approach is that some sub-sets would prove to be much better sub-tasks than other subsets, so how do we choose which subsets to use? Another approach is to create our own subtasks through some type of logical analysis of the primary task. This too has issues associated with it, like what type of analysis to use. In the learning schemes described later, we will use both of these approaches to subtask creation and the details of how these issues were resolved will be addressed. One algorithm that will be particularly useful in this endeavor is an approach called split flocking. The details of split flocking are addressed in the next section.

3. Intermediate Learning Algorithms

The intermediate learning algorithms described in the following two sections are really the heart and soul of this research. At least one of these two algorithms will form the basis for each of the final learning algorithms discussed. Split flocking, the algorithm put forth in the first of the two sections, is a method that meticulously divides up the input space using hyperplanes. Convergence training, the algorithm put forth in the second section, is a method that seeks to converge hyperplanes in order to exploit inherent consistencies or symmetry within the input space. Both these algorithms were fashioned with the end goal of improving ANN generalization.

3.1 Split Flocking

Split flocking is an algorithm that utilizes backpropagation, flocking, and low-level multi-task training in order to "split-up" the training set into logical subsets. At heart it is very much a clustering algorithm, however unlike other basic clustering algorithms, split flocking utilizes supervised training to initialize the network for self-organized clustering. Further, split flocking has the ability to actually modify the weights of a particular neuron such that the hyperplane it represents is the mechanism by which the subset clusters are determined. In other words, instead of clustering examples into subsets and training a hyperplane to split these subsets (as would be the case using most classic forms of clustering), the split flocking algorithm places the hyperplane itself through flocked training and then extracts the subsets by analyzing the hyperplane's placement.

Let us first examine the split flocking algorithm from the clustering point of view. The first step is to split the training set into two clusters, then each of those clusters is split into two clusters, and so on until a suitable "stop splitting" criteria is met for all resulting subclusters. The outcome of this process is a very straightforward and informative binary tree representation of our data, which gives us in many respects, a hierarchical roadmap of generalization within the input space. This in and of itself is still not much more than what would be provided using a classical form of clustering such as k-means. Two things that make split flocking stand out from classic forms of clustering are the fact that supervised training (i.e. backpropagation) is used to initialize the self-organization and the fact that split flocking places neuron hyperplanes in the input space in order to identify clusters.

The supervised backpropagation portion of the algorithm is used to initialize neuron weights in a way that moves a neuron's hyperplane toward an input space location that provides an error reduction common to a majority of patterns in the training set. This initialization process is needed in order to give each output neuron a significant output response, which can then be used in the flocking algorithm. The low-level multi-task training simply refers to the idea that only example patterns within the cluster being split are used in the split training. The flocking algorithm used is however actually a slightly modified version of the core flocking algorithm discussed above.

Everything about the new flocking algorithm is the same except for the reflection point. In the core flocking algorithm, the reflection point is the zero crossing; i.e. if a particular pattern's net Input is negative, it flocks to the negative mean, and visa-versa if the pattern's net input is positive. In split flocking however, instead of using the zero crossing,

the weighted average of all net Inputs is used as the reflection point. This weighted average is the algebraic mean of the net inputs (not the mean which is calculated using the absolute values). With this one exception, the remainder of the algorithm is exactly the same; nevertheless this one change allows the clusters to be split in a logical fashion regardless of where the hyperplane originally starts out in the input space.

The following split flocking equations reflect the changes made to the core flocking algorithm:

$$avg_h = \frac{\sum_{j=1}^{P} \left(EI_{jh}\right) \left(net_{jh}\right)}{Nef_{jh}}$$

$$\left(\frac{\partial \Psi_{ph}}{net_{ph}}\right) = \begin{cases} \left(EI_{ph}\right)\left(\left(1-\alpha_{ph}\right)net_{ph} - \beta_{ph}\left(1-\alpha_{ph}\right)\right) & if\left(net_{ph} \geq avg_h\right) \\ \left(EI_{ph}\right)\left(\left(1-\alpha_{ph}\right)net_{ph} + \beta_{ph}\left(1-\alpha_{ph}\right)\right) & if\left(net_{ph} < avg_h\right) \end{cases}$$

To see the effects of using split flocking on a data set, refer to section 7, which presents the implementation and results of a learning method, based almost solely on the split flocking algorithm. The other learning algorithms that will be discussed all use split flocking in combination with the remaining intermediate algorithm, convergence training, which is addressed in the following section.

3.2 Convergence Training

The idea of convergence training revolves around using multi-task training in order to get hyperplanes in the network to converge to locations that exploit inherent symmetry in the input space. This goal is accomplished through using the binary tree data structure produced in the split flocking algorithm along with a form of multi-task training that actually produces its own subtasks (as opposed to the form of multi-task training used in split flocking that simply uses a subset of the training set). The new subtasks produced are each representative of one of the clustering splits catalogued in the binary tree data structure. When the multi-task training takes place, the hyperplanes being taught each try to accommodate as many of the clustering splits as possible, and thus exploit much of the inherent symmetry within the input space.

We will examine this method in more detail by first taking a closer look at the details of the binary tree data structure and how it is used to create the subtasks that will be used in the multi-task training. Recall that the tree structure created by split flocking is composed of nodes, where each node represents a set of training patterns. The root node is the set of all training patterns in the training set, and each additional node at each successively lower level of the tree structure represents a subset of that training set. The subsets within the two child nodes of any parent node are always mutually exclusive and the union of the two child nodes will always produce the exact set of patterns contained in the parent node. Using this model a single subtask is created from each non-leaf node in the data structure. The example patterns in each subtask are "temporarily modified" during this part of training. The difference between the example patterns of the training set and the example patterns of the

subtasks are the patterns' targets. All the patterns for each subtask are assigned a single target output, and this target output depends solely on which of the two child nodes the current example pattern is found in. In other words, all example patterns for a particular parent node that are repeated in the first child node receive the same target (of approximately zero), and all example patterns repeated in the second child node receive the same target (of approximately one). In this fashion subtasks are created which can then be used to train the network to recognize the splits represented in the binary tree data structure.

At this point one might note that each split represented by each of the newly created subtasks is in fact already represented in the network at the conclusion of split flocking. So what then is the point of going through all this trouble to create subtasks that the network already has hyperplane representations for? The answer to this question lies in the fact that simply learning these subtasks is not what we are setting out to do. Our goal is rather to utilize these subtasks in a form of training that will exploit symmetry within the input space (i.e. similarities across all the inputs), and that's where the real multi-task training comes in.

Once our subtasks have been created, the convergence process finally begins. Just as in other forms of multi-task training we will use our subtasks to pre-train the network. Unlike most forms of training, however, we will be training on multiple subtasks at once. There are a number of conceivable approaches that could be taken to choosing which subtasks to train in unison (many of which would likely achieve positive results), but we will focus on one particular approach because it will be the one used in the learning algorithms of sections 5 & 6. In this approach, the subtasks corresponding to a particular level of the binary tree are all trained in unison. So for example, the subtask corresponding to the root

node would be trained all by itself, but the subtasks corresponding to the root node's two children nodes would both be trained in unison. Training subtasks in unison for a particular input pattern is simply a matter of not ignoring the output neurons that correspond to each of these subtasks ("ignoring" the output neurons that correspond to the other subtasks is accomplished by having the other neurons backpropagate an error of zero).

The learning scheme used during a session of convergence training is a combination of backpropagation and flocking (just like split flocking). In convergence training the neurons are trained on the subtasks using standard backpropagation either for a certain specified number of epochs or to within a certain target SSE. After that, the hidden layer neurons being convergence trained are trained for a short period of time using the flocking algorithm for a specified number of epochs. This final flocking stage helps to assure that the hyperplane settles into a symmetric spot that influences a large a number of examples. Future research should allow us to develop a refined control of this process.

Since each subtask represents a particular clustering split, training these subtasks in unison causes the formation of a pseudo-subtask that will be most easily solved when hyperplanes can be shared among the subtasks that the pseudo-subtask is composed of. This sharing of hyperplanes is exactly the exploitation of symmetry that was mentioned earlier as the goal of convergence training. Convergence training plays critical role in the upcoming learning algorithms of sections 5 & 6. For a detailed analysis of how convergence training behaves on an experimental data set, please refer to these sections.

4. Problem Statement

In this section we will define a data set for the purposes of testing our learning algorithms. When creating a data set there are a few things to keep in mind with respect to our particular situation. One, we want to start simple. The more complex the data set is the harder it is to analyze the affects of a learning algorithm on that data set. Two, we want to be able to easily analyze the placement of hyperplanes in the input space. This means restricting the inputs to two dimensions (with the hope that similar results can be achieved for data sets that are n-dimensional). Finally the last thing to keep in mind is symmetry (i.e. consistencies). We want to analyze the learning algorithm's ability to exploit consistencies in the input space, so make sure the data set has some symmetry to exploit. With these restrictions in mind consider the following ideal data set:

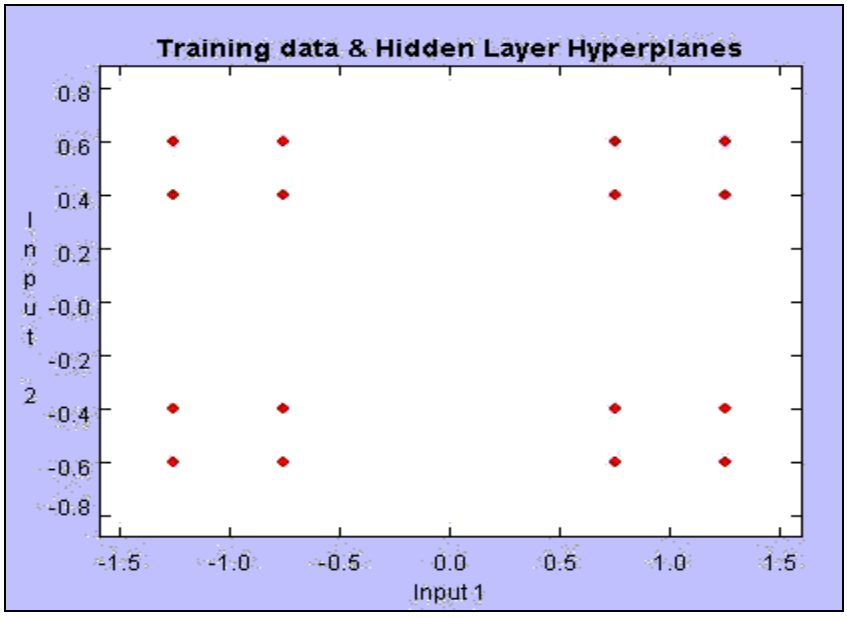

Figure 4.1 The Ideal Data Set

In this ideally contrived data set there are four distinct clusters centered around the points (1,.5), (1,-.5), (-1,.5), and (-1,-.5). Each of these clusters is then split into four distinct points in a similar fashion but on a smaller scale. Imagine also that each of these sixteen points corresponds to a distinct class or output in the problem space, i.e. when given the coordinates corresponding to a single pattern, the output for that class should go high, while the outputs associated with the other fifteen classes should go low.

In order to simulate the fluctuations associated with any real world data, we next create what will be our actual example pattern set by adding random values to each of the points in the original ideal data set. Specifically, for each class three patterns are created in the final set that have the exact same associated target output as the original pattern but each new pattern's input values are modified by adding a random value between plus and minus 1 tenth. Doing so gives us the following data set, which we will be using to analyze each of our learning algorithms.

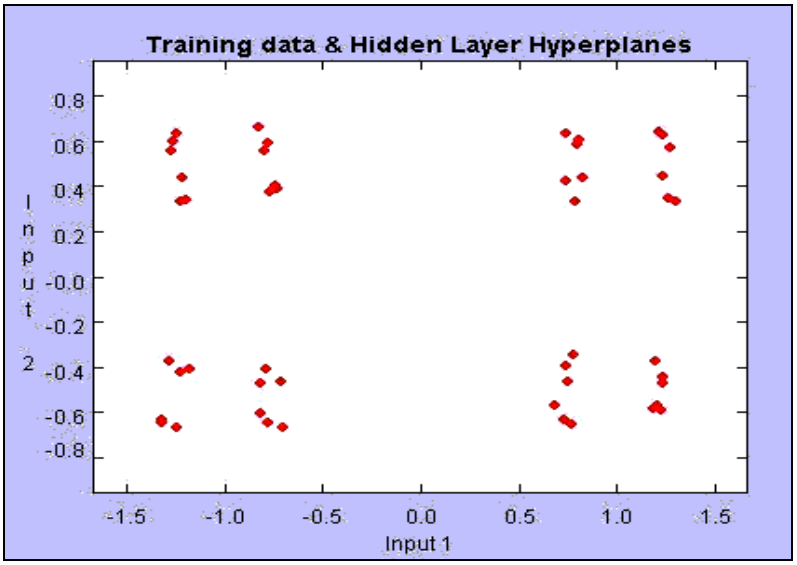

Figure 4.2 The Actual Data Set

The only thing left to do is divide our data set up into a training set and test set. Normally this is done by randomly selecting a certain percentage of the data set. In our contrived pattern set however, we select a training set by randomly choosing one of the three example patterns associated with each of the sixteen distinct target outputs. The remaining thirty-two example patterns are then designated as the test set. Here are the final training and testing sets that will be used in all the following learning algorithms:

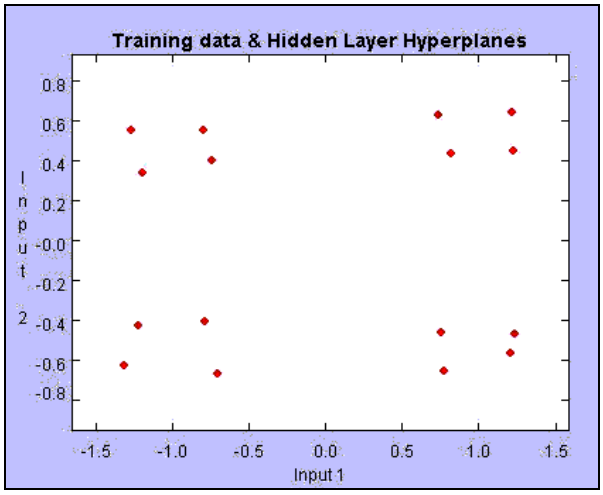

Figure 4.3 The Training Set

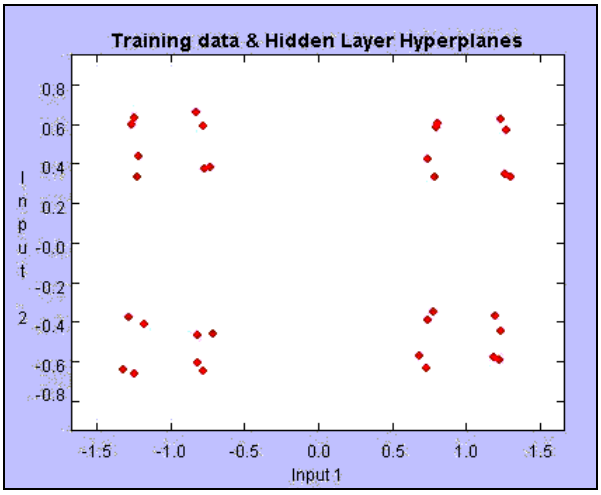

Figure 4.4 The Test Set

Normally the training set would have more example patterns than the test because the idea is to use as much data as possible during training. Since our goal is to analyze the networks abilities on unseen data however, we have chosen to do just the opposite in order to get good measures of generalization capability. With our problem defined, the next section begins discussion of the learning algorithms and their performance on this data set.

5. Bringing It All Together (Primary Algorithm One)

This section is devoted to the first primary learning algorithm that is the focus of this paper. This algorithm combines the intermediate learning algorithms discussed in section 3 and is the base algorithm used to derive the learning algorithms discussed in sections 6 and 7. Unless otherwise stated, it is assumed that further mention of ANN refers to a single or multi-hidden layer feed-forward network consisting of neurons using sigmoid transfer functions and utilizing batch training. In all cases input data is assumed normalized between plus and minus one, target output data is assumed normalized between zero and one, and weights assumed to have been initialized prior to the first epoch of training to a low random value (usually to between plus/minus one one-hundredth).

5.1 Methods of Approach

This approach uses the training and testing sets defined in section 4 to analyze the generalization capabilities of the first primary learning algorithm as compared to that of normal backpropagation (the control algorithm) on identical network architectures. Since the first primary algorithm actually implements a dynamic network architecture (utilizes an expanding hidden layer), the control network architecture was selected post training of the primary algorithm. However, for clarity the control network will be defined before moving into the results of the experimentation.

Recall that both the split flocking algorithm and the convergence training algorithm are iterative in nature. During each iteration of the split flocking algorithm a new level of the

binary tree data structure is created, and during each iteration of the convergence training algorithm a new level of the binary tree data structure is consumed. This parallel relationship between the two intermediate algorithms is utilized in the primary algorithm such that each iteration of the primary algorithm results in a single iteration of the split flocking algorithm and a single iteration of the convergence training algorithm.

The network architecture of the primary algorithm implements a dynamic hidden layer. At the start of training the network architecture contains but a single hidden layer neuron, and the output layer contains one neuron for each output specified in the data set (our data set has 16 outputs and therefore 16 different output neurons). During each iteration of the primary algorithm, the number of neurons in the hidden layer is both increased and decreased; however all hidden layer neurons in existence at the end of each iteration retain a permanent position in the network. These permanent neurons become locked, i.e. the neuron's bias and all incoming synapses become locked (unchanging) for the remainder of the learning process.

From this point forward we will refer to a particular iteration with a corresponding phase number (the very first iteration of the algorithm will be considered phase 1, and each iteration after that will be phase 2, phase 3, ect.). The process of iterating through all the phases will be referred to as "hidden layer training". With these preliminaries in mind, we next examine the algorithm steps that take place during a single iteration of hidden layer training:

HIDDEN LAYER TRAINING for PRIMARY LEARNING ALGORITHM:

STEP 1: Add neurons to the hidden layer. The number of neurons to add corresponds to the number of clusters in the binary tree that are available for splitting (for phase 1 there is a single cluster composed of all example patterns in the training set)

STEP 2: Use split flocking to separate each cluster with a hyperplane

STEP 3: Analyze the hyperplane produced by each newly added hidden layer neuron and update the binary tree accordingly. Remove all neurons that were added in STEP 1

STEP 4: Use the new binary tree level produced in STEP 3 to create new convergence training subtasks and assign each new subtask to its own output layer neuron (if not enough output layer neurons exist, add temporary output neurons)

STEP 5: Ignore the output of output layer neurons representing classes unrelated to each STEP 4 subtask (remember that each STEP 4 subtask is composed of a subset of example patterns, therefore this step results in each example pattern having its own set of ignored output neurons)

STEP 6: Add a hidden layer neuron and do convergence training using the newly created subtasks. If the specified target SSE is not reached at the end of a max number of training epochs, repeat this step again leaving all weights of previous hidden layer neurons added in this step unlocked.

STEP 7: Remove any redundant hidden layer neurons produced in STEP 6 and any temporary output layer neurons produced in STEP 4

STEP 8: If there are clusters that can be split, proceed to STEP 1 and begin the next phase training. Otherwise all hidden layer training is complete

Once the above hidden layer training has completed, the final stage of the primary learning algorithm begins. This final stage consists of keeping all hidden layer weights locked and training just the output layer neurons using all of the example patterns contained in the original training set. It is not until this final stage of training that we can even compare the primary algorithm to that of normal backpropagation. Before this point all measures of training and testing error have large fluctuations due to flocking, randomization, and the addition/removal of neurons.

After the complete training of an ANN using the primary algorithm on the data set specified in section 4, we are left with a solution network architecture that contains six hidden layer neurons. The next step in the experimentation process is to create a control network with which to compare our findings. In order to provide the best comparison possible, our control ANN will also contain six hidden layer neurons. Also the primary ANN (in its final stage) will train for the same number of epochs using the same learning rates as the control ANN. Unless otherwise stated in the algorithm, all other parameters of the two networks will be identical.

In the following results section, pertinent information regarding the training and ultimate generalization capabilities of both the control and the primary ANN will be

displayed, compared, and discussed in detail. Keep in mind however, that much more research must be done to fully understand and develop these ideas. The testing done in this paper is really just a promising indicator that research should be continued.

5.2 Results & Analysis

This section serves not only to relay the findings of this research, but also to provide a visual supplement to the algorithm descriptions above. We will start by visually examining the results of the control network during a typical training session. Then we will visually trace the progress of the primary algorithm on the specified data set by viewing the state of the network at the conclusion of each consecutive phase. After that we will take a look at how the control ANN faired on the data set during multiple runs where the only variable is the random weight initializations. We will then conclude by applying the same multiple run technique to the ANN using the primary learning algorithm, and by contrasting the results of the multiple runs from both ANNs, we hope to gain some positive feedback and additional insight into our quest for improved ANN generalization.

