A Framework for an Automated Neural Network Designer using Evolutionary Algorithms

by
Markian D. Hlynka
B.C.Sc. (University of Manitoba) 1997

A dissertation submitted in partial satisfaction of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Manitoba,
Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

A Framework for an Automated Neural Network Designer Using Evolutionary Algorithms

BY

Markian D. Hlynka

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

A Framework for an Automated Neural Network Designer using Evolutionary Algorithms

"And because, in all the Galaxy, they had found nothing more precious than Mind, they encouraged its dawning everywhere. They became farmers in the fields of stars; they sowed, and sometimes they reaped.

And sometimes, dispassionately, they had to weed."

Arthur C. Clarke
3001: The Final Odyssey

Acknowledgments

Many people have contributed to the completion of this dissertation in many ways: explicitly, unwittingly, serendipitously. I hope to name them herein. As my memory is encased in a biological substrate, it is fallible. If your name does not appear here, forgive me: I am thankful to you as well.

In no particular order, I would like to thank:

My advisor, Dr. David Scuse, for his support and encouragement.

My committee members, Dr. Robert Tait and Dr. John Anderson.

My parents, for their own unique form of support.

My grandmother, who kept asking, "are you finished yet?"

My brother Anthony.

The professors who have most supported and encouraged me:

Dr. Bill Kocay
Dr. John Bate
Dr. Neil Arnason
Dr. Dekang Lin
Dr. John Anderson
Dr. Peter Graham
Dr. Hugh Williams

Gilbert Detillieux and Tom Dubinski, who keep our computers running.

Lynne Romuld, without whom I wouldn't have managed to get anything done.

Susan Harder, who has now managed to put up with me for years.

My friends and fellow students, of whom I can not possibly name but a few. Significant contributions of debugging, proofreading, and designing were made (in no particular order) by:

Hamish Carr
Andrea Mantler
Ryan Szypowski
Daniel Neilson
Doug Hamilton
Brian Doob
Patrick Pantel
Jenny Chadee
Cathy Leung
Sara Arenson

Finally, thanks to some of my greatest sources of inspiration and motivation:

R. Daneel Olivaw
Susan Calvin
HAL 9000
Rossum's Universal Robots
Deep Thought
The Shockwave Rider
Dr. Who
K-9

To all who became involved in this thesis, principally and peripherally, I extend my gratitude and my thanks.

Markian Hlynka
The University of Manitoba
July 1999

Abstract

A Framework for an Automated Neural Network Designer using Evolutionary Algorithms

by

Markian Hlynka

Master of Science in Computer Science

University of Manitoba

One of the major stumbling blocks of neural networks is the difficulty of designing the networks. Networks must be created by experts who understand both the problem domain and the process of developing neural networks. For complex problems, the process, even for experts, can be an intuitive rather than ratiocinative process. Evolutionary and genetic algorithms are a robust, probabilistic search strategy that excel in large, complex problem spaces. Research involving the application of evolutionary algorithms to neural networks for purposes of both training and selection of an optimal network has been carried out. The focus of such research, however, has been to generate an optimal network of a given structure. No generic framework exists which allows for the automation of the network creation process – the selection and design of the architecture – for a particular problem. This thesis is concerned with the design of such an evolutionary framework. The system is subsequently evaluated with backpropagation networks on an unknown data set. A new method of evolution, probabilistic Lamarkian learning transfer, appears to produce the desired results.

Table of Contents

Acknowledgments iv

Abstract vi

CHAPTER 1 Introduction 1

1.1 Neural Networks 1
1.2 Evolutionary Algorithms 2
1.3 Overview of Thesis 4

CHAPTER 2 Neural Networks 5

2.1 Neural Networks: A brief history 5
2.1.1 In The Beginning 5
2.1.2 Learning 6
2.1.3 Overcoming the Block 7
2.1.4 Onward 8
2.2 Neural Network Basics 8
2.2.1 Overview 8
2.2.2 The Neuron 9
2.2.3 Mechanics 9
2.2.4 The Activation Function 10
2.2.5 Learning 11
2.2.6 The Layer 12
2.2.7 The Network 13
2.2.8 Linear Separability 14
2.3 Backpropagation 16
2.3.1 The Network Model 16
2.3.2 Supervised and Unsupervised Learning 17
2.3.3 The Learning Algorithm 18
2.3.4 The Hidden Layer 19
2.3.5 Momentum 20
2.3.5.1 The Problem 20
2.3.5.2 The Contradiction 21
2.3.5.3 The Solution 22
2.3.6 Bias Values 23
2.4 Training Problems in Backpropagation 24
2.4.1 Initial Weights 24
2.4.2 Number of Hidden Units 25
2.4.3 Length of Training 26
2.4.4 Evaluation Strategies 26
2.5 Summary 27

CHAPTER 3 Evolutionary Algorithms 28

3.1 Background 28
3.2 Mechanics 29
3.2.1 Overview 29
3.2.2 A Simple Example of an Evolutionary Algorithm 30
3.2.3 A Detailed Example of a Genetic Algorithm 34
3.2.4 Summary 39
3.3 Comparison to Other Search Methods 40
3.3.1 Calculus Based 40
3.3.2 Enumerative 41
3.3.3 Random 41
3.3.4 Random Versus Randomized 41
3.3.5 Genetic Algorithms 41
3.4 Neural Networks as Candidates for Genetic Search 42
3.4.1 Motivation for Evolutionary Networks 43
3.5 Neural Networks and Evolution in the Literature 44
3.5.1 Initial Weights 44
3.5.2 Training and Evolution 45
3.5.3 Adaptive Neuroevolution 46
3.5.4 Other Mentions 47
3.6 Summary 47

CHAPTER 4 Structure of the System 48

4.1 Goal 48
4.1.1 Implementation 48
4.2 Neural Network Implementation 49
4.2.1 The Layer 49
4.2.2 The Network 50
4.2.3 The Backpropagation Class 50
4.2.4 Network Evaluation 51
4.2.5 System Verification 52
4.3 Evolutionary Algorithm 53
4.3.1 Representation 54
4.3.2 Pre-reproduction approaches 55
4.3.3 The Fitness Function 57
4.3.4 Standard Genetic Operations 58
4.4 Consolidation: The System 59
4.4.1 Mating Trained Weights 60
4.4.1.1 Lamarkian Evolution 60
4.4.1.2 Mixing Weights in Neural Networks 61
4.4.1.3 Nurture overcomes Nature 62
4.4.2 Mating Initial Weights 63
4.4.2.1 The Baldwin Effect 63

4.4.2.2 Baldwin Evolution in Neural Networks . . . 63
4.4.2.3 Distributed Learning Trials and Baldwin Evolution . . . 64
4.4.3 Probabilistic Lamarkian Learning Transfer . . . 64
4.4.4 The Algorithm Revisited . . . 66
4.5 Summary . . . 67

CHAPTER 5 Experimentation . . . 69

5.1 Description . . . 69
5.2 Purpose . . . 69
5.3 Materials . . . 69
5.3.1 Computer . . . 69
5.3.2 Dataset . . . 70
5.3.2.1 Features . . . 71
5.4 Methodology . . . 73
5.4.1 The Train/Test Strategy . . . 73
5.4.1.1 k-fold cross validation . . . 74
5.4.1.2 Separating the Validation of the Algorithms . . . 75
5.4.2 Training by hand . . . 75
5.4.3 N-fold Crossvalidation . . . 76
5.5 Experiment . . . 76
5.5.1 Setup . . . 77
5.6 Conclusion . . . 80

CHAPTER 6 Conclusion . . . 82

6.1 Summary . . . 82
6.1.1 Successes . . . 82
6.1.2 Caveats . . . 82
6.1.2.1 Dataset . . . 83
6.1.2.2 Limited flexibility . . . 83
6.2 Future Work . . . 84
6.2.1 Improved Implementation . . . 84
6.2.2 Curved Fitness Functions . . . 84
6.2.3 Other Genetic Operators . . . 86
6.2.4 Network Types . . . 87
6.2.4.1 Mixed Population . . . 87
6.2.4.2 Segregated Population . . . 87
6.2.4.3 Cooperative Population . . . 87
6.2.5 Survival Traits . . . 88
6.2.5.1 Prevent Inbreeding . . . 88
6.2.5.2 Age-correlated Fitness . . . 88
6.2.5.3 Dataset Partitioning . . . 89
6.3 Conclusion . . . 89

Bibliography 91

Appendix A A-1

A.1 Long run, full EA with PLLT A-1
A.1.1 First Five Generations A-1
A.1.2 Final Five Generations A-13

Appendix B B-1

B.1 Neural Network Code B-1
B.1.1 Debug.h B-1
B.1.2 NLayer.h B-1
B.1.3 neural.h B-3
B.1.4 backprop.h B-6
B.1.5 NLayer.cc B-6
B.1.6 neural.cc B-10
B.1.7 backprop.cc B-21
B.2 Evolutionary System and Supporting System Code B-28
B.2.1 genetic.h B-28
B.2.2 netparms.h B-28
B.2.3 genetic.cc B-29
B.2.4 main.cc B-36

1 Introduction

This Chapter presents a brief overview of neural networks and their main failing, parameter selection.

1.1 Neural Networks

Neural networks are a non-symbolic approach to pattern recognition. Based on a loose paradigm of neurons in the brain, neural networks are able to pick out pertinent patterns in data, often when the data is incomplete, corrupted, noisy, or uncertain. While their training processes can be slow, completed neural networks are generally quite fast in application. Their strengths include the ability to generalize large numbers of patterns into classes, and to learn from a presentation of example problems and solutions. One major impediment to the design of neural networks is the selection of an ideal set of parameters for a particular problem.

Neural networks are hand-crafted by experts with years of experience in the field. Two major drawbacks of this approach are a lack of experts, and a lack of a rigorous design methodology. The first problem is straightforward enough: there simply are not enough experts to attend to all the potential neural network projects the world has to offer. The second problem is somewhat more subtle. No tractable algorithm exists to optimally determine the parameters for a particular neural network application. The result of this complex-

ity is that the science of designing neural systems is imprecise at best. At worst, the process is guided by intuition alone. To this end, a system is required to determine neural network designs more efficiently and in greater number than the experts can manage.

It is unreasonable to expect that any single neural network will be able to solve any problem regardless of complexity. To address this problem, research is being conducted into modular neural networks. In these systems, several networks cooperate to solve a problem which would be unsolvable by any single neural network architecture. While the power and flexibility of the resulting gestalt has the potential to outperform simple neural networks, the combination of multiple networks increases the difficulty of crafting the system. Whereas before an expert had to craft only a single network, the problem becomes one of designing multiple networks while simultaneously enabling them to cooperate on the problem at hand. The work load rises exponentially with the size of the system.

Clearly, a method is needed to free experts from the imprecise drudgery of hand-crafting networks. The recent trend to modular networks only exacerbates this need. One promising method of solving both problems is through the use of evolutionary algorithms (EAs). This thesis presents a systematic approach to automating the design of neural networks through the use of EAs.

1.2 Evolutionary Algorithms

Genetic algorithms were developed by John Holland at the University of Michigan. With his students and colleagues, Holland set out to achieve two goals. First, to "abstract and rig-

orously explain the adaptive processes of natural systems", and second, to "design artificial systems software that retains the important mechanisms of natural systems." [Goldberg] In other words, Holland was trying to find out how natural, biological systems manage to be so adaptive, and how this knowledge might be applied to artificial systems.

Due to the inherent difficulty in the process of creating neural networks, genetic algorithms have become a focus of study in the field. Using genetic algorithms, it is possible to remove some of the burden of trial and error design from the designer. Rather, the genetic algorithm is used to search a solution space for effective neural network parameters.

Genetic algorithms have been used to select various features of neural networks. These include learning parameters, hidden units, topology, connections, and even to evolve the synaptic weights themselves (a task usually achieved by the learning algorithm) [Korning; Caudill].

Possibly the most successful neuroevolutionary algorithm to date is the SANE algorithm devised by David Moriarty [Moriarty]. The SANE algorithm was highly successful in evolving effective neurons simultaneously to evolving effective cooperation among the neurons. The result was highly effective networks for sparse reinforcement problems: that class of problem where a series of decisions must be made before feedback is achieved. Often this feedback is very general; for example, success or failure.

Research seems to indicate that the combination of genetic algorithms and neural networks results in robust systems able to tackle a wide variety of problems.

1.3 Overview of Thesis

The remainder of this thesis will be consist of several parts. First, an in-depth discussion of neural networks will review the basic problems encountered when designing application specific networks. This will be followed by and overview of genetic algorithms, and a discussion of previous applications of EAs to neural networks. Chapter IV will present the structure of the system developed for this thesis, and Chapter V will discuss the experiments performed and results obtained. Finally, conclusions will be drawn in Chapter VI, and directions for future work suggested.

2 Neural Networks

The process of designing neural networks is subject to many pitfalls. This chapter provides an introduction to neural networks, followed by a more studied review of the backpropagation algorithm. Finally, the most common problems encountered in training backpropagation networks, initial weights, number of hidden units, training time, and evaluation method, are discussed; the most common solutions for dealing with these problems are presented simultaneously.

2.1 Neural Networks: A brief history

2.1.1 In The Beginning

Since the creation of simple computing machines and automata, researchers have been trying to make computers more that just complex calculators. They have been trying to make a machine that would think and reason as a human might. It is not surprising, then, that neural networks have their roots in psychology [Blum]. Early models of the workings of the brain were to lay the foundations for research into how the brain solves problems, and later, how to build machines that would solve problems in a manner similar to the human brain. Hence, the neural network approach is that of designing an algorithm modeled after the way the human brain works; a computer that ultimately thinks in the same manner as people.

The first neural network model was formulated by McColloch and Pitts in 1943 [Zurada]. Their network used digital neurons, and had no ability to learn [Blum]. The idea of learning ability in a neural network came from psychology with the work of psychologist Donald Hebb. Hebb devised the model by which learning is achieved through changes in synaptic strengths within the brain [Zurada]. This model, which came to be called Hebbian learning, was the foundation for a neural system which could learn.

2.1.2 Learning

Neuropsychologist D. O. Hebb described a rule for updating synapse strength between neurons in two distinct layers. Known as Hebbian learning, it was described by Hebb as follows: [Haykin; Mehrotra]

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

Mathematically, this can be described by the equation:

$$\Delta w_{i,j} = c x_i x_j$$

where c is a small constant often referred to as the learning rate and $w_{i,j}$ designates the strength of the connection from the jth node to the ith node. x_i and x_j denote the activation levels of the nodes. Thus, the change to the strength of the connection between two nodes is a proportion of the product of their activations; j's efficiency at firing i is increased.

This idea was implemented by Frank Rosenblatt in 1958 to create a two-layer network called a perceptron. [Blum, 1992 & Zurada, 1992] Rosenblatt's learning rule centered on the calculation of weight adjustment of synapses as a proportion of the error between the output neurons and the target, or expected, values. Rosenblatt also developed a theorem, the perceptron convergence theorem, by which he was able to prove that the weights of his model would converge to produce the desired results if such results were possible. Rosenblatt attempted to create a three-layer perceptron, but he was unable to conceive of a sound way of updating the weights between the input and middle, or hidden, layer of neurons.

Due to problems of linear separability, which will be discussed later, the application for two-layer networks was limited. The lack of a provable algorithm for updating hidden layer synapses in multilayer networks was a major problem. Some work continued to be done with two-layer networks, but despite such applications as associative memory and other learning based on Stephen Grossberg's models of the brain, the practical application of neural networks was limited. [Blum, 1992] Research into neural networks entered a stagnation phase, partly due to the multilayer problem, but at least partially due to the "modest computational resources available" then [Zurada, 1992].

2.1.3 Overcoming the Block

Finally, in 1974, the backpropagation network was developed by Werbos. It was largely ignored by the scientific community [Mehrotra]. The algorithm was independently redeveloped in 1985 by Parker and LeCun [Haykin]. The modern version of the algorithm, however, was popularized by Rummelhart, Hinton, and Williams [Mehrotra, Haykin].

Backpropagation allowed the training of those hidden layers that had been a stumbling block for so long. With this block removed, the potential application domain of neural networks was widened immensely. Backpropagation is a powerful algorithm for problem solving which, unlike two-layer networks, is able to effectively deal with non-linear pattern recognition. For some types of problems backpropagation allows solutions to be found which would be very difficult with conventional computer science techniques. [Blum, 1992]

2.1.4 Onward

With the discovery of backpropagation, neural networks were freed from their limited two-layer incarnation. The ability to use multiple layer networks opened up a huge realm of possibilities. Though currently not useful for all types of problems, neural networks are full of unexplored potential. It is "this mix of failure and success [that] offers the tantalizing suggestion that research will eventually produce artificial systems capable of performing a large percentage of the tasks that now require human intelligence, hence the exponentially increasing growth of neural network research." [Wasserman]

2.2 Neural Network Basics

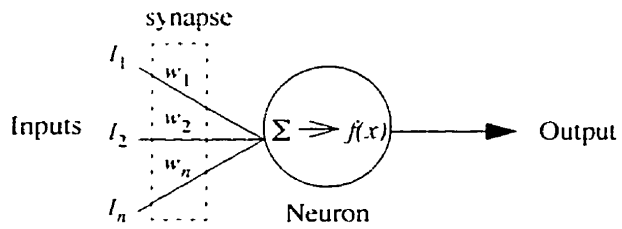
2.2.1 Overview

The basic unit of a neural network is a neuron. Neurons compose layers, and layers in turn compose a network. The following sections will detail each component of a neural network.

2.2.2 The Neuron

A neuron is a computational unit which takes a vector of input values and produces an output value. (Figure 1.1) Inputs can be received from other neurons or directly as input. A single output value is generated, which is either sent to each of the neurons in the next layer or becomes part of the final output of the network.

Figure 2.1. A simple neuron [Blum, 1992]



2.2.3 Mechanics

Inputs into the neuron are multiplied by the weights in the synapses. Biologically speaking, the synapse is that area between the end of one neuron and the start of the next one. The input synapses of biological neurons are normally located on dendrites – cellular filaments which consist, in essence, of many places for 'upstream' neurons to 'plug in'. The dendrite sums the activity from the synapses that occur on its surface. If the sum exceeds a threshold value, then the neuron discharges, or sends an output to the next neurons in the chain. Thus, a high activation of a particular neuron may cause a subsequent neuron to fire even if other neurons connecting to the same target neuron have low activations. Repeated propagation

across a synaptic cleft results in the strengthening of its ability to propagate signals [Haykin; Mehrotra; Mitchell].

In the neural network model used in artificial intelligence, the synaptic strengths are represented as a vector of weights, one per input to the neuron. The incoming values are multiplied by the weights in the synapses, and the products summed. The accumulated input to the neuron is the dot product of the input vector and the synaptic weights vector:

$$x = \sum_{i=0}^{n} I_i w_i \qquad \textbf{Equation 2.1.}$$

This weighted sum is often referred to as the neuron's stimulus. Next, the neuron then applies an activation function, $f(x)$, to the sum x. This activation value is the output of the neuron. In some cases, a threshold or bias value, θ, is added to x before the activation function is applied. Thus, like its biological counterpart, an artificial neuron fires when the sum of its inputs meets the criteria determined by the activation function.

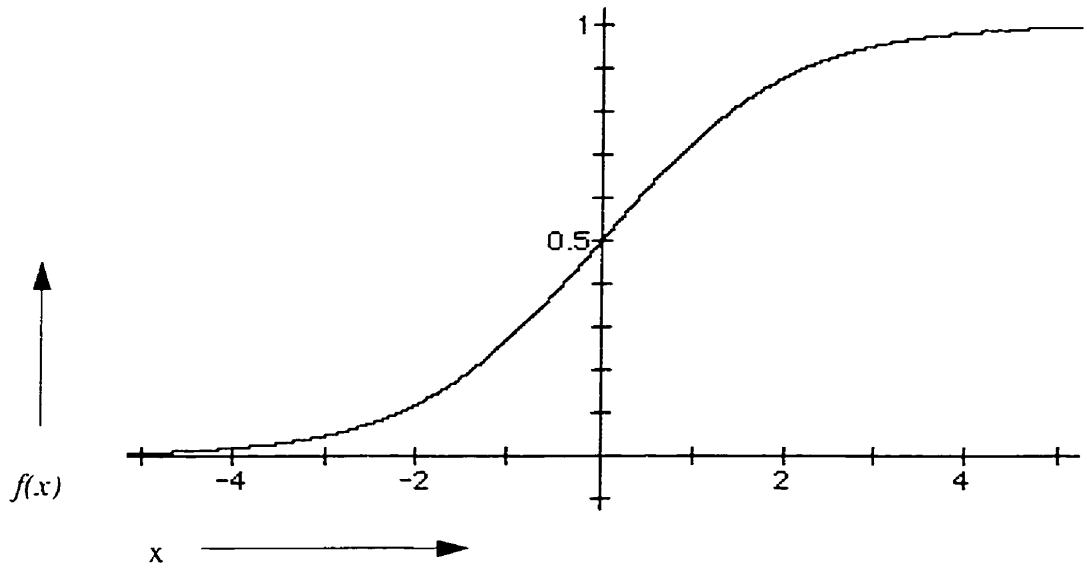
2.2.4 The Activation Function

The activation function, $f(x)$, is usually a nonlinear function which indicates the effect on the neuron of the inputs. For discrete neural networks a function f(x) such that f(x) produces 1 if $x > 0$, -1 where $x < 0$, and doesn't change the previous value of f(x) if x=0 might be appropriate [Blum, 1992]. For nondiscrete and analog networks, sigmoid functions are often appropriate, such as

$$f(x) = \frac{1}{1 + e^{-x}}$$

where f(x) is in the range [0,1]. This function is illustrated in figure 1.2.