We begin by examining how a typical backpropagated neural network behaves on a typical data set. Among other things, the principle behaviors we will be looking for are how the hyperplanes get utilized during the training process and how the network performs on unseen data. Recall that our control network specifically consists of six hidden layer neurons for better comparison to the primary algorithm later on. The following diagram is a representation of the network architecture used for the control ANN. Note that all the synapses are colored red, this means that all the weights are in an unlocked state. Future

network architecture diagrams containing locked weights will color such synapses black. Note also that there are two input neurons and sixteen output neurons corresponding exactly to the requirements of the data set (in general all ANNs must adhere to this property):

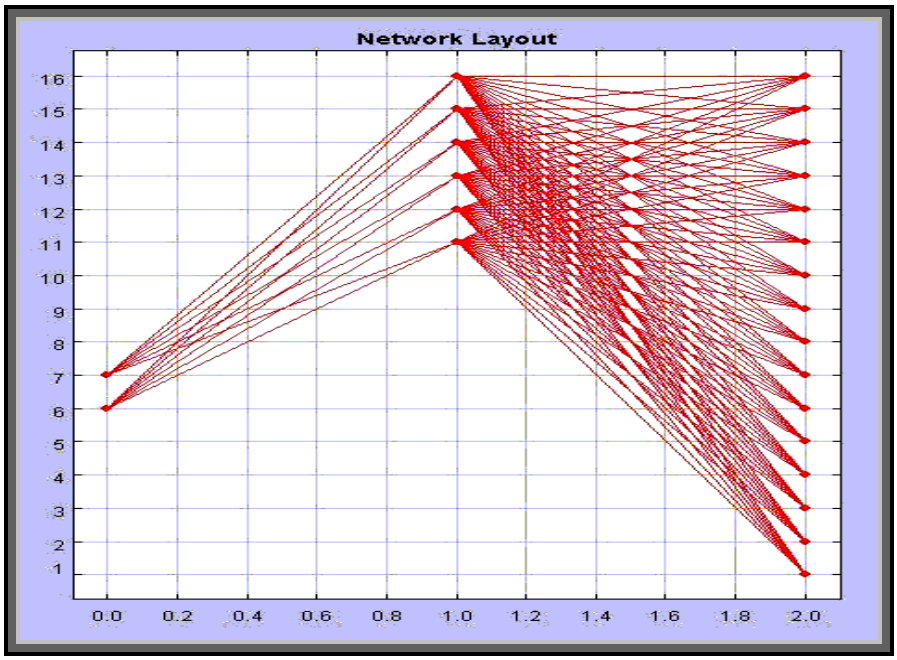

Figure 5.1 Network Architecture of Control ANN

Using this architecture the control ANN is batch trained for a total of 10,000 epochs using standard backpropagation. After the first 6,000 epochs the learning rate is raised slightly to increase convergence. This exact training scheme is used for all control network training sessions as well as for all final training sessions of the ANNs using the primary learning algorithm. Trying to relay every implementation detail of these training sessions would be far to cumbersome considering the 1,000s of lines of code involved, so for exact details of all network training processes please refer to Appendix 2 which contains further

implementation details at the code level and also provides links to where you can download a copy of the actual code used to conduct this research.

At the conclusion of the training session described in the paragraph above, we are left with the following hyperplane representations of the six hidden layer neurons within the control network:

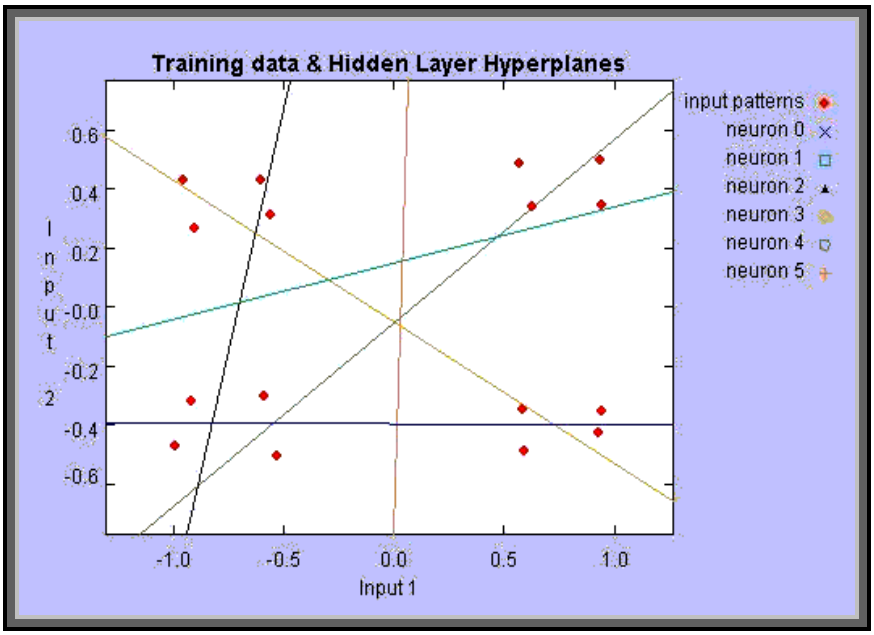

Figure 5.2 Control Network Hyperplanes

We can see from this figure that ANNs trained using standard backpropagation appear to find very non-intuitive solutions. It is very hard to tell exactly how the network is solving the problem by looking at the hidden layer hyperplane representations shown above. This is due in part to an ANN's ability to solve problems in a very non-linear fashion and search the solution space for solutions that would otherwise never really be conceived of.

Next we examine the sum-squared-error graph produced during the training process.

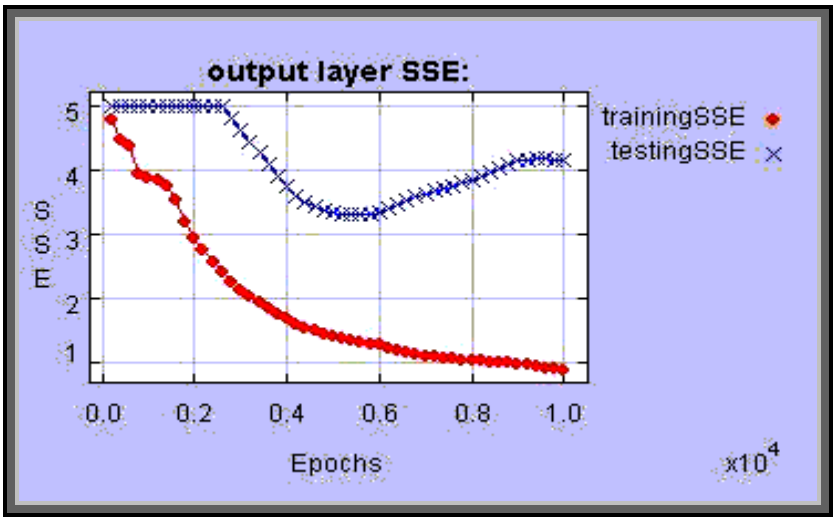

Figure 5.3 Control Network SSE

In the above graph we can see the sum-squared-error resulting from both the training and the testing set during the entire 10,000 epochs of the network's training process (In this and all remaining SSE graphs, the max SSE displayed is 5 in order to better view the pertinent results). Just as you would expect, the test set error decreases along side the training set error until the network starts to enter the memorization stage at which point the test set error begins to increase. Remember this is only the result gained from one particular random weight initial condition, and as previously stated, starting the network with different random weights can give vastly different results. Later in this section we will provide the results of multiple runs, which will give us a much better idea as to the behavior of backpropagated ANNs in the general case. Next we will again trace the behavior of a single ANN, but this time train it using the primary learning algorithm.

According to the hidden layer learning scheme described for the primary learning algorithm the network begins with one hidden layer neuron, which is added in STEP 1. This single neuron is then used for split flocking and convergence training. For the duration of Phase 1 all network synapses remain unlocked and the network architecture of the primary ANN during this phase resembles the following diagram.

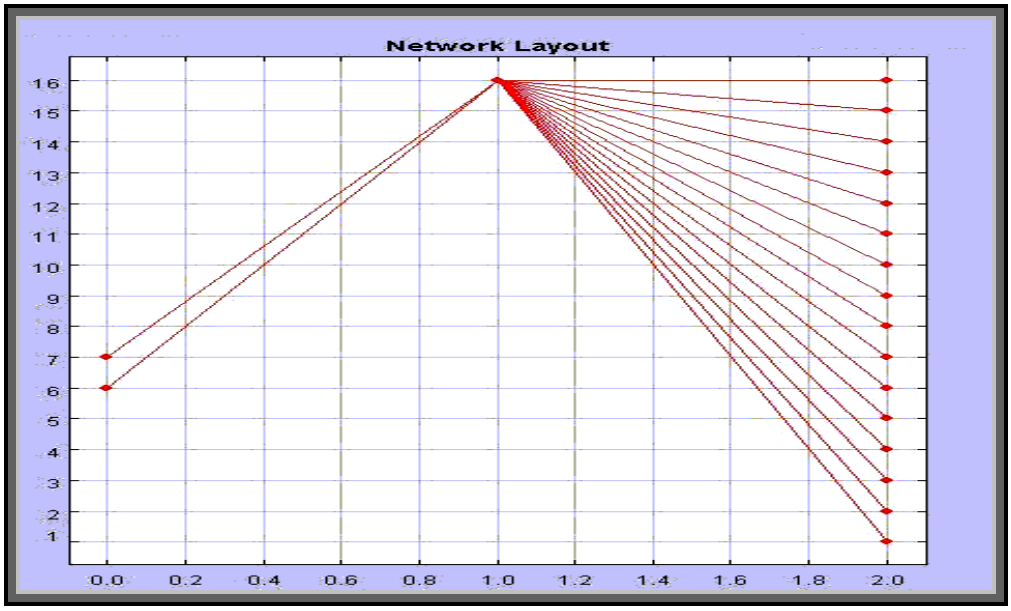

Figure 5.4 Network Architecture for duration of Phase 1

Since there is only one neuron in the split flocking phase, the convergence training phase ends up being redundant. Both the flocking and convergence algorithms train on the entire data set, and they both train to learn the same thing. As a result the hyperplane diagrams at the end of each algorithm's execution are very similar as the following figure displays:

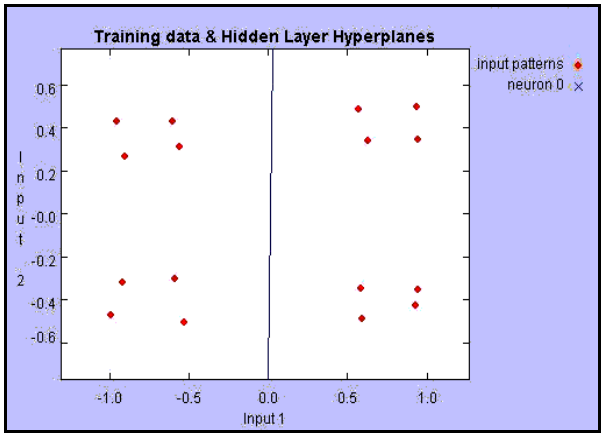

Figure 5.5
Phase 1 post split flocking

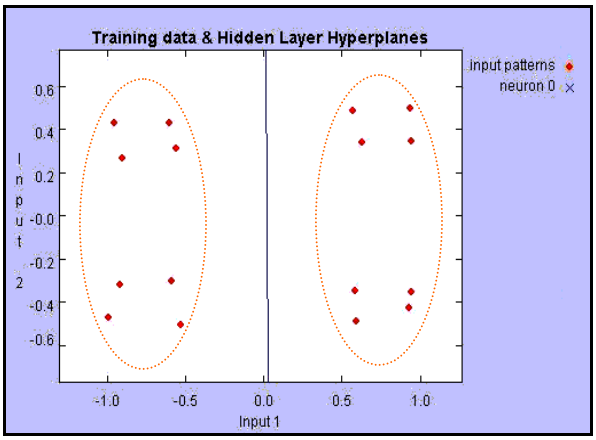

Figure 5.6
Phase 1 post convergence training

The dotted ovals in Figure 5.6 show that phase 1 has resulted in a very intuitive first division of the input space with respect to the example patterns. As far as generalization is concerned, this seems like a very good start. In the next phase, the synapses of neuron 0 will become locked and two new neurons will be added to the hidden layer. The exact same splitting process will then be carried out for all the patterns in each of the two ovals. Then we will try to converge (i.e. merge) these two splits (represented by the two newly added neurons) using the convergence algorithm. Remember that even though all the output layer weights are unlocked during split flocking, most of these outputs are being ignored. Whether a synapse is actually learning depends on whether the current training pattern belongs to the subtask that a particular output neuron represents. The first of the following two figures depicts the network architecture during the stage where the first of the two newly added

neurons is being split flocked. The second of the two figures depicts the state of the hyperplanes at the conclusion of the phase 2 split flocking.

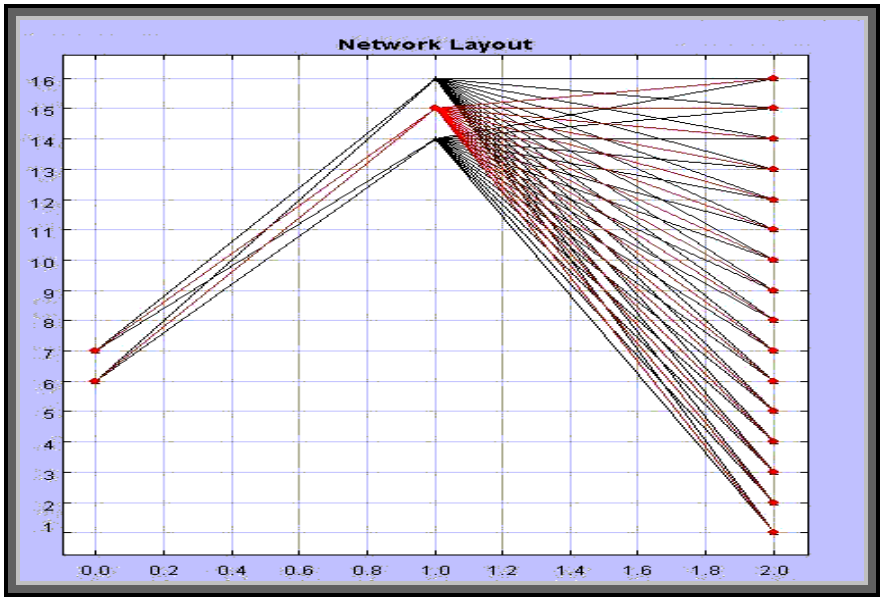

Figure 5.7 Phase 2 split flocking neuron 1

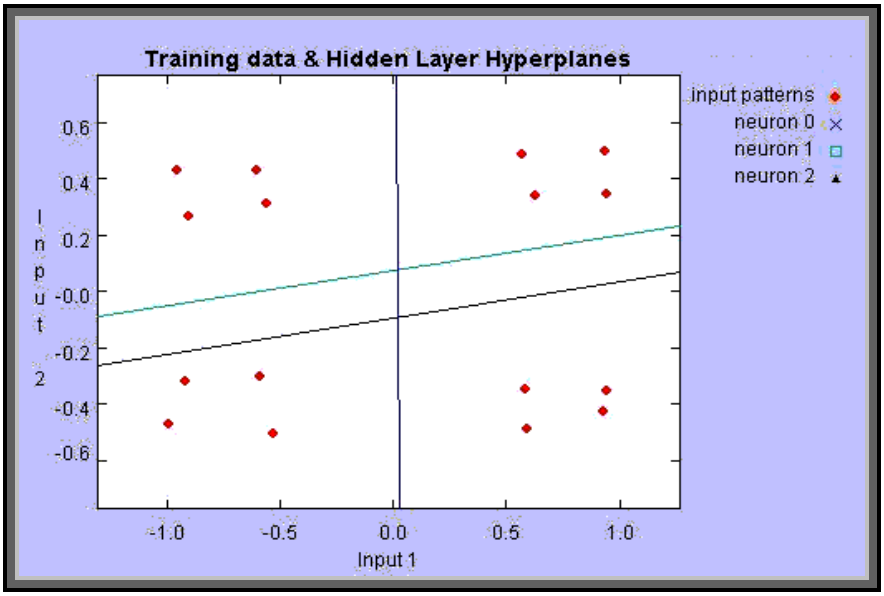

Figure 5.8 Phase 2 hyperplanes post split flocking

These figures depict the network architecture and final state of the hyperplanes from the phase 2 convergence training.

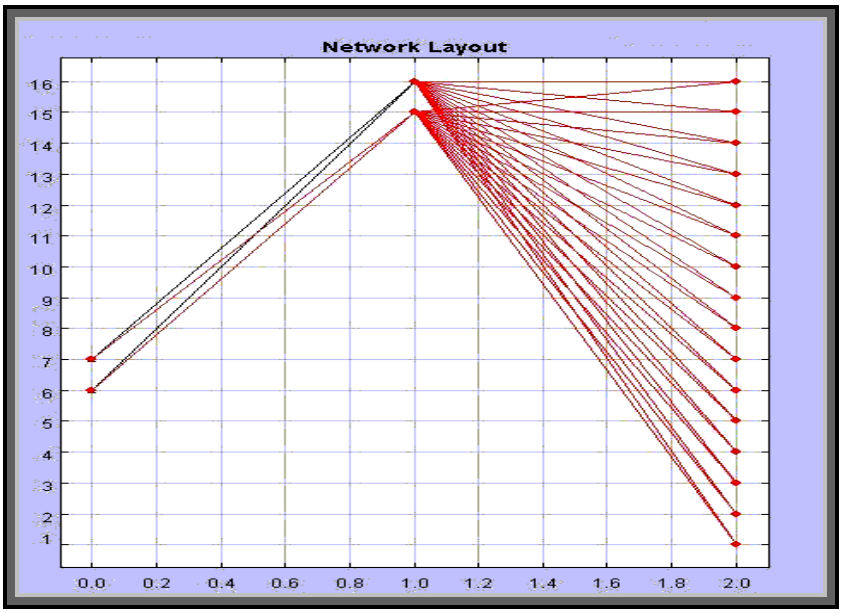

Figure 5.9 Phase 2 convergence training

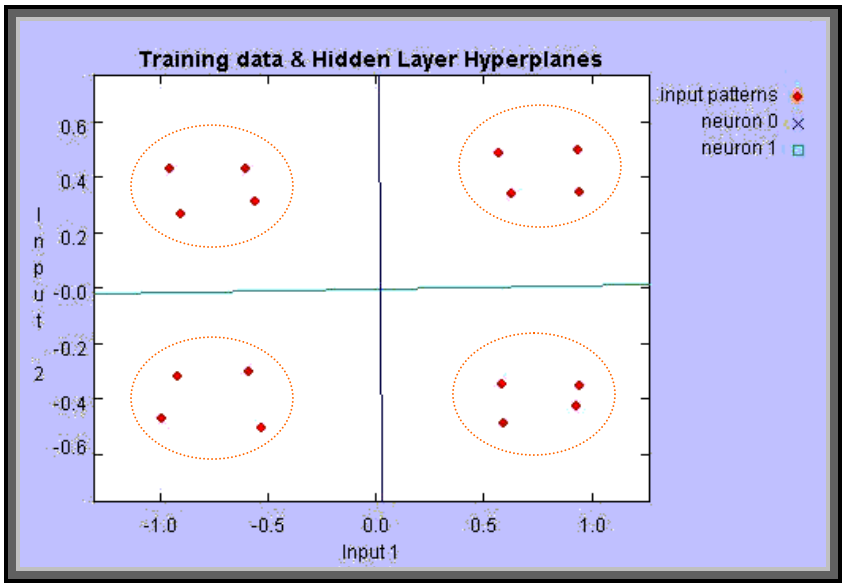

Figure 5.10 Phase 2 hyperplanes post convergence training

The dotted ovals in Figure 5.10 depict the clusters that will form the next level in the binary tree below the clusters depicted in Figure 5.6 (recall that these are the clusters that form the subtasks used for multi-task training). It can again be easily seen that the learning algorithm has provided some very intuitive splits within the input space. We are now ready to move on to phases 3 & 4. After the fourth phase, as you have probably guessed, each cluster will contain but a single example pattern, and the hidden layer training process will at that point be halted. The following figures provide a visual summary of the remainder of this training process (Figures 5.19 & 5.20 below represent the final hidden layer outputs and the subtasks that were created during split flocking respectively, also the following hyperplane plots are not to scale).

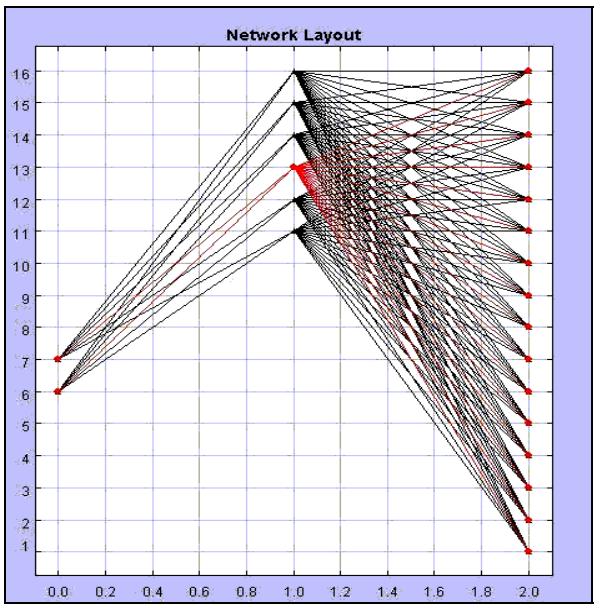

Figure 5.11
Phase 3 split flocking neuron 3

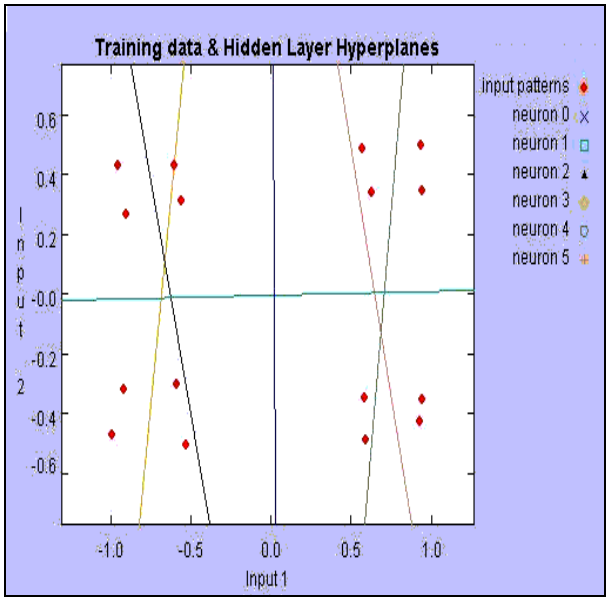

Figure 5.12
Phase 3 hyperplanes post split flocking

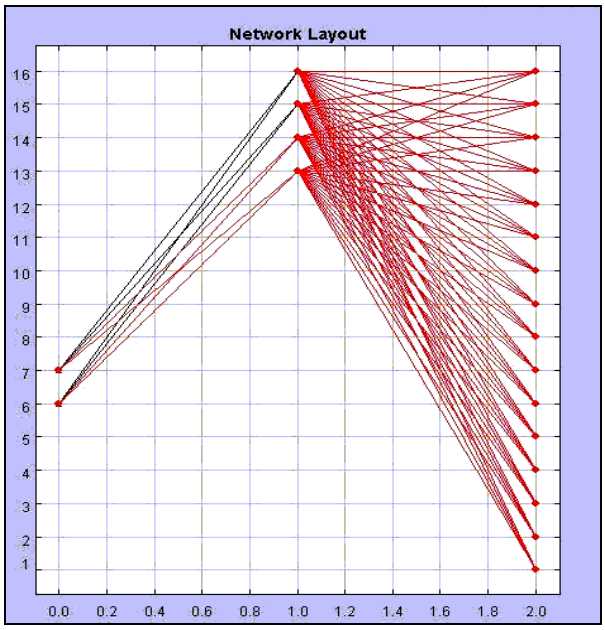

Figure 5.13
Phase 3 convergence training

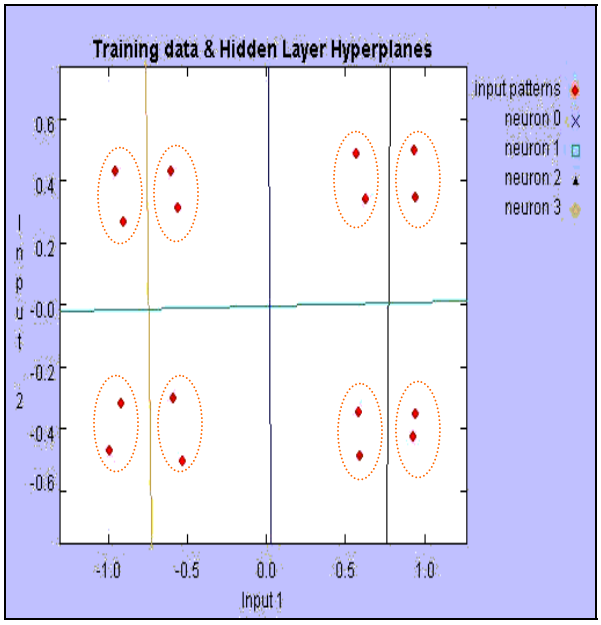

Figure 5.14
Phase 3 hyperplanes post convergence training

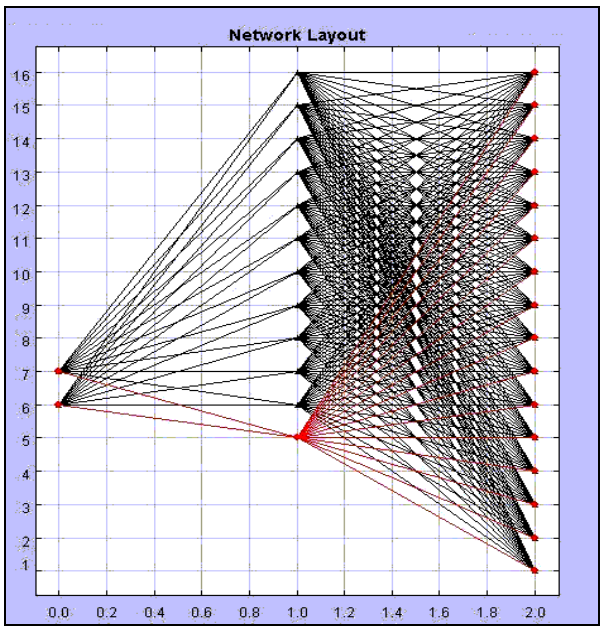

Figure 5.15

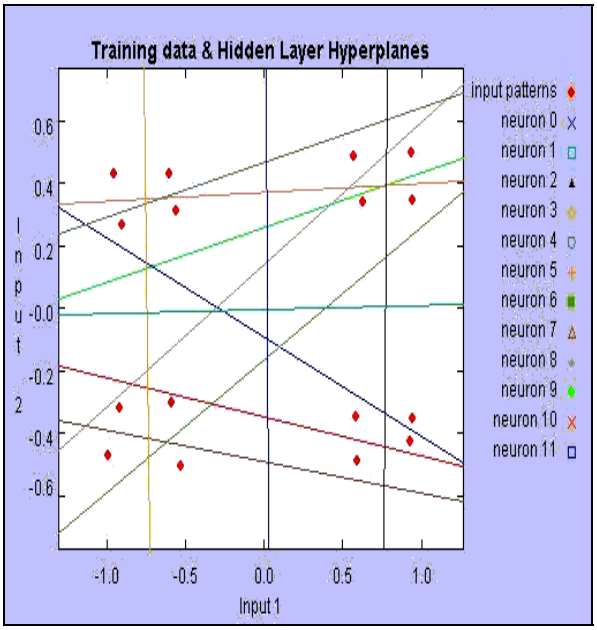

Figure 5.16

Phase 4 split flocking neuron 11

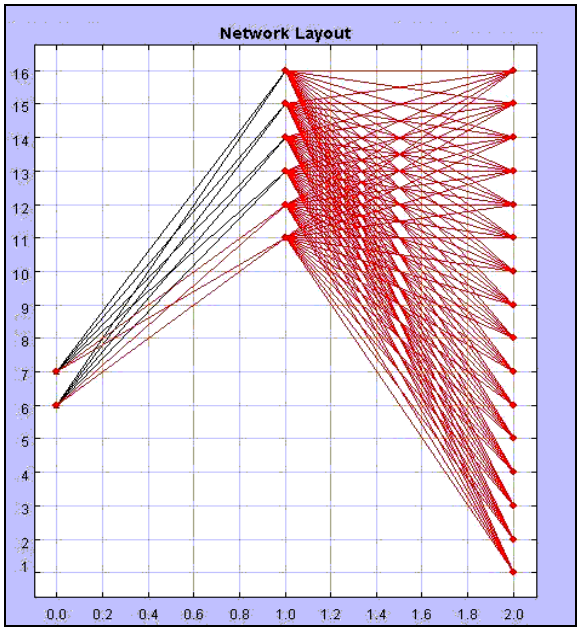

Figure 5.17
Phase 4 convergence training

Phase 4 hyperplanes post split flocking

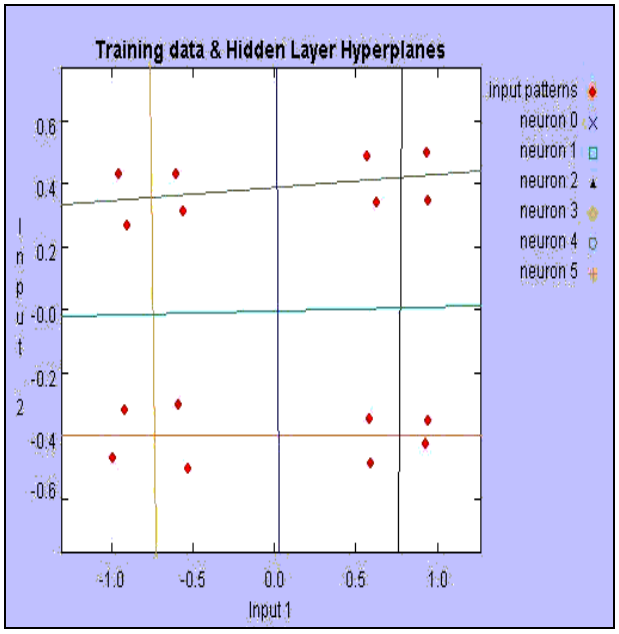

Figure 5.18
Phase 4 hyperplanes post convergence training

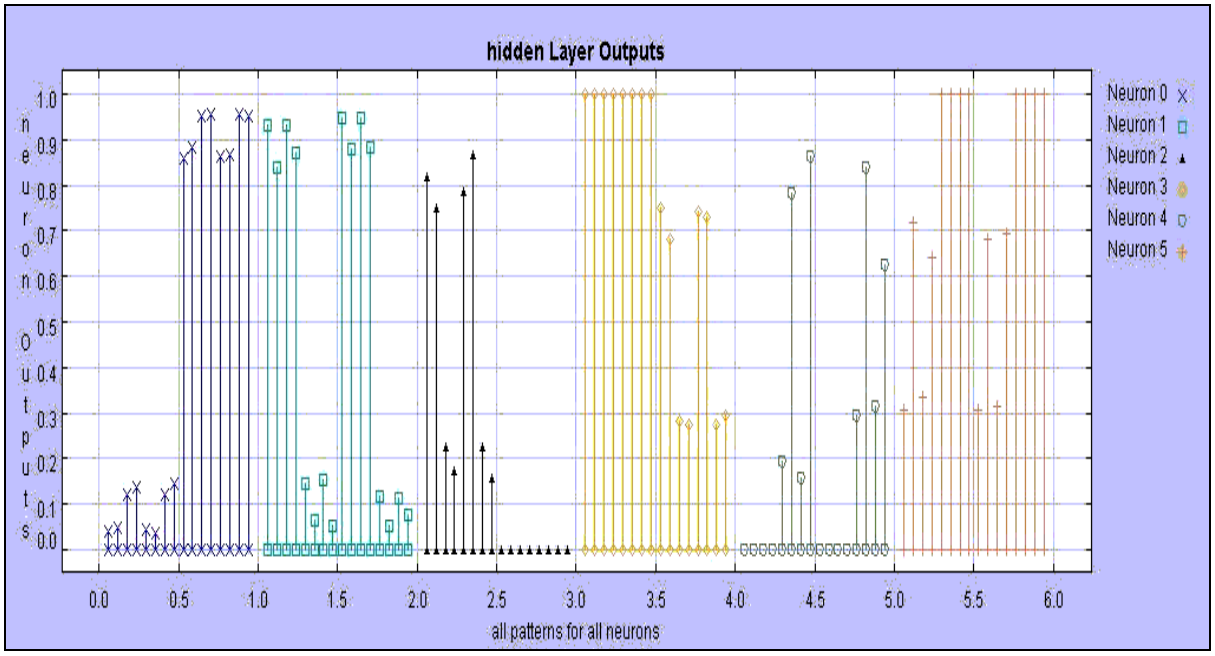

Figure 5.19 Final Hidden Layer Output Responses

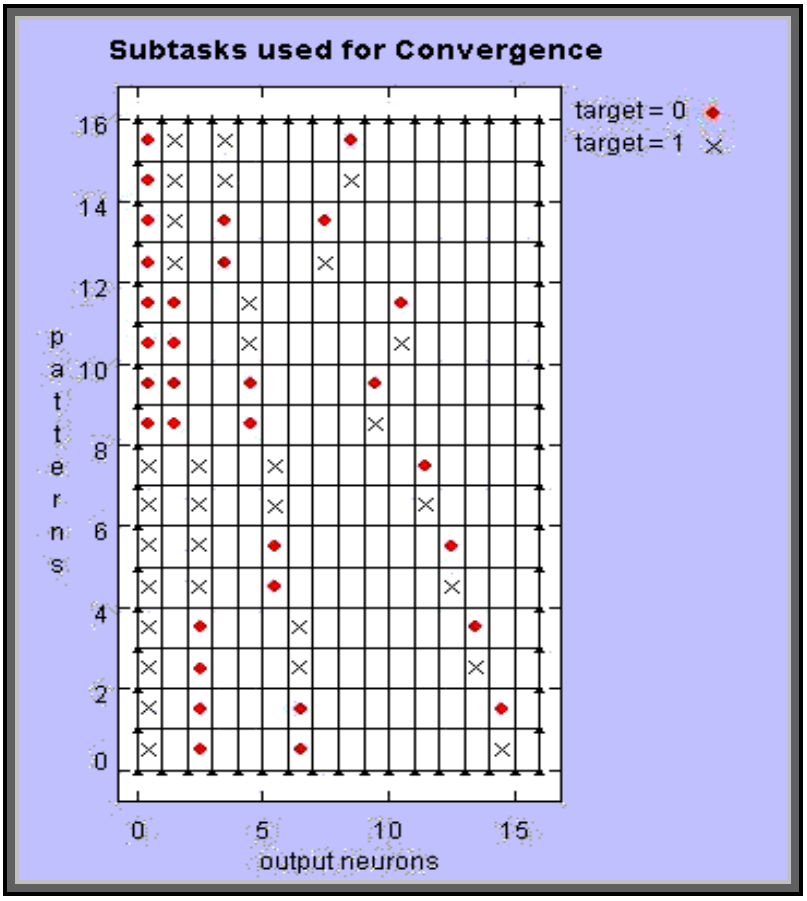

Figure 5.20 Subtasks created by split flocking

The hidden layer training stage of the primary learning algorithm produced a six neuron hidden layer solution that hierarchically generates “high margin” splits. This however means little if it does not produce good generalization results on unseen data. During the final stage of training, recall that the primary algorithm locks all weights and biases going into the hidden layer. The synapse weights going into the output layer are then re-randomized to low values before final training begins using the original training set. The following figures depict the final network architecture during the last stage of training and

the sum-squared-error graph for both the training set and the testing set during this final training process.

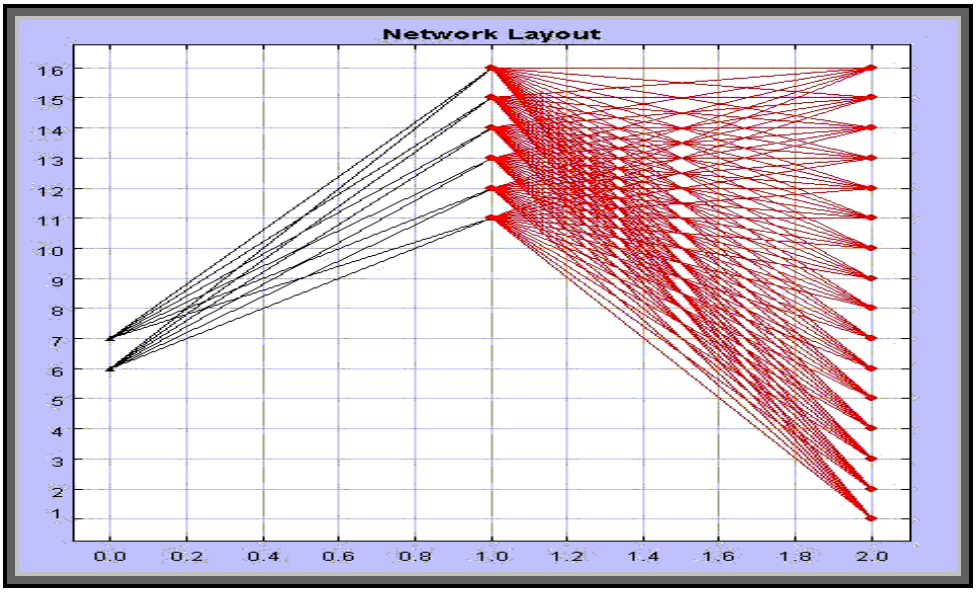

Figure 5.21 Network Architecture during final training stage

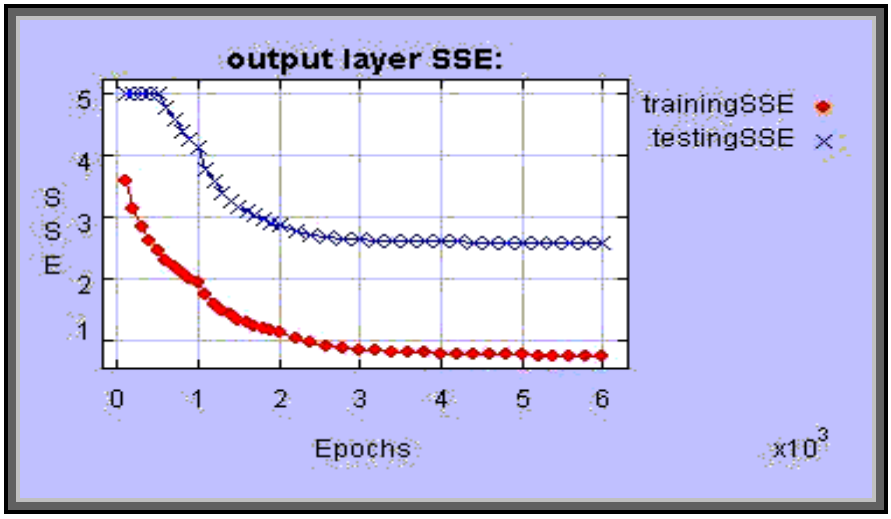

Figure 5.22 Sum Squared Error during final training stage

As you can see the primary learning algorithm has produced a network that generalizes better than the control network (compare to Figure 5.3). But how representative are these results? Will the primary learning algorithm consistently produce networks that generalize better on unseen data? To get a better idea of the behavior of both learning algorithms we need to run more tests. The following two figures are hidden layer hyperplane representations just like the ones seen previously, except that these figures overlay all the hyperplanes produces during the training of ten different ANNs. All ANNs in each of the two figures are identical except for the random weight initializations.

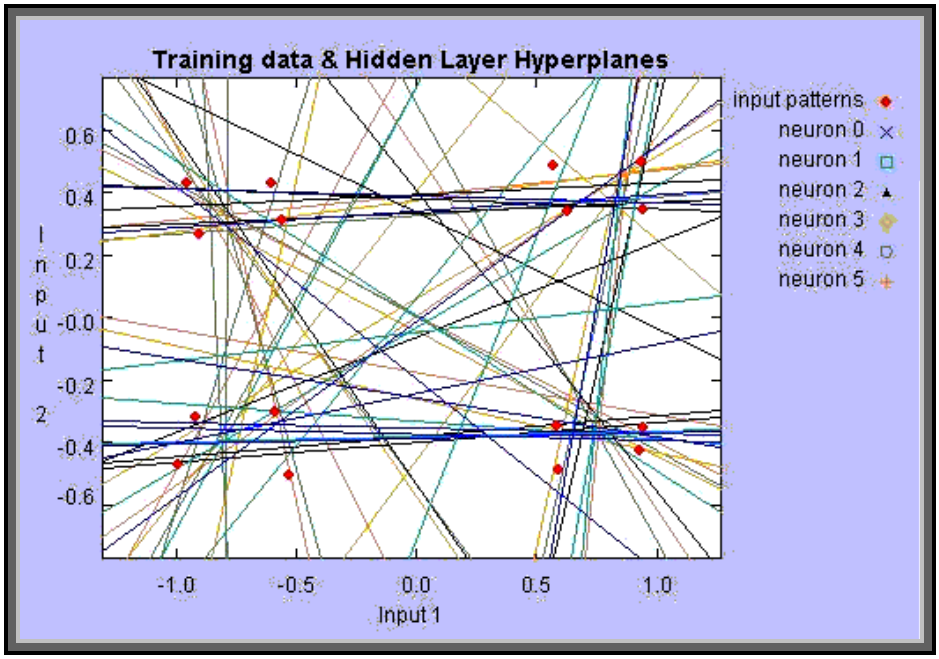

Figure 5.23 Control ANN Hyperplanes
(From 10 runs)

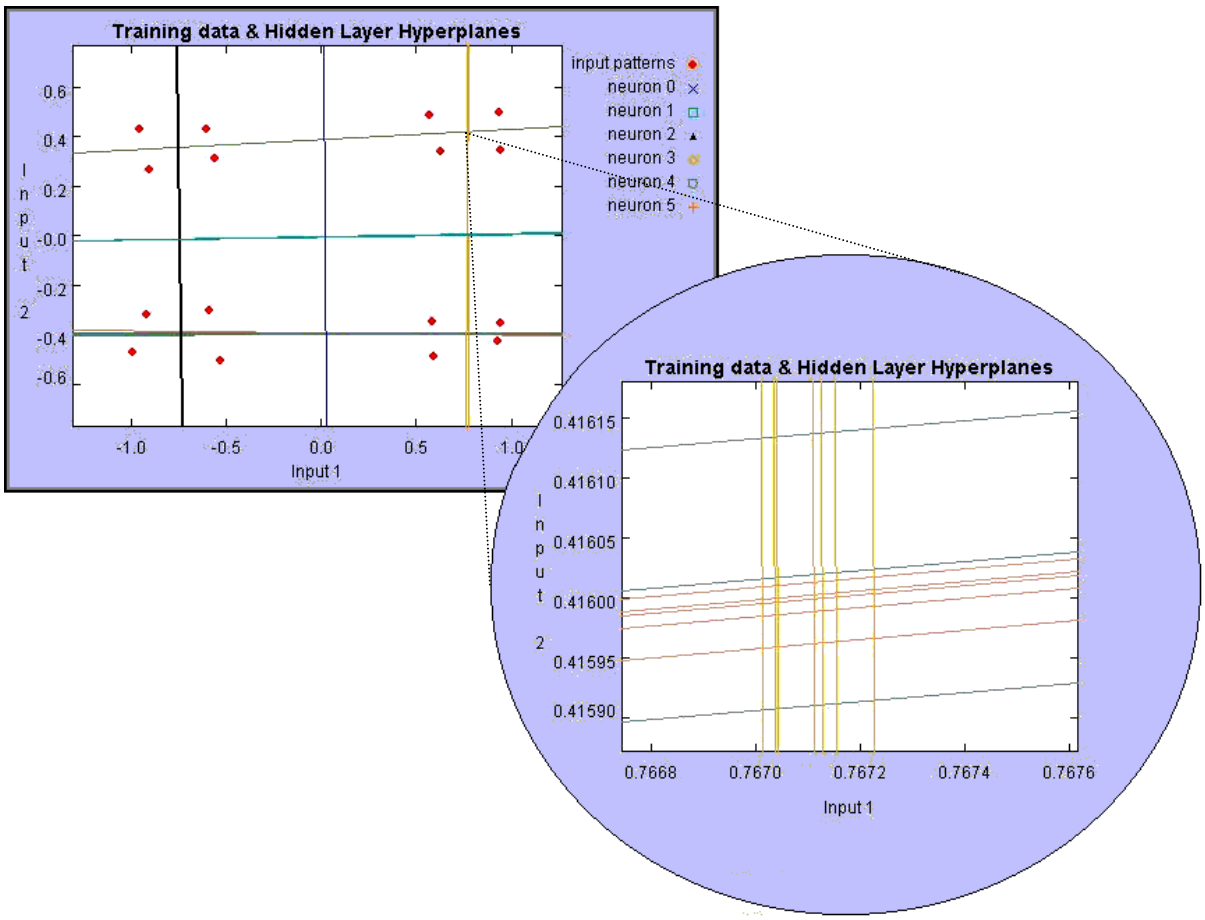

Figure 5.24 Primary ANN Hyperplanes (From 10 runs)

As you can see, the primary learning algorithm consistently finds very similar hyperplane solutions, while the control algorithm tends to converge to many different various hyperplane solutions. Is it necessarily better to be so consistent? Remember that the standard method for finding a backpropagation network that generalizes well is to train many networks and choose the best. Does this approach prove more beneficial in the end? The next two figures display the overlaid SSE graphs from the networks that produced the hyperplane figures shown above.