Figure 2.2. Graph of sigmoid function.



2.2.5 Learning

A neuron's learning is effected through the adjustment of the synaptic weights. Learning takes place through the use of a learning rule, which is an algorithm for the adjustment of the synaptic weights. The speed of learning is moderated via a learning rate, α. The learning rate is usually in the range 0 - 1, and is used to calculate the change to each synaptic weight. There are many algorithms for synaptic weight adjustment [Blum, 1992]. A general learning rule is expressed in equation 2.2.

$$\Delta w = \alpha I e \qquad \textbf{Equation 2.2.}$$

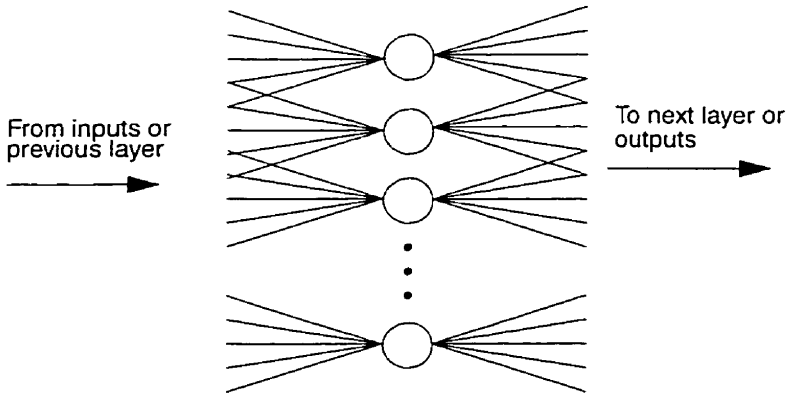
Here, the change to the weight is expressed as a product of the learning rate, α, the input, I, and the error in the neuron's output, e. I can be thought of as the proportion of the error, e, that the input is responsible for contributing. The input to the neuron could either be direct input from the environment or the activation of the previous neuron. Similarly, the

error could be a measure of the difference between the produced value and the expected value in the environment, or the difference between the actual and desired activation of the next neuron. Note that this adjustment is effected once for every input into the neuron.

2.2.6 The Layer

A layer is a number of neurons operating together, each neuron receiving the same inputs. What makes a layer an effective computational element is that each neuron has different synaptic weights which, when multiplied with the inputs, give each neuron a different value to which it applies its activation function. Normally all the neurons in a layer have the same activation function. It is also possible, however, for different neurons in a layer to have different activation functions. This is a highly advanced, often domain-specific subject, and is beyond the scope of this thesis.

Figure 2.3. A Layer of Neurons



The main purpose behind a layer of neurons is that it can learn more patterns than a single neuron, and it can produce multiple output values rather than the single output which a single neuron produces. This allows the distribution of a problem over many neurons.

The learning of a layer lies in the adjustment of its synaptic weights. This happens much as in section 2.2.5, with the learning formula applied to each neuron in the layer. Obviously some of the parameters of Equation 2.2 can be calculated once for the entire layer. Input to the layer can be represented as a single vector, as each neuron in the current layer gets input from all inputs in the preceding layer. Since this general neural network model has every neuron in the current layer connected to every neuron in the subsequent layer, the error for a layer can be calculated once. However, this is only one possible approach to training layers of neurons.

Thus, a neuron is a basic functional unit. Many neurons combined comprise a layer. It has been noted that a layer can be connected to inputs and outputs in the environment directly, or to other layers of neurons. Layers of neurons, rather unsurprisingly, comprise a network.

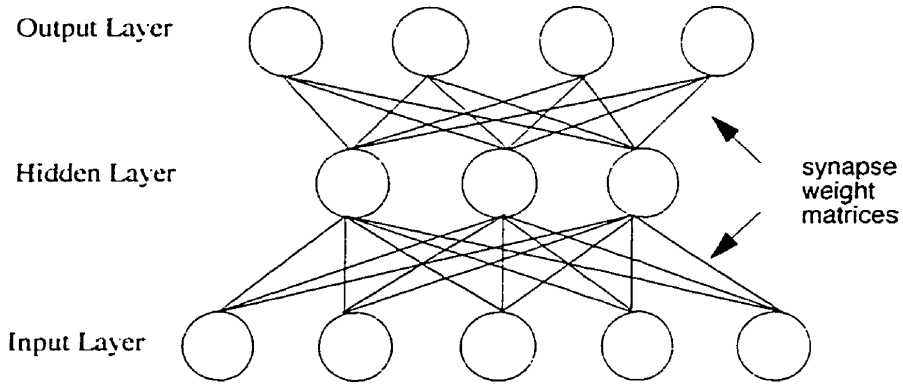
2.2.7 The Network

A network is the end result when layers of neurons are connected in sequence. The first layer receives inputs from the environment, and the final layer returns outputs to the environment. Hence, these two layers are respectively termed the input and output layers. The neurons making up a layer are often referred to as units, and as such the neurons comprising the input and output layers are the input and output units [Weiss & Kulikowski].

There are many different possible layouts for a network involving physical attributes such as connection of the units, and with differences in learning algorithms and their application. Some networks feed their outputs back into their inputs, while others modify their own structures as a result of their learning. Different network models vary in the range of prob-

lems they can be used to solve. In general, however, any network with more than two layers of neurons is referred to as a multi-layer network. Layers between the input and output layers are called hidden layers.

Figure 2.4. A Multi-layer Network



2.2.8 Linear Separability

Though the concept of multi-layer networks has been introduced, no justification for complicating the network layout with multiple layers has been given. Why not just use two-layer networks with many nodes and adjust the weights accordingly? The solution reason involves the comparative topology of two-layer and multi-layer networks. Two-layer networks are, in essence, linear entities. By their nature they can only classify data that is linearly separable.

Consider a set of data that is divisible into two classes. The data can be graphed in two dimensions and the two classes separated by a straight line as in Figure 1.5. For multidimensional data of n dimensions, the data will be separable with an n-dimensional separation. That is, data in 3 dimensions will be separable with a plane, and higher dimensions will be separable with an appropriate hyperplane [Wasserman].

Figure 2.5. Linearly Separable Data

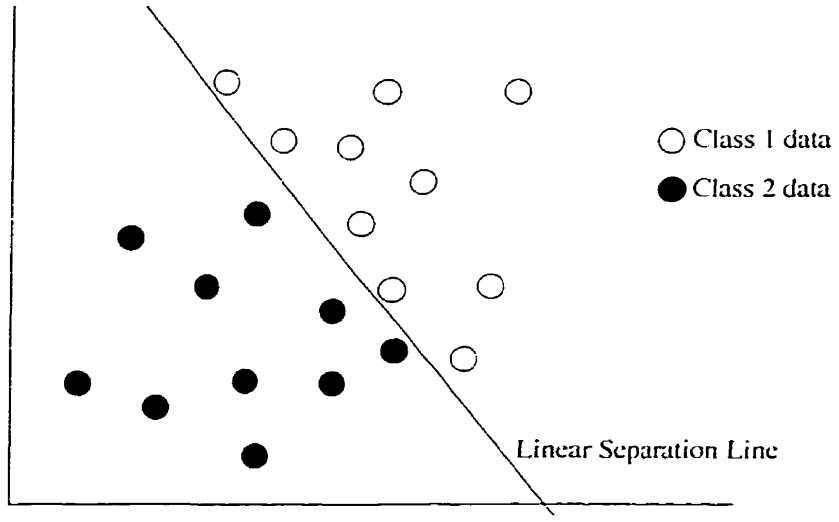
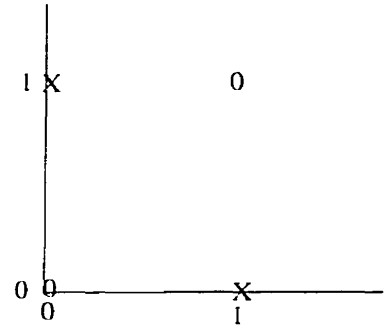


Figure 2.6. Linearly Inseparable Data: The xor function [Blum, 1992]



Some data, however, are not separable in this manner. Consider the graph of the exclusive-or function in Figure 2.5. There is no way to draw a line which separates the X's from the 0's. In some problems, there might be additional data which could be used to add another dimension to the problem. Imagine, for example, an extra piece of data which would indicate a move to another plane for either of the two X values. In that case, a simple 2-layer network could be used with an extra input factor [Wasserman]. In the case of the exclusive-or function, this extra data could be a logical *and* or a logical *or* of the x and y values. This

addition would serve to place one of the special cases in a different plane, thus rendering them separable by a 2-layer network. Ordinarily, however, the data is linearly inseparable. The use of additional data in this manner is not always feasible, as analysis of the dataset to discover such data may be a non-trivial task. Nevertheless, because a third dimension that would create separable data can be conceived of, this problem is solvable by a multi-layer network which will not require the use of additional input factors.

The purpose, therefore, of multi-layer networks is to solve problems in which the data is not linearly separable. If the data can be made separable by the addition of further input factors, this may be desirable as the resulting neural network would be simpler. However, as this is not always possible, multi-layer networks are required.

Having discussed the fundamentals of neural networks, and explored the rationale behind multi-layer networks, it is possible to focus on a particular type of multi-layer network: the backpropagation network.

2.3 Backpropagation

2.3.1 The Network Model

The problem with the neural network learning models described thus far is that they define weight changes for the output layer only; the weight changes are based on an error term only available at the output layer. This was the problem that Rosenblatt encountered: a lack of a 'teacher' (error term) for the hidden units. To solve linearly inseparable problems, multi-layer networks are required. Thus, a method of training the hidden layer is called for.

Backpropagation refers to the backwards distribution of error used to train a multi-layer network. In particular, backpropagation proposes a method of estimating the error of a hidden layer in a neural network and so permits the use of the learning law for hidden units. This allows for adjustment of the hidden layer's synapses even though the desired output of the hidden units is not known. Though a 3-layer network will be discussed here, the process could be recursively applied for more hidden layers. However, this is rarely necessary for more than one or two hidden layers [Wasserman].

Backpropagation is effectively able to solve a wide range of problems, and as such is a good model to study with regards to new research. Backpropagation is one of the most commonly used supervised training algorithms [Blum, 1992]. However, because backpropagation is a supervised learning algorithm, it is required that a set of "fact" data be obtainable which associates input patterns with correct outputs. Also, backpropagation has few if any self-organizing aspects and as such a very good "sense" of the problem with regards to network topology (number of units per layer, for example) is necessary [Blum].

2.3.2 Supervised and Unsupervised Learning

Supervised learning is a process of presenting input patterns to the neural network and comparing the produced output to the desired result. The synaptic weights of the network are then updated according to the learning rule of the network model being used. The key factor is that it is required that the desired outputs be known, and that learning is done separately from the recall process.

Unsupervised learning, by contrast, makes weight adjustments which are not based on comparison with target output values. It is also known as self-organization [Blum, 1992]. Backpropagation is a supervised learning paradigm; unsupervised learning is beyond the scope of this thesis.

2.3.3 The Learning Algorithm

The backpropagation learning algorithm operates in several steps. First, a pattern of data is presented to the network, and the current activations are computed from the inputs to the outputs. [Weiss and Kulikowski, 1991] Then the output layer error is computed and hidden layer error is approximated in turn. Finally, the second and then the first (hidden) layer of synapses is updated. The formulae used are as follows:

1. Calculation of hidden-layer neuron activations:

$$h = F(iW1)$$ **Equation 2.3.**

In this case, h is the vector representing the hidden layer neurons, i is the input vector to the network, W1 is the first set of synaptic weights (a matrix), between the input and hidden layers, and F is a sigmoid activation function.

2. Calculation of output-layer neuron activations:

$$o = F(hW2)$$ **Equation 2.4.**

The new variables here are o, the output layer vector, and W2, the matrix of the second set of synaptic weights, between the hidden and output layers.

3. Calculation of the error in the output layer. This is the observable, non-hidden error. The output layer error is represented by the vector d, and is calculated by comparison with the target output values in the vector t as follows:

$$d = o(1-o)(t-o)$$ **Equation 2.5.**

$o(1-o)$ is the first derivative of the sigmoid function, used to calculate the error gradient.

4. Estimation of the hidden layer error which will be called e. This is the crux of the back-propagation model. The difference between the target and produced values for the hidden layer is estimated by taking the product of the output layer error and the second set of synaptic weights:

$$e = h(1-h)W2d \qquad \textbf{Equation 2.6.}$$

Hence the name backpropagation: the output layer error is weighed *backwards*, with the synaptic weights, such that it approximates the error of the hidden layer. In other words, the error of the output is being weighed to estimate how big the error must have been before it reached the hidden layer.

5. The synaptic weights of the second layer, W2, are then adjusted by adding to W2 the term:

$$\alpha hd + \Theta \Delta W2_{t-1} \qquad \textbf{Equation 2.7.}$$

The second term is the product of the momentum term, Θ, and the change in synaptic weights on the previous pass (time t-1).

6. Finally, the hidden layer's synaptic weights are adjusted by adding to W1 the term:

$$\alpha ie + \Theta \Delta W1_{t-1} \qquad \textbf{Equation 2.8.}$$

Again the momentum term is used, a discussion of which follows.

2.3.4 The Hidden Layer

A significant issue in backpropagation is the choice of the number of hidden units. This depends entirely on the particular problem to be solved, and the desired performance of the trained network. For example, a classification problem might be easily solved by having one hidden unit per training pattern. With this model, it can be shown that each hidden unit would effectively learn to recognize a single pattern. This might be a good approach for a network expected to perform a repetitive task with no unexpected data; that is, one with a "complete" set of facts known and available for training. However, if the network were

expected to generalize its learning to new information not in its training facts, it would likely do poorly. This is due to the fact that while each hidden neuron is good at effectively recognizing a single pattern of the problem, there exists no overlap in the learning to allow the network to effectively generalize its "knowledge". The exact number of hidden units required for a generalization problem is variable, and dependent on the exact problem. This is why experts are often required to implement neural network systems.

2.3.5 Momentum

The momentum term is multiplied with the previous change in a synapse's weight. This is shown in equations 2.7 and 2.8. The resulting product is added to the learning adjustment to produce the total adjustment to the current synaptic weight. The obvious question is: what is the momentum term for?

2.3.5.1 The Problem

In training a neural network, there are two difficulties which occur. The first is that, with a low learning rate, a network may take a long time to learn a set of facts. This will be further demonstrated later in the thesis. The difficulty problem is that a network with a high learning rate may encounter a thrashing problem. As the network converges upon the ideal value for the data, the high learning rate causes it to overshoot that ideal value. Then, on the next training cycle, the network will generate a synaptic update in reverse (the error will have the opposite sign) in an attempt to remedy the "over-learning" on the previous pass. It is therefore possible for the error calculation to produce the same numbers, with only the signs changing (depending on which side of the target value the actual output is on), on both

sides of the target value. Since the learning rate is constant, the network will "thrash"; that is, it will alternate between values on both sides of the desired value without ever actually converging to the desired value. It is also possible for this to happen over a period of several weight adjustments. This problem is endemic to high learning rates in backpropagation, and to say that it is undesirable is an understatement.

2.3.5.2 The Contradiction

An ideal neural network is one which quickly and accurately converges to the target values. Unfortunately, the available approaches thus far each preclude this ideal. If a small learning rate is used, the network takes an inordinate amount of time to train. Hence, speed is sacrificed for accuracy. If a large learning rate is used the network trains much faster. However, several detrimental situations arise. A large learning rate may cause the network to start with weights of the wrong sign, which will then take a long time to reverse. Even if this problem is circumvented, the network may have trouble converging because the learning rate may be too large to bring the network to its optimal point; thrashing will result. These situations defeat the purpose of a large learning rate for faster learning, and sacrifice accuracy. Thus, the only usable option of the two is the former, as accuracy is paramount. The result is slow training. The only other option is to reduce the learning rate during the training. As this slows down the learning, it will not be dwelt upon. Rather, another solution, momentum, is the means by which this problem is most easily solved.

2.3.5.3 The Solution

The momentum term puts a weight on how much a synapse's previous weight adjustment (it's learning) should effect its current weight adjustment. The momentum term is multiplied by the previous result of the learning formula, that is, the previous weight adjustment. In essence, the second term of Equations 7 an 8 is equivalent to adding a percentage of the last weight adjustment to the current adjustment.

The benefit of a momentum term is twofold, effectively dealing with both the major problems discussed above. First, the time it takes the network to train drops. This is due to the momentum term influencing the change in synaptic weights. Once the network is training in one direction toward the ideal point, the momentum term allows it to pick up speed. Since momentum is applied to each iteration, the effect "snowballs". The training actually picks up speed, making increasingly larger jumps toward the target value until it arrives at or passes over the target value. This leads to the second case, that of passing over the target value.

Momentum also solves the thrashing problem discussed earlier. Consider that when a network oversteps its target value, the next pass may recalculate the same amount of "correction" as the original error (or some portion thereof to enable a cycle over several updates). With momentum, the adjustment in the new, opposite direction is added to a percentage of the direction in which the network was previously moving. In the case of an overstepped target, these two values will have opposite signs. While this may cause an overstep in the opposite direction, it must be less than the previous overstep due to the momentum term.

This process continues, with each overstep of the target value becoming smaller as the momentum term influences the current weight change with the previous one. Eventually, the synaptic weights will converge upon the target values.

Thus, momentum allows a network to train faster, both by permitting a higher learning rate and "snowballing" synaptic weight adjustment. When using high learning rates, momentum also tempers a backpropagation network's tendency to "thrash" around the target values without ever actually achieving them. Momentum allows a twofold gain in performance at a usually low implementation cost. All the implementor must do is maintain a list of the last adjustment to each synapse. Momentum makes backpropagation much easier to deal with and far less temperamental in its training.

2.3.6 Bias Values

In section 2.2.3, it was mentioned that sometimes a threshold, or bias, value is added to the term in the summation of Equation 1. This would give the equation:

$$x = \sum_{i=0}^{n} I_i w_i + \theta_i \qquad \textbf{Equation 2.9.}$$

The bias values are adjusted on each pass with the product of the learning rate and the error in the neuron's output. [Blum, 1992] The effect of bias values is to give the network a short cut to achieving its pattern separation. Referring to the linear separability problem of figure 2.5, consider that the threshold value corresponds to the y-intercept of the line separating the classes. Thus, the network can focus more exclusively of learning the essential slope of

the line. Meanwhile, the y-intercept position is learned by the threshold values. Naturally, this extrapolates into higher dimensions.

2.4 Training Problems in Backpropagation

2.4.1 Initial Weights

The most obvious problem encountered in training backpropagation networks is that of their initial weights. That is, what values should the initial weight matrices be set to in order to obtain optimum performance? In order to answer this question, it is necessary to consider why the initial weights are important.

Gradient descent refers to the practice of minimizing the error of a function over several iterations. In a backpropagation network, a generalized least mean squared algorithm is used to modify network weights. The goal is to minimize the mean squared error between the desired and actual outputs of the network. [Mehrotra] Where the error for a pattern p is given by $E_p = \sum_k (l_{p,k})^2$, with k the node from the output layer and l the squared error between the output and desired value, backpropagation must discover a vector that minimizes E_p. Since the output of the network is a function of its weights, so must E be a function of the network weights. [Mehrotra] Thus, the starting weights of a neural network affect not only the initial outputs but also the error and, thus, the gradient descent. In other words, every set of starting weights for a backpropagation network has a different gradient descending to the state of minimum error. Some may take many iterations, some only a few, and some may become stuck in local minima, unable to progress. Since the number of pos-

sible combinations of weights is infinite for real weights, and the danger of poor weights so great, how does one choose a starting set of weights with a reasonable gradient descent?

The most commonly used method to combat the problem of initial starting weights is to run multiple trials of multiple networks. The idea is to eliminate the problem by running a particular network architecture with a number of different starting weights. The performance of a particular set of network parameters is determined by considering all the sample runs of that network and comparing it to those of networks with other parameters. Since there are so many network parameters, such as hidden units, layers, activations, and so forth, adding even more trials makes the number of potential runs far too large for an exhaustive search. Rather, trials tend to be guided either by previous knowledge about the data, or the intuition of the expert crafting the network. Clearly a system which removes the burden of this trial and error process from the neural network professional is desirable. The running time of such a system need not be an improvement over the previous method; it is the selection strategy that must first be optimized.

2.4.2 Number of Hidden Units

One of the most difficult choices a neural network designer must make in designing a back-propagation network is how many hidden units to employ. To begin, a small number of hidden units is usually better at generalizing to unseen data. A large number of hidden units tends to be a superior memorizer; however, the data to be learned can make a significant difference. For simple data for which the dimensionality, or number of classes, is known, it is often optimal to choose one hidden unit per data class. Unfortunately, data sets from

the real world are not always well structured. Classes may overlap, be discontiguous, or have other properties which mean that a greater (or fewer) number of hidden nodes may actually be optimal. Combine this with the fact that each network has to be run many times with different starting values, and a guaranteed optimal solution becomes intractable.

2.4.3 Length of Training

Once a few networks are chosen with numbers of hidden units that are likely to work well, the designer must decide how long to train the network for. If, as is usually the case, the network will have to generalize previously unseen data, then training the network for a long period of time may be counterproductive; the network will memorize the data rather than extract the patterns contained therein. This is less of a problem if one of the networks has an appropriate number of hidden nodes to generalize sufficiently. However, the previous section discussed the problems in determining such a number to any degree of certainty. Similarly, training the network for too short a time results in a network that performs sub-optimally on the known data. Ideally, if a method could be devised for selecting the number of hidden units with near-optimality, this problem would largely disappear. Alternately, a method of stopping training at the optimal point in training would also solve this problem. Again, trial runs of different lengths only compound the number of required trials, as they must be combined with the previous problems.