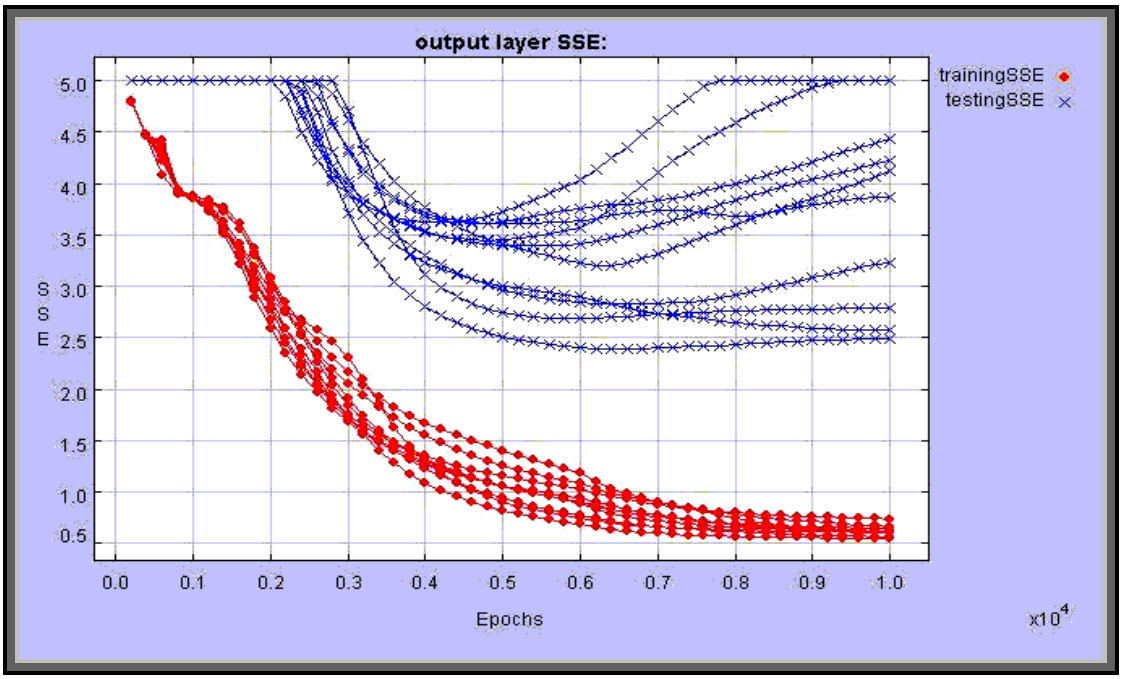

Figure 5.25 Control ANN Sum Squared Error (From 10 runs)

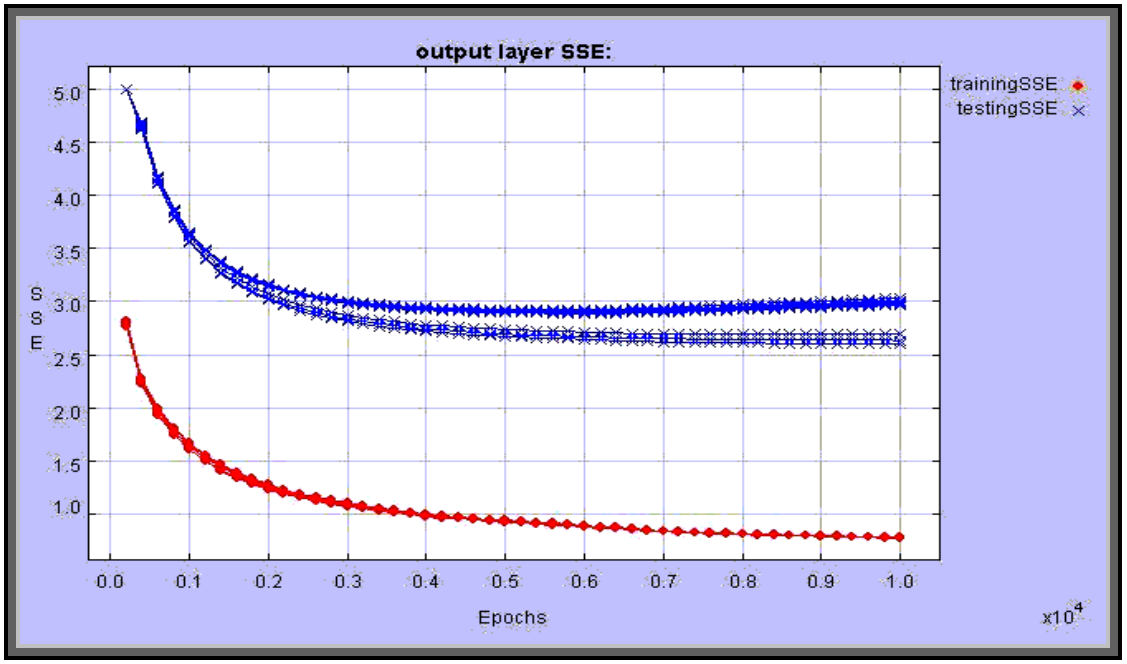

Figure 5.26 Primary ANN Sum Squared Error (From 10 runs)

As you can see, there are a lot of positives to the primary learning algorithm. It much more reliably produces a network that can generalize well. However out of the twenty networks represented in the two figures above, the one that generalized the best on the unseen data still came from the control group. This issue raises a number of questions. Assuming that this algorithm can be expanded to handle more complex data sets, just how many networks would have to be trained using standard backpropagation before you found a network that would generalize better than a network produced using the primary learning algorithm? Also would the time required to do this be justified? Many other basic questions still exist, not the least of which would be to simply answer the question of how robust this method actually is when applied to a variety of data sets.

In the next section we will examine a learning algorithm that takes the primary learning algorithm discussed in this section a step further by augmenting it with the ability to expand into multiple hidden layers. This new algorithm will provide some encouraging results especially with regard to hyperplane sharing.

6. Adding Multiple Hidden Layers (Primary Algorithm Two)

The learning algorithm discussed in this section has many of the same components as the primary learning algorithm discussed in section 5. The split flocking and convergence training of individual neurons is essentially the same as before, with the major modification being the adaptation of the algorithm for use with multiple hidden layers. In this new learning algorithm each phase results in the creation of a new hidden layer, which is then provided context inputs by the outputs of the previous hidden layer. This modification can have some very interesting consequences with regard to hyperplane sharing, as you will see in the results section. First we will describe in more detail the modifications made to the primary algorithm.

6.1 Methods of Approach

Recall that each phase in the primary algorithm concentrates on learning a new level of the binary tree data structure representing the subtasks produced from split flocking. At the conclusion of each phase the neuron weights representing the subtasks learned in that phase become locked. This is still relatively true in the multi-hidden layer version. Instead of proceeding to training another neuron in the same layer during the next phase however, a new layer is inserted in-between the old hidden layer and the output layer. This new layer not only contains synapse connections to the neurons of the old hidden layer, but also contains connections to the original inputs. The phase training at that point resets all the way back to phase 1 and proceeds to train neurons in the new layer for consecutive phases until

the new layer has trained one more phase than the neurons in the previous layer. Then another new layer is formed and the process repeats. Here is a formal breakdown of the new learning algorithm:

HIDDEN LAYER TRAINING for PRIMARY LEARNING ALGORITHM 2:

STEP 1: Add neurons to the hidden layer (layer closest to output layer). The number of neurons to add corresponds to the number of clusters in the binary tree that are available for splitting (for phase 1 there is a single cluster composed of all example patterns in the training set).

STEP 2: Use split flocking to separate each cluster with a hyperplane

STEP 3: Analyze the hyperplane produced by each newly added hidden layer neuron and update the binary tree accordingly. Remove all neurons that were added in STEP 1.

STEP 4: Use the new binary tree level produced in STEP 3 to create new corresponding convergence training subtasks and assign each new subtask and each previous subtask to its own output layer neuron (if not enough output layer neurons exist, add temporary output neurons)

STEP 5: Ignore the output of output layer neurons representing classes unrelated to each STEP 4 subtask (remember that each STEP 4 subtask is composed of a subset of example patterns, therefore this step results in each example pattern having its own set of ignored output neurons)

STEP 6: Add a hidden layer neuron and do convergence training using the subtasks corresponding to the next level of the binary tree data structure (Each phase starts STEP 6 using the very top level node's single subtask). If the specified target SSE is not reached at the end of a maximum number of training epochs, repeat this step again leaving the weights as they are and using the same set of subtasks.

STEP 7: Lock the weights of the neuron or neurons just trained in STEP 6. Increment the level of the binary tree used to select the subtasks of STEP 6. If this new level is greater that the level produced in STEP 3 proceed to STEP 8 otherwise go back to STEP 6.

STEP 8: Unlock the weights of all hidden layer neurons (the layer closest to the output layer) and do convergence training using ALL subtasks that have been created thus far.

STEP 9: Remove any temporary output layer neurons produced in STEP 4 and lock all remaining weights of the current hidden layer

STEP 10: If there are clusters that can be split, add a new hidden layer directly in front of the output layer and then proceed to STEP 1 to begin the next training phase using this new layer. Otherwise all hidden layer training is complete

Once the above hidden layer training has completed, the final stage of primary algorithm two begins. This final stage is exactly the same as the final stage of the first

primary learning algorithm and consists of keeping all hidden layer weights locked and training just the output layer neurons using all of the example patterns contained in the original training set.

6.2 Results & Analysis

We start this results section by examining a typical training session using the new learning algorithm. One of the big differences between this algorithm and the previous single hidden layer version is the fact that the hyperplanes represented by the hidden layer neurons are no longer solely two-dimensional. This means that we cannot create insightful hyperplane view graphs as we have done with all the other simulations conducted in this paper. Instead we will have to rely on other means in order to interpret the hyperplanes within the network. For the most part this means simply examining the output responses themselves and interpreting the divisions that they impose in the input space.

Since the hidden layer learning algorithm described above results in a dynamic network architecture that is hard to visualize, lets start by examining the network layout at different key points in time while training on the training set outlined in section 4. This should provide a better mental picture of how the network is behaving during the training process of primary algorithm 2. The following series of figures convey the continuously changing network architecture and concludes by showing the final SSE graph obtained during the final stage of training.

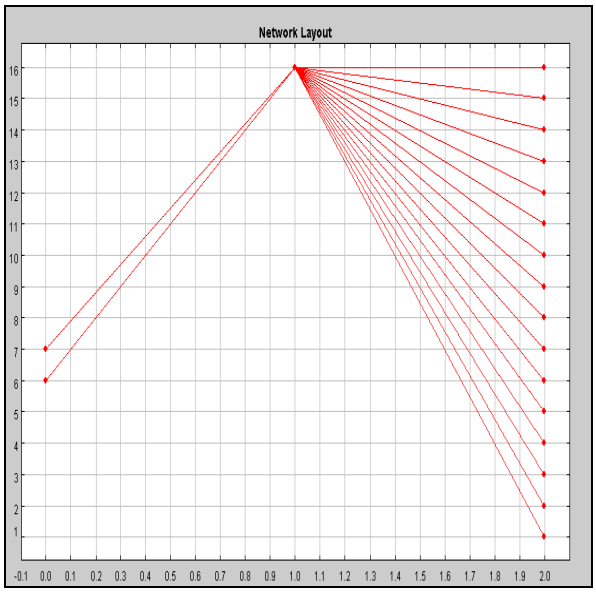

Figure 6.1
Neuron 0 for duration of Phase 1

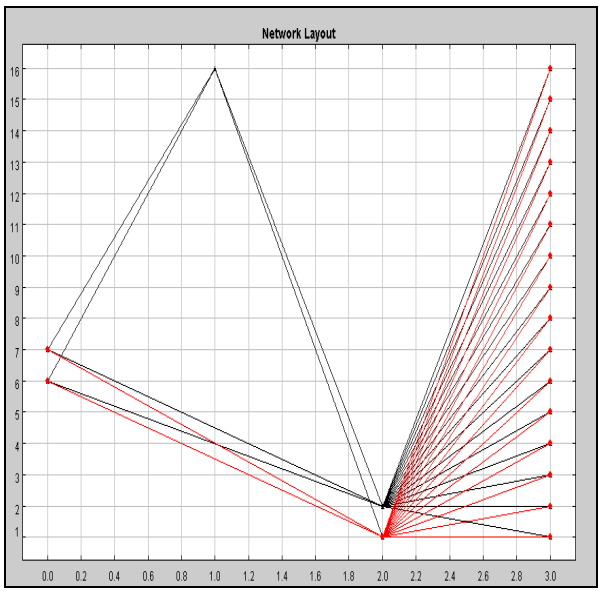

Figure 6.2
Phase 2 split flocking neuron 1

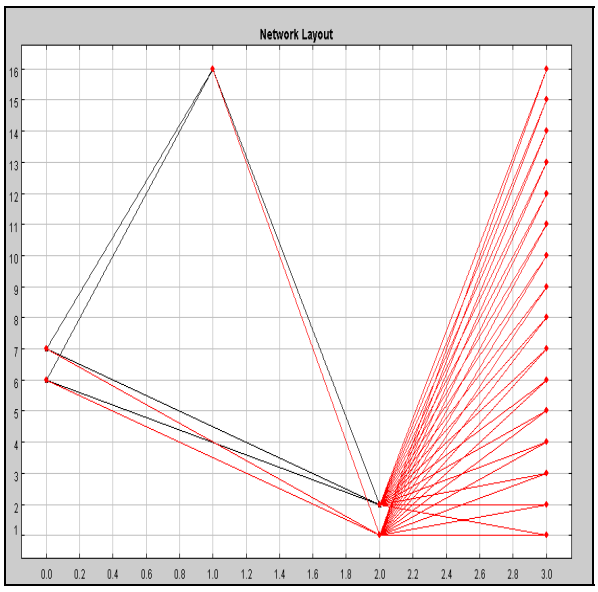

Figure 6.3
Phase 2 convergence training neuron 1

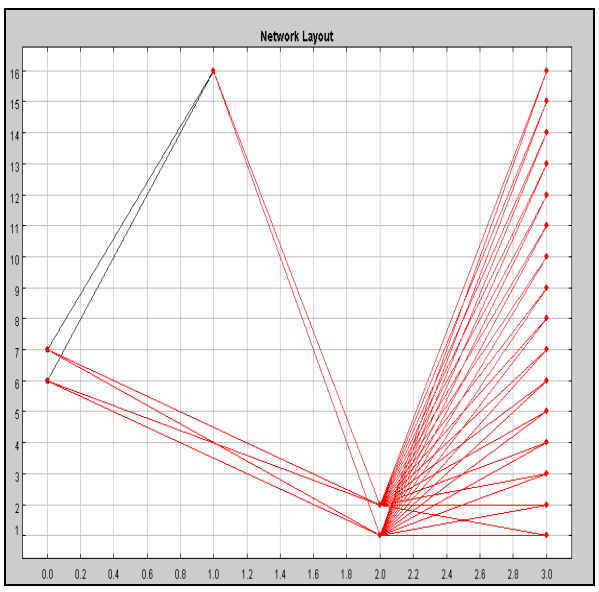

Figure 6.4
Phase 2 final convergence training

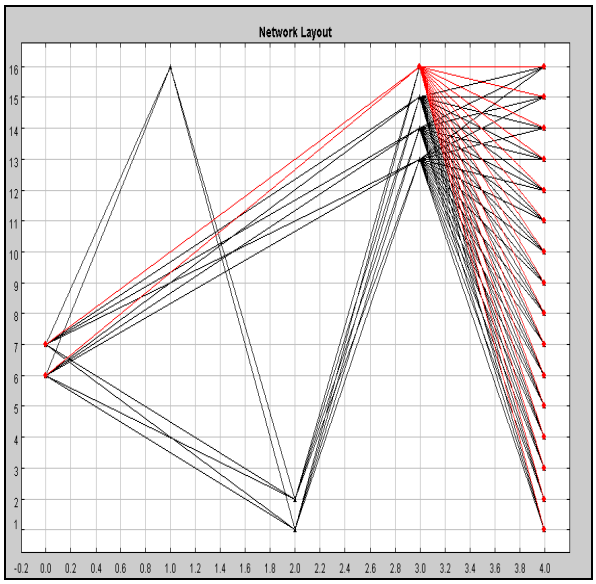

Figure 6.5
Phase 3 split flocking neuron 0

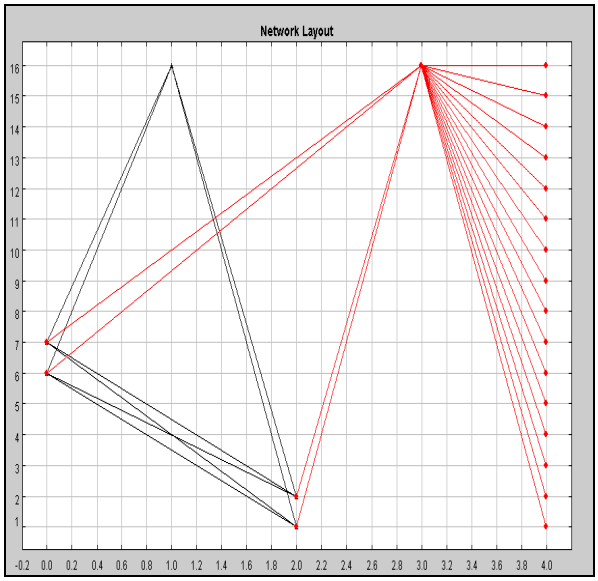

Figure 6.6
Phase 3 convergence training neuron 0

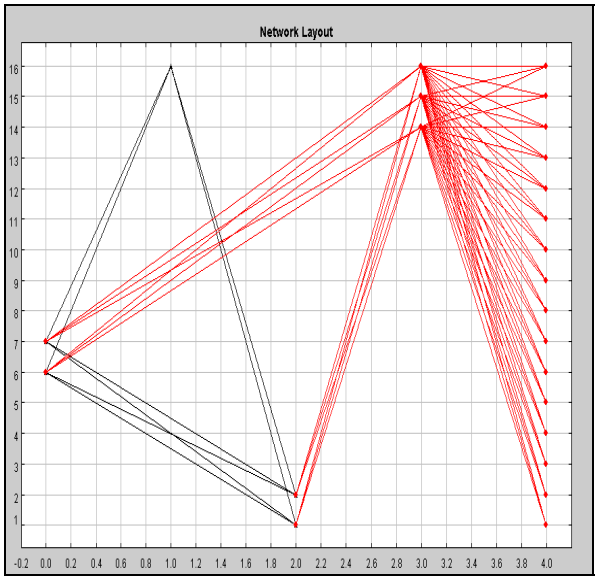

Figure 6.7
Phase 3 final convergence training

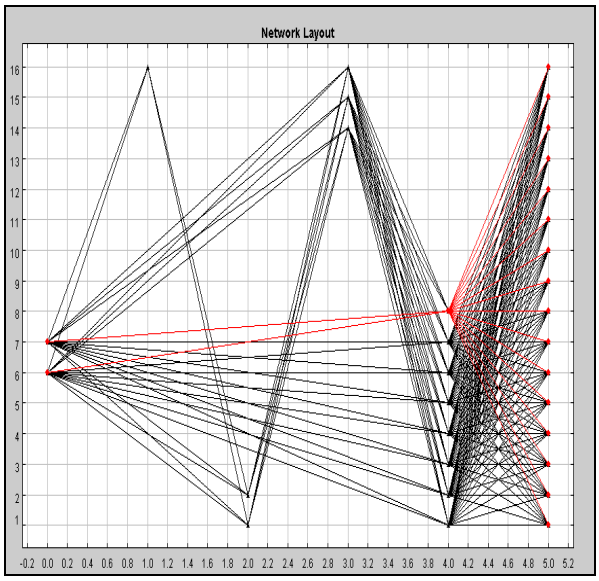

Figure 6.8
Phase 4 split flocking neuron 0

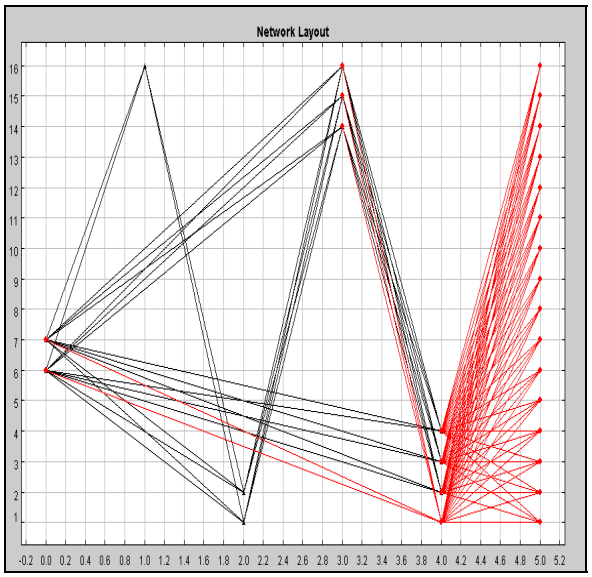

Figure 6.9
Phase 4 convergence training neuron 3

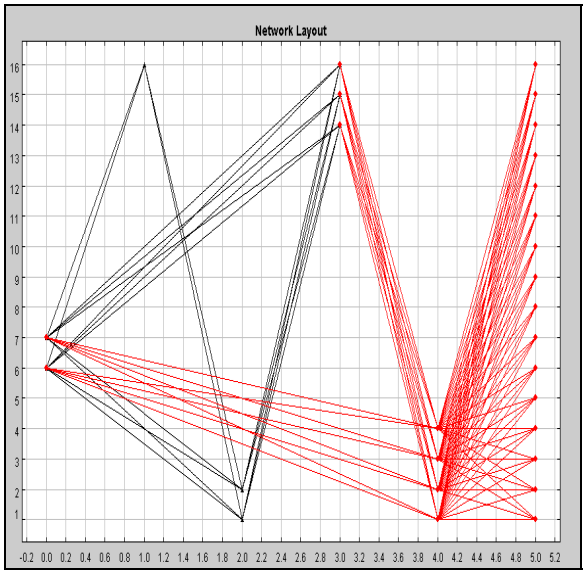

Figure 6.10
Phase 4 final convergence training

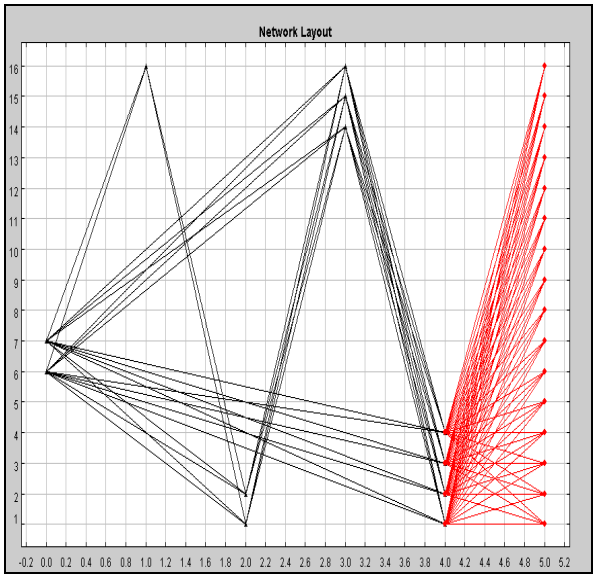

Figure 6.11
Layout during final training stage

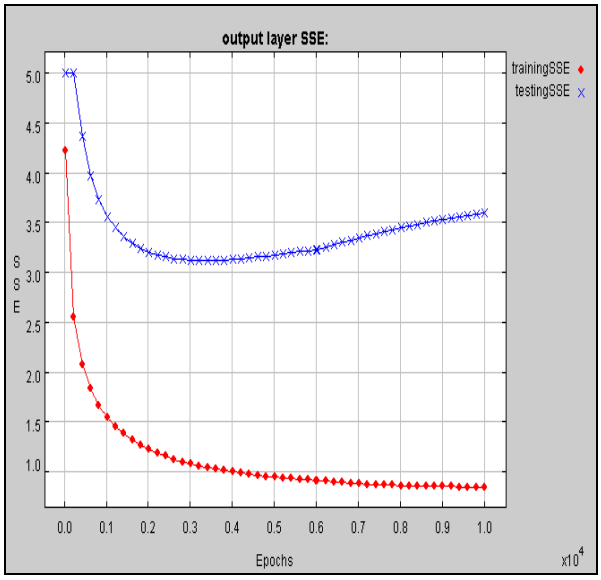

Figure 12
Final training stage SSE graph

The SSE on unseen data doesn't seem to be better than what was obtained with the single hidden layer primary algorithm, but the really interesting results come from examining other aspects of the network. Notice that at the conclusion of final convergence training for each phase (STEP 8 in the algorithm of section 7.1) that the weights for the layer trained in that phase become locked for the remainder of the entire training process. This means that the layer's output responses will also remain unchanging for the remainder of the entire training process. Now lets take a look at the final output responses of the four hidden layers created during this above training process.

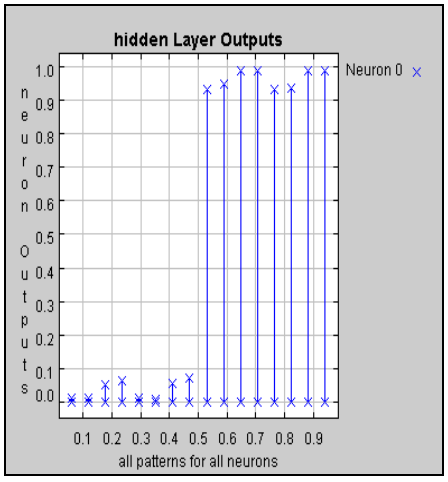

Figure 6.13
Hidden Layer 1 output response

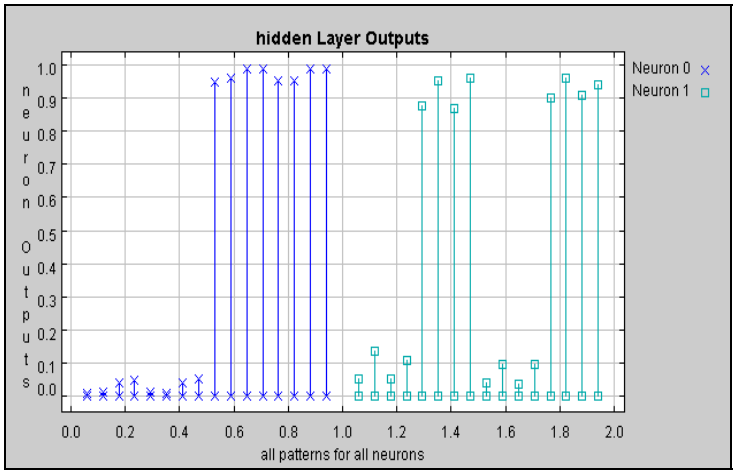

Figure 6.14
Hidden Layer 2 output responses

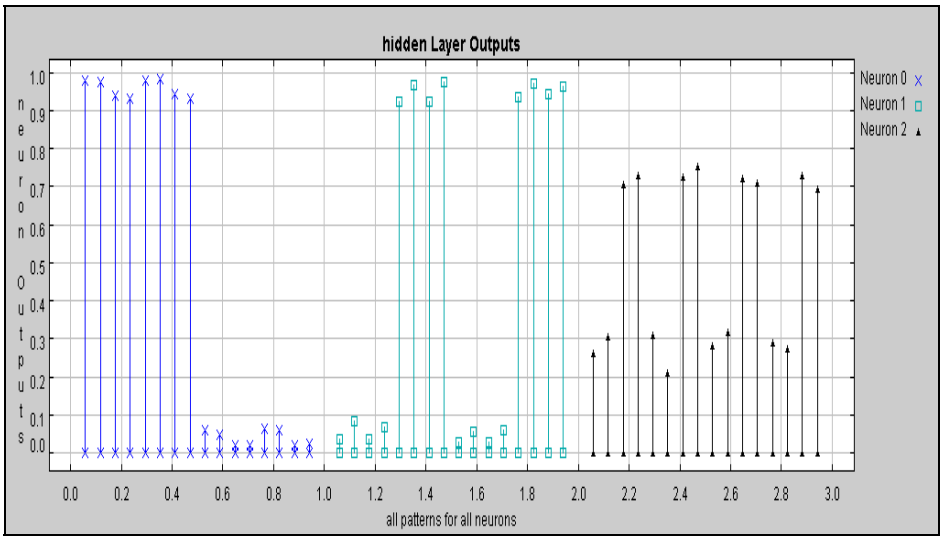

Figure 6.15 Hidden Layer 3 output responses

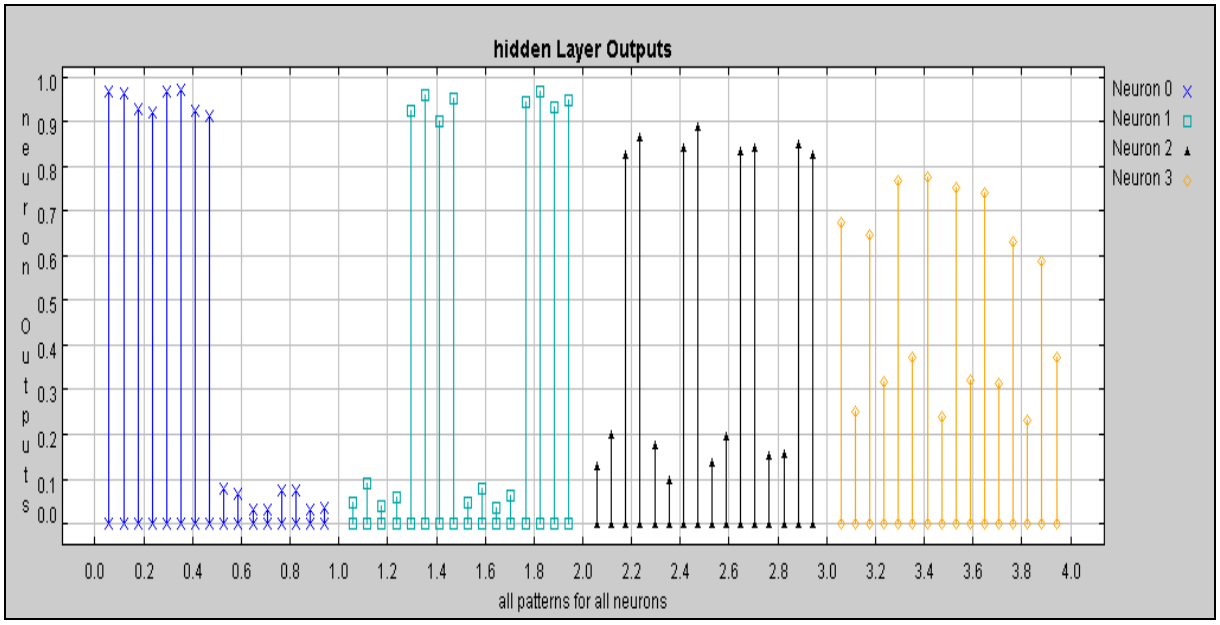

Figure 6.16 Hidden Layer 4 output responses

Notice that in each consecutive layer, the convergence training process begins by mimicking the outputs of the layer previous to it (though it is a matter of chance whether this mimicking turns out to be exactly opposite of the previous layer's general responses). This mimicking process happens at a much faster pace than the original training it took to achieve the same responses because of the of the "context" inputs received from the previous hidden layer. Judging by the response of neuron 2 in both layers 3 and 4, they become better established as well.

Of particular significance are the output responses of the fourth layer since layer 4 represents the only input that will be given to the output layer during final training on the original data set. Notice that if we turned each response into a zero or a one based on whether it was above or below 0.5 and used each of the four neurons as a single bit in a four bit binary number, then each of the 16 input patterns can be exclusively identified by a unique binary number. Also, each move in the direction from least to most significant bit in this number represents a different level of categorization within the binary subtask tree (and therefore a different level of generalization).

Since each neuron in the fourth hidden layer is receiving five inputs, each neuron in the fourth hidden layer represents a four dimensional hyperplane in 5-space. Obviously we cannot visualize such a hyperplane verbatim, however since the fourth hidden layer resulted in such nice divisions, we have plotted a representation of the decision boundaries with respect to the two original inputs. This plot is displayed in the following figure:

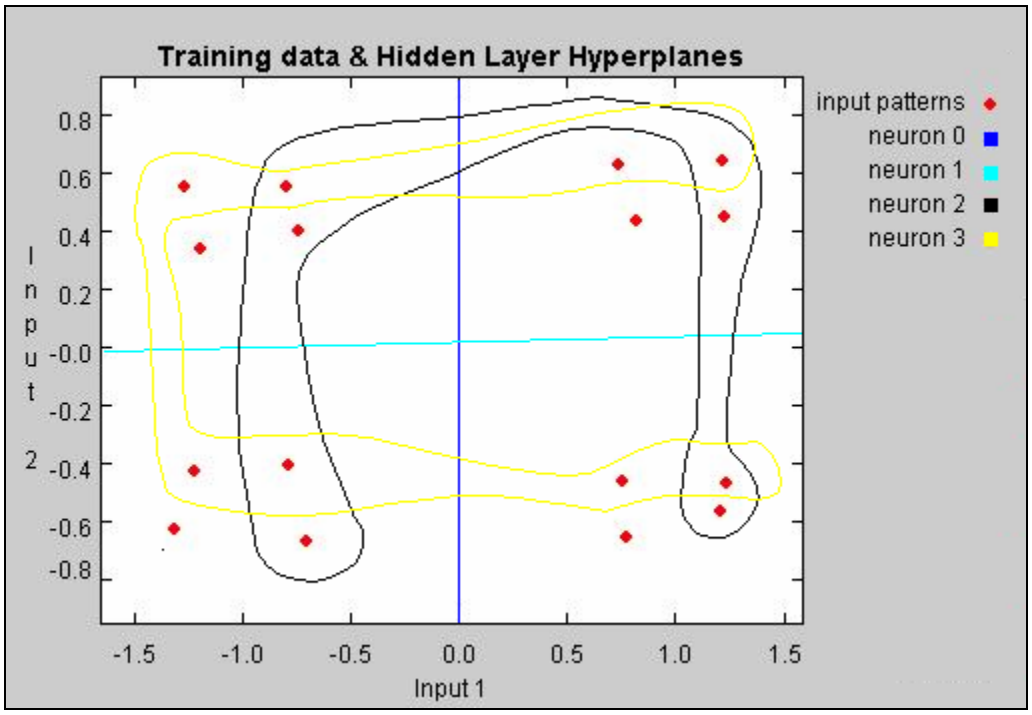

Figure 6.17 Hidden Layer 4 hyperplane representations

As you can see, the multi-hidden layer approach allows us to produce decision boundaries that would be physically impossible with only a single hidden layer. No matter how many four-neuron single hidden layer networks you trained, you would never be able to divide up the input space in such a way as to exclusively separate every pattern from every other pattern in this particular data set. Using multiple hidden layers to provide context thus allows for a much more compact final hidden layer solutions that can exploit hyperplane sharing in higher dimensions of the problem space.

Next I want to talk about a particularly promising outcome of the multi-hidden layer learning algorithm; one which was not discussed in the single hidden layer approach. This outcome revolves around the flocking done during the final stage of convergence training. In the single hidden layer approach, the multi-task backpropagation training on the subtasks (during the convergence component of training) did pretty much all the work with respect to actually converging the hyperplanes to their respective divisions in the input space. The flocking portion of the convergence training algorithm was a "fine-tuning" to improve generalization; it did not affect which training patterns were being split by the hyperplane before flocking started. In the multi-hidden layer approach this was not the case, and in fact the nice binary split results in figure 6.16 would likely have been very hard to achieve were it not for the flocking that occurred during convergence training.

In the following figures the output response of neuron 2 in hidden layer 3 and of neuron 3 in hidden layer 4 have been isolated. There are two depictions of each neuron's output response before and after the flocking stage in the convergence training of that neuron. In neuron 2 of hidden layer 3, it was already fairly obvious where the split was likely to occur, but the results of this neuron definitely show the important role that flocking plays. The two figures depicting neuron 3 of hidden layer 4 display the more remarkable results that this flocking stage can achieve in that it was not very obvious at all what effects the flocking stage was going to have. The final responses that were achieved in this neuron changed which patterns were being split by the hyperplane considerably and in the process achieved the extremely useful hyperplane sharing depicted in figure 6.17.

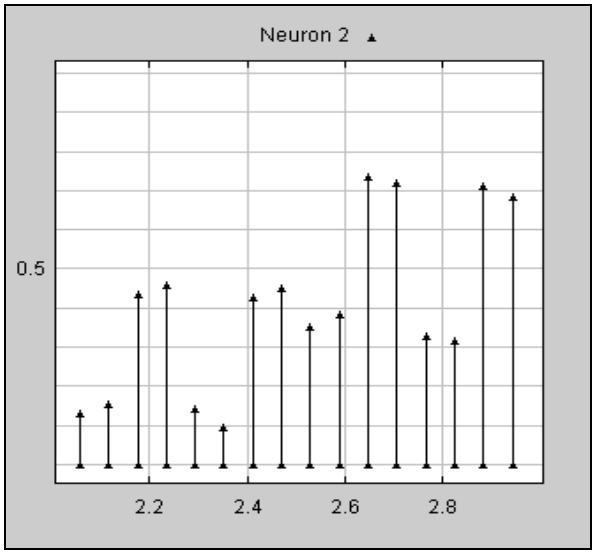

Figure 6.18
Layer 3 Neuron 2 Preflock Response

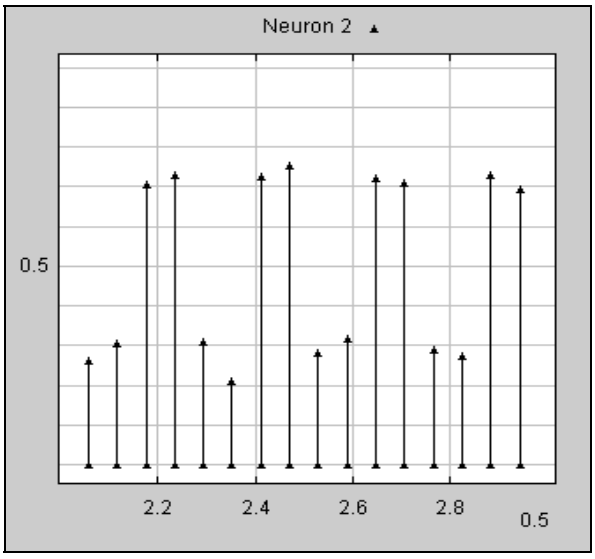

Figure 6.19
Layer 3 Neuron 2 Postflock Response

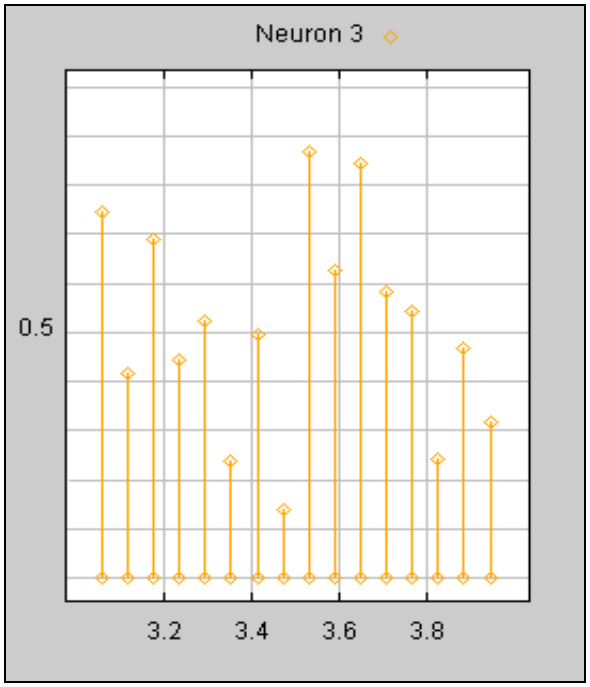

Figure 6.20
Layer 4 Neuron 3 Preflock Response

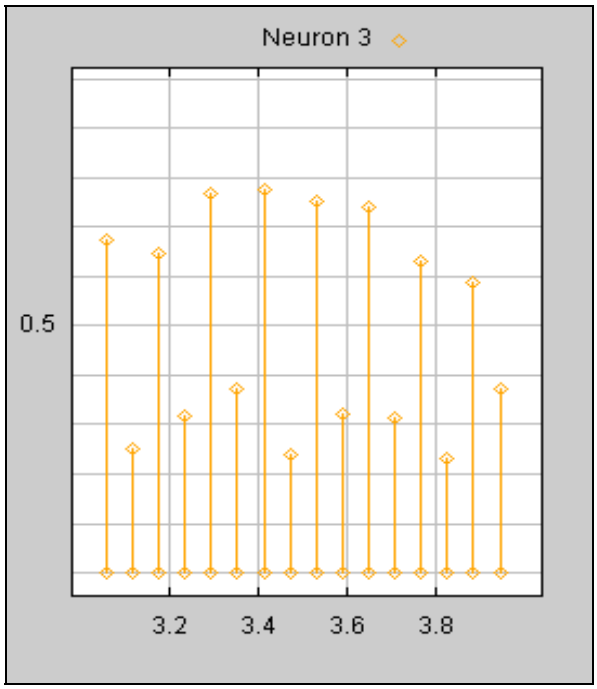

Figure 6.21
Layer 4 Neuron 3 Postflock Response

Now that we have thoroughly examined the results of a single training session using the multi-hidden layer learning algorithm, it's time to answer the question of how consistent the algorithm is. We will again overlay the results of 10 consecutive training sessions on top of one another and compare this to the results of the control network described in section 5 (Recall that the control network described in section 5 is a six neuron single hidden layer network trained using standard backpropagation). Since we cannot compare the hyperplane graphs, we will instead view the overlaid output response graphs of the two very different networks. This should at least be some indication of the consistency with which the hyperplanes are formed using the multi-hidden layer approach. The overlaid SSE graph following these two figures was produced during the same 10 consecutive runs of the multi-hidden layer learning algorithm. This SSE graph displays much the same consistency in its results as the results that were was displayed in the single hidden layer primary learning algorithm of section 5 (for comparison please refer to Figure 5.26).