2.4.4 Evaluation Strategies

The final stumbling point in this maze of pitfalls is the evaluation strategy. That is, in what manner does one determine the performance, both significant and relative, of a neural net-

work? If one uses data that the network has been trained on, this biases the networks in favour of memorized patterns. If one uses new data the networks are biased in favour of generalization, but usually at the cost of accuracy on the training set. A combination of these numbers might be desirable, but what combination? Finally, using test data – data withheld during the training phase – to evaluate the network and determine how to use that network's parameters in future iterations can be considered to contradict the idea of test data. That is, the held out data is, in fact, influencing the network architecture. In some cases it is therefore deemed necessary to hold out a third set of data as the final test set. Thus, in addition to selecting the discussed training parameters of the networks, selecting the evaluation strategy is itself no trivial decision.

2.5 Summary

This chapter has provided an introduction to the topic of neural networks. A brief history was followed by a more in-depth look at the subject. Finally, the backpropagation algorithm was reviewed, and the major problems in its use examined. To be able to discuss the application of evolutionary algorithms to neural networks, it is now necessary to examine evolutionary algorithms in more detail.

3 Evolutionary Algorithms

3.1 Background

Genetic algorithms, also called evolutionary algorithms (GAs or EAs), were developed by John Holland at the University of Michigan [Goldberg, 1989]. With his students and colleagues, Holland set out to achieve two goals. First, to "abstract and rigorously explain the adaptive processes of natural systems", [Goldberg, 1989] and second, to "design artificial systems software that retains the important mechanisms of natural systems." [ibid]

In other words, Holland was trying to find out how natural, biological systems manage to be so adaptive, and how this knowledge might be applied to artificial systems. The power of genetic algorithms lies in their robustness: they effectively balance the need for efficiency with the need and ability to survive in many environments. It is this higher level of adaptation which is sought for artificial systems. No artificial system is as flexible, efficient, or robust as a biological system. Biological systems demonstrate self-repair, self-guidance, and reproduction on a level which the most sophisticated artificial systems cannot even begin to attempt. [Goldberg, 1989]

Thus, Holland's reasoning was that if robustness is desired, why not model artificial systems after the most robust behavior known: that of biological systems. However, this is not just a "shot in the dark", or an appeal to the elegance or aesthetics of natural systems:

"Genetic algorithms are theoretically and empirically proven to provide robust search in complex spaces." [ibid]

3.2 Mechanics

3.2.1 Overview

Evolutionary optimization schemes are patterned after Charles Darwin's theory of natural selection. That is, the fittest members of a population survive to pass their traits on to their offspring. This results, hopefully, in the propagation of beneficial traits.

When applied to artificial intelligence (AI), this means that a population of potential solutions to a problem is constructed. Simultaneously, a representation capable of describing a member of the population is defined. Each member of the population is evaluated to determine how well it solves the problem in question. Based on this fitness value, the representations are manipulated by operators such as crossover, mutation, reproduction, and so forth, according to some algorithm. Conversion of the new representations to solutions results, hopefully, in a superior solution combining traits of previous, partially successful solutions [Goldberg].

An understanding of the three basic genetic operators, crossover, reproduction, and mutation, will be helpful in the succeeding sections. Crossover is the exchange of genetic information between two (though possibly more) individuals in a population [Goldberg]. "Genetic" information refers to the fact that the process involves an exchange of the substance of the uniqueness of each individual; an exchange of inherited traits.

Reproduction, also referred to as selection, is the propagation (selection) of individuals into a new population. This selection is based upon the fitness of the individual judged relative to the population. Thus, a highly fit individual may be copied several times into a new population [Goldberg].

Mutation, the last of the basic operators, is the occasional random alteration of the value of a gene; a spontaneous change in the genetic code of an individual. In evolutionary algorithms mutation plays the role of preventing the loss of potentially important genetic material, especially due to overzealous reproduction and crossover within the population [Goldberg]. The mechanics of each of these operators will become clear through the rest of this chapter.

3.2.2 A Simple Example of an Evolutionary Algorithm

Consider the problem of navigating a simple maze. The possible operations at any point are:

- turn left 90 degrees
- turn right 90 degrees
- go forward to next wall or intersection

Consider also that it is known that the maze can (or must) be navigated with no more than ten consecutive operations. It does not matter if the maze is navigated in less than ten operations.

Figure 3.1. A Simple Maze. Action begins at start, facing in the direction of the arrow.

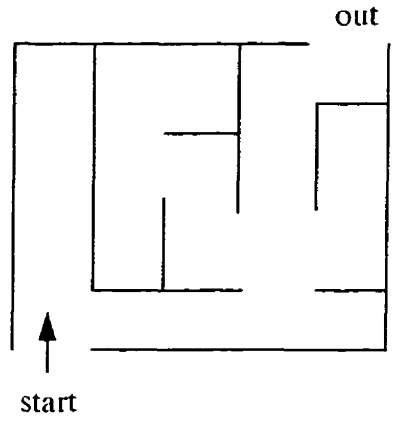


Table 3.1. Potential Solutions

A	B	C	D
forward	forward	forward	forward
forward	right	right	right
forward	forward	forward	forward
left	forward	left	left
forward	forward	forward	forward
forward	forward	left	forward
right	right	forward	right
forward	forward	right	forward
left	left	forward	right
forward	forward	forward	forward

The success of a potential solution can be measured by first determining the point at which it was furthest along the correct path. Then, the distance is calculated by counting the number of forward steps required to complete the maze. Thus, a population of potential solutions might appear as in table 3.1. These potential solutions are named A, B, C, and D.

A sequence of steps follows. For the sake of brevity, the above operations are abbreviated to left, right, and forward, respectively. These operations are the encoding of the movements of each potential solution. They comprise the genome, or genetic code, of the individuals. If it is not possible to go forward at a particular point in an individual's path when that individual's genes indicate a forward move, no movement occurs. It is reasonable to assume that the final step for each solution is forward, as the potential solutions must step out of the maze to be successful.

Figure 3.2. Position of Solutions

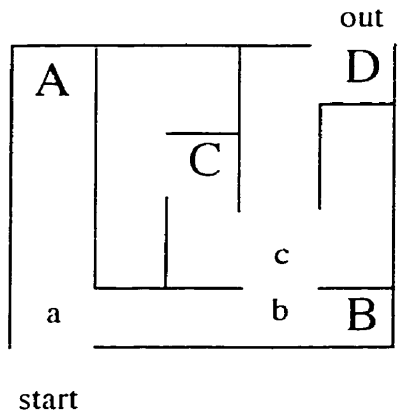


Figure 3.2 illustrates the position of each solution after they terminate. The current direction is not indicated. The lowercase letter indicates the point at which the corresponding solution was furthest along the correct path. Note that no solution has exited the maze (exiting the maze is one more step; that is, the solution must end up on out), and that there is no lowercase D because it would be in the same place as the uppercase D. The distance from the exit of the closest points are 5, 4, 3, and 1 for A, B, C, and D, respectively. Distance has been measured as the number of forward moves necessary, not counting turns.

Now, four new solutions are created based on the performance of the previous 4. These will be created, to maintain the simplicity of this example, by combining each of the two best solutions with one of the inferior solutions. That is, A will combine with C, and B with D. This is accomplished by choosing a random point and mixing the solutions – a technique known as crossover. For this example, all solution will be mixed between the fifth and sixth operations. The result is shown in table 2.

Table 3.2. New solutions after mixing

A'	B'	C'	D'
forward	forward	forward	forward
forward	right	right	right
forward	forward	forward	forward
left	forward	left	left
forward	forward	forward	forward
left	forward	forward	forward
forward	right	right	right
right	forward	forward	forward
forward	right	left	left
forward	forward	forward	forward

Examination will prove that potential solutions C' and D' now solve the maze as required. Naturally, this is a much simplified example, but it serves to demonstrate the salient points of genetic algorithms: that through a process of combining partial solutions, improved and possibly optimal solutions can be obtained.

Thus, genetic algorithms function by manipulating a pool of 'chromosomes' in a population. A chromosome is a string which represents a coding for a parameter set; a representation of a potential solution to a problem. A genetic algorithm evaluates each chromosome for its fitness: how well it is performing relative to other chromosomes in the population. Based on the fitness of the chromosomes, the genetic algorithm selects members of the population (chromosomes from a group of chromosomes) for reproduction, crossover, and mutation. This selection is done randomly, but is weighted by the relative fitness of the chromosomes. That is, a particularly fit member of a population has a much better chance at reproduction than an unfit member, and so forth. An detailed example serves to illustrate this process further.

3.2.3 A Detailed Example of a Genetic Algorithm

Consider a black box consisting of 5 switches each with two positions. For every combination of on and off values for the switches, the box produces a numeric output. The problem is to determine the setting of switches which produces the largest output value from the box. A genetic search for this value works as follows:

First, a representation for a chromosome must be constructed which represents the parameters of the problem. Since the problem has five binary switches as parameters, the chromosome can be a string of 5 binary numbers. Next, an initial population is generated randomly. Say an initial population of 5 chromosomes is desired. With five bits per chromosome, a coin would be flipped 25 times to generate the initial population. Thus, the initial population is:

```
1 1 0 1 1
0 1 1 1 1
1 0 1 0 1
0 0 1 1 0
1 0 0 1 1
```

The black box will serve as the evaluator of each string's fitness. The switches on the box are set to reflect the 1's and 0's in the strings, and the value output from the box is recorded. Consider that the black box produced the following, where fitness is the black box's output:

Chromosomes	Fitness
1 1 0 1 1	729
0 1 1 1 1	225
1 0 1 0 1	441
0 0 1 1 0	36
1 0 0 1 1	361

However, the genetic algorithm requires that the fitness of each chromosome be known relative to the others. To achieve this, simply sum the fitness values and take each chromosome's relative fitness as a percentage of the total fitness of the population:

Chromosomes	Fitness	Relative Fitness (%)
1 1 0 1 1	729	40.7
0 1 1 1 1	225	12.6
1 0 1 0 1	441	24.6
0 0 1 1 0	36	2
1 0 0 1 1	361	20.1

Total fitness: 1792

With this information, the reproduction operator is applied. The purpose of the reproduction operator is to select, based on the fitness value, which members of the population will have the opportunity to mate. Thus, this is not reproduction in the biological sense; it could more correctly be called selection for reproduction. This selection is achieved by creating

a cumulative probability based on the relative fitnesses. Goldberg calls this a "roulette wheel" because each chromosome is allotted slots on the wheel according to how fit that chromosome is relative to the others. Chromosome 1, for example, would be allotted slots 0 through 40.7, chromosome 2 would be in slots 40.7 through 53.3 (40.7+12.6) and so forth. The effect of this is that for each member of the population a random number between 0 and 100 is generated. That number will determine which member of the population to reproduce. A random number of 50, for example, indicates that chromosome number 2 should be reproduced as 50 is greater or equal to 40.7 and less than 53.3. Note that the implication is that the same chromosome may be selected more than once. This means that selection for reproduction can completely remove some members from the population. However, the gain from this behaviour is that successfully selected chromosomes will mate more often. Since they are, probabilistically, more fit, they will pass on their better genes to the next generation in greater numbers.

Now consider the following 5 random numbers chosen by the genetic algorithm. The chromosomes they select for reproduction are indicated in the next column:

Random	Chromosome selected
46.6	2
61.8	3
52	1
98	5
19	1

Thus, the new population looks like this:

Chromosomes
0 1 1 1 1
1 0 1 0 1
1 1 0 1 1
1 0 0 1 1
1 1 0 1 1

The next step is to allow the chromosomes to mate. Mating is carried out by selecting two chromosomes at random for this example. However, some underlying scheme could be used if desired. Since there are five members in the population, only 4 will be paired. Hence random numbers between 1 and 5 are generated in pairs, each indicating a pair of chromosomes to "mate". This process is stopped when one chromosome is left unpaired: (if a random number is generated more than once, it will be ignored and a new one generated)

```
Pairs:
4 5
2 1
```

Thus, chromosomes 4 and 5 are paired, as are 2 and 1. Chromosome number 3 is left alone and merely passes through this section of the algorithm untouched.

Crossover is now applied to the pairs. To effect this, a random number is chosen for each pair which represents the number of points at which a chromosome could be broken. Since the example contains chromosomes of five bits, there are 4 potential breaking points, or loci. They are generated as:

```
Pair    Locus
4-5     3
2-1     2
```

Crossover is applied by swapping the chromosome pairs around the locus point:

```
Chromosome 1:0 1  1 1 1
             (the space indicates the locus)
Chromosome 2:1 0  1 0 1

Crossover completed:

Chromosome 1':0 1 1 0 1
Chromosome 2':1 0 1 1 1
```

The other pair is crossed over in a similar manner. Note that this coincidentally produces chromosomes identical to the parents. The new population now appears as follows, with new fitness values. This new population completely replaces the original population.

Chromosomes	Fitness	Relative Fitness (%)
0 1 1 0 1	169	6.7
1 0 1 1 1	529	21
1 1 0 1 1	729	29
1 0 0 1 1	361	14.3
1 1 0 1 1	729	29

Total fitness: 2517

Notice that the overall fitness of the population has increased to 2517 from the original 1792. Note also the chromosome with the lowest initial fitness was dropped completely from the population during reproduction. The current population now contains two chromosomes of the form "1 1 0 1 1". This means that the chance of that chromosome being selected for reproduction is even higher in subsequent iterations. If the genetic process of reproduction and crossover is applied repeatedly to this population, within a relatively short period of time the optimal value of "1 1 1 1 1" will be achieved. It has probably been observed by the reader that the black box performs a square on the decimal equivalent of the five bit binary number represented by the chromosome. However, the beauty of the genetic approach is that the actual function is irrelevant; an optimal coding will be found in the case of most "black boxes".

The only genetic operator not used in the preceding example is the mutation operator. In nature, genetic robustness of a species is aided by random mutation of genes. When not

influenced by external sources, mutation is actually very rare, and chances are mutants will die off fairly quickly. Occasionally, however, a mutant has some new trait which makes it more fit to survive in its environment. Thus, the new trait is added to the gene pool.

In genetic algorithms, it is possible to lose genetic material, as it were, or not to have had it in the first place. This can be caused by overzealous reproduction and crossover operators selecting what appear to be the most fit members but losing potentially useful genetic material in the process, or can be merely random. (Recall reproduction's "roulette wheel" approach. A fit member may die off; there's only a probability that it won't.)

Consider the previous example. Suppose that each population member had a zero at position two. That is, each member of the population is of the form x 0 x x x, where x is a 1 or 0. Notice that no matter how much reproduction or crossover takes place, that position in the chromosome will always contain a zero. With a mutation operator, genes on the chromosome are randomly changed according to some probability. In this case, changing a gene merely involves flipping the bit. De Jong's study of genetic algorithms for function optimization suggested a low mutation probability inversely proportional to the population size. [Goldberg, 1989] Goldberg suggests approximately one mutation per thousand bit position transfers [ibid]. Thus, mutation plays a secondary yet important role in genetic algorithms.

3.2.4 Summary

It has been shown that genetic algorithms allow a randomized search of a complex search space. They effectively use "random choice as a tool to guide a search toward regions of the search space with likely improvement". [Goldberg, 1989] Yet because they operate on

a coding of the problem, the problem itself is often irrelevant; the genetic algorithm itself can operate on any problem once its parameters are coded. It is this "directed randomness" combined with the ability to search many options in parallel that makes genetic algorithms a powerful search tool.

3.3 Comparison to Other Search Methods

At this point it would be useful to discuss genetic algorithms in comparison to other types of search. This will give a greater insight into how genetic algorithms are able to do what they do, and how they overcome some of the major pitfalls of other search algorithms. To begin, Goldberg identifies three basic types of search: calculus-based, enumerative, and random.

3.3.1 Calculus Based

Calculus-based search methods involve two classes. One class operates through solving nonlinear sets of equations. These equations result from "setting the gradient of the objective function equal to zero." [Goldberg]. That is, restricting the search to points with a slope of zero in all directions. The second class of calculus search methods seek to move up a local gradient. These are the classical "hill-climbing" algorithms. The trouble with these methods is that they are susceptible to localized peaks in the domain space. If a start point is selected close to a low peak, a higher peak will likely be missed all together. Also, calculus based methods require continuity of the domain and existence of derivatives [ibid] In the real world, this is not necessarily the case. Thus, except in limited, known domains, calculus-based methods are insufficiently robust [ibid].

3.3.2 Enumerative

An enumerative scheme is one that evaluates each point in the search space in turn, seeking the optimal point. The problem with this is obvious: except for very small search spaces, this type of search will be terribly inefficient [Goldberg].

3.3.3 Random

Random search algorithms, in effect, attempt to rectify the pitfalls of calculus-based and enumerative schemes. A random search involves sampling points in the search space at random with the goal of locating an optimal point. While this may overcome the tendency to get stuck on local peaks, in the long run this can be considered no more efficient than an enumerative search [Goldberg].

3.3.4 Random Versus Randomized

While random techniques suffer from the same efficiency problems as enumerative techniques, they should not be confused with randomized techniques such as genetic algorithms and simulated annealing. Randomized search algorithms use random elements to guide a search for an optimum value [Goldberg].

3.3.5 Genetic Algorithms

Genetic algorithms are a randomized search method. They use randomness combined with laws of probability to direct search in a direction where improvement is likely. More correctly, genetic algorithms direct a search in many directions that are likely. Goldberg identifies 4 ways that genetic algorithms differ from the traditional methods discussed above:

1. Genetic algorithms "work with a coding of the parameter set, not with the parameters

themselves". [Goldberg, 1989] That is, the algorithm generates many possibilities simultaneously, and then evaluates them.

2. Genetic algorithms use a population of potential solutions. They are inherently parallel, and not restricted to considering a single point at a time as other methods are. This also the reason for their robustness and ability to overcome local peaks in a search space.
3. Genetic algorithms use a fitness, or objective function to determine viability of potential solutions. They do not use derivatives, or "other auxiliary knowledge." [ibid] Thus they can be tailored to any domain where some judgement of the "goodness" of a result can be made, regardless of whether or not the domain has a search space that conforms nicely to the laws of calculus.
4. Genetic algorithms guide the transitions in the search space using probabilistic, not deterministic rules. Unlike a hill climbing search, for example, which might have a rule to the effect of "if a higher point exists adjacent to the current one, choose it", genetic algorithms assign likelihood of a good search direction based on the results of the pay-off function relative to other directions being explored.

Thus, genetic algorithms produce a robust search algorithm which works across a wide variety of domains, many of which are not suited to traditional search algorithms. Their inherent parallelism allow them to search a space more efficiently and quickly than many traditional algorithms. Though they are randomized, they are not random techniques; they use randomness as a tool to direct search in promising directions.

3.4 Neural Networks as Candidates for Genetic Search

Having discussed the basics of backpropagation networks and genetic algorithms, the question of combining the two is raised. While neural networks can be powerful tools for pattern recognition, optimization, classification, and mapping problems in general, they are by no

means easily constructed. Traditionally, neural networks are designed and implemented by specialists – professionals with in-depth knowledge of the strengths and weaknesses of various network architectures. While this results in well designed networks, it can also give rise to certain problems. While the underlying algorithms may be relatively simple, network parameters such as learning rate, momentum, initial weights, number of layers, and number of units per hidden layer play a large part in the ability of a particular network to solve a given problem.

Even when selected and implemented by an expert with knowledge both of neural networks and the problem domain, the process is often little better than trial and error. A better way to determine optimal parameter settings for a neural network is required. The goal of applying an evolutionary algorithm is to automate the now largely ad hoc process of neural network design.

Thus, it is established that neural networks, by virtue of the complexity of their design, are potentially good candidates for genetic search. In order to employ such an approach, however, it is necessary to ask whether evolutionary techniques are in fact suited to neural networks.

3.4.1 Motivation for Evolutionary Networks

What is the advantage of evolving neural networks? EAs offer a much more flexible approach. Neural networks are, in essence, a hill climbing search. As such, they are subject to the pitfalls, discussed in section 3.3.1, of getting stuck on local features of the solution space. Neural networks use an error calculation to compute a gradient to direct the search;

for example, the backpropagation network [Haykin]. These methods require smooth, continuous activation functions in order to derive gradient information [Moriarty]. In contrast, evolutionary algorithms do not perform direct calculation of gradients. Instead, they focus on blanketing the search space with potential solutions. This results in a far more global search which is much less likely to succumb to local features of the solution space. These advantages give evolutionary algorithms a much wider range of options; an EA might use linear thresholds, splines, or product units where traditional neural networks might require a smooth sigmoid function [Moriarty]. Further, computation of gradients in the more complex neural networks, such as recurrent networks, can be quite costly. EAs do not require these expensive calculations.

Thus, EAs complement the traditional neural network gradient-descent techniques quite well. Their simultaneous global search allows large, irregular search spaces to be covered in an automated manner, removing human drudgery, and human error, from the equation.

3.5 Neural Networks and Evolution in the Literature

Applying genetic algorithms to neural networks is not a new idea. Maureen Caudill, David Stork, Ronald Keesing, Peter Korning, and others have discussed genetic algorithms as a means of optimizing network structure.

3.5.1 Initial Weights

Caudill proposes an algorithm which focused on initial network weights to produce a network more suited to learning a particular problem [Caudill]. She refers to this as a networks's "nature", that is, its inherent ability to solve a particular problem. This idea was

explored in more detail by Stork and Keesing, who went on to patent a technique [USP5245696].