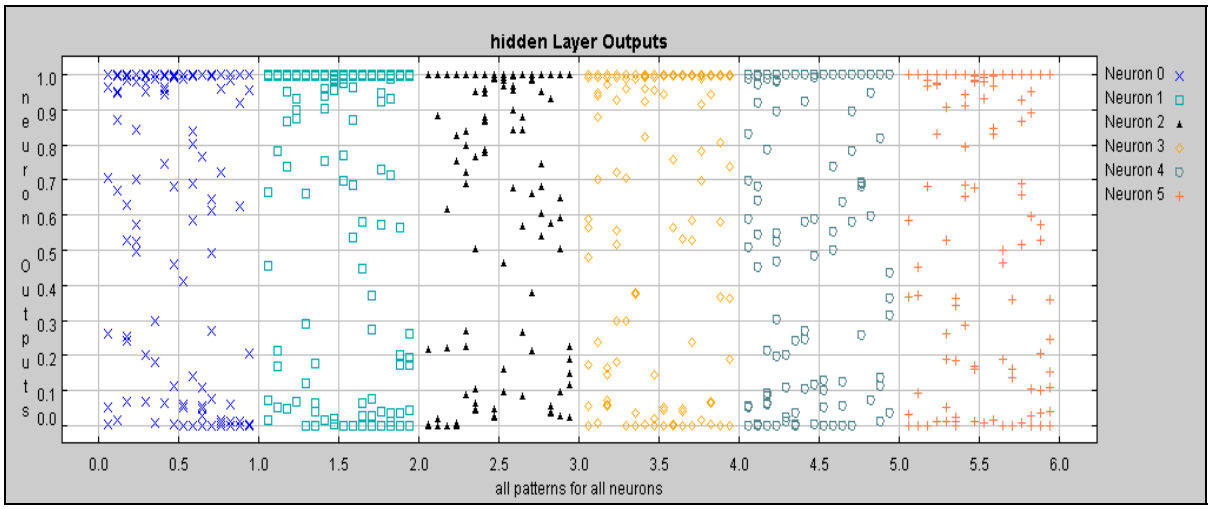

Figure 6.22 Overlaid output responses of 10 trained Control Networks

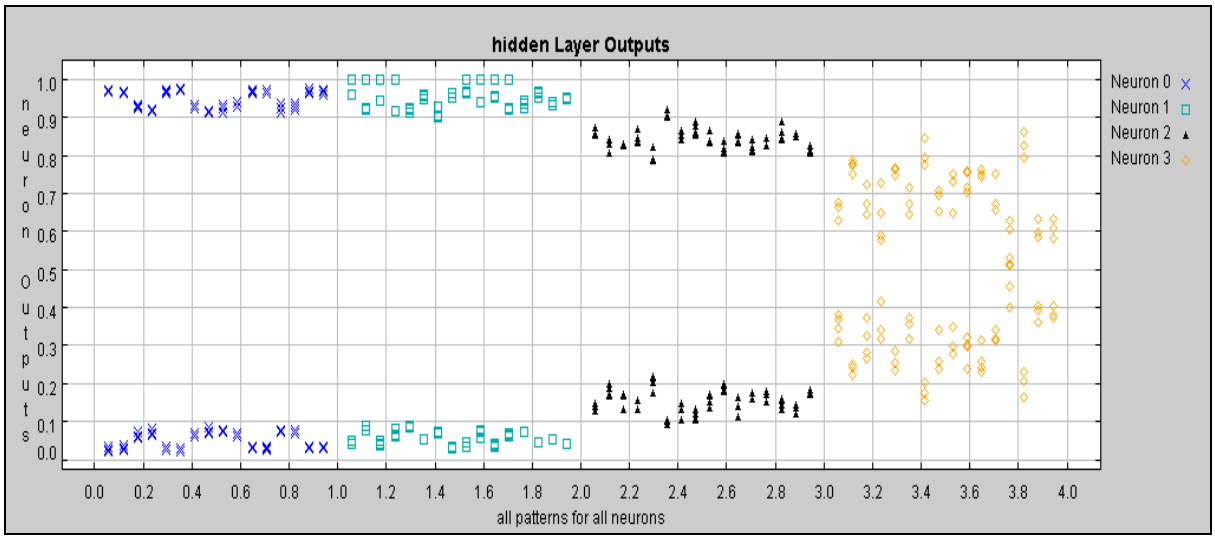

Figure 6.23 Overlaid output responses of 10 trained Multi-hidden Layer Networks

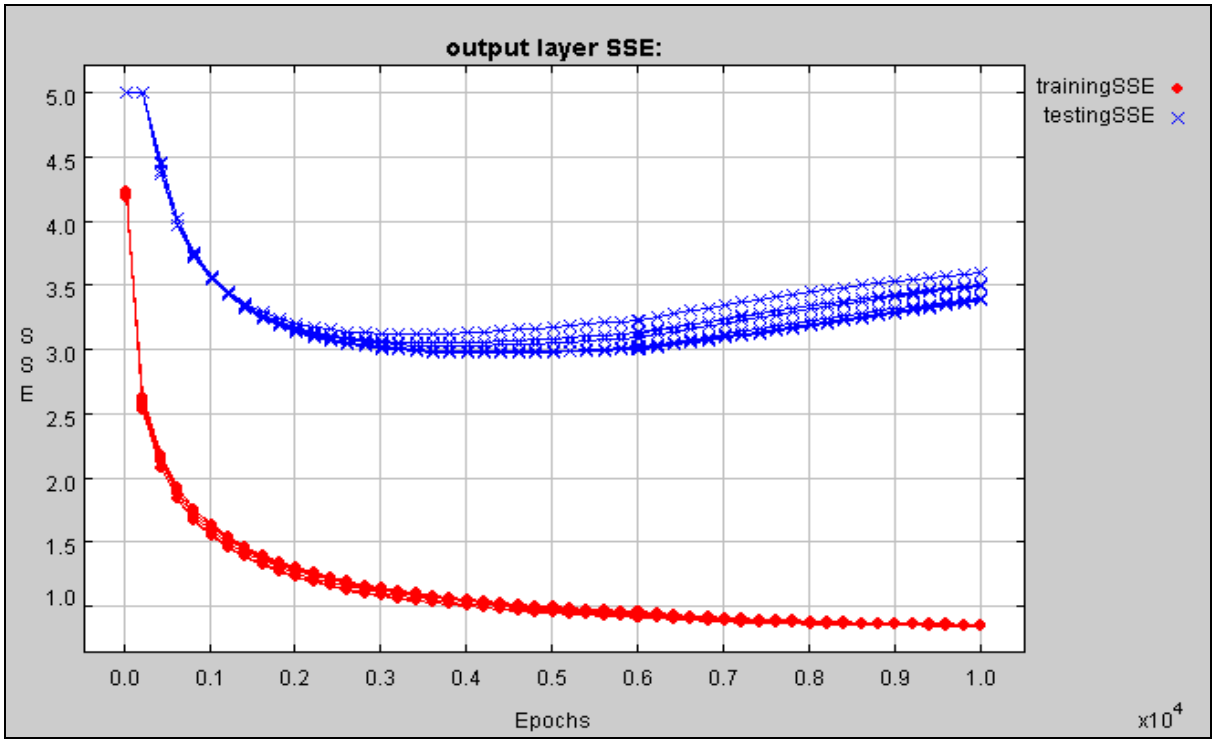

Figure 6.24 Overlaid SSE graphs of 10 Multi-hidden Network training sessions

As you can see the multi-hidden layer approach provides a very consistent solution to the data set but not quit as good generalization on unseen data as the single hidden layer implementation (compare to Figure 5.26). I believe this is largely due to the fact that the output responses seen in layer 4 neuron 3 are too far inside the linear region of the sigmoid function. This makes the responses much less stable when applied to unseen data. I also believe that this is a problem that could be fairly easily fixed given further research trials. As the above trials demonstrate, the hard part of getting the hyperplanes to actually converge to the hyperplane sharing positions can be accomplished. Getting the divisions to then become better pronounced can conceivably be done in a variety of ways whether it is through some form of further training or through methods similar to those to be described in the following section.

In the next section an ad-hoc approach involving just the split flocking algorithm will be examined which produces much better generalization than either of the two primary learning algorithms discussed above, but at the same time this new approach has a major drawback as you will see.

7. Ad-hoc Split Flocking (Supplemental Algorithm)

In this section we will examine the consequences of simply removing the convergence training portion of the primary learning algorithm just discussed. The real purpose of this section is not to suggest a viable algorithm so much as to hint at possibilities. At the conclusion of split flocking the hidden layer has all the divisions in it needed to identify each individual example pattern in the input space. So why not just go ahead and train the output layer? Well, even though the hyperplanes are in fact placed in such a way as to completely and conclusively separate every pattern from every other pattern, the synapse weights going into the hidden layer are so low as a result of flocking that the hidden layer output responses are too insignificant to allow learning to take place in the output layer.

To demonstrate this fact, take a look at what happens when we remove the convergence training portion of the primary algorithm (details of exactly how this is done will be described shortly in section 6.1) and then train for 200,000 epochs with incremental increases in learning rates (the last 180,000 being a learning rate of 20!):

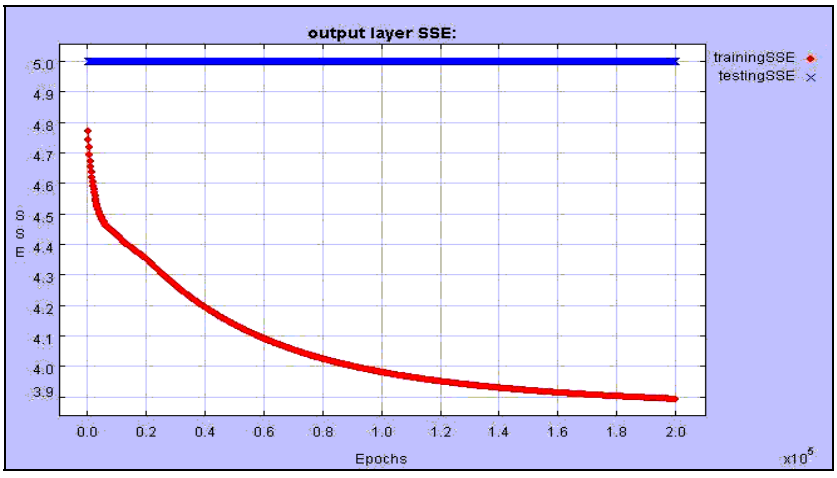

Figure 7.1 unmagnified split flock training session

The above figure simply demonstrates that using the split flocking algorithm alone to train the hidden layer is not very useful because it will usually leave the weights too low to provide a substantial hidden layer output response. That's where the ad-hoc approach comes in. The algorithm in this section cheats a little to obtain some very interesting results. I believe it is an encouraging motivator of further research in this area.

7.1 Methods of Approach

The only difference between the algorithm used to train the network in Figure 7.1 and the primary hidden layer training algorithm discussed in section 5 is essentially the removal of steps 4, 5, and 6. This fundamental change results in the following hidden layer training algorithm:

HIDDEN LAYER TRAINING for SUPPLAMENTAL ALGORITHM:

STEP 1: Add neurons to the hidden layer. The number of neurons to add corresponds to the number of clusters in the binary tree that are available for splitting

STEP 2: Use split flocking to separate each cluster with a hyperplane

STEP 3: Analyze the hyperplane produced by each newly added hidden layer neuron and update the binary tree accordingly.

STEP 4: Remove all redundant hidden layer neurons

STEP 5: If there are new clusters that can be split, proceed to STEP 1 and begin next phase training. Otherwise hidden layer training is complete

The ad-hoc approach that will solve the low weight problem is to simply magnify the weights at the end of hidden layer training. Then the output layer training continues exactly as before. Magnifying the weights does not cause the hyperplanes to shift position in the input space in any way, but it does cause the strength of the divisions they represent to increase in magnitude. In other words, instead of all the example patterns causing hidden layer outputs close to .5, the example patterns on one side of the hyperplane get closer to one, and the example patterns on the other side of the hyperplane get closer to zero.

7.2 Results & Analysis

Under normal ANN conditions magnifying the weights of a network like this is not advisable. For comparison purposes we will contrive a control network that will demonstrate this fact. Since the split flocking approach results in different various numbers of hidden layer neurons for different various initial weight conditions, it is hard to create a control network architecture that exactly models the architecture of the ad-hoc split flocking network. The results however are not all that different for different numbers of neurons in the control's hidden layer, and the control network that we will examine utilizes 16 hidden layer neurons. It will also be trained in a similar fashion to the control network used for comparison to the two primary networks. The only difference is that at the end of normal backpropagation training we will magnify the hidden layer weights, lock them, re-randomize the output layer weights, and then continue training for a period of time on the training set.

In the following figures, we examine the outcomes of the network training sessions for both architectures. Just as before we will train 20 networks total with each learning algorithm training on 10 networks each. The SSE graph for the control network will represent all the epochs of training so that the effects of magnification can be very easily seen. The ad-hoc split flocking SSE graph however, is only representative of the final stage of training due to the useless nature of the SSE graph during hidden layer training.

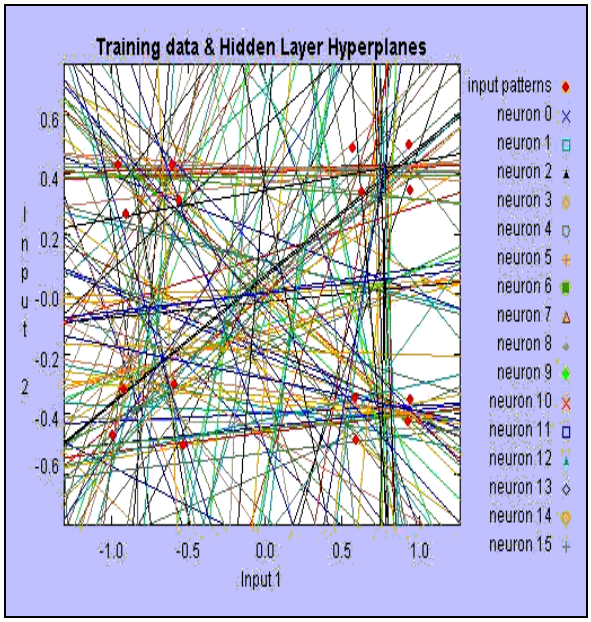

Figure 7.2
Control Network Hyperplanes

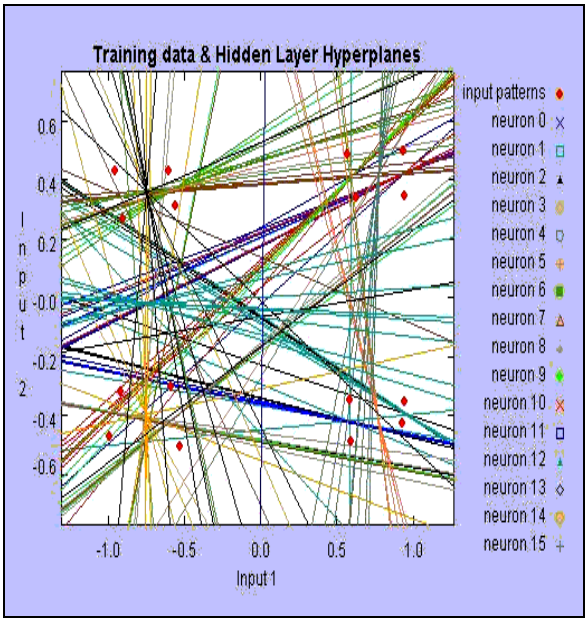

Figure 7.3
Ad-hoc Network Hyperplanes

As you can see the ad-hoc networks are not nearly as chaotic as they might first seem when they are contrasted with the final hyperplane solutions of the control networks. Next let us examine the SSE graphs for the networks that produced these hyperplanes.

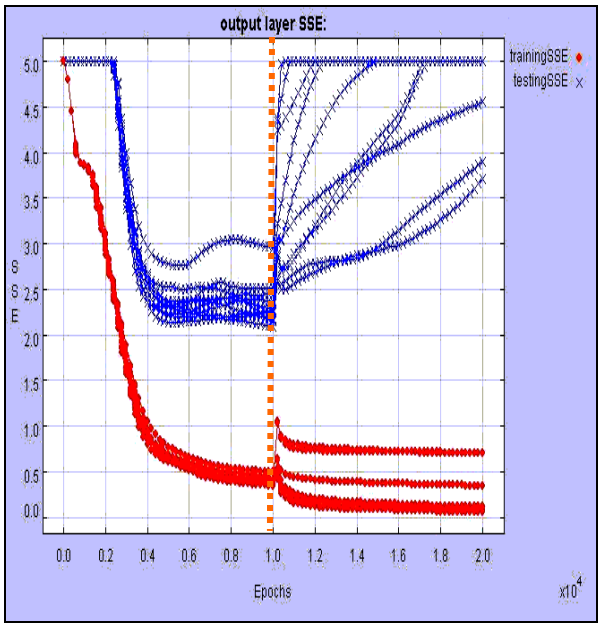

Figure 7.4
Control Network SSE

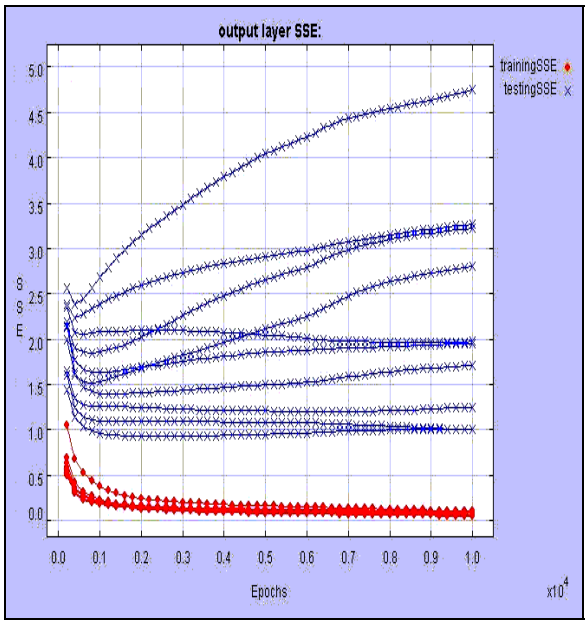

Figure 7.5
Ad-hoc Network SSE

The dotted orange line in Figure 7.4 represents the point at which the weights were multiplied during the control network's training process. The training to the right of this dotted line is what we are concerned with comparing to the ad-hoc network. It is self evident that weight multiplication was not a good idea for the control network scenario. The ad-hoc split flocking network on the other hand shows much more resistance to the destabilizing effects of the weight multiplications, and in fact would never have been able to train at all if not for the weight magnifications occurring. Future algorithms should try to exploit this property without causing the magnification induced weight saturations that cause this method to be considered ad-hoc.

8. Ideas for Future Research

There are a variety of jumping off places where this research could be taken in other different directions. For starters simply trying these algorithms out on other various data sets is a much-needed addition to the material that has been discussed. Actually just pulling different random samples from the ideal data set used above would be a good idea. Besides using other data sets or pulling additional random samples, there are a number of questions that this research has raised. The split flocking and convergence training algorithms are by no means a final answer to improved generalization, and by simple examination of the details in the code an exceptionally large number of algorithm offshoots is not hard to imagine. Simply experimenting with these different offshoot algorithms could easily occupy ones research interests for an extended period of time.

With regards to specific ideas that I have for future research, I have two that come to mind and that I would like to mention in further detail. The first idea involves the multi-hidden layer primary learning algorithm. In this algorithm it was noted that each successive layer mimicked the layer preceding it by training neurons with similar output responses to the previous layer before adding neurons to the layer which finally resulted in original output responses. The final hidden layer therefore ends up with neurons that are highly representational of the output responses of all other neurons in all other hidden layers. It is not hard to imagine that if the respective neurons with similar responses in each layer actually had identical responses, that the output of the network would be little changed as a result of it. Given this fact it is not at all infeasible that all four layers could be collapsed into

a single hidden layer recurrent network with only four neurons in the hidden layer. The architecture would have the neuron representing the first split feed back to the next three neurons, the neuron representing the second split feed back to the next two neurons, and the neuron representing the third split would feed back to the final neuron. This recurrent architecture would have some interesting properties in that you would be able to measure the output responses during each feedback iteration and watch the output hone in on the final output response. If the correct response happened to be achieved in the first few feedback cycles, it would be a measure of confidence in the categorization that the network produced for that particular example pattern.

The second suggestion was actually one of the goals that was hoped would be achieved at the onset of this research, and it is a much more alluring research opportunity. Basically the goal would be to successfully combine the backpropagation and flocking equations into a set of unified equations for use with the gradient descent algorithm. The possibilities of such an algorithm are very attractive. I don't know of any other ANN learning algorithms that contains both supervised and unsupervised components within the same equations. This would be a very interesting achievement indeed and likely to be well worth any time that was invested.

9. Conclusions

The two aims of this research were to try and unravel some of the mystery surrounding the means by which artificial neural networks generalize and to utilize some new learning tools to try and improve generalization. To this end we have made the suppositions that generalization is improved by raising the number of examples that influence individual neuron decisions and it is also improved by placing neuron hyperplanes in such as way that they are shared among the decision regions of the problem space as much as possible.

In order to tackle the first supposition we utilized example influence as a tool for measuring both the individual influence of example patterns as well as the effective number of examples that influence an individual neuron. Example influence was then utilized in the self-organizing learning algorithm: flocking, which is a powerful algorithm that tries to promote uniform decision regions where example patterns equally influence decisions within a network. In order to take advantage of the capabilities of flocking, we inserted the algorithm into two very different intermediate learning algorithms: split flocking and convergence training.

In order to tackle the second supposition (regarding the sharing of hyperplanes) we implemented a multi-task training scheme that employed the results of clustering to create subtasks that represent intuitive splits within the input space. We then trained on groups of these subtasks in unison using backpropagation in the intermediate convergence-training algorithm in order to promote the sharing of hyperplanes in the network.

The split flocking algorithm was mainly used for clustering in both primary learning algorithms, but as the results of the Supplemental Algorithm show, it is much more than just

a clustering algorithm and it has much potential to be integrated into a learning algorithm of its own. By far the best generalization results of all the learning algorithms tried in this research were in fact achieved by the supplemental ad-hock split flocking algorithm.

The convergence training algorithm also provided much to be pleased about. In both the single hidden layer and multiple hidden layer primary algorithms, convergence training brought a strong sense of reliability. All ten networks in both multiple trial runs produced very consistent results that were on the whole much better that the average results achieved using backpropagation alone. Also both the single and multiple layer primary algorithms were able to achieve extremely high degrees of hyperplane sharing. This not only tends to improve generalization, but also brings a sense of efficiency to the network by accomplishing the division task with as few neurons as possible.

The results observed in the final stages of hidden layer training using the multiple hidden layer algorithm were particularly interesting and indeed completely unexpected. The fact that the final solution involved a four neuron layer that basically provided a binary number representation for each pattern was quite elegant but not predicted. We also showed that this elegant solution would not have been arrived at using the multi-task training alone. Flocking played an essential role in causing the hyperplanes to converge to this solution. Indeed in the case of the final neuron trained in layer four the solution arrived at without flocking would have been drastically different and no doubt provided much worse generalization on the test set.

All and all, I believe the ANN experiments conducted in this research should be considered a success. We have gained at least a small understanding with regards to how

ANNs might be made to generalize better, and we certainly have gained some insight into how to make them generalize more consistently. The ideas related to flocking and multi-task convergence training have certainly been shown to have a real relevance to the ultimate goal of creating smarter and better generalizing ANN learning algorithms. It is hoped that this research has been a contribution, however small, if not to answering any of the big questions then hopefully to helping lay the foundation for asking more of the smaller ones.

Artificial Neural Networks are, simply put, an amazing problem solving tool. Besides the mere fact that they are able to provide solutions to enormously complex data sets, they do it in a completely localized fashion that can provide the generalization necessary to be extremely resilient to the fluctuations encountered in the real world. Yet for all this, even more amazing is just how little we understand about them. We know extremely little about the true nature of the non-linear relationships that exist on synapses in between the neurons of a network. How do these relationships form? By what means can we more accurately predict and focus their awesome learning potentials? Only through meticulous analysis can we hope to achieve such goals. To understand their true nature we must continuously strive for new perspectives, investigate new ideas, and catalogue our successes and failures.

LIST OF REFERENCES

[1] Randall C. O'Reilly and Yuko Munakata, 2000. Computational Explorations in Cognitive Neuroscience. Massachusetts Institute of Technology, Cambridge, MA.

[2] Basheer, I.A., Hajmeer M., 2000. Artificial neural networks: fundamentals, computing, design, and application. Journal of Microbiological Methods 43, 3-31.

[3] Daniel Klerfors, Artificial Neural Networks. Saint Louis University School of Business & Administration: http://hem.hj.se/~de96klda/NeuralNetworks.htm

[4] Janet, J. A., Sutton, J. C., 1998. Computational Intelligence: Supervised and Unsupervised Learning with Neural Networks. TMI Robotics, Inc., Raleigh, NC

[5] Carlos Gershenson. Artificial Neural Networks for Beginners. University of Sussex: http://www.cogs.susx.ac.uk/users/carlos/doc/FCS-ANN-tutorial.htm

APPENDICES

Appendix 1: A Flocking Demonstration

This appendix was included as a resource for better demonstrating the underlying mechanisms of flocking with regards to different various network components. In the following simulation we will first examine the consequences of gradient descent backpropagation on a single hidden layer neuron (the only one in the network), and then we examine the effects of flocking on that neuron. The data set being used for training in the simulation is contrived in such a way as to highlight how the underlying mechanisms of flocking contrast with the basic tendencies of backpropagation. Here is a depiction of the basic network topology:

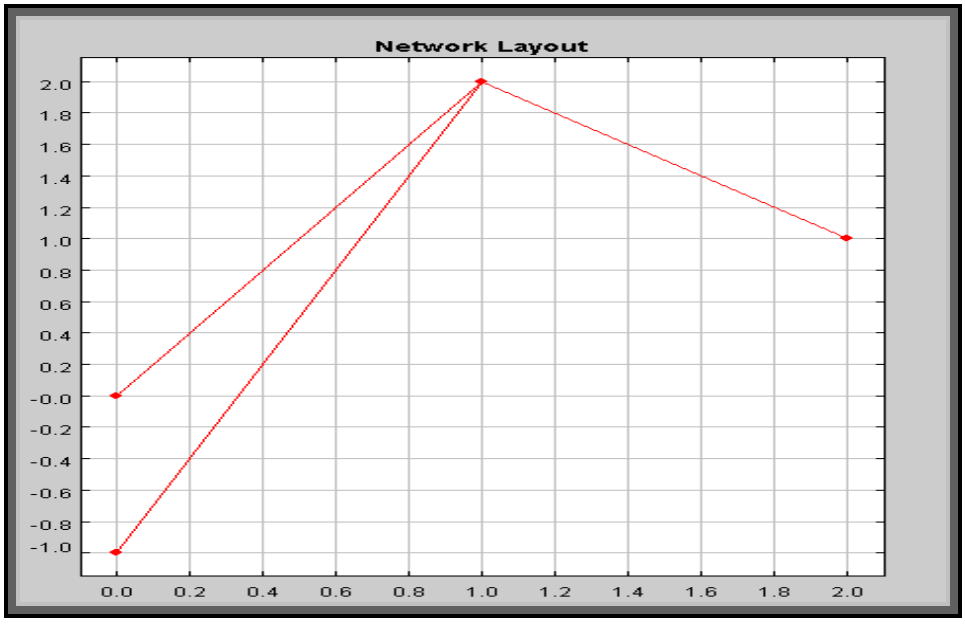

Figure 11.1 Flocking Demo Topology

Next here is a depiction of the eight 2-dimensional input example patterns of the training set overlaid with the hyperplane that represents the initial conditions of the single hidden layer neuron prior to any training:

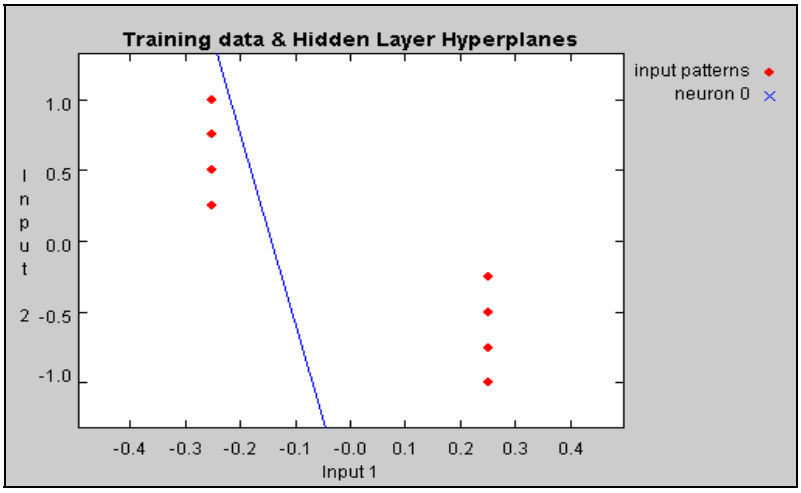

Figure 11.2 Demo data set with overlay of initial hidden neuron hyperplane

In this contrived training set the example patterns are split into two groups that more or less are representative of points along two different vertical lines. All four points along the first vertical line have the exact same target output of one, and the four points along the other vertical line all have the exact same target output of zero. On the following page the initial conditions of other various network components are displayed. These components include the initial hidden neuron output response and example influence for each pattern as well as the initial weights and a flocking graph that plots example influence as a function of net Input and includes a depiction of the weighted flocking mean and Nef_h .

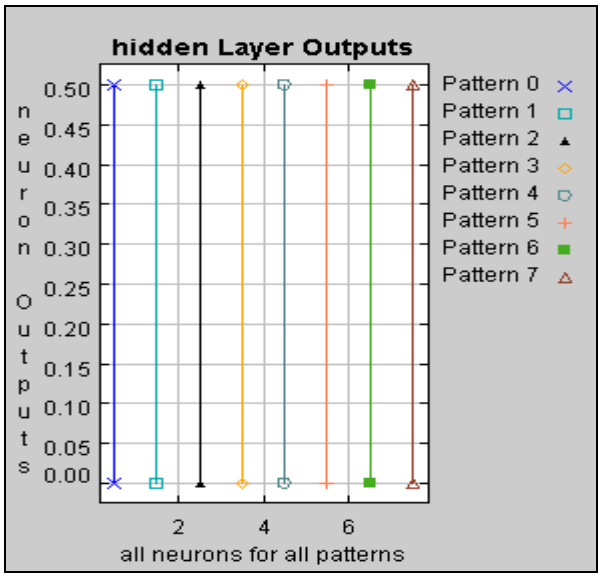

Figure 11.3
Initial hidden neuron Output Response

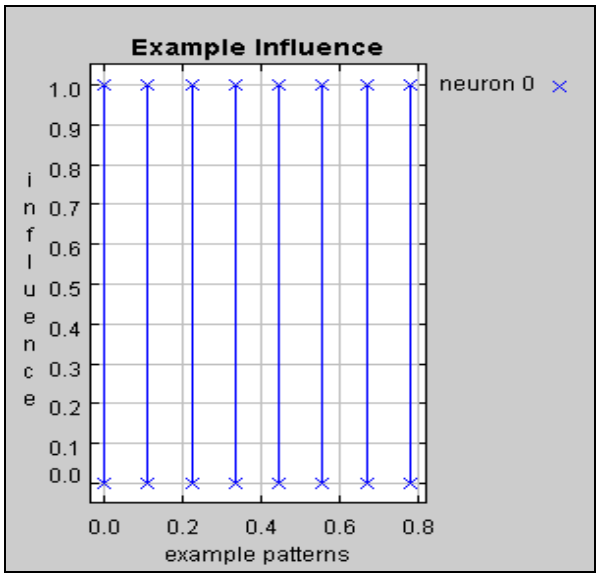

Figure 11.4
Initial influence of each example

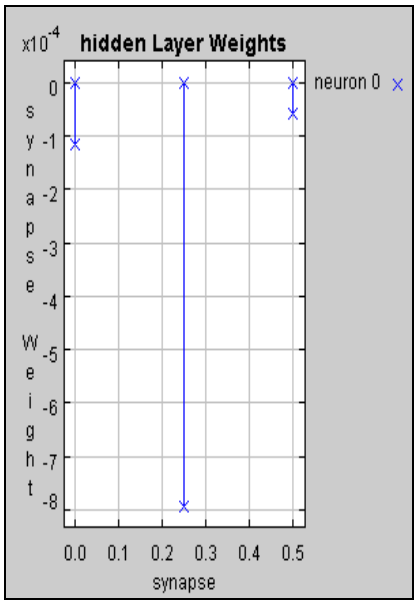

Figure 11.5
Initial hidden neuron Weights

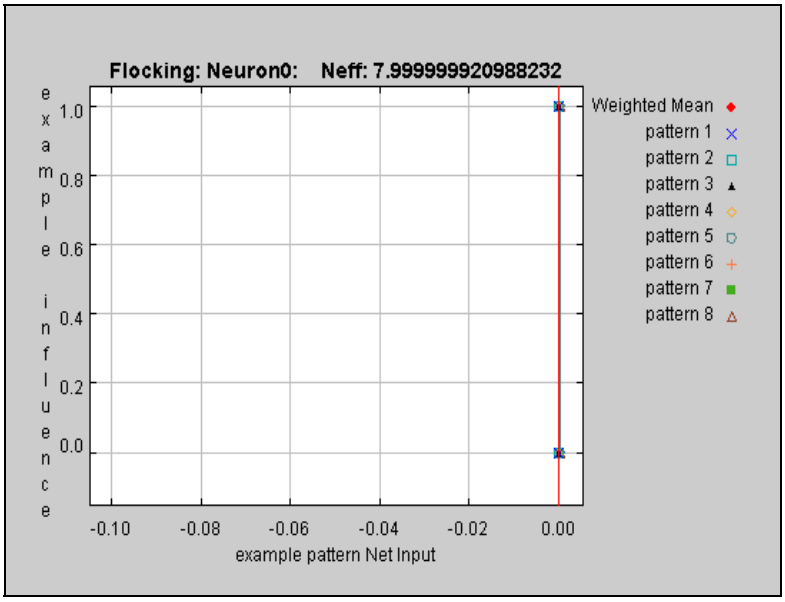

Figure 11.6
Initial example influence as a Function of Net Input

Now that we have a good idea of the state of the hidden layer neuron prior to training, next lets examine the state of the neuron after undergoing 1000 epochs of standard backpropagation training. In the following figure we see the hyperplane division that such a training session arrives at (this result is indicative of what will happen regardless of initial weight conditions).

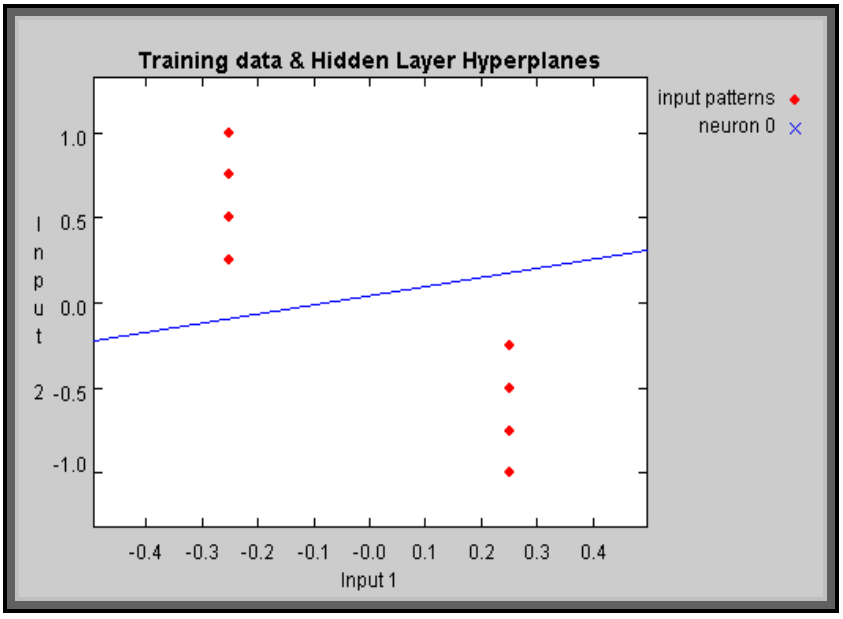

Figure 11.7 hidden neuron hyperplane after backpropagation training

Obviously if the example patterns are indeed representative of a more general data set consisting of two straight lines, then the solution arrived at by backpropagation training may not provide us with good generalization on unseen data. Even though we can see this striking tendency of the patterns to "line up" and be divided using input 1, backpropagation sees an even larger division with respect to input 2, which is why the hyperplane is more horizontal than vertical. Following are the other neuron components after backprop training.

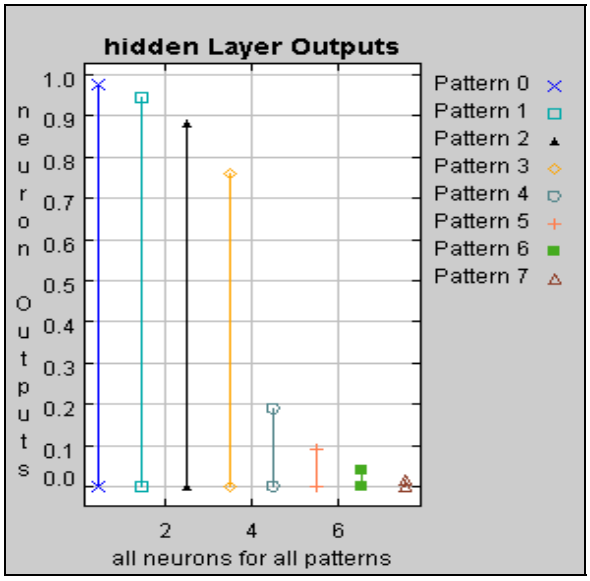

Figure 11.8
Preflock hidden neuron Output Response

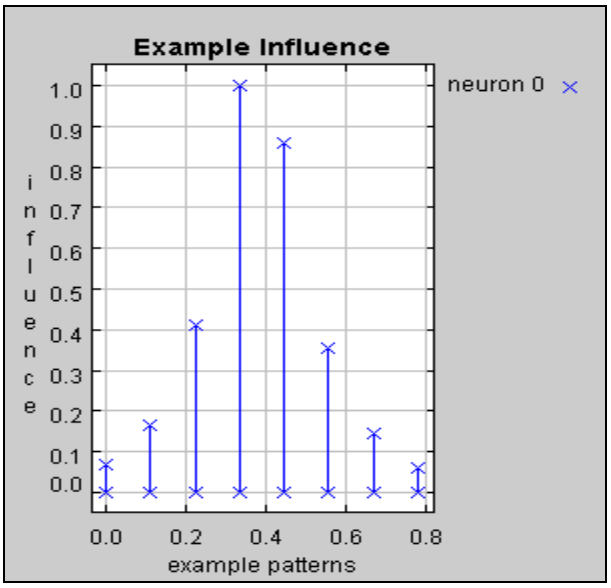

Figure 11.9
Preflock influence of each example

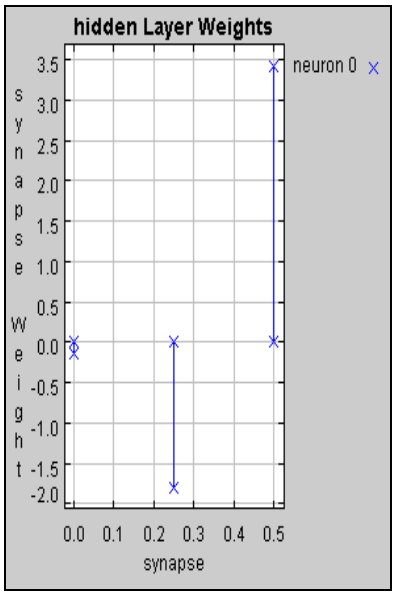

Figure 11.10
Preflock hidden neuron Weights

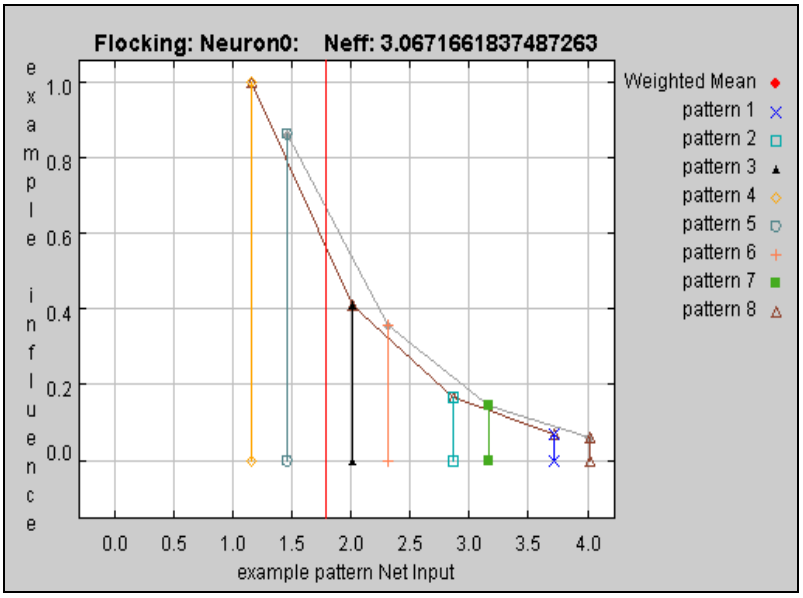

Figure 11.11
Preflock example influence as a Function of Net Input

There are a couple things to note from the figures on the previous page. For starters, look at the unevenness of the output responses for the different example patterns; ideally every pattern in the same target output group should receive the same output response from the hidden layer neuron. Next look at the example influences. Even though all example patterns should theoretically be considered equally when trying to influence the decision of a neuron, due to the placement of the hyperplane the two examples directly on either side of the hyperplane carry much more influence than do the two examples furthest from the hyperplane. As far as the neuron weights, simply note that they have increased dramatically in magnitude.

In the flocking figure, we note that the example patterns have moved apart from one another and that each has established a unique net input, which is used in combination with the example influence to calculate the displayed flocking mean. The "uniqueness" of these net inputs is the direct cause of the "uniqueness" of the previously displayed output responses. Recall that in the mean calculation all net inputs are reflected across the y-axis. This concept is displayed by connecting all the negative example patterns to one another as well as all the positive example patterns to one another. Also note that the closer an example pattern is to the zero crossing of the hyperplane, the larger its example influence, and that the effective number of examples has dropped dramatically since beginning of training (from 7.999 to 3.067).

In the final stage of this simulation we apply the flocking algorithm to the network that we just trained using backpropagation. The following figures represent the state of the

hidden neuron hyperplane at the conclusion of 750 flocking epochs and the sum squared error graph for duration of the entire training process.