The general idea is to use evolutionary algorithms to discover networks which are predisposed to learning a particular problem. The alternate view, that of training the network incrementally, is discussed in the next section. All the reviewed research has focused on similar networks. That is, the network structure is not an issue – all the networks have the same number of layers, hidden units, and so forth. The question is whether or not networks of varying topologies may be the focus of the same evolutionary algorithm. This would result in an EA which is able to explore a much larger solution space and hence find a better solution.

3.5.2 Training and Evolution

A different approach is to train the networks for a set amount of time, run the EA on them, and then train them further. These methods can run into problems, however, if the weights of the network are manipulated by the EA. To overcome this problem, sometimes the EA is used as the sole method of training or, alternately, the EA is used until it can do no better, and the job is finished with gradient descent.

Korning discusses the various attempts to apply genetic algorithms to neural networks. Whitley, Miller, Romaniuk, Zhang, and Myhlenbein have all attempted genetic optimization of network architecture. [Korning] Others have attempted training a neural network's weights and thresholds. Korning criticizes these attempts for using too high a mutation rate, and too short a chromosome.

Korning argues that very long chromosomes, approaching 10000 bits, are required to effectively allow a genetic algorithm effect "hyperplane sampling". [ibid.] Also, Korning argues that the qualitative nature of a genetic search requires a fitness function other than the commonly used "least mean square" function from backpropagation training. [ibid.] Korning goes on to show how his method produces very good results, and points out that the generalization abilities of his method are quite high. Thus, he believes that the future of genetic algorithms and neural networks lies in more work being done in the representation and evaluation areas.

3.5.3 Adaptive Neuroevolution

A new method of applying EAs to neural networks has recently been developed. Symbiotic, adaptive neuro-evolution (SANE) is a new neuro-evolutionary technique for solving complex sequential decision tasks [Moriarty; Weeks & Burgess]. In a domain where traditional neural network and evolutionary techniques have been at best moderately successful, SANE demonstrates an ability to excel at tasks which offer small, infrequent feedback. Developed by Moriarty and Miikkulainen, the key to SANE is a two-pronged attack at the search space. The algorithm evolves neurons separately from networks. Thus, networks are formed of evolved neurons and neurons are evaluated based on their participation in successful networks. Since no single neuron alone can solve a complex problem, it behooves the neurons to learn to cooperate. Thus the SANE algorithm is exemplary at maintaining population diversity without resorting to high mutation rates.

3.5.4 Other Mentions

Other writers such as Timothy Masters and Philip Wasserman have also discussed genetic algorithms within the context of neural networks, but have proposed no particular implementation [Masters; Wasserman].

3.6 Summary

Research into genetic algorithms and their application to neural networks is ongoing. There remain areas with very little research, such as "sexual reproduction", diploidy, and dominance. [Wasserman] It can be hoped that further research produces newer and better neural network training systems.

4 Structure of the System

This chapter describes the system designed and implemented for this thesis.

4.1 Goal

It is the goal of this thesis to study evolutionary algorithms applied to neural networks as a method of automatic neural system generation. The ultimate goal is to create a system which can be 'turned on', and will autonomously design a neural system equal in performance to what a human expert would design. To this end, a system is presented which consists of algorithms designed to construct backpropagation networks. All the algorithms presented in this chapter were implemented and tested. The advantage of this system, of course, is that it generates neural networks without human intervention.

4.1.1 Implementation

The implementation of the system consisted of two phases: first, the implementation of the neural network architecture to be manipulated; second, the design and implementation of a suitable evolutionary algorithm.

The system was implemented in as close to standard C++ as possible. The backpropagation network was designed as a C++ class, while the evolutionary algorithm was implemented, for convenience mostly, in standard C. It is important to note the while the neural networks are implemented in C++, care was taken not to sacrifice the speed of the implementation.

Thus, the object-oriented design was carefully constructed to avoid common C++ overhead problems.

4.2 Neural Network Implementation

Backpropagation was chosen as the neural network model for the system. The advantage of backpropagation is that it is a well known, proven and studied algorithm. The techniques for using it are as rigorous and standardized as possible in AI.

The backpropagation network, as previously described, was implemented in C++. The main classes were a layer class, an abstract network class, and finally a class defining a specific type of network. In this case the third class was obviously backpropagation. The classes were designed with two goals in mind: to simplify the coding of types of networks other than backpropagation, and to retain as much of the speed of C while garnering some of the benefits of C++. For this reason, complex C++ constructs requiring run-time overhead were not employed. Rather, simpler but leaner solutions were sought.

4.2.1 The Layer

In the class hierarchy, the base class is a layer. Arguably, a neuron might be the ideal base case in such a design. However, this would add an extra level of object dereferencing to access every weight. Additionally, the weights would not be easily accessible in other orders. For example, assuming a neuron consisted of incoming connections, it would require one object dereference to obtain all the weights into that neuron. Conversely, to obtain one input weight for each neuron in the layer would require as many dereferences as there are neurons. It is possible this could be circumvented by retaining a link to the previ-

ous neuron, but at best this incurs the same cost as the first case: an extra dereference. Thus, the layer was chosen as the basic unit of the network.

4.2.2 The Network

The network class is an abstract class; that is, it cannot itself be instantiated. Rather, another class must inherit it and define certain key functions such as the learning algorithm. The network class consists of generic, network-specific parameters, such as learning rate, bias terms, and number of layers, as well as a set of layer objects. To overcome C++ overhead, the network was initially allowed access to the private members of a layer. However, C++ inheritance rules made this infeasible later in the implementation phase. Therefore, most, if not all, member functions used to access layers, and some used to access networks, are inline functions. This results in larger code, but this is not especially significant to this dissertation.

4.2.3 The Backpropagation Class

The final class in the neural network hierarchy is the backpropagation class. This class inherits the basic functionality of the network class and adds backpropagation-specific items. Specifically, the backpropagation class adds class-specific constructors, a learning function, activation function, and miscellaneous access and I/O functions. Note that while backpropagation inherits from the network class, the network class *contains*, rather than inherits, layers.

Thus, the backpropagation network encapsulates the following functionality. A backpropagation network is instantiated with a constructor which specifies the number of layers, the

length of the input and output vectors, the number of patterns (for training), pointers to the desired outputs, and an array in which to place results. The network additionally contains bias terms and a learning rate. The learning rate is fixed for the purposes of this experiment, but it could easily be modified to be adjustable. The backpropagation class also contains a learning function, an activation function (the sigmoid function described in chapter 2), and an output function. The learning function works by being passed a number of cycles for which to learn; the weight updating is done on-line; that is, after each pattern.

4.2.4 Network Evaluation

As discussed in chapter 2, it is necessary to partition the data presented to a neural network into training and testing sets. The purpose of this practice is to better evaluate a network's future performance on new data. If a backpropagation network is trained on a single large data set, it may begin to memorize some patterns instead of learning the general patterns of the dataset. By presenting the network with testing data not used in training, it is possible to better evaluate the network's ability to generalize. Several terms derive from this practice. The *classification rate* is the ratio of patterns correctly classified compared to the total number of patterns classified. Conversely, the *rejection rate* is equal to 1, less the classification rate. Finally, the *accuracy* is the ratio of the number of correctly classified patterns to the total number patterns. Naturally, these number exist for both the test and training sets.

An evolutionary algorithm requires some form of fitness evaluation to determine the relative goodness of the members in the population. The question when dealing with backpropagation networks is whether to base the fitness on the training set, the test set, or a

combination of the two. Since the intent is to produce networks more capable of learning the generalities of the data, the first option is not useful for reasons described above. Due to the difficulty in combining the performance numbers for the test and training sets into a meaningful performance evaluation, it was decided to evaluate the networks solely on their ability to classify the test set. To that end, the fitness of a network was calculated as the number of correctly classified patterns as compared to the total number of patterns. It was mentioned in chapter two that a common practice in neural networks is to withhold a third partition for the very end upon which to test the final networks. This method was not used, as this experiment was concerned with the ability of the evolutionary algorithm to create networks and not the generation of networks by the standard design methods. Thus, it was felt that to effectively measure the performance of the evolutionary algorithm, it must have at its disposal all of the available data which a human designer would have.

4.2.5 System Verification

Following the creation of the backpropagation network, it was necessary to make certain that it worked correctly. To perform this validation, several tests were performed:

- Training on the xor problem.
- Comparing the number of training cycles required to attain a certain level of performance with a network trained by hand via another system.
- Comparing detailed output of the above for a few cycles to validate the algorithm. That is, verifying that both networks produced precisely the same calculations for an entire pass through the data set.

The first test was to create and train a backpropagation network on the xor problem. Any backpropagation network should be able to learn this data, provided it has a minimum of 2 hidden units. Failure to do so would indicate an implementation error.

Once a backpropagation network was successfully learning the xor patterns, the network was compared to a network which was known to operate correctly. Two identical networks should achieve the same performance given the same training, regardless of the system. If, from the same starting state, the two networks did not converge to the same answer in the same number of cycles, a more subtle implementation error must be at work.

Finally, after the first two tests were successfully passed, every step of the network's algorithm was compared to the control network for two epochs of learning. This ensured that precisely the same results were being produced. Additionally, final values some five to ten epochs into training were also compared. This final validation ensures that no further minor errors exist to corrupt the algorithm.

4.3 Evolutionary Algorithm

After implementing and verifying the backpropagation network, work began on the development of a suitable evolutionary algorithm. This development was done in incremental steps, not dissimilar from the evolutionary prototyping paradigm of software engineering. The approximate steps taken in the development of the EA are summarized below.

1. Run a single network for n epochs at a time until performance starts to degrade.
2. Run multiple networks as in 1.

3. Run multiple networks, n epochs at a time, removing the worst performers after every set of n. Stop after a fixed number of sets or when only a few networks remain.
4. Apply reproduction to trained networks.
5. Apply reproduction crossover and mutation to trained networks.
6. Apply reproduction, crossover, and mutation to initial network weights using both distributed and uniform learning epochs.
7. Method 6, but allow the best performers to pass on their learned genome with some probability.

For methods one through four, a fitness function was developed. By the time method five was reached, the fitness function had been finalized. The following sections discuss the representation upon which the EA operates, and then discuss the steps above in further detail.

4.3.1 Representation

In order to operate on neural networks, the evolutionary algorithm requires a representation of those networks which is easily manipulable. The salient features are the network weights, the number of layers, and the number of hidden units.

The number of layers is deemed irrelevant for the purposes here, as extra layers rarely add much to the neural network. This is due to Kolmogorov's theorem [Bishop]. Thus, it is assumed that all networks in the system are two layer networks. (An input layer, one hidden layer, and an output layer.)

The network weights are the key features for the evolutionary algorithm, as the EA is looking for an optimal set of weights. As there are two distinct layers, it makes sense to encode the weights on two distinct chromosomes; mixing of weights across layer boundaries is not

investigated for the purposes of this thesis. Thus, two strings are created which catenate each weight matrix into a single vector. These two vectors are stored together, and consist of the EA's representation of a neural network.

Fortunately, the number of inputs and outputs for each networks are fixed, as they depend upon the data in use. Therefore, the precise number of genes, representing weights, in a chromosome created in the above manner can be calculated at any time. Additionally, a value indicating the length of a chromosome is stored in the first position of that chromosome. This obviates the need for extra calculations by the algorithm. This system of encoding is easily extensible to additional network types and topologies.

4.3.2 Pre-reproduction approaches

The first method used to develop the EA was the most trivial possible method. The goal was to automate the network training to continue running until the performance of the network on either the test data, training data, or some combination thereof, began to degrade. Initially, the system was run with all the data in an attempt to automate the process of learning all the data. This is similar to what experts do with a new set of data. By training the networks on the entire set of data, they are able to determine the overall "learnability" of the data set. Preferably this is done with a small number of hidden units. As the number of hidden units increases, the network begins to memorize more of the specific records. This behaviour is undesirable: since the network is intended to work with data it has not been trained on, learning the general patterns in the data is more important. Thus, a smaller

number of hidden units indicates a more likely ability to learn the general patterns in the data.

The algorithm worked by running the network for several sets with n epochs in a set, where N was a suitable number that was predefined. Every set, the system would check to see if performance had improved, degraded, or stayed the same. If it had degraded, then the system would stop. The problem with this approach is that gradient descent algorithms often have 'bumps': sections in the solution space where the error becomes temporarily worse before becoming substantially better. Later versions of this basic algorithm took into account 'bumps' in the gradient descent algorithm by requiring two consecutive degradations to stop the run.

The obvious next step to this simple algorithm was to run a number of networks in this manner simultaneously. The first iterations of this improvement merely ran all of the networks for a fixed number of epochs and presented the best ones. Later, networks whose performance had degraded prior to the end of the run were not trained further. A final adjustment was to remove poorly performing networks from the run entirely. This was accomplished through a variety of methods. The most successful involved eliminating networks whose performance was below the nth percentile for that iteration, or generation. In other words, the current average fitness of the population, minus some percentage of the standard deviation. For example, killing off networks in every generation which were performing below the average less the standard deviation meant that the top 84% of networks were retained. Conversely, the worst 16% were removed from the population. Note that it

is not possible to remove networks performing too close to the average, or else when the population begins to converge (which will happen due to continued training) all the networks will suddenly be removed! This iteration of the algorithm did not attempt to replace the removed members, but rather ran either for a fixed number of sets or until only a few networks remained.

4.3.3 The Fitness Function

Throughout the previous development, the fitness function was also in constant flux. Initially, when the entire data set was being learned, the total sum squared error (TSS) was used. This quickly gave way to the number correct, or the percent correct, as much less unwieldy numbers than the TSS. Upon the introduction of separate test and training sets, a new fitness evaluation had to be formulated. For example, the percent correct in the test set was somewhat effective in encouraging the networks to learn to generalize. Unfortunately, as this does not take into account the performance on the training set, the ability to identify some special-case patterns in the training set might be lost. Also, as the networks increased their performance on the training set, test set performance tended to fall off; the networks were overfitting the training set [Haykin; Mehrotra]. A happy medium was required which sought out a balance between the two. One method was the product of the classification rates of the test and training sets. The *classification rate* is the percentage of correctly classified patterns in the total number of patterns classified. Thus the network is not penalized for patterns if it is able to indicate that it is uncertain as to their classification. Thus, a higher evaluation with this function indicates a higher ability to make accurate predictions.

Once again, however, increased performance in the test set causes an undesirable decline in training set performance. For this reason, and because the data set was known to be trainable (able to be learned to a high degree of accuracy) [Wolberg; Mangasarian; Scuse], the final evaluation used in the later stages of the experiments was the percent performance on the test set only. This ensured the evolution of networks with the highest ability to generalize.

4.3.4 Standard Genetic Operations

Before discussing the standard genetic operators, it is necessary to briefly review the representation chosen for the system. Recall that network layers consist of weight matrices. Thus, for this system, a multi-chromosome representation was chosen where each layer, or rather, the weight matrix preceding the layer, was transformed into a separate chromosome. Thus, each chromosome becomes a linear representation of the corresponding weight matrix. For the sake of simplicity, this study will be limited to two-layer networks; that is, networks with one hidden layer. It follows that only chromosomes that originate from the same relative layers will be allowed to cross. By thus restricting the motion of genetic material, it is possible to focus more upon the effects of the base system. Also, this would appear to be the approach taken by biological systems: only homologous chromosomes cross over, not any chromosomes at random. Future work may indeed include networks of differing numbers of layers and cross-layer mating.

The next step towards the evolutionary algorithm is implementing the standard genetic operators of reproduction, crossover, and mutation.

The EA developed for this thesis uses standard roulette wheel reproduction as discussed in Chapter 3. Mutation is also standard, and the mutation rate was set to one mutation in one thousand weight copy operations, or 0.1%. The mutation rate is purposely kept at this low level, as this is not a study of the effects of mutation, but rather an attempt to learn about the evolutionary potential of these networks. Therefore, a high mutation rate would be counterproductive, as it would confuse the results.

Crossover, on the other hand, involved some new ideas. Most of the literature involves populations of networks of the same architectures, that is, the weight matrices have the same dimensions. The authors have been concerned with performance rather than architecture optimization [Stork & Keesing, Caudill, etc.]. Since the system constructed herein differs in that aspect, crossover is implemented by choosing a crossover point on the shorter of the two chromosomes (if one is shorter). This effectively deals with the problem of differing network structures.

4.4 Consolidation: The System

The combination of the neural network and the evolutionary algorithm constitute the entire system. While the basics of genetic representation and operation have been presented, there remain a variety of ways in which the two systems can interact. [Caudill, Stork, Moriarty]. The evolutionary system devised herein is a synthesis of several ideas. Caudill proposed the mating of trained network weights in 1991. Her work followed that of Keesing and Stork in 1990, which focused on the mating of initial weights based on the performance of partially trained systems. With the addition of the innovation of distributed learning epochs

(to be discussed shortly), Stork and Keesing went on to patent this technique a few years later [USP5245696]. A conversation with Dr. Stork indicates that further work was not realized and the system was left where it was.

Indeed, the literature is rife with examples of evolutionary algorithms used to optimize neural network performance [Caudill; Stork; Dodd; Chang]. However, on the subject of automating the creation of neural networks, especially when considering increasing the EA's degrees of freedom to operate on the networks, there appears to be a marked absence. This dissertation briefly explores each of these ideas as applied to backpropagation. Finally, while mating of both trained and initial weights has been explored, a combination of the two does not seem to have been attempted. Thus, the final incarnation of the system involves an attempt to probabilistically allow trained weights to be passed into the subsequent population to mix with the gene pool of initial weights. The results, presented in the next chapter, are encouraging.

4.4.1 Mating Trained Weights

The first, and perhaps the most obvious, way to combine neural networks with evolutionary algorithms is to alternate cycles of neural network learning and performance evaluation with evolutionary optimization. This approach is straightforward, and at first would seem to be the obvious method. However, there are several inherent drawbacks to this technique.

4.4.1.1 Lamarkian Evolution

Lamark was a scientist of the late nineteenth century who proposed that evolution was directly influenced by the experiences of individual organisms in their lifetime [Mitchell].

The classic example of Lamarkian evolution is that of a giraffe with a short neck which stretches to reach leaves in tall trees all its life. Subsequently, its offspring have longer necks, which they stretch further, and so forth. While current scientific evidence contradicts this model rather vehemently, there is naturally a tendency to attempt Lamarkian techniques for use in artificial systems such as AI and neural networks. Indeed, studies have shown that these processes can improve artificial evolutionary algorithms [Mitchell, Caudill].

However, the fact remains that biological systems do not operate in this manner. [Mitchell, Stork & Keesing] The exceptions are the lowest forms of life, such as planaria. Since artificial intelligence is concerned with modeling higher brain functions, should the evolutionary optimizations used upon learning systems be a reflection of those used upon the lowest forms of stimulus-response processing?

4.4.1.2 Mixing Weights in Neural Networks

A neural network, particularly a backpropagation network, depends upon the gradient descent of the error in order to learn. Networks are typically robust in that, because the learning is distributed among all the network weights, damage to the network or noise in the data does not necessarily cause complete failure of the network. However, recall that backpropagation begins with a random starting point and then applies the learning algorithm to minimize the error over the training time; different starting weights lead to completely different learning curves. By crossing over networks with two different sets of starting weights, it is unlikely that the resulting amalgam will be any better than a third net-

work with a random starting point. The fact that the system presented herein also mixes networks of varying topology (specifically the number of hidden units) merely compounds the unlikelihood of a genetic operation resulting in a network that is anything more than a new random starting point for gradient descent learning. Finally, the further the networks are along in their total training, the more specialized their weights, and the less likely that genetic operations will produce anything useful.

4.4.1.3 Nurture overcomes Nature

Another problem arising from the mating of trained weights is the fact that the learning algorithm, backpropagation in this case, is highly effective. The design of the algorithm allows it to learn from nearly any starting point. This means that the longer the network is trained, the greater its ability will become; this is the whole point of gradient descent. However, when combined with an evolutionary system, the two become inextricably intertwined: the learning process is performing local search, which moves all population members closer to potential solutions. The EA in turn performs a global search. However, the diversity of its population has been reduced by the previous local search. The network weights which were once widely different may now be tending down the error gradient to the same local minimum in the solution space. The more training sessions that are used, the greater the effect it has on the evolutionary algorithm. Thus, the inherent goodness of any particular network structure or starting point is lost, overwhelmed by the efficient learning strategy. The nurturing of the system has overcome any inherent nature that may have been discernible.

4.4.2 Mating Initial Weights

The alternative to the method of section 4.4.1 is the mating of initial weights. In short, this method trains networks for a short period of time, evaluates their fitness, and then mates their *initial* weights based on this information. In essence, this is akin to true biological evolution: the success of the parent influences its ability to pass on its *initial* genetic makeup. How then do improvements occur? In biology, species are improved through the Baldwin effect [Mitchell].

4.4.2.1 The Baldwin Effect

Proposed by J. M. Baldwin near the end of the nineteenth century, the Baldwin effect recognizes that environmental pressure may favour individuals with the capability to learn [Mitchell]. Individuals more capable of learning are more likely to survive, and therefore reproduce. It follows that an organism able to learn many traits will be less dependent upon genetically encoded traits. Thus, a more diverse gene pool is supported which allows for learning to compensate for traits not fully developed by the genetic code. The ability to learn indirectly accelerates the rate of evolution.