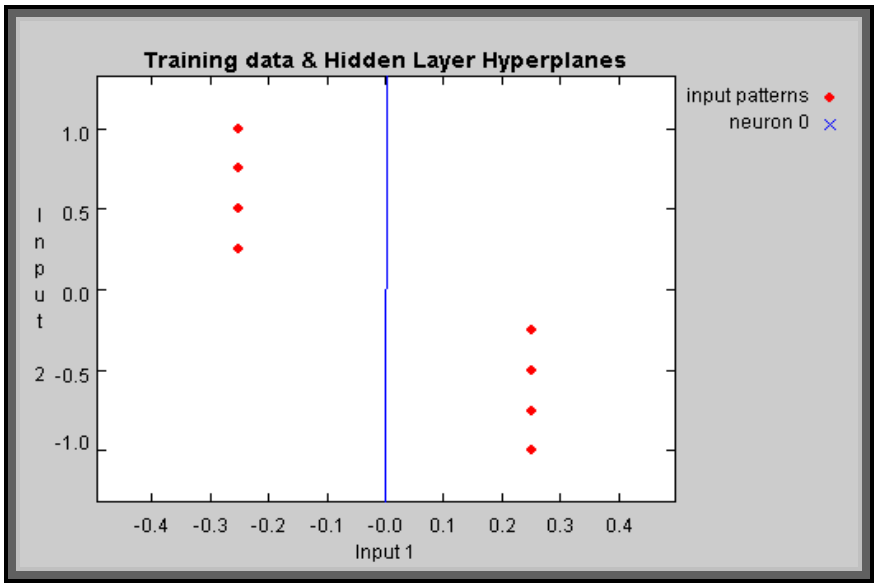

Figure 11.12 Hidden neuron hyperplane post backpropagation training

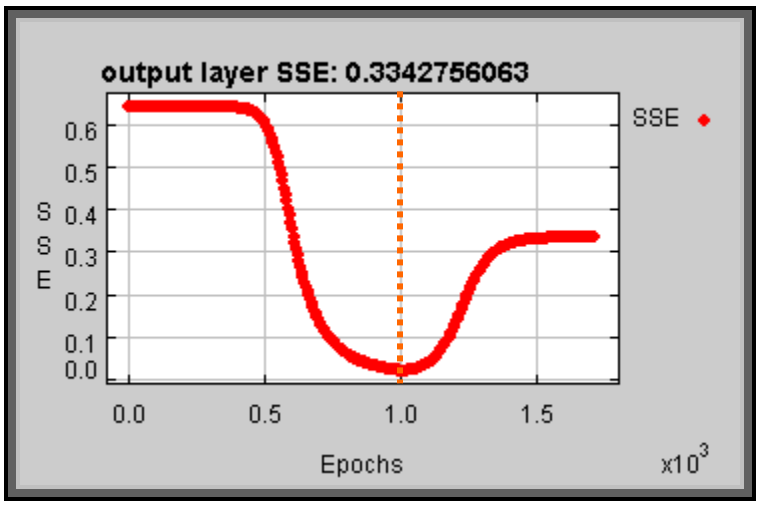

Figure 11.13 Sum Squared Error during backpropagation and flocking

And here are the states of the hidden neuron components at the conclusion of flocking:

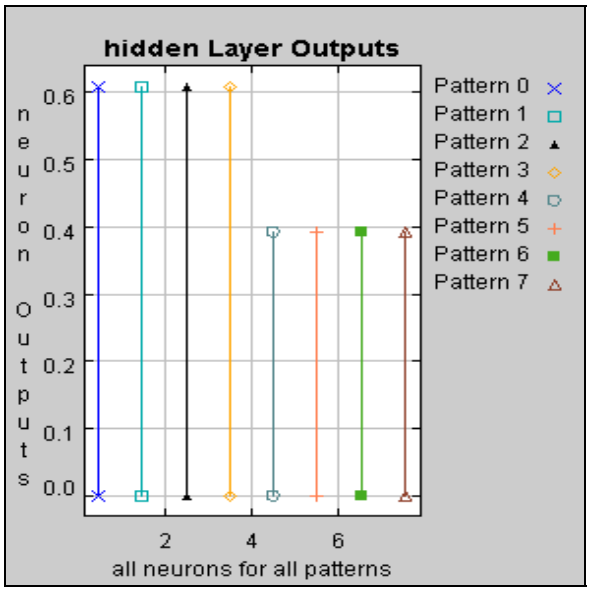

Figure 11.14
Final hidden neuron Output Response

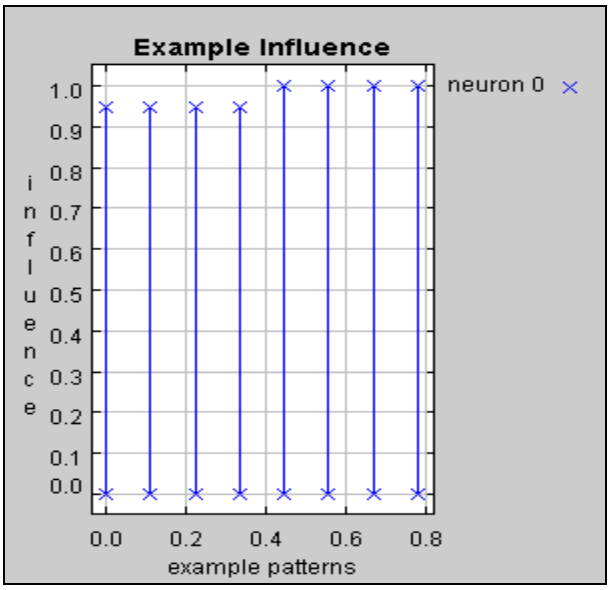

Figure 11.15
Final influence of each example

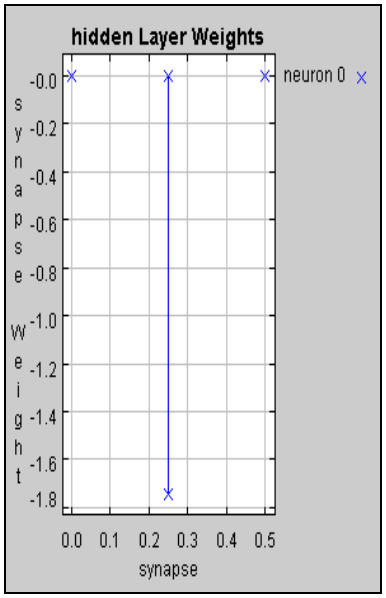

Figure 11.16
Final hidden neuron Weights

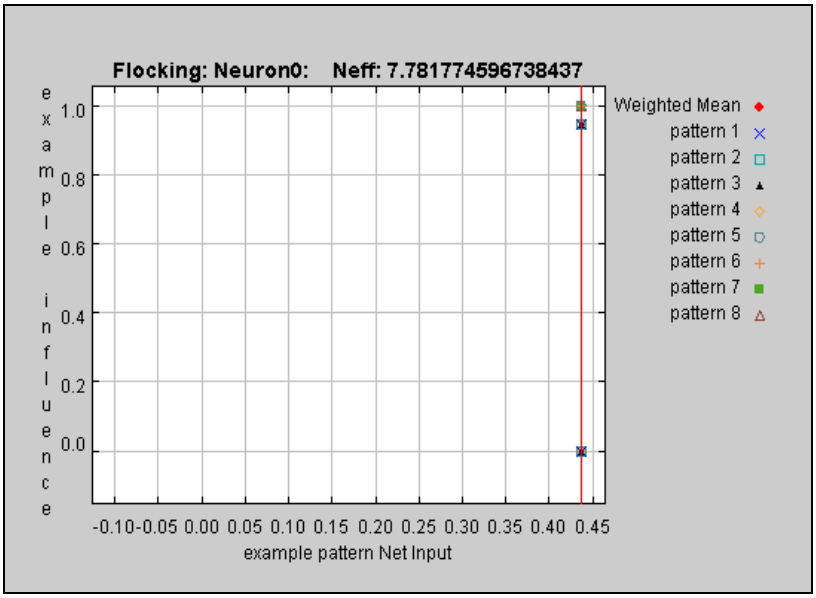

Figure 11.17
Final example influence as a Function of Net Input

From the above simulation, we can conclude that there are certainly some positive aspects to the flocking algorithm. The resulting hyperplane is a nice intuitive division and the output responses for the example patterns in each group have converged nicely. Also all the examples now have approximately the same influence on the network outputs, which has resulted in the effective number of examples being raised to 7.781.

There are some downsides to flocking too however. For one there is a very noticeable increase in error as a result of flocking (shown in Figure 11.13). Also look at the decrease in the mean net input from Figure 11.11 to Figure 11.17. Both these negative effects are directly due to the weight decreasing property of flocking. If we examine the final weights of the hidden layer neuron depicted in Figure 11.16, we see that the first and last weights representing the bias and the input 2 weights respectively are both very close to zero in comparison to the middle input 1 weight. This says that for all practical purposes, the input 1 weight is the sole decision maker in the network. The fact that the input 1 weight did not decrease with the other two is the result of using such an ideal data set. Most real world data sets will not converge to such a nice state because no perfect solution usually exists to the flocking error criterion other that the state where all weights are zero.

This is a very simplistic simulation, but it has none-the-less demonstrated the core behaviors that the flocking algorithm exhibits on a given data set. Some of these behaviors must be exploited while others must be dealt with if the flocking algorithm is to be harnessed to improve generalization.

Appendix 2: Implementation Details

This appendix is intended to provide a brief overview of the coding scheme used in the implementation of this research. For a more detailed explanation please refer to the code itself and the comments contained therein. A copy of the code is freely available and is included as a zip file in the same directory as this thesis paper. Inside the zip file you will find all of the Java code and all of the text training files used to conduct each of the experiments described in this paper.

Zip file containing Java code and training files

The following information will briefly review some of the core components of the JOONE framework but will mainly focus on my implementation over top of that framework and how this implementation was used to conduct the experiments described in this paper. For more information on using the JOONE core engine Java classes or to obtain a copy of the JOONE core engine code, please visit the JOONE web site at the following location:

http://joone.sourceforge.net/

The graphics classes of my implementation were all written over top of a Java framework called Ptolemy. For more information on Ptolemy and using the this framework to write graphics software, please visit Ptolemy's web site at the following location:

http://ptolemy.eecs.berkeley.edu/java/ptplot/

As stated in section 2.4, JOONE (Java Object-Oriented Neural Engine) is a framework for constructing neural networks out of *Layer* and *Synapse* objects. The JOONE core engine actually provides much more than just an object-oriented approach, it provides a multi-threaded approach as well. Each Layer object implements the Java interface *Runnable*, which means that all *Layer* objects are actually executed as individual threads at runtime. The motivation behind this is two fold. First it provides a nice level of abstraction when considering the parallel nature of artificial neural networks. Second the code is actually written so that it can be physically executed on multiple processors.

Each *Layer* object contains a *Vector* of input *Synapses* and a *Vector* of output *Synapses*. During execution a *Layer*'s run() method enters a while() loop whose entire execution takes it through the processing of a single example pattern forward and backward through all input and output *Synapses*. The four critical methods called in a single pass through the while() loop are the following: fireFwdGet(), fireFwdPut(), fireRevGet(), and fireRevPut(). These methods call respective methods in all connected input and output *Synapses,* and the whole process can be thought of as getting neuron net Inputs, putting neuron output responses, getting backpropagated error being passed back, and putting new backpropagated error to previous *Layers*.

Both the *Layers* and *Synapses* contain weights that affect the operation of the network. The *Layer* objects contain the individual layer neuron bias weights and the *Synapses* objects contain all the weights that form the synapse connections between the *Layers*. Both the *Layers* and *Synapses* also update their respective weights in a similar fashion by calling their internally defined protected methods forward() and backward().

These two methods contain all the feed-forward and backpropagation equations that power a backpropagated artificial neural network. The forward() and backward() methods employ the OO methodology discussed in section 2.4 in order to implement the equations in a relatively simplistic fashion.

Among the other more important components in the JOONE core engine are the *NeuralNet* class, the *Monitor* class, and the *NeuralNetListener* interface. The *NeuralNet* class is essentially just a container for managing *Layer* and *Synapse* objects but provides a very nice layer of abstraction. Each *NeuralNet* object contains a single *Monitor* object that monitors the network for certain events and interfaces the outside world when they occur. The *Monitor* contains a list of *NeuralNetListener* objects given to it by whatever code is being laid overtop the JOONE framework. When an event is fired the *Monitor* hears it and calls the event's respective method in each of the *NeuralNetListener* objects that the *Monitor* has in its list. In this way Objects that implement the *NeuralNetListener* interface can gain access to the network at critical points in time during the execution. I will now move on to describing how my coding implementation uses and extends the JOONE framework.

The top-level package in my code is called: *thesiswork2*. This package contains six classes and two additional packages. Four of the six contained classes directly extend different various *Layer* and *Synapse* objects in JOONE, and the other two are implementers of the *NeuralNetListener* interface previously described. The remaining two packages are called *graphics* and *tests*, and they contain additional graphics classes and simulation test code respectively.

The four classes in the *thesiswork2* package that extend existing JOONE components are the *SmartSigmoidLayer*, the *SmartFullSynapse*, the *SmartTeachingSynapse*, and the *SmartTeacherSynapse* and each extends the JOONE class *SigmoidLayer*, *FullSynapse*, *TeachingSynapse*, and *TeacherSynapse* respectively (for details on the implementation of the existing JOONE classes, again please visit the JOONE web site at the URL given above).

The *SmartSigmoidLayer* extends the functionality of the *SigmoidLayer* class quite extensively. Following is a list of functionality that the *SmartSigmoidLayer* provides in addition to the functionality provided by the parent JOONE class:

- The ability to add and remove neurons from a layer at will
- The ability to lock individual bias weights
- The ability to magnify bias weights
- The ability to batch train
- The ability to use adaptive learning rates
- The ability to prevent the layer from learning
- The ability to prevent error from being backpropagated to the previous layer
- The ability to calculate flocking derivatives
- The ability train in flocking mode or split flocking mode
- The ability to store netIns, neuron outputs, backpropagated netIns, and backpropagated output derivatives for all example patterns in an epoch

The class methods that provide the listed functionality are for the most part self-explanatory and commented; so for detailed information on the implementation of the *SmartSigmoidLayer* please refer to the actual code.

The next core extension of the JOONE framework is the *SmartFullSynapse*. The *SmartFullSynapse* class extends the functionality of the *FullSynapse* class in many of the same ways that the *SmartSigmoidLayer* extended the *SigmoidLayer*. This new class is responsible for creating the objects that have been described up to this point as simply *Synapse* objects (both the *Layer* and *Synapse* classes are actually abstract and do not create objects directly). Here is a list of functionality that the *SmartFullSynapse* class provides:

- The ability to lock individual synapse weights
- The ability to magnify individual synapse weights
- The ability to batch train
- The ability to use adaptive learning rates
- The ability to prevent the synapse from learning
- The ability to prevent the synapse from backpropagating error
- The ability to train in flocking mode

For more information on the individual methods of the *SmartFullSynapse* class and its implementation please refer to the actual code.

The third extension of the JOONE framework is the *SmartTeacherSynapse*, which extends the *TeacherSynapse* class. The extended functionality in this class provides the means for all the multi-task training that takes place in the experiments of this paper. Following is a short list of the new abilities added to the *TeacherSynapse*:

- The ability to ignore (or unIgnore) any output for any example pattern in the training set
- The ability to train in flocking mode
- The ability to decide whether or not to control the global error used by other objects

For more information on the individual methods of the *SmartTeacherSynapse* class, please refer to the actual code.

The remaining framework extension is the *SmartTeachingSynapse* class and this extension is a much more minor one. The *SmartTeachingSynapse* class provides no real new functionality to speak of, but serves simply as a wrap around class in order for the *SmartTeacherSynapse* class to function properly.

This brings us to the most important class in the code used to conduct these experiments, the *MasterNet* class. This class is an implementer of the JOONE *NeuralNetListener* interface and is responsible for a great deal of things including the execution of both the flocking and convergence training intermediate learning algorithms. The *MasterNet* class actually contains the JOONE *NeuralNet* object that it is listening to, so

it is and all-inclusive class for implementing artificial neural networks. In fact, three simple lines of code are all that's needed in order to create and train a default network:

```
MasterNet mNet = new MasterNet("networkSaveFileName", 2, 6, 16);
mNet.setDataFile("testSetFileName", true);
mNet.execute("trainingMode");
```

The *MasterNet* class is only capable of representing networks with a single hidden layer (multiple hidden layer networks are constructed using the *ExpandingNet* class, which extends *MasterNet* and is the sixth and final class in the *thesiswork2* package). All *MasterNet* networks must be constructed by specifying an initial number of inputs, hidden layer neurons, and output layer neurons (2, 6, 16 respectively in the code above), but these parameters can be modified dynamically after construction. Also the network will remain inoperable until some sort of input file is specified for it.

There are four methods that *MasterNet* is required to define in order to implement the *NeuralNetListener* interface. These four methods are the cicleTerminated(), errorChanged(), netStarted(), and netStopped() methods. The *Monitor* object that was previously described calls all four methods. The last two only get called at the beginning and end of training and are not all that useful; the first two however are quite useful. The cicleTerminated() method is called at the termination of each full epoch of training on all example patterns in the training set and the errorChanged() method is called immediately after any sort of update to the global error (SSE).

All other methods in the class are there by my own design and here is a brief list of the functionality that these remaining methods provide:

- The ability to create a data set array of doubles from a given a text file
- The ability to add and remove graphics (of the type specified in the *graphics* package) that animate different various network components during training
- The ability to execute the network in a variety of different Modes
- The ability to retrieve *Layers* and *Synapses* by name
- The ability to print out a display of all currently locked weights
- The ability to print out a display of all currently ignored output neurons (and the respective example patterns for which they are ignored)
- The ability to print any *Layer'*s current output responses for all patterns in the previous epoch
- The ability to randomize *Layer* and *Synapse* weights
- The ability to turn on, turn off, or reset adaptive learning rates for all weights
- The ability to save the *NeuralNet* object (and all its "*Smart"* components) to a binary file for restoration at a later date
- The ability to turn batch training on and off while the network is training
- The ability to specify either a single training set or a training and testing set In the form of a text file
- The ability to set pertinent split flocking and convergence training parameters

- The ability to set the refresh rate at which all current graphics are updated (this parameter directly affects training times)
- The ability to update graphics directly to reflect the current state of the network
- The ability to set the network's learning rate, momentum, maximum training epochs, or target sum squared error while the network is executing

As stated in the above functionality list, the *MasterNet* has the ability to execute itself in different various modes specified by the *String* parameter of the execute() method. The execute() method itself is a simple design that prepares the network to execute in a particular mode by calling setMode() (which does all the necessary preparation for the network to run in the specified mode) and then kicks off a network execution by calling some pertinent methods in the *NeuralNet* and *Monitor* objects. All necessary JOONE network components are then spawned off as independently executing threads and the execute() method then simply puts the main thread it is executing on to sleep and checks periodically to see if the network has finished training.

Currently the *MasterNet* is setup to handle six different modes of operation, but this is easily expandable by simply creating another entry in the setMode() method. The first mode of operation is "trainingMode" which will result in straightforward backpropagation training. The second mode of operation is "recallMode" which results in simple feed forward network execution without modifying any weights. The third mode of operation is "flockingMode" which sets all necessary parameters in the network required to use the flocking algorithm on

the hidden layer neurons of the network. While the network is executing in "flockingMode" the *SmartTeacherSynapse* always returns a one for the error associated with all unignored outputs (ignored outputs return a zero regardless of the mode of operation) and all associated output layer synapses are locked and backpropagating the flocking error derivatives to the hidden layer neurons. The remaining three modes of operation are essentially combinations of these first three operational modes and as a result they each have some important functionality defined in the previously described cicleTerminated () method.

"splitFlockingMode" is the first of the remaining three modes. Recall that in the split flocking algorithm neurons are trained for a period of time using backpropagation before being trained with flocking. While operating in "splitFlockingMode" pertinent code within the cicleTerminated() method allows the network to actually switch back and forth between "trainingMode" and "flockingMode" without halting network execution (which takes up a lot of valuable time due to having to kill off of all the threads involved and then restart them for the network to execute in the specified mode).

"convergenceTrainingMode" is the second of the remaining three modes of operation and works in a very similar fashion to the "splitFlockingMode" in that it too jumps back and forth between "trainingMode" and "flockingMode" in order do convergence training. This mode however, also requires additional code in the errorChanged() method (which is used when executing in "trainingMode" to identify when the target SSE has been reached) to identify when enough neurons have been added and trained to meet the convergence training SSE criterion.

The final mode of operation is called “validationTrainingMode”. When executing in this mode the network jumps back and forth between “trainingMode” and “recallMode” also using pertinent code within the cicleTerminated() method. The purpose of this mode is to allow simultaneous training on a training set while testing on a test set.

When assigning the training set or a train/test set for use in the *MasterNet*, a filename is specified which refers to a simple ASCII text file. The format of this file is very straightforward; basically any character other than a number character, a ‘.’ character or a “-“ character is interpreted as a comment or data separating character. This means that you can include all the comments you want anywhere you want as long as the comments don’t contain actual numbers, dots, or dashes. The actual data should be specified numerically with a different pattern on each line of the text file. This means that each row of the text file should contain all the inputs and target outputs for a particular example pattern. Other than that the patterns just need to be specified consistently. The actual methods in *MasterNet* that extract the patterns from the text file are called setDataFileWithValidationSet() and setDataFile(). Simply specify row and col ranges in the parameters of these methods to designate the inputs from the targets and training patterns from testing patterns. Copies of the text files used to specify the data sets used in this research should be freely available at the same location as this pdf file.

After extraction of the actual data from the text file, *MasterNet* creates a JOONE object called a *MemoryInputSynape* and another JOONE object called a *MemoryOutputSynapse*. Each of these objects simply performs the duty of injecting the

required data into the network when their respective fwdGet() methods are called by the input layer or teacher components respectively (for details on the operation of JOONE *MemorySynapse* components please refer to the JOONE web page at the URL given above)

The *ExpandingNet* class is a basic extension of the *MasterNet* class that provides multiple layer functionality. Really the only major difference between the two classes are the respective constructors and the addition of an addNewLayer() method. This new method provides very specific layer insertion in accordance with the multiple hidden layer learning algorithm described in section 6. With a little effort this method could be generalized in order to produce a more robust layer insertion routine, but exists in its current state solely for the purposes of this algorithm.

The next point of interest in the implementation of this research is the bundle of classes contained in the previously mentioned *graphics* package. Almost all the figures displayed in the body of this text come from screen captures of these graphics objects. Examination of this package will reveal an abstract class called *AbstractGraphic* from which all other classes in the package are extended. The reason this hierarchy exists is for easy handling of the graphics object from within the *MasterNet*. All graphics that extend the *AbstractGraphic* class are required to implement a critical method called update() and the *AbstractGraphic* itself extends the Java swing component *JFrame* so it can be displayed and repainted. In this way the *MasterNet* can simply have a *Vector* of *AbstractGraphic* objects

and call all their individual update() routines every so often in order to provide an animated look into network components that make up the *MasterNet*.

None of the graphic component classes would be possible without the help of the Java Ptolemy framework. The foundation code provided in Ptolemy allows for the relatively easy creation of nice looking fast graphic plotting software. All of the graphics in the *graphics* package utilize a Ptolemy defined class called *Plot* which can be used in a multitude of ways to create nice GUI plotting interfaces. Each of the different graphics classes is given access to relevant components of the *MasterNet* and simply uses these components as the parameters in their individual update() routines to modify their Ptolemy *Plot*. After that, the update() routines simply repaint the *Plot* itself which exists on the *ContentPane* of the of the current graphics object executing the update() method (remember that each graphics object is extend from the Java *JFrame class*).

The final component of the *thesiswork2* package is the package entitled *tests*. In this package lies all the top-level test and simulation code. All classes that have any direct relationship to the simulations described in this paper will begin with the prefix "SIM", all other classes in this package are tests not directly discussed in this thesis paper. I won't go into much detail with regards to this package because I have already described the major algorithm simulations that these tests conduct in the main body of this text and all the tools that these test use in the body of this appendix. The simulations that begin with "A_", "B_", "C_", and "D_" are just slightly tweaked versions of the other tests with the same name (these are the simulations I used during my final oral exam). The format of each test is

simply a class consisting of all static methods of which there is always a main() method so that the class can be executed. The basic scheme of each test should be self evident from examination this main() method inside each test class. To create additional tests that employ the use of *MasterNet* objects or *ExpandingNet* objects simply follow the basic scheme outlined in the executable test classes contained within the *test* package.

In addition to the code included in the accompanying files, one will also need access to the basic Java API. The following Java packages are also required for the software to function properly: *java.util*, *java.io*, *java.awt.event*, and *java.lang*. If you have any questions or comments about this code or the research in this thesis paper please feel free to contact me. My email address is matthewpotter@msn.com. Cheers!