4.4.2.2 Baldwin Evolution in Neural Networks

The realization of the Baldwin effect in neural networks is straightforward. Instead of mating weights which have been trained by the learning algorithm, the initial weights are mated. The fitness of the network, however, is evaluated through the performance of the network after a set amount of learning. In essence, this modification to the evolutionary algorithm gives the neural networks a 'life' which is distinct from their genetic makeup. A network's performance in its life determines the relative fitness of its genome compared to

other individuals in the population. This approach is successfully demonstrated by Stork and Keesing. Further, they discovered that distributed learning trials produce better performance.

4.4.2.3 Distributed Learning Trials and Baldwin Evolution

Stork and Keesing discovered that randomly choosing the number of learning epochs for each individual in a population of neural networks gave much improved performance than using the same number of leaning trials for all networks. [Keesing & Stork] This makes intuitive sense when one considers the biological paradigm. No two organisms are exposed to exactly the same amount or type of stimulus. Modifying the number of learning epochs on an individual basis mirrors this biological truth. Unfortunately, there is no straightforward way to modify the type, as opposed to amount, of stimulus other that to break up the training data into random partitions. Due to lack of data, this is not always feasible.

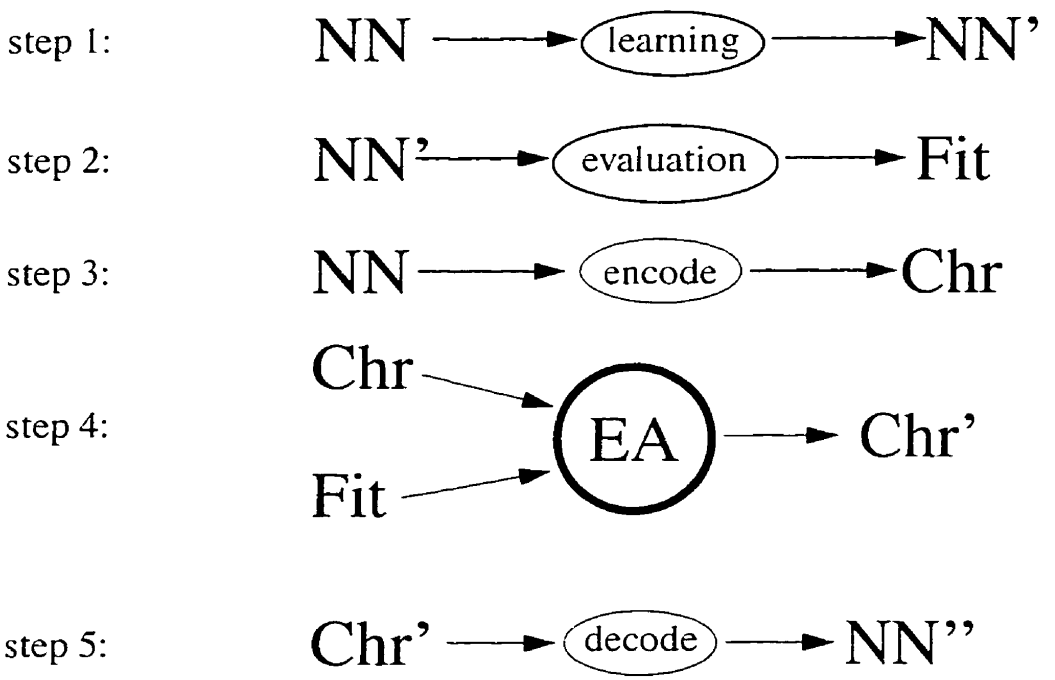
4.4.3 Probabilistic Lamarkian Learning Transfer

The final enhancement presented by this dissertation is that of probabilistic Lamarkian learning transfer (PLLT). Earlier, section 4.4.1.1 mentioned that studies had shown that Lamarkian techniques can improve artificial evolutionary algorithms. The question this raised was under what circumstances Lamarkian techniques could improve the previously discussed Baldwin-type models. It was postulated that this could be accomplished by allowing the best networks in the system of section 4.4.2 transfer their trained weights, rather than their initial weights, to their offspring with a certain probability. By restricting

the frequency of this occurrence, it is possible to artificially boost the performance of the evolutionary system via a pseudo-Lamarkian process.

PLLT allows new genetic material to be introduced into the population. Because there is a possibility of a network mating with itself, thus producing identical progeny, this also, in effect, allows particularly successful networks to, in essence, 'live' longer. Thus, a well performing network has the chance of keeping it's learning through a generation (or perhaps more), instead of always losing it.

Figure 4.1. The Basic Evolutionary Algorithm



4.4.4 The Algorithm Revisited

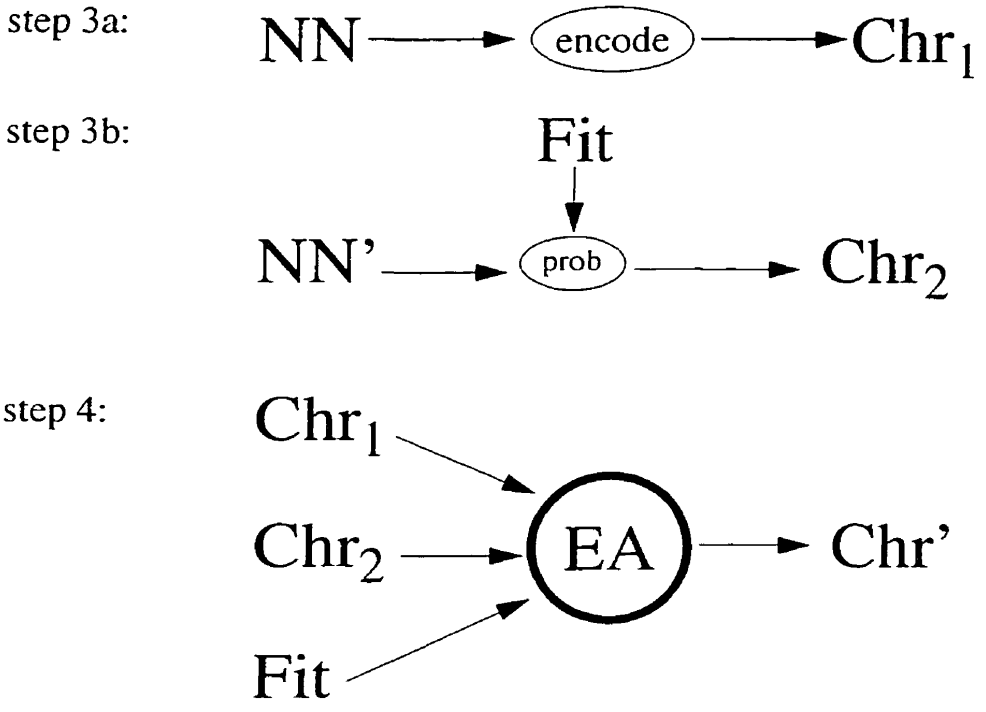
The evolutionary system developed uses the basic algorithm described in figure 4.1. The algorithm begins with a population of neural networks, NN. These are trained via the learning process to obtain trained networks, NN'. Next, the trained networks are evaluated to produce a fitness measure, Fit, for each member of the population. Step 3 takes the original networks and encodes them into chromosomes, Chr, with which the evolutionary algorithm can work. The chromosomes and their corresponding fitnesses are sent to the evolutionary algorithm in step 4. The result is a new set of encodings, Chr', which represents the offspring of the previous generation. Finally, in step 5, the new encodings are converted to a new set of networks, NN''. The process repeats for as many generations as desired.

This presents the basic algorithm. In the case of PLLT, a few extra steps occur. Step three has the addition indicated in figure 4.2: with a probability influenced by the fitness, some members of the set NN', the trained networks, are encoded into chromosomes. This selection process uses a lower and upper bound. When a network's fitness is greater than the lower bound, a uniform variate is generated between these two bounds. If the network's fitness exceeds this variate, its trained, rather than initial, weights are encoded into the gene pool. Next, these members are added to the evolutionary algorithm in step four.

In order to maintain a population of constant size, after the PLLT chromosomes are created (Step 3b), enough additional chromosomes are added in the traditional manner (Step 3a) to make a total population equal in size to the previous generation. It is important to note that the genes created in step 3b are passed into the EA for mating. They do not immediately

find their way into the next generation. Such behaviour might indeed be desirable, but is in the scope of future work.

Figure 4.2. PLLT Modified Evolutionary Algorithm



4.5 Summary

The focus of this thesis is one of unifying a learning system and a search system. The combination should result in an evolutionary learning system, where the powers of each system both support the strengths and compensate for the lacks of the other. The goal of this work is to remove the drudgery from neural network design: Once a task has been identified as being a candidate for a neural network approach, the ideal situation would be the mere acti-

vation of a system which would automatically generate the most appropriate, accurate network.

This system is a step in that direction. While the network type and architecture are limited, no feasible reason has been encountered to suggest that multiple network types and configurations could not be added to the same genetic pool. Thus, the lessons learned in this application of genetic algorithms to backpropagation networks should hold true for at least several other architectures. Minor modifications will probably suffice to add more network types to the system.

Freeing humans from the unnecessary drudgery of network creation will allow research in more elaborate areas of learning and evolution to take place. The ability to set up neural systems without human intervention is a vital step on the path to fully adaptive, autonomous intelligent systems.

5 Experimentation

5.1 Description

This chapter will describe the performance of the system described in chapter 4 in an experimental environment. Specifically, a dataset will be selected on which the system will be trained. The system's results will be compared to that of known results of expert-designed neural networks.

5.2 Purpose

The purpose of this experiment is to demonstrate the practical validity of the theoretical designs presented in previous chapters. It will be demonstrated that the system described herein is a viable and effective alternative to having experts design neural networks by hand.

5.3 Materials

The materials used in the experiment consisted of a computer and associated hardware, the system whose design was described in Chapter 4, and a dataset from an AI data repository on the internet.

5.3.1 Computer

For this experiment, the system was run on a Power Macintosh 8600/300. This computer system runs on a 300 Mhz Motorola PowerPC 604ev microprocessor on an Apple Com-

puter designed motherboard with 128 megabytes of interleaved RAM, and 1 megabyte inline level two cache running at 100 MHz. This machine was running MacOS version 8.5. As previously mentioned, the evolutionary system has been designed to be highly portable, and contains very little platform-specific code.

The evolutionary system itself was written using the Metrowerks Codewarrior compiler, release 4. The accompanying SIOUX-WASTE (Simple Input and Output User eXchange) console library was used for console output. This library actually aids portability because it allows standard C++ console input and output to be used on a platform without a standard command-line console. The code was written entirely in C++; however C++ specific features, such as classes and overloading, were used sparingly (only when their use was deemed to overcome the traditional speed loss in using C++ over plain C).

5.3.2 Dataset

The dataset chosen for this experiment was the Wisconsin Diagnostic Breast Cancer (WDBC) dataset. The set consists of five hundred sixty-nine records, each with thirty variables. Each record is classified as either malignant or benign. Three hundred fifty-seven cases are benign, while two hundred twelve are malignant. The data are known to be nearly linearly separable when all thirty input features are used. There are a few patterns which are on the class boundaries; thus, achieving 100% accuracy is a problematic endeavor. Previously, the best predictive accuracy obtained at the University of Wisconsin, Madison, is 97% through the use of repeated ten-fold crossvalidations [WDBC.doc].

5.3.2.1 Features

The thirty features of this dataset are computed from a digital image of a fine needle aspirate (FNA) of a breast mass [WDBC]. "The... [image analyzer] ...computes values for each of ten characteristics of each nuclei, measuring size, shape and texture. The mean, standard error and extreme values of these features are computed, resulting in a total of 30 nuclear features for each sample." [WDBC; Mangasarian]

Figures 5.1 and 5.2 show two examples of the images processed by the image analyzer and turned into records in the dataset. The features computed for each cell nucleus by the image analyzer are as follows: [WDBC.doc]

a) radius (mean of distances from center to points on the perimeter)
b) texture (standard deviation of gray-scale values)
c) perimeter
d) area
e) smoothness (local variation in radius lengths)
f) compactness $\left(\frac{perimeter^2}{area - 1.0}\right)$
g) concavity (severity of concave portions of the contour
h) concave points (number of concave portions of the contour
i) symmetry
j) fractal dimension ("coastline approximation" -1)

When the mean, standard error, and extreme values are added to these ten features for each cell, the result is the thirty features per sample.

Figure 5.1. Benign Breast Mass [92-5311]

Figure 5.2. Malignant Breast Mass (Cropped) [91-5691]

This thirty-feature dataset was normalized – converted to values in the range [0,1] – for use in this experiment. Furthermore, the benign-malignant result was converted into two features per record. A 1 in one category indicated a malignant mass, while a 1 in the other category indicated a benign mass. A pair of ones or a pair of zeros is undefined, and indicates no classification. For the purposes of this experiment, the ID tag was dropped from the dataset: it has no bearing on the diagnosis.

5.4 Methodology

In order to assess the performance of the evolutionary system, a comparison must be made to a known quantity. Since the goal of the system is to eliminate the need for humans in the neural network design process, it seems reasonable to compare the system's performance to that of hand-designed neural networks. To this end, several results were available. First, the results from the University of Wisconsin where the data was first used [Mangasarian]. Second, two neural network designers (the author of this dissertation, and the main advisor) independently created networks in an attempt to determine the best possible rate of classification.

5.4.1 The Train/Test Strategy

One of the most frequently used strategies in creating neural networks is the train/test strategy. Because of the ever-present danger that a neural network will memorize a set of patterns rather than generalize them, the dataset is split into two portions. The network is trained on the first, but tested on the second. Unfortunately, the choice of how many and which patterns to allocate to each set is something of a devil's alternative. Allocating more

patterns to the training set ensures that the network learns these patterns well. However, the testing data may therefore not be representative of the entire dataset, and the network may perform poorly on the test set.

Consider, for example, the possibility of an entire class being inadvertently placed into only the testing set. The trained networks will perform inordinately poorly on the training set. Conversely, one might opt for adding more data to the test set. Unfortunately, this means that now the training set may not be representative. Thus, with either choice, there are risks.

5.4.1.1 k-fold cross validation

It is possible to estimate the maximum possible neural network performance by using k-fold cross validation. In this type of validation, the data is divided into k partitions. One partition is chosen to test, and the remainder to train. This data is used to train several networks, whose performance is either averaged, or the best performer is chosen. Then, the process is repeated choosing a different partition as the test set. This is repeated for each of the k partitions.

The average of the results is a good predictor of how well the system will perform. If this is done with a partition size of one pattern, that is k = n where n is the total number of patterns, the average performance represents an approximation of the maximum possible performance obtainable with the neural network model. Clearly, this method is computationally very expensive. However, there is another problem with this technique: after the trials are complete and a number representing the projected system performance is obtained, there is no single 'best' network. That is to say, while this technique is very

good at predicting theoretical network capabilities, it does not, upon completion, generate a usable network. Instead, such a network must be created from the knowledge garnered during the crossvalidation runs. By contrast, an evolutionary algorithm functions by maintaining a population of real solutions. As such, the EA can at any time, implementation permitting, produce a neural network of known, as opposed to theoretical, ability.

5.4.1.2 Separating the Validation of the Algorithms

For this experiment, the focus is on the validation of the evolutionary algorithm as a viable and effective method of designing neural networks. Therefore, using k-fold cross validation within the system would incur needless computational expense. Thus, the experiment involving the evolutionary system uses a straight holdout approach to testing. The holdout chosen for the aforementioned dataset is 500 training patterns, 69 test patterns. The results of the system on this dataset will be compared to networks designed by hand, as well as to the best results obtained using leave-one-out cross validation.

5.4.2 Training by hand

When training networks by hand, varying numbers of hidden units were employed with a wide variety in training epochs. The results were compared, and the best were selected. The performance measure used was the accuracy on the test set, the classification rate, and the value cutoffs, or thresholds. These cutoffs are the values for which a network output is taken to mean 1, 0, or neither. For all runs in this experiment, numbers greater than 0.9 were taken to be 1, numbers less than 0.1 were taken to be 0, and numbers in between 0.1 and 0.9 were taken to be neither.

Using between 4 and 8 hidden units appeared to give the best results, with some networks performing well with fewer or greater numbers of hidden units. The best networks classified 98.55% of unseen data. These were rare, and usually had around six hidden units. By far the most common 'best' result was 97.0588%. Again, it was apparent that provided the networks did not have so many hidden units that they memorized the training data rather than learning the inherent patterns, a variety of networks with different numbers of hidden units could potentially perform well. Thus, for this particular dataset, there is a range of possible numbers of hidden units that produce equally good results. The observed variability comes from the initial weights of the network.

5.4.3 N-fold Crossvalidation

An independent test using n-fold crossvalidation, with n=569, showed that it definitely possible to classify 98.55% of unseen data. This confirms the results obtained above by the human expert. That is, a human designing networks by hand was able to achieve the theoretical maximum accuracy as determined by the n-fold crossvalidation method [Scuse].

Finally, it should be noted that 97.5% is the best figure obtained at the University of Wisconsin, where this data originated.

5.5 Experiment

After creating hand-trained networks to provide a basis for comparison, the evolutionary algorithm was then employed. First, some trial runs were carried out. The purpose of the trial runs was to determine the best combinations of features discussed in chapter 4. Then long runs were performed to determine the best results attainable with the combinations of

features determined in the previous step. This was necessary because testing every possible combinations of features would have been computationally prohibitive with short runs, let alone trying to do it with long runs.

5.5.1 Setup

To begin, small numbers of networks were created and run for a small number of generations. Once the system was determined to be performing correctly, twenty networks were created by a network-generating function. These networks ranged in their number of hidden units from one to ten, with two networks of each. These networks were run for varying numbers of generations, starting with ten, but eventually settling with fifty as a reasonable number of generations. As explained in chapter 4, and with the exception of PLLT, all genetic operations were performed with initial weights as opposed to trained weights. Where PLLT was used, the parameters were 0.9 to 0.97. (That is, a uniform variate between 0.9 and 0.97 was generated. If the current network's fitness was greater, the network was allowed to pass on its trained weights.) These networks gave rise to the following observations.

The training from section 5.4.2 indicated that five thousand to ten thousand epochs were more than sufficient to learn the data in most cases. Literature suggests that for evolutionary algorithms, using five to ten percent of the total required epochs seems to give the best performance [Caudill; Stork]. This information, combined with experimentation, lead to the choice of 250 to 500 as the number of epochs per generation.

- Each of distributed learning and PLLT, gave some increase in performance.
- Using 250 (5% of 5000) epochs for every network (uniform learning), the networks failed to achieve even the 'easy' result of 97% accuracy on unseen data.
- With the exception of uniform learning, all of the aforementioned were capable of finding the 'easy' answer. That is, 97% accuracy on unseen data. However, their abilities varied.
- Distributed learning, using a uniform variate with a mean of 250 (from 0 to 500) for the number of epochs, produced the 97% figure, but with 20 networks, this result appeared only once in 50 generations.
- Distributed learning with PLLT produced the 97% figure over 20 times in 50 generations.

These results indicated that distributed learning with and without PLLT were the best performers. Thus, these two models were to be used for several extended runs. To accomplish this, the population size was increased to 100. This was accomplished by applying the same algorithm as before, only now there would be ten networks of each number of hidden units. Thus, the new population has many times the diversity of the previous test. With more starting positions, it was hypothesized that the distributed learning would be able to produce the 97% figure, and possibly the 98% figure. It was also hypothesized that the distributed PLLT algorithm would be able to produce the sought-after 98% accuracy on new data. The results follow.

- With 100 networks, the distributed learning produced networks achieving 97% accuracy more than 20 times, especially in the latter part of the run.
- The distributed learning alone did not produce 98% accuracy.
- Using distributed PLLT, 97% accuracy was to be had anywhere from over 100 times to over 1000 times in the course of a run. It existed in each and every generation.

- Using distributed PLLT, 98% accuracy occurred anywhere from 2 through 8 times in a run. It was almost always in the very late part of the run, if anywhere. It is important to note that the way the PLLT parameter was set, networks with this performance *always* pass on their trained weights.

Due to the probabilistic nature of evolutionary algorithm, it is not surprising that once the 98% figure appears, it does not always carry through to the next generation; the continued training, selection, mating, and mutation, are almost certain to evolve that set of weights into something different. Because of the delicate nature of backpropagation networks, this new network is not necessarily the equal of its parents. There are a number of possible ways of overcoming this shortcoming which will be discussed in chapter 6.

Another point that is of note is that the crossing over of trained weights did not cause an overabundance of poor networks. One might suspect that crossing weights of trained networks would produce garbage more often than not. However, consider the following extract. Of a population of 100 networks, 69 passed on their trained weights. This means that their minimum accuracy was 90%. Of the subsequent generation, 79 networks had a resulting fitness of greater than 0.9. Seven networks had a fitness in the range $0.7 \leq x < 0.8$, one network had a fitness in the range $0.6 \leq x < 0.7$, seven networks had a fitness in the range $0.2 \leq x < 0.3$, and six had a fitness of 0. With 69 networks using trained weights in this population, it would be expected that much more of the population would be poor performers. There are several explanations for this. First, the learning may completely correct for this. Second, networks which are approaching the same minimum in the solution space may be crossing over, thus helping each other to get closer to the minimum. Finally, it may

be that there is an abundance of networks all following the gradient descent algorithms to the same minimum, and as such they all reproduce amongst themselves, giving an erroneous impression as to the true diversity of the population. This answer can be determined by testing the fitness of a population before training as well as after. If the networks have a uniform fitness distribution, then that will rule out the second possibility in favour of the first or last. The last possibility is the trickiest to weed out. It would require an algorithm to compare the weights of several networks and, by calculating if they were within some tolerance of each other, determine if they belonged to the same local minimum on the solution space.

5.6 Conclusion

To summarize, it does appear that distributing the number of learning epochs results in a higher occurrence of well-performing networks. This confirms the work of Keesing and Stork. The use of PLLT not only increases the number of good networks, but provides the impetus required to get the networks over the last obstacle to get the 98% theoretical maximum accuracy.

The most success was had with the Probabilistic Lamarkian Learning Transfer method discussed in 4.4.3. Note that this does not mean that the non-PLLT method of section 4.4.2 was unsuccessful: rather, the PLLT method produced the best results more consistently and in greater number.

Using a large population, over many generations, it has been established that the evolutionary system can produce networks which correctly classify 98.55% of unseen data. This is consistent with the best classification obtained to date on this dataset. Further, the evolutionary system is able to obtain this result with relatively low computational expense on today's desktop hardware. (Runs of 100 networks for 50 generations took from 6 to 8 hours). Finally, and most importantly, these results were achieved without the intervention or supervision of an expert. While numerous refinements can undoubtedly be made to this approach, this experiment has demonstrated that automatic generation of neural networks via evolutionary algorithms is capable of attaining the maximum accuracy on a particular dataset. It remains to be determined as to whether the performance would be equally high on other datasets. However, while the generic applicability of a particular evolutionary system requires further testing, evolutionary algorithms do indeed provide a viable and effective alternative to having experts design neural networks by hand.

6 Conclusion

6.1 Summary

6.1.1 Successes

This study has developed a framework which allows for the automation of the neural network creation process. As demonstrated in chapter 5, the system developed within this framework was successful in attaining the maximum accuracy on unseen data. Further, this result was reached in a reasonable amount of time on a common desktop computer system.

The system was able to determine the best backpropagation architecture to use with regards to the number of hidden units. This ability opens new possibilities which will be discussed in section 6.2. Of particular note are two novel approaches taken by this system: the mating of chromosomes of unequal length, and the use of a pseudo-Lamarkian process, dubbed PLLT, to enhance the performance of the algorithm. While these results are encouraging, there are, naturally, several caveats to be considered.

6.1.2 Caveats

Like any study, the successful results must be considered in light of their shortcomings. It is important to consider a few important points: the dataset, the limited flexibility, and other factors.

6.1.2.1 Dataset

The system was tested, as described in chapter 5, with the Wisconsin Diagnostic Breast Cancer dataset. The creation of the system was not directly biased by the dataset, as the experimenter was blind to the nature of the dataset. That is, the dataset was presented as raw data, and was not identified to the experimenter until the experiment was complete. However, the fact remains that to date the system has only been tested on the WDBC dataset. To properly validate the ideas in this dissertation, further testing on a variety of datasets will be necessary.

6.1.2.2 Limited flexibility

The experiment in chapter 5 was designed as a testbed for a number of evolutionary ideas. As such, the actual freedom of the system was somewhat constrained; the evolutionary algorithm was free only to choose the number of hidden units in the networks. It is felt that this is a significant achievement over previous methods which focus more on evolutionary algorithms as an optimization method. However, this is only one step in the development of a general-purpose evolutionary system. It is hypothesized that the system will work equally well when additional degrees of freedom are given it. For example, no impediments are known which would prevent the system from being able to choose the number of layers in addition to the number of networks (though this is unlikely to be beneficial as noted in 4.3.1), or to create hierarchies of networks for use in gating or voting systems, or even consider networks of several different types (for example, backpropagation, radial basis, and Hopfield networks) all in the same population. However, these facets remain to be tested. They will be discussed in somewhat more depth in section 6.2.

6.2 Future Work

This thesis has left open a plethora of avenues for future work. The most obvious of these will be detailed herein.

6.2.1 Improved Implementation

While an attempt was made to keep the system efficient, it is nearly always possible to do a better job. A more rigorous software design, with a focus on efficiency of the neural networks and interaction between the evolutionary algorithm and the networks would only benefit the system. Possibly a language other that C++ is better suited to the problem. Fortran, for example, excels at mathematical calculation, and Pascal or Modula give a high degree of structure. While this is an incidental concern, not directly related to the ability of the system or the validity of the evolutionary framework, it should be a concern in any serious software implementation.

6.2.2 Curved Fitness Functions

It was remarked in chapter 5 that even when an accuracy of 98% was achieved, it did not necessarily carry over to the next generation. Part of the reason behind this irritating behaviour is that when the total and percent fitnesses are calculated, 98% does not achieve a much bigger portion of the fitness space than 97%. Consider a population of 4 networks, three of which have a fitness of 0.97, and one of which has the sought-after 0.98 fitness. The total fitness for this population is 3.89. When it is time for the reproduction phase, the roulette wheel approach gives 24.94% of the wheel to each of the 0.97 networks, and only 25.19% to the one network with 98% accuracy. Clearly, this approach does not favour the 98%

accuracy much over the 97% accuracy. If such accuracy is important to the application of the system, either a different reproduction scheme or a different fitness function may be required.

For example, the experiment in chapter 5 used 69 patterns in the test set. Thus, the maximum number of correct patterns is 69. The elusive 98% accuracy figure corresponds to 68 correct patterns. Now consider the equation

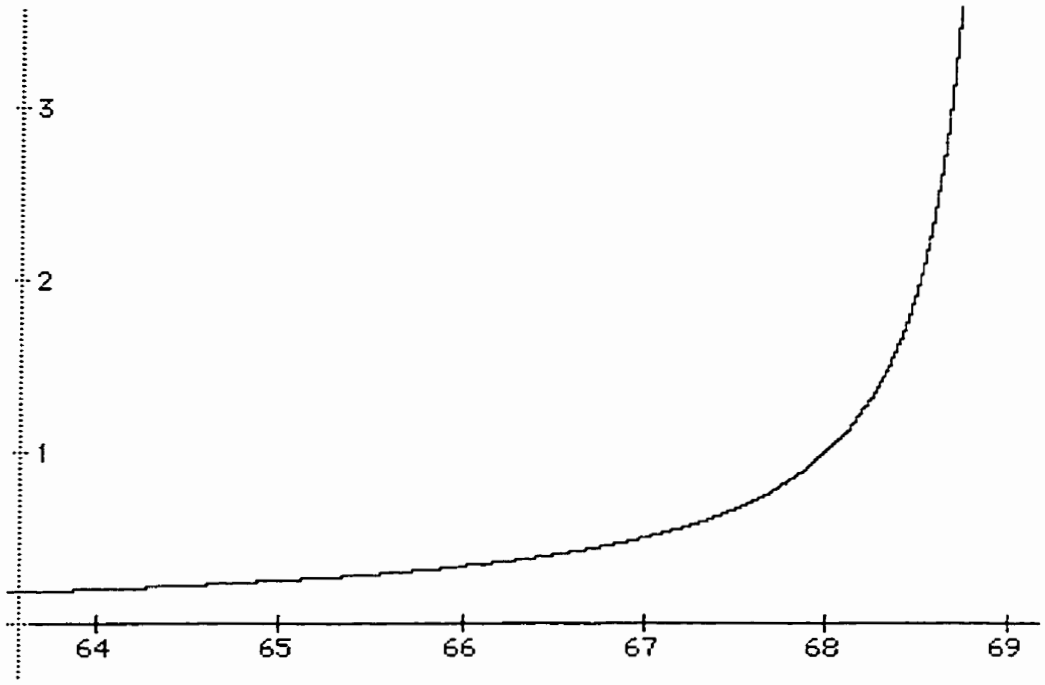
$$y = \frac{1}{69 - x}$$

where x is the number of test patterns correct for a network. This equation produces the graph in figure 6.1. By using the value of y for the fitness evaluation of a network successfully classifying x patterns, it is possible to make the change between 97% and 98% accuracy non-linear. To further adjust the curve, an exponent could be added to x, producing

$$y = \frac{1}{69 - x^n}$$

This is one example of how the fitness function could be tailored to produce a non-linear evaluation. Thus, the evolutionary system would perceive 98% to be significantly better than 97%, depending upon the tuning of the exponent n.

Figure 6.1. Graph of $y = \frac{1}{69 - x}$



Naturally, it is important not to evaluate this function for x=69 and n=1. Using such a fitness function (69.5 or 70 in place of 69 are two examples of the possibilities) would help ensure that better networks always survived.

6.2.3 Other Genetic Operators

The experiments performed to date have made use of the most basic genetic operators: reproduction, crossover, and mutation. There exist a variety of other operators: for example, double point crossover, sexual versus asexual reproduction, elitism, and so forth. Each of these methods has advantages and drawbacks. Any future work in this area should certainly include some exploration of different genetic operators.

6.2.4 Network Types

For the experimental purposes of this thesis, it was explained that backpropagation networks were chosen. It was further suggested that while the current system only selects different numbers of hidden units, there was no foreseeable reason why other attributes, such as the number of layers, might not also be selected. Indeed, taking this idea one step further, there is no reason to stick with a single type of network.

Different neural network models learn in different ways. Radial basis networks employ a learning strategy that excels at learning local features of the solution set. Contrast this to the global approach of backpropagation. Other network models have other strengths. There are several ways that additional network models could be added to the system.

6.2.4.1 Mixed Population

One possibility is to have a population consist of networks of several different types. Mating might be only among like members, or possibly even among unlike members (though this seems improbable). The evolutionary algorithms would search for the best networks without regard to their type, merely their ability.

6.2.4.2 Segregated Population

This method is similar to the above, but in this case the best networks from each type of network would be selected separately, thus enforcing a wider diversity of the population.

6.2.4.3 Cooperative Population

In this extension of the aforementioned methods the evolutionary algorithm would be designed to take into account each network's unique abilities. Thus, situations where a

backpropagation network could do broad classification and a radial basis network subclassification might arise. This approach could give rise to the automatic generation of gating, voting, or hierarchical networks.

6.2.5 Survival Traits

The next two possibilities lie in the area of rewarding or restricting certain actions based upon the fitness of individual networks.

6.2.5.1 Prevent Inbreeding

Biologically speaking, inbreeding, the mating of closely related organisms, is considered harmful to the gene pool. This is because the practice amplifies poor traits without the possibility of a corresponding introduction of new, possibly superior traits. Thus, the gene pool is said to stagnate. No research appears to have been done using the prevention of inbreeding as a means of promoting population diversity in an artificial evolutionary system.

6.2.5.2 Age-correlated Fitness

In the system as described in chapter 4, probabilistic Lamarkian learning transfer is introduced as a way of introducing new genes into the gene pool. This practice, while referred to as Lamarkian, can also be viewed as a method of increasing the granularity of the aging process for the population members. That is, it gives certain members of the population the ability to live longer than a single generation. As discussed in section 4.4.4, this might include allowing the PLLT-generated chromosomes to pass directly into the next generation. This idea could be further exploited by correlating the fitness of an individual to its age. Thus, older individuals which had managed to survive longer would be considered to

have better genes, and thus would rate a higher fitness. This idea in combination with that in 6.2.2 might produce some very interesting results, as it would overcome some of the main problems encountered in the design of this system: difficulty in keeping good networks in the population, and difficulty in selecting marginally better networks for reproduction due to the high fitness of the population.

6.2.5.3 Dataset Partitioning

One of the tenets behind the use of distributed learning cycles was that in the biological world, every organism does not receive the same amount of learning. Similarly, in the biological realm, not every organism receives the same *type of* learning. An avenue for future exploration would be the partitioning of the dataset among the networks of the population such that the networks each receive a different training set. A new partition would be generated for each network in each generation. Testing would be performed on a partition that none of the networks had been trained on. Naturally, there are many possible permutations of this approach. For example, the ability to partition a dataset for distribution to the members of a neural network population could be of use in the creation of gating, voting, and hierarchical networks as discussed in section 6.2.4.3. The use of such partitioning methods would theoretically increase the diversity of the gene pool and result in more robust individuals of higher fitness.

6.3 Conclusion

This dissertation has presented a framework to allow for the automatic creation of neural networks to solve a task. The implication of such a system is two-fold. By removing the

requirement that an expert design a neural system, the technology comes within reach of a greater number of potential applications. Secondly, the divorcing of the expert from the drudgery of day-to-day creation will allow for the study of more interesting problems.

The system designed and presented herein successfully produced networks with the maximum theoretical accuracy on previously unseen data. The benefit is that it runs unattended as opposed to requiring the constant attention of an expert. The system is by no means a finished product; many avenues are open for future exploration. It is felt, however, that the basis of the underlying theories is promising, and that future work will result in increasing flexible automated neural network design systems.

Once it is possible to generate neural networks in an accurate, automated fashion, problems involving more complex applications can be considered. Network collaboration and sparse reinforcement problems are two such areas. Indeed, the ability to generate neural networks for small tasks will lead to their application in ever-larger tasks. The future holds the promise of systems which can create hierarchical systems of networks working in concert, much as the biological brain. Within a few short years, the computational power of computers will rival that of the human brain [Kurzweil]. All that is required, then, is a system to organize that power into an artificial intelligence. The twenty-first century holds great promise as the century that the Holy Grail of AI may finally come within our grasp.

Bibliography

Aleksander, I., & Morton, H. (1993). *Neurons and symbols: the stuff that mind is made of.* New York: Chapman & Hall.

Baldi, P., & Brunak, S. (1998). *Bioinformatics: the machine learning approach.* Cambridge, MA: MIT Press.

Bishop, C. M. (1995). *Neural networks for pattern recognition.* Oxford: Clarendon.

Blum, A. (1992). *Neural Networks in C++: An Object-Oriented Framework for Building Connectionist Systems.* New York: John Wiley & Sons.

Caudill, M. (1991, March). Evolutionary Neural Networks. *AI Expert*, pp. 28-33.

Chang, E. I., & Lippmann, R. P. (1991). Using genetic algorithms to improve pattern classification performance. In R. Lippmann, J. Moody, & D. Touretzky (Ed.), *Advances in neural information processing systems 3* (pp. 797-803). San Mateo, CA: Morgan Kaufmann Publishers.

Dodd, N. (1990). Optimisation of network structure using genetic techniques. In *IJCNN international joint conference on neural networks* (Vol.3) (pp. 965-970). San Diego.

Freeman, J. A., & Skapura, D. M. (1991). *Neural networks: algorithms, Applications, and programming techniques.* Reading, MA: Addison-Wesley.

Fullmer, B. & Miikkulainen, R. (1992). Using Marker-Based Genetic Encoding of Neural Networks To Evolve Finite-State Behaviour In F. J. Varela and P. Bourgine (editors) *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life* (ECAL-91, Paris, France). 255-262. Cambridge, MA: MIT Press.

Goldberg, D. E. (1989). *Genetic Algorithms is Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Haykin, S. (1994). *Neural networks: a comprehinsive foundation*. New York: Macmillan College.

Keesing, R., & Stork, D. G. (1991). Evolution and learning in neural networks: the number and distribution of learning trials affect the rate of evolution. In R. Lippmann, J. Moody, & D. Touretzky (Ed.), *Advances in neural information processing systems 3* (pp.804-810). San Mateo, CA: Morgan Kaufmann Publishers.

Korning, Peter G. (n.d.). *Training Neural Networks by means of Genetic Algorithms Working on Very Long Chromosomes*. [On-line]. Available: ftp://archive.cis.ohio-state.edu/pub/neuroprose/korning.nnga.ps.Z

Kurzweil, R. (March 1, 1999). When machines think. *Maclean's*, pp. 54-57. Toronto: Maclean Hunter Publishing.

Langton, C. G. (Ed.). (1997). *Artificial life: an overview*. Cambridge, MA: MIT Press.

Law, A. M., & Kelton, W. D. (1991). *Simulation Modeling and Analysis*. New York: McGraw-Hill.

Luger, G. F., & Stubblefield, W. A. (1993). *Artificial intelligence: Structures and strategies for complex problem solving* (2nd ed.). Menlo Park, CA: Benjamin/Cummings.

Mangasarian, O. (1999). *Machine learning for cancer diagnosis and prognosis*. [On-line]. Available: http://www.cs.wisc.edu/~olvi/uwmp/cancer.html

Masters, T. (1993). *Practical Neural Network Recipes in C++*. San Diego, CA: Academic Press.

Mehrotra, K., Mohan, C. K., & Ranka, S. (1997). *Elements of artificial neural networks*. Cambridge, MA: The MIT Press.

Mitchell, T. M. (1997). *Machine Learning*. Boston, MA: McGraw-Hill

Montana, D. J., & Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In Sridharan, N. S. (Ed.), *Proceedings of the eleventh international joint conference of artificial intelligence* (pp.762-767). San Mateo, CA: Morgan Kaufmann.

Moriarty, D. E. (1997). *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. Ph.D. Dissertation; Technical Report AI97-257, Department of Computer Sciences, The University of Texas at Austin.

Pinker, S. (1997). *How the mind works*. New York: W. W. Norton & Company.

Rao, V., & Rao, H. (1995). *C++ neural networks and fuzzy logic* (2nd ed.). New York: MIS Press.

Rawlins, Gregory J. E. (Ed.). (1991). *Foundations of Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann.

Rogers, J. (1997). *Object-oriented neural networks in C++*. San Diego: Academic Press.

Rumelhart, D. E., McClelland, J. L., et. al. (1986) *Parallel distributed processing: explorations in the microstructure of cognition: volume I : foundations*. Cambirdge, MA: The MIT Press.

Russell, S., & Norvig, P. (1995). *Artificial intelligence: a modern approach*. Upper Saddle River, MJ: Prentice Hall.

Schildt, H. (1995). *C++: The Complete Reference* (2nd ed.). Berkeley, CA: Osborne McGraw-Hill.

Schildt, H. (1995b). *C: The Complete Reference* (3rd ed.). Berkeley, CA: Osborne McGraw-Hill.

Scuse, D. (1999). Personal communication: July, 1999.

Stevens, W. R. (1992). *Advanced Programming in the UNIX Environment*. Reading, MA: Addison-Wesley.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: an introduction*. Cambridge, MA: MIT Press.

United States Patent. Stork, D. G., & Keesing, R. C. (1993). *Evolution and learning in neural networks: the number and distribution of learning trials affect the rate of evolution.* Patent Number: 5,245,696.

Wasserman, P. D. (1993). *Advanced Methods in Neural Computing.* New York: Van Nostrand Reinhold.

WDBC. University of Wisconsin, Madison. (1999). *machine-learn.* [On-line]. Available: ftp://ftp.cs.wisc.edu/math-prog/cpo-dataset/machine-learn/

WDBC.doc. University of Wisconsin, Madison. (1999). *machine-learn.* [On-line]. Available: ftp://ftp.cs.wisc.edu/math-prog/cpo-dataset/machine-learn/WDBC/WDBC.doc

Weeks, E. R. & Burgess, J. M. (1997, August). Evolving artificial neural networks to control chaotic systems. *Physical Review E* (Vol. 56, No. 2). pp. 1531-1540.

Weiss, S. M., & Kulikowski, C. A. (1991). *Computer Systems that Learn.* San Francisco, CA: Morgan Kaufmann.

Whitley, L. D. (Ed.). (1993). *Foundations of genetic algorithms 2.* San Mateo, CA: Morgan Kaufmann.

Winston, P. H. (1993). *Artificial intelligence* (3rd ed.). Menlo Park, CA: Addison-Wesley.

Wolberg, W., Street, W. N., & Mangasarian, O. (1995). *Wisconsin Diagnostic Breast Cancer.* [On-line]. Available: ftp://ftp.ics.uci.edu/pub/machine-learning-databases/breast-cancer-wisconsin/wdbc.names

Zurada, J. M. (1992). *Introduction to Artificial Neural Systems.* St. Paul, MN: West Publishing Company.

Appendix A

This appendix consists of a sample run output.

A.1 Long run, full EA with PLLT

This run has been truncated. Only the first five and final five generations are shown in the interests of space conservation.

A.1.1 First Five Generations

```
Sat Jul  3 00:06:49 1999

Random Seed: 6458

569 patterns, 30 inputs, and 2 outputs.
Reading in...

Finished Reading.

Splitting...
  Allocating
  Copying training input
  allocating training output
  Copying training target
  Copying testing input
  allocating testing output
  Copying testing target

Split off first 500 patterns to train,
69 to test.

Seed in Netset: 8874

Final Random Seed  = 8974

Starting runs for 100 networks.
50 generations.
250 epochs of training per set.

Distributed Learning
Continued learning range: 0.9 to 0.97
```

Generation 0

Network:	Fitness:	Epochs:
0	0	209
1	0	202
2	0	369
3	0	242
4	0	489
5	0.956522	469
6	0.84058 23	
7	0	2
8	0.927536	246
9	0.913043	83
10	0.942029	185
11	0.927536	359
12	0	4
13	0.942029	400
14	0.956522	482
15	0.869565	43
16	0	299
17	0.956522	463
18	0.942029	262
19	0.927536	205
20	0	308
21	0	131
22	0	20
23	0	384
24	0.942029	439
25	0.898551	340
26	0.927536	117
27	0.927536	235
28	0.956522	487
29	0.956522	430
30	0.927536	280
31	0.942029	302
32	0.927536	224
33	0.927536	191
34	0	118
35	0.942029	391
36	0.855072	23
37	0.913043	71
38	0.913043	132
39	0.927536	333
40	0	382
41	0.927536	199
42	0	465
43	0.927536	190
44	0.927536	453
45	0.927536	157
46	0.942029	367
47	0.927536	198
48	0.942029	437

49	0.927536	166
50	0.898551	111
51	0	491
52	0.898551	457
53	0.898551	118
54	0.927536	346
55	0.927536	415
56	0.927536	266
57	0.913043	110
58	0.898551	120
59	0.942029	456
60	0	392
61	0	415
62	0.913043	66
63	0	107
64	0.913043	100
65	0.884058	55
66	0	461
67	0	197
68	0	253
69	0	409
70	0.913043	180
71	0.956522	491
72	0.956522	458
73	0	3
74	0.956522	240
75	0.913043	109
76	0.913043	83
77	0.927536	439
78	0.956522	448
79	0.869565	53
80	0	23
81	0	99
82	0	371
83	0.942029	393
84	0.927536	311
85	0.898551	74
86	0.927536	121
87	0.463768	414
88	0.956522	446
89	0.956522	476
90	0.913043	183
91	0.913043	262
92	0.826087	32
93	0.826087	55
94	0.942029	273
95	0.956522	387
96	0.898551	101
97	0.942029	297
98	0.956522	465
99	0.927536	340

```
Total fitness:  66.913

Real average:   0.66913

Miss-0 avg:     0.916617

starting continuance
ending continuance
Continued nets: 30


Generation elapsed run time: 449 s, or 7.48333 min.
Generation 1
Network:        Fitness:    Epochs:
    0           0.971014    202
    1           0.971014    243
    2           0.246377    15
    3           0.942029    211
    4           0.971014    395
    5           0           53
    6           0.913043    64
    7           0.913043    234
    8           0.927536    275
    9           0           157
    10          0.927536    348
    11          0.927536    147
    12          0.956522    35
    13          0.710145    369
    14          0           11
    15          0.927536    199
    16          0.913043    65
    17          0.898551    77
    18          0           282
    19          0.927536    222
    20          0.927536    71
    21          0.956522    109
    22          0           177
    23          0           417
    24          0.971014    385
    25          0.971014    432
    26          0.927536    280
    27          0.942029    412
    28          0.217391    187
    29          0.927536    226
    30          0.927536    226
    31          0.956522    497
    32          0.927536    143
    33          0.927536    140
    34          0.942029    234
```

35	0.956522	214
36	0.927536	174
37	0.927536	260
38	0.956522	296
39	0	223
40	0.927536	354
41	0.913043	105
42	0.927536	405
43	0	119
44	0.927536	164
45	0.913043	179
46	0.942029	395
47	0.927536	117
48	0	91
49	0.956522	393
50	0.927536	292
51	0.942029	276
52	0.927536	301
53	0.956522	393
54	0.942029	248
55	0.956522	196
56	0.942029	376
57	0.927536	94
58	0	447
59	0.927536	339
60	0.942029	345
61	0.956522	468
62	0	74
63	0.956522	417
64	0	1
65	0.942029	380
66	0.927536	217
67	0	5
68	0.913043	221
69	0	22
70	0	302
71	0	308
72	0	298
73	0	241
74	0.956522	434
75	0.942029	243
76	0.956522	498
77	0.927536	306
78	0.927536	230
79	0.927536	284
80	0.942029	337
81	0.942029	415
82	0.869565	37
83	0.927536	199
84	0.927536	311
85	0.927536	479

```
    86          0.927536    110
    87          0.927536    154
    88          0.898551    63
    89          0           1
    90          0.666667    133
    91          0.942029    300
    92          0.927536    217
    93          0           65
    94          0.927536    183
    95          0.942029    316
    96          0.927536    375
    97          0.927536    265
    98          0.956522    495
    99          0.884058    40

Total fitness:  72.8841

Real average:   0.728841

Miss-0 avg:     0.911051

starting continuance
ending continuance
Continued nets: 35


Generation elapsed run time: 477 s, or 7.95 min.
Generation 2
Network:        Fitness:    Epochs:
    0           0.942029    143
    1           0.710145    29
    2           0.246377    435
    3           0.971014    265
    4           0.956522    492
    5           0           323
    6           0.927536    294
    7           0.942029    223
    8           0.956522    173
    9           0.971014    35
    10          0.956522    467
    11          0.956522    488
    12          0.927536    328
    13          0.956522    490
    14          0.971014    342
    15          0.942029    466
    16          0.971014    380
    17          0.913043    47
    18          0.217391    114
    19          0           432
    20          0.956522    460
```

21	0.913043	52
22	0.927536	114
23	0.971014	371
24	0.956522	499
25	0.942029	338
26	0.971014	494
27	0.971014	481
28	0.188406	43
29	0.942029	387
30	0.956522	186
31	0.927536	279
32	0.913043	92
33	0.246377	20
34	0.942029	312
35	0.217391	79
36	0	498
37	0.927536	214
38	0.956522	464
39	0.942029	309
40	0.927536	229
41	0	412
42	0	23
43	0.971014	400
44	0	38
45	0.927536	435
46	0.927536	446
47	0.927536	246
48	0.927536	346
49	0.942029	351
50	0.927536	184
51	0	453
52	0	371
53	0.956522	417
54	0.927536	162
55	0.927536	369
56	0.927536	306
57	0.942029	445
58	0.956522	488
59	0.913043	130
60	0.869565	102
61	0.942029	208
62	0.927536	307
63	0.927536	209
64	0	133
65	0.927536	249
66	0.956522	490
67	0.942029	188
68	0.927536	313
69	0.927536	187
70	0.898551	56
71	0.913043	258

```
    72          0           442
    73          0.956522    397
    74          0.826087    20
    75          0.927536    166
    76          0.942029    338
    77          0.927536    162
    78          0.869565    74
    79          0.913043    100
    80          0.318841    55
    81          0.927536    159
    82          0.927536    282
    83          0.927536    246
    84          0.956522    220
    85          0.956522    395
    86          0           207
    87          0.956522    495
    88          0.971014    303
    89          0.956522    439
    90          0.913043    140
    91          0.884058    47
    92          0.956522    355
    93          0           73
    94          0.942029    361
    95          0.971014    384
    96          0           15
    97          0.927536    211
    98          0.217391    93
    99          0.956522    355

Total fitness:  76.4203

Real average:   0.764203

Miss-0 avg:     0.878394

starting continuance
ending continuance
Continued nets: 42



Generation elapsed run time: 538 s, or 8.96667 min.
Generation 3
Network:        Fitness:    Epochs:
    0           0.173913    450
    1           0.971014    444
    2           0.927536    324
    3           0.797101    206
    4           0.971014    214
    5           0.942029    398
    6           0.724638    418
```

7	0.971014	400
8	0.173913	431
9	0	414
10	0.971014	325
11	0.724638	113
12	0.913043	97
13	0.956522	476
14	0.971014	292
15	0.971014	485
16	0.710145	455
17	0.927536	282
18	0.927536	439
19	0.971014	430
20	0.971014	212
21	0.202899	57
22	0.681159	10
23	0.956522	194
24	0.956522	391
25	0.217391	240
26	0.956522	473
27	0.927536	271
28	0	169
29	0.927536	210
30	0.927536	197
31	0.942029	235
32	0.942029	322
33	0.73913 479	
34	0.927536	183
35	0	273
36	0.971014	126
37	0.971014	14
38	0.913043	100
39	0.956522	471
40	0.971014	327
41	0.956522	127
42	0.913043	96
43	0.927536	154
44	0.855072	42
45	0	87
46	0.971014	127
47	0.956522	316
48	0.927536	195
49	0.956522	384
50	0.927536	279
51	0.927536	451
52	0.898551	357
53	0	319
54	0.971014	332
55	0	319
56	0.913043	82
57	0	337

```
    58          0           34
    59          0.956522    449
    60          0.913043    121
    61          0.927536    278
    62          0.942029    334
    63          0.971014    434
    64          0.927536    101
    65          0.942029    467
    66          0.927536    43
    67          0.927536    295
    68          0           49
    69          0.927536    313
    70          0.956522    175
    71          0           483
    72          0           352
    73          0           344
    74          0.956522    438
    75          0.927536    300
    76          0           23
    77          0           341
    78          0.927536    115
    79          0           297
    80          0.956522    386
    81          0.492754    4
    82          0.971014    321
    83          0.971014    433
    84          0.927536    25
    85          0.942029    317
    86          0           365
    87          0.898551    56
    88          0.913043    117
    89          0.84058 24
    90          0           209
    91          0.971014    494
    92          0.173913    11
    93          0.942029    405
    94          0.942029    253
    95          0.956522    305
    96          0.927536    153
    97          0.927536    268
    98          0.956522    428
    99          0.724638    483

Total fitness:  72.4203

Real average:   0.724203

Miss-0 avg:     0.872534

starting continuance
ending continuance
```

Continued nets: 43

Generation elapsed run time: 573 s, or 9.55 min.
Generation 4

Network:	Fitness:	Epochs:
0	0.956522	449
1	0.942029	291
2	0.971014	249
3	0.956522	483
4	0	63
5	0.971014	129
6	0	67
7	0.246377	178
8	0.956522	393
9	0.971014	114
10	0.956522	194
11	0	4
12	0.971014	314
13	0.971014	372
14	0.942029	400
15	0.695652	209
16	0.898551	123
17	0.971014	197
18	0.231884	481
19	0.927536	420
20	0.971014	232
21	0.971014	158
22	0.942029	432
23	0.971014	462
24	0.927536	94
25	0.971014	351
26	0.971014	265
27	0.710145	235
28	0.217391	129
29	0.927536	335
30	0.913043	89
31	0.942029	101
32	0.956522	108
33	0.246377	60
34	0.217391	295
35	0.956522	428
36	0.927536	242
37	0.971014	328
38	0.956522	486
39	0.956522	230
40	0.971014	456
41	0	87
42	0	469
43	0.913043	58

44	0.942029	88
45	0	99
46	0.724638	461
47	0.927536	86
48	0.956522	489
49	0.927536	286
50	0.971014	336
51	0.942029	44
52	0.927536	263
53	0.942029	295
54	0.927536	40
55	0.710145	425
56	0.217391	135
57	0.681159	86
58	0.855072	29
59	0.956522	148
60	0.913043	171
61	0.913043	80
62	0.217391	339
63	0.913043	84
64	0.217391	349
65	0	173
66	0.913043	142
67	0.898551	129
68	0.927536	197
69	0.956522	158
70	0.956522	411
71	0.971014	240
72	0	3
73	0.927536	353
74	0	37
75	0.652174	99
76	0.971014	312
77	0.971014	481
78	0.913043	20
79	0.956522	352
80	0.956522	209
81	0.942029	103
82	0.898551	91
83	0	56
84	0.971014	117
85	0.217391	144
86	0.942029	362
87	0.942029	294
88	0.188406	88
89	0.971014	315
90	0.927536	386
91	0.956522	109
92	0.956522	332
93	0	473
94	0.927536	215

```
    95          0.913043    174
    96          0.217391    464
    97          0.898551    222
    98          0.942029    329
    99          0.927536    393

Total fitness:  74.5362

Real average:   0.745362

Miss-0 avg:     0.837486

starting continuance
ending continuance
Continued nets: 43


Generation elapsed run time: 525 s, or 8.75 min.
```

A.1.2 Final Five Generations

```
Generation 45
Network:        Fitness:    Epochs:
    0           0.971014    311
    1           0           332
    2           0.971014    321
    3           0.927536    97
    4           0           137
    5           0.971014    367
    6           0.971014    211
    7           0.942029    131
    8           0.231884    258
    9           0.956522    176
    10          0.956522    89
    11          0.942029    47
    12          0.956522    27
    13          0.956522    228
    14          0.942029    23
    15          0.971014    220
    16          0.956522    434
    17          0.956522    417
    18          0.971014    85
    19          0.971014    369
    20          0.956522    324
    21          0.971014    467
    22          0.956522    449
    23          0.971014    372
    24          0.971014    389
    25          0.971014    448
```

26	0.942029	12
27	0.971014	110
28	0.927536	103
29	0.956522	158
30	0.956522	69
31	0.956522	443
32	0.652174	40
33	0.971014	346
34	0.971014	362
35	0.942029	179
36	0.956522	219
37	0	11
38	0.753623	31
39	0.956522	132
40	0.956522	425
41	0.971014	261
42	0.971014	276
43	0.971014	161
44	0.971014	159
45	0.956522	476
46	0.73913 266	
47	0.956522	255
48	0.942029	146
49	0.956522	241
50	0.956522	103
51	0.753623	65
52	0.971014	78
53	0	297
54	0.753623	379
55	0.971014	464
56	0.956522	499
57	0.913043	495
58	0.971014	131
59	0.246377	8
60	0.985507	24
61	0.942029	482
62	0.956522	234
63	0.971014	404
64	0.956522	294
65	0.956522	97
66	0.942029	26
67	0.753623	340
68	0.956522	352
69	0.971014	233
70	0.971014	228
71	0.956522	300
72	0.753623	360
73	0	122
74	0.927536	437
75	0.942029	202
76	0.956522	321

```
    77          0.971014    300
    78          0.971014    248
    79          0.971014    491
    80          0.956522    239
    81          0.971014    223
    82          0.971014    306
    83          0.956522    27
    84          0.927536    381
    85          0.753623    274
    86          0.971014    319
    87          0.971014    393
    88          0.956522    66
    89          0.956522    145
    90          0.942029    354
    91          0.942029    471
    92          0           151
    93          0.942029    323
    94          0.956522    435
    95          0           404
    96          0.956522    139
    97          0.956522    175
    98          0           261
    99          0.231884    293

Total fitness:  84.2464

Real average:   0.842464

Miss-0 avg:     0.915721

starting continuance
ending continuance
Continued nets: 72



Generation elapsed run time: 340 s, or 5.66667 min.
Generation 46
Network:        Fitness:    Epochs:
    0           0.246377    151
    1           0.927536    313
    2           0.971014    169
    3           0.956522    440
    4           0.942029    437
    5           0.956522    183
    6           0.710145    202
    7           0.942029    158
    8           0.956522    103
    9           0.942029    56
    10          0.971014    233
    11          0.927536    177
```

12	0.942029	382
13	0.971014	468
14	0.971014	208
15	0.927536	329
16	0.927536	236
17	0.971014	142
18	0.971014	146
19	0.927536	237
20	0.956522	366
21	0	159
22	0.710145	365
23	0.246377	215
24	0	240
25	0.246377	185
26	0	126
27	0.971014	271
28	0.231884	105
29	0.971014	278
30	0.231884	478
31	0.971014	72
32	0.246377	122
33	0	478
34	0.971014	359
35	0.956522	482
36	0.956522	65
37	0.956522	184
38	0	243
39	0.565217	16
40	0.927536	247
41	0.231884	42
42	0.971014	392
43	0.217391	350
44	0.942029	96
45	0.971014	335
46	0.971014	489
47	0.971014	209
48	0.898551	59
49	0.971014	105
50	0.956522	223
51	0.971014	282
52	0.942029	248
53	0.956522	490
54	0.927536	104
55	0.956522	483
56	0.971014	498
57	0.956522	21
58	0.956522	66
59	0.971014	206
60	0.942029	277
61	0.971014	346
62	0.246377	150

```
    63          0.971014    400
    64          0.942029    94
    65          0.971014    132
    66          0.898551    314
    67          0.971014    98
    68          0           342
    69          0           117
    70          0.971014    168
    71          0.971014    248
    72          0.913043    255
    73          0           59
    74          0.956522    183
    75          0.971014    451
    76          0.710145    380
    77          0.942029    47
    78          0.681159    167
    79          0.956522    481
    80          0.231884    248
    81          0.971014    435
    82          0.971014    51
    83          0.971014    178
    84          0           177
    85          0.971014    358
    86          0.971014    300
    87          0.956522    89
    88          0           352
    89          0.971014    78
    90          0.956522    472
    91          0.971014    454
    92          0           127
    93          0.956522    143
    94          0.971014    234
    95          0.956522    408
    96          0.942029    214
    97          0.913043    92
    98          0.956522    390
    99          0.956522    495

Total fitness:  76.4638

Real average:   0.764638

Miss-0 avg:     0.859143

starting continuance
ending continuance
Continued nets: 60


Generation elapsed run time: 336 s, or 5.6 min.
```

Generation 47

Network:	Fitness:	Epochs:
0	0.942029	157
1	0.971014	462
2	0.942029	432
3	0.956522	356
4	0.666667	42
5	0.956522	21
6	0.956522	291
7	0.971014	406
8	0.246377	7
9	0.956522	175
10	0.942029	137
11	0.942029	179
12	0.971014	9
13	0.942029	79
14	0.956522	4
15	0.971014	163
16	0.971014	389
17	0.942029	60
18	0.956522	82
19	0.956522	441
20	0.971014	14
21	0.942029	129
22	0.956522	472
23	0.927536	462
24	0.956522	334
25	0	24
26	0.971014	317
27	0.985507	75
28	0.942029	208
29	0.956522	439
30	0.942029	41
31	0.956522	385
32	0	111
33	0.971014	388
34	0.956522	183
35	0.681159	221
36	0.246377	89
37	0.971014	352
38	0.217391	53
39	0.956522	255
40	0.971014	58
41	0.956522	141
42	0.942029	205
43	0.942029	92
44	0.956522	317
45	0.971014	143
46	0.971014	452
47	0.971014	347
48	0.942029	373

49	0.927536	382
50	0.956522	415
51	0.971014	31
52	0.956522	458
53	0.942029	51
54	0.913043	330
55	0.971014	282
56	0.971014	321
57	0.956522	451
58	0	115
59	0.971014	242
60	0	173
61	0.202899	3
62	0.753623	408
63	0.942029	115
64	0.942029	392
65	0.231884	250
66	0.681159	82
67	0.956522	227
68	0.956522	337
69	0.927536	32
70	0.971014	458
71	0.188406	104
72	0	394
73	0.956522	247
74	0	203
75	0.956522	395
76	0.942029	102
77	0.971014	371
78	0.971014	157
79	0.956522	167
80	0.956522	481
81	0.927536	168
82	0.956522	52
83	0.971014	370
84	0.710145	398
85	0.971014	242
86	0.956522	386
87	0.956522	34
88	0.246377	119
89	0.956522	379
90	0.956522	405
91	0.971014	476
92	0.971014	362
93	0.971014	330
94	0.724638	459
95	0	76
96	0.956522	493
97	0.971014	459
98	0.971014	190
99	0.956522	416

```
Total fitness:  82.3478

Real average:   0.823478

Miss-0 avg:     0.88546

starting continuance
ending continuance
Continued nets: 64


Generation elapsed run time: 336 s, or 5.6 min.
Generation 48
Network:        Fitness:    Epochs:
    0           0.956522    171
    1           0           14
    2           0.971014    294
    3           0.956522    52
    4           0.927536    222
    5           0.956522    3
    6           0.0434783   9
    7           0           381
    8           0           395
    9           0.898551    63
    10          0           179
    11          0.637681    47
    12          0.971014    52
    13          0.971014    432
    14          0.971014    176
    15          0.956522    314
    16          0.942029    241
    17          0.956522    236
    18          0           272
    19          0.971014    465
    20          0.971014    72
    21          0.956522    63
    22          0.956522    265
    23          0.971014    457
    24          0.971014    468
    25          0.898551    366
    26          0.971014    454
    27          0.956522    81
    28          0.971014    448
    29          0.942029    456
    30          0.971014    35
    31          0.217391    214
    32          0.971014    391
    33          0.971014    46
    34          0           193
```

35	0.956522	34
36	0.942029	415
37	0.942029	100
38	0	458
39	0	301
40	0	330
41	0.942029	454
42	0.942029	99
43	0.927536	72
44	0.971014	313
45	0.971014	459
46	0.246377	124
47	0.956522	452
48	0.956522	236
49	0.956522	119
50	0.956522	165
51	0.927536	240
52	0.971014	315
53	0.956522	458
54	0.971014	316
55	0.942029	84
56	0.913043	10
57	0.971014	119
58	0.971014	192
59	0.942029	3
60	0.956522	335
61	0.971014	398
62	0.971014	289
63	0.971014	382
64	0.246377	11
65	0.971014	224
66	0.971014	408
67	0.927536	305
68	0.971014	129
69	0.942029	313
70	0.956522	224
71	0.956522	330
72	0.942029	247
73	0.956522	467
74	0.753623	267
75	0.956522	69
76	0.753623	85
77	0.956522	271
78	0.971014	379
79	0.971014	179
80	0.971014	257
81	0.927536	162
82	0.971014	441
83	0.971014	130
84	0.623188	193
85	0.971014	101

```
    86          0.971014    417
    87          0           349
    88          0.927536    182
    89          0.971014    351
    90          0.246377    1
    91          0.971014    110
    92          0.971014    211
    93          0           65
    94          0.942029    434
    95          0           273
    96          0.956522    14
    97          0           198
    98          0.956522    273
    99          0.942029    230

Total fitness:  78.3768

Real average:   0.783768

Miss-0 avg:     0.900883

starting continuance
ending continuance
Continued nets: 69


Generation elapsed run time: 321 s, or 5.35 min.
Generation 49
Network:        Fitness:    Epochs:
    0           0.971014    284
    1           0.927536    259
    2           0.956522    218
    3           0           430
    4           0           16
    5           0           226
    6           0.971014    282
    7           0           208
    8           0.956522    177
    9           0.971014    438
    10          0.942029    178
    11          0.971014    229
    12          0.971014    241
    13          0.927536    349
    14          0.927536    156
    15          0.971014    198
    16          0.971014    157
    17          0.246377    31
    18          0.724638    179
    19          0.956522    478
    20          0.913043    97
```

21	0.942029	342
22	0.956522	187
23	0.246377	10
24	0.956522	372
25	0.942029	491
26	0.956522	37
27	0.956522	284
28	0.724638	214
29	0.956522	258
30	0.956522	160
31	0.971014	468
32	0.753623	329
33	0.971014	316
34	0.942029	241
35	0.971014	64
36	0.724638	401
37	0.971014	431
38	0.956522	467
39	0.956522	202
40	0.971014	462
41	0.971014	497
42	0.971014	127
43	0.246377	409
44	0.971014	311
45	0.971014	143
46	0	135
47	0.971014	497
48	0.956522	388
49	0.956522	414
50	0.971014	245
51	0.956522	398
52	0.971014	292
53	0.956522	232
54	0.927536	33
55	0.971014	72
56	0.971014	352
57	0.956522	155
58	0.971014	413
59	0.956522	346
60	0.913043	425
61	0.913043	134
62	0.956522	268
63	0.942029	291
64	0.231884	355
65	0.246377	284
66	0.971014	455
67	0.956522	491
68	0.942029	244
69	0.942029	371
70	0.956522	269
71	0	324

72	0.956522	475
73	0.956522	39
74	0.927536	130
75	0.971014	111
76	0.971014	151
77	0.971014	467
78	0.971014	61
79	0.956522	235
80	0.956522	371
81	0.246377	475
82	0.724638	247
83	0.724638	95
84	0.753623	18
85	0.971014	304
86	0.956522	219
87	0.942029	11
88	0.956522	478
89	0.956522	36
90	0.956522	468
91	0.913043	429
92	0.231884	469
93	0.956522	422
94	0.681159	20
95	0.942029	34
96	0.927536	386
97	0.971014	70
98	0.956522	297
99	0.971014	142

Total fitness: 83.029

Real average: 0.83029

Miss-0 avg: 0.883287

Total elapsed run time: 21676 s, or 361.267 min.

Appendix B

This appendix contains the C++ code for the evolutionary and neural systems.

B.1 Neural Network Code

B.1.1 Debug.h

```
//#define VERBOSE_NN
#include <iostream.h>
```

B.1.2 NLayer.h

```
//Layer.h

#include <string.h> //for memset
#include "Debug.h"

typedef double (*SimpActFunc)(const double in);
typedef double * (BatchActFunc)(const double* in, const double* out);

class NLayer
{


public:

      //layer Constructor:

      NLayer();

      NLayer(int numin, int numnodes, double wrange, double brange);

      //Destructor
      ~NLayer();

      //Set fuctions
      bool SetBatchActivation(double * (*Batch)(const double* in, const
double* out));
      bool SetSimpleActivation(double (*Simple)(const double in));

      inline const double *SetInput(double *in){return (Input=in);}
      inline const double *SetOutput(double *out){return (Output=out);}
```

```
      inline bool OutputIsNull(){return (Output==NULL);}
      inline bool InputIsNull(){return (Input==NULL);}
      inline double *GetOutput(){return Output;}
      inline const double *GetInput(){return Input;}
      inline double **SetWeights(double** wts){return(Weights=wts);}
      inline const int NumNodes(){return outsize;}
      inline const int NumIn(){return insize;}

      inline double *GetET(){return ErrorTerm;}
      inline const double SetETElement(int i, double towhat){return
ErrorTerm[i]=towhat;}

      inline const double GetETElement(int x){return ErrorTerm[x];}
      inline const double GetOutElement(int i){return Output[i];}
      inline const double GetInElement(int i){return Input[i];}

      inline const double GetWeightElement(int i, int j){return
Weights[i][j];}
      inline const double SetWeightElement(int i, int j, double x){return
Weights[i][j]=x;}

      inline const double GetBiasElement(int i){return Bias[i];}
      inline const double SetBiasElement(int i, double x){return
Bias[i]=x;}


       void pieceofship(){return;}

      //recurrent vectors must have an initial input. (Later)

      //Other
      double* Evaluate();//Evaluate input*weights, apply activation.
      double* BackPass(double*source, double* dest);//Take array pointed
to by
                                          //double * and pass through
weights.
                                          //MUST be the right size array.

      void DumpWeights();



private:

      int insize, outsize;
      const double *Input;//Allocated outside. We don't go
                                                //about modifying our
own inputs.
      double *Output;//Allocated outside
      double **Weights;//Allocated inside
      double *ErrorTerm;//Allocated inside
```

```
      double *Bias;//Bias term allocated inside


      //Activation mechanisms
            //All this activation stuff is easily solved. Layers have ONE
activation
            //function. Multiple layers at the same level are possible.
That deals with
            //everything, because in a fully connected graph, it's all
isomorphic anyway.
            //We will provide 2 activation functions. A batch one and a
simple one. Either
            //or both can be used.

            //This is the Batch Activation. It takes an array of double
in, a pointer
            //to sufficient memory, and returns a pointer to that memory
that it "filled in"
      double* (*BatchActivation)(const double* in, const double* out);

            //This is the simple Activation. It takes a double and returns
a double
      double (*SimpleActivation)(const double in);

            //It may also be beneficial to add one which modifies its
paramaters.


      //Recurrence mechanisms (Ignore for now)
      bool Done;
      int RecurrenceCount;
      int RecurrenceLimit;
      //Recurrence test function here: tests whether it's time to
terminate a recurrence

};
```

B.1.3 neural.h

```
//Neural.h
#include "NLayer.h"

//If you want debug output, #define VERBOSE_NN
#include "Debug.h"
//#include <string.h> //for memset

class NeuralNet
{

      protected:
```

```
            double **Input;//an array of input vectors; ie an array of
double arrays.
            double **Output;//Array of output vectors

            double **Desired;//Array of Output Vectors

            double **layerVecs;//an array of double arrays.
                                        //Each layer of the network
points to the
                                        //memory "between"
                                        //allocate each one as output
when we allocate each NLayer.
                                        //Thus, processing layer [0]
outputs to layerVec[0], etc.
                                        //Obviously, there is one less
of these than the number of
                                        //layers, as the last one
points to output.
                                        //unnecessary:
                                        //layervecs[0] is the "fan" ie
puts input into everything.

            int numLayersizesSet;//How many layers are allocated with
sizes

            int numLayers;//Number of layers in Network
            int numActivations;
            int numPatterns; //Number of Input Patterns to be processed.
            int whichPattern;//Input pattern currently being processed.

            int inVecSize;//number of elements in input vector.
            int outVecSize;//number of elements in output/desired vector.

            NLayer**layers;//ptr to ptr so we can allocate one by one

            SimpActFunc* ActRecord;

            double weightRange;//note that these are not arrays of
values, one
            double biasRange;//for each layer; they should be, but there
is no
                                    //reason for this at this time.

            double lrate;     //the learning rate;

      public:
            bool Run(bool output);  //Produce Output to Input

            bool Run(bool output, double **in,//Produce Output to Input
for
                  double **out, int num);//given non-training array
```

```
            bool Evaluate();                //Evaluate a single pattern
            virtual bool Learn(int cycles) = 0;//Learn Input

            //NeuralNet(int nlayers, int inputSize, int outputSize, int
numpats,
            //                double **invecs, double**outvecs);
            NeuralNet(int nlayers, int nactivations, int inputSize, int
outputSize,
                              int numpats, double** invecs, double**
outvecs, double** desvecs,
                              double learn=.1);

            ~NeuralNet();

            bool AllocateActivationList(int howMany);

            //The way Assign and SetSimple -Activation work is as follows.
            //SetSimple- is called to set the network's activation array
to point to the
            //activation function. AssignActivation is used to associate
an activation in
            //this master list (in the network) with a particular layer.
The caveat to this
            //is IFF there is only one activation function for the
network,
            //SetSimpleActivation will sense this and automatically fill
in all the layers,
            //precluding the need to call AssignActivation.
            bool AssignActivation(int which, int layer);

            bool SetSimpleActivation(int which, SimpActFunc thefunc);


            bool SetLayerSize(int whichlayer, int layerin, int
layernodes);

            bool Connect(int source, int destination);
            bool Connect(NLayer source, NLayer destination);

            bool TestConnections();

            bool SetLayerWeights(int thelayer, double** theWeights);

            double GetLayerWeight(int theLayer, int i, int j)
                        {return layers[theLayer]-
>GetWeightElement(i,j);}

            double SetLayerWeight(int theLayer, int i, int j, double
value)
```

```
                        {return layers[theLayer]-
>SetWeightElement(i,j,value);}

            void DumpNet();
            virtual void PrintStats(ostream& foo=cout);
            inline const int GetNumLayers(){return numLayers;}
            inline const int GetLayerSize(int which){return
layers[which]->NumNodes();}
            inline const int GetLayerInSize(int which){return
layers[which]->NumIn();}
            inline const double GetWeightRange() {return weightRange;}


};
```

B.1.4 backprop.h

```
//backprop.h
#pragma once

#include "neural.h"

class BPNet: public NeuralNet
{
      private:
            static inline double Sigmoid(const double in);


      public:
            BPNet(int nlayers, int* sizes, int insize, int outsize, int
numpats,
                  double** inv, double** outv, double** desv);

            bool Learn(int cycles);
            void PrintStats(ostream& foo=cout);

};
```

B.1.5 NLayer.cc

```
//Layer.cc
#include <stdlib.h>
#include <iomanip.h>
#include "NLayer.h"

//Layer Definitions

NLayer::NLayer()
{
}
```

```
NLayer::NLayer(int numin, int numnodes, double wrange, double brange)
{
	//Sets size, but leaves input and output to be connected.
	//Set up layer size
	insize = numin;
	outsize = numnodes;
	Input=NULL;
	Output=NULL;

	#ifdef VERBOSE_NN
		cout<<"Initial ranges; weights: "<<wrange<<" bias:
"<<brange<<endl<<endl;
	#endif

	Weights = new double*[insize];//DON'T FORGET TO RANDOMIZE!!

	//this is the random seed I used for testing with Dr. Scuse's data
	//srand(1574);

	for(int i=0;i<insize;i++)
	{
		Weights[i]=new double[outsize];
		for(int j=0;j<outsize;j++)
			Weights[i][j]= (wrange- -wrange)*(double(rand())/
RAND_MAX)+ -wrange;
	}

	ErrorTerm=new double[outsize];

	Bias=new double[outsize];
	for(int i=0;i<outsize;i++)
		Bias[i]=(brange- -brange)*(double(rand())/RAND_MAX)+ -
brange;

	//Set activation functions to NULL
	BatchActivation=NULL;
	SimpleActivation=NULL;


}

NLayer::~NLayer()
{
	delete[] Bias;
	for(int i=0;i<insize;i++)
		delete[] Weights[i];
	delete[] Weights;
	delete[] ErrorTerm;
}
```

```
bool NLayer::SetBatchActivation(double * (*Batch)(const double* in, const
double* out))
{
      //Set up Batch Activation

      return BatchActivation = Batch;
}

bool NLayer::SetSimpleActivation(double (*Simple)(const double in))
{     //why can't this be inline?

      //Set up Simple Activation
      return SimpleActivation = Simple;
}


double* NLayer::Evaluate()
{
      //Outputs a pointer to the array we filled up, or NULL if
      //something went wrong (though the array may be filled up
      //anyway. it probably is.

      //We assume that the Output array is zeroed... who should do that?
      //We had better, because it has to be done every time.
      //This has been tested in the debugger; it works.
      memset((void*)Output,'\0', sizeof(double)*outsize);


      //Now, the way this works, is for every input value, ie position
      //on the input vector, i, each is multiplied by every one of it's
      //associated weights, [i,j], and added to the appropriate output
      //vector element, j. Thus, on each pass, with each subsequent input
      //vector, the outputs are accumulated until i hits the top (insize)
      //and at that point they're all there. This was done this way because
      //The inputs are accessed over and over again, and this method
groups
      //these accesses. The weights are accessed once each, so it doesn't
matter.
      //However, each output vector element is accessed as many times as
the
      //number of input elements. Thus, since writing is slower, this
could
      //be a bottleneck in performance. To access the output vector
elements
      //once each (at the cost of the input element optimization) we can
swap
      //the two 'for' conditions. The rest stays the same.

      //Note: if we want to be really anal, we can determine which
      //will have the greater number of accesses, input or output, and
      //run the appropriate code.
```

```
	//Zero the output array with bzero or memzero or something.

	for(int i=0;i<insize;i++)
		for(int j=0;j<outsize;j++)
		{
			Output[j]+=Input[i] * Weights[i][j]+Bias[j];//Every
Output is the

		}

	if (BatchActivation)
	{	if (Output==BatchActivation(Output, Output))
			return Output;
	}

	else if (SimpleActivation)
		for(int i=0;i<outsize;i++)
			Output[i]=SimpleActivation(Output[i]);


	return Output;
}

double* NLayer::BackPass(double*source, double*dest)
{
	//The difference between this and Pass is that this function uses
	//insize and outsize for the bounds in the opposite way of pass.
	//source and dest are provided for convenience, and must be EXACTLY
	//the right size. that is, dest is of size insize and source is of
	//size outsize
	//returns a pointer to dest.
	//dest will be set to zero and modified.

	memset((void*)dest,'\0', sizeof(double)*insize);

	for(int i=0;i<insize;i++)
		for(int j=0;j<outsize;j++)
		{
			dest[i]+=source[j] * Weights[i][j];//Every Output is
the

		}

	return dest;
}

void NLayer::DumpWeights()
{
	for(int i=0;i<insize;i++)
```

```
	{
		for(int j=0;j<outsize;j++)
			cout<<setprecision(15)<<Weights[i][j]<<"  ";
		cout<<endl;
	}
}
```

B.1.6 neural.cc

```
//Neural.cc
#include <iostream.h>

#include "neural.h"


//----------------------------------------
//NeuralNet Definitions

//
//
// NeuralNet::NeuralNet(int nlayers, int inputSize, int outputSize, int
numpats,
//                              double **invecs, double**outvecs)
// {
//    //Allocate number of layers
//    layers = new NLayer*[nlayers];
//    numLayers=nlayers;
//    numLayersizesSet=0;
//
//    ActRecord=NULL;
//    numActivations=0;
//
//    //set all the layers to NULL
//    for (int i=0;i<numLayers;i++)
//          layers[i]=NULL;
//
//
//    //now, allocate the memory between the layers. ie the containers
//    layerVecs = new double*[nlayers-1];
//    //allocate the input container
//          //no. point at Input as necessary.
//          //layerVecs[0] = new double [inputSize];
//
//    numPatterns=numpats;//set the number of input/output patterns to
process.
//    whichPattern=0;   //set which pattern we're currently processing.
//
//    Input=invecs;
//    Output=outvecs;
// }
```

```
//
.....
.........


//
.....NeuralNet.....
.........
NeuralNet::NeuralNet(int nlayers, int nactivations, int inputSize, int
outputSize,
                              int numpats, double** invecs,
double**outvecs, double** desvecs,
                              double learn)
{
      //Set learning rate
      lrate=learn;

      //Allocate number of layers
      layers = new NLayer*[nlayers];
      numLayers=nlayers;
      numLayersizesSet=0;
      inVecSize=inputSize;
      outVecSize=outputSize;

      //Allocate number of activations
      ActRecord=new SimpActFunc[nactivations];

      numActivations=nactivations;

      //set all activations to NULL
      for(int i=0;i<nactivations;i++)
            ActRecord[i]=NULL;

      //set all the layers to NULL
      for (int i=0;i<numLayers;i++)
            layers[i]=NULL;

      //now, allocate the memory between the layers. ie the containers
      layerVecs = new double*[nlayers-1];
      //allocate the input container
            //no. point at Input as necessary.
            //layerVecs[0] = new double [inputSize];

      numPatterns=numpats;//set the number of input/output patterns to
process.
      whichPattern=0;   //set which pattern we're currently processing.

      Input=invecs;
      Output=outvecs;
      Desired=desvecs;
```

```
	//set default range values
	weightRange=1;
	biasRange=1;

#ifdef VERBOSE_NN
	cout<<invecs[0][0]<<invecs[0][1]<<endl;
	cout<<invecs[1][0]<<invecs[1][1]<<endl;
	cout<<invecs[2][0]<<invecs[2][1]<<endl;
	cout<<invecs[3][0]<<invecs[3][1]<<endl;
#endif

}
//
.....................................................................
.........

//
.....~NeuralNet.....................................................
.........

//Need a destructor!!!!
NeuralNet::~NeuralNet()
{
	delete[] ActRecord;
	for(int i=0;i<numLayers;i++)
		delete layers[i];
	for(int i=0;i<(numLayers-1);i++)
		delete[] layerVecs[i];
	delete[] layerVecs;
	delete[] layers;

}

//Later, we will need to differentiate between simple and batch
activations.
//
.....................................................................
.........


//
.....AllocateActivationList.........................................
.........

bool NeuralNet::AllocateActivationList(int howMany)
{
	if (numLayersizesSet!=numLayers)
	{
		#ifdef VERBOSE_NN
		cout<<"Size of Layer not allocated in
NeuralNet::AllocateActivationList"<<endl;
```

```
            #endif
            return false;
      }
      //how many activation functions will we need?
      if (ActRecord!=NULL)          //if it already exists
            delete []ActRecord;           //delete it

      numActivations=howMany;       //allocate the new memory
      return ActRecord=new SimpActFunc[howMany];

}
//
.....................................................................
.........


//
.....AssignActivation.....................................................
.........
bool NeuralNet::AssignActivation(int whichact, int whichlayer)
{
      //This function assigns the layer layers[whichlayer] the activation
      //function found in ActRecord[whichact]. It returns an error for
overflow etc.

      if (whichact>=numActivations || whichlayer>=numLayers)
      {
            #ifdef VERBOSE_NN
            cout<<"Error - out of range layer or activation index
in"<<endl;
            cout<<"NeuralNet::AssignActivation"<<endl;
            #endif
            return false;
      }

      layers[whichlayer]->SetSimpleActivation(ActRecord[whichact]);

      #ifdef VERBOSE_NN
      if (ActRecord[whichact]==NULL)
            cout<<"Warning - assigning null function to layer
"<<whichlayer<<endl;
      #endif

      //This could also be done as
      //layers[whichlayer].SetSimpleActivation(ActRecord[whichact]);
      return true;
}
//
.....................................................................
.........
```

```
//
·····SetSimpleActivation·····
·····
bool NeuralNet::SetSimpleActivation(int which, SimpActFunc thefunc)
{
      //Set the activation of ActRecord[which] to thefunc
      if (!numActivations) //ie if it's zero
      {
            #ifdef VERBOSE_NN
            cout<<"Error - no activation functions allocated in
SetSimpleActivation"<<endl;
            #endif
            return false;
      }

      //if we're trying to se a function slot that doesn't exist
      if (which>=numActivations || which<0)
      {
            #ifdef VERBOSE_NN
            cout<<"Error - Activation function slot is out of range
in"<<endl;
            cout<<"NeuralNet::SetSimpleActivation"<<endl;
            #endif
            return false;
      }

      ActRecord[which]=thefunc;

      //if we only have 1 activation function, set it for all the layers
      if (numActivations==1 && which==0)//do a double check so we don't
screw up
            for (int i=0;i<numLayers;i++)
                  AssignActivation(0,i);
                                                            //This is
real bad programming practice. <grin>
      return true;
}
//
·····
·····


//
·····SetLayerSize·····
·····
bool NeuralNet::SetLayerSize(int whichlayer, int layerin, int layernodes)
{
      //whichlayer is the index of the layer we're setting, starting at
zero
```

```
      //layerin is how many inputs, and layernodes is how many nodes. ie
layerin
      //is the number of nodes in the previous layer.
      //maybe later I'll make it automatic :)

      //NLayer foo(layerin, layernodes);

      //first the debugging
      if (Input==NULL || Output==NULL||whichlayer>=numLayers)
      #ifdef VERBOSE_NN
      {
            cout<<"Error: Input and Output to Neural Network not"<<endl;
            cout<<"     allocated or Layer doesn't exist in "<<endl;
            cout<<"     NeuralNet::SetLayerSize"<<endl<<endl;
      #endif
            return false;
      #ifdef VERBOSE_NN
      }
      #endif

      //now the important stuff

      bool initialstatus=layers[whichlayer]; //use this to determine if
we increment
                                                                  //
numLayersizesSet
      bool toreturn;



      if (layers[whichlayer]!=NULL)
            delete layers[whichlayer];

      //Sets the size for a layer

      if(layers[whichlayer]=new NLayer(layerin,
layernodes,weightRange,biasRange))

            //if the allocation worked
      {
            if(numLayersizesSet<numLayers && !initialstatus)
                  numLayersizesSet++;
            if (whichlayer==0)
                  layers[whichlayer]->SetInput(Input[whichPattern]);

            //Allocate the array to store this layer's output. There are
layernodes
            //outputs, one for each neuron/node in the layer.
            //don't do it if we're the last layer.

            if (whichlayer!=numLayers-1)//if it's not the last layer
```

```
                  toreturn=layers[whichlayer]-
>SetOutput(layerVecs[whichlayer] =
                                          new double[layernodes]);

            else
                  toreturn=layers[whichlayer]-
>SetOutput(Output[whichPattern]);

            #ifdef VERBOSE_NN
            if (!toreturn)//if we pointed at a null
            {
                  cout<<"Warning: Pointing at Null in
NeuralNet::SetLayerSize"<<endl;
                  cout<<"          While setting layer
"<<whichlayer<<"."<<endl<<endl;
            }
            #endif

            return toreturn;
      }//if

      #ifdef VERBOSE_NN
      cout<<"Memory Allocation Failure in
NeuralNet::SetLayerSize"<<endl<<endl;
      #endif
      return false;
}
//
.....................................................................
.........


//
.....Connect.........................................................
.........
bool NeuralNet::Connect(int source, int destination)
{

      //Connect source layer to destination layer. Exploit the fact that
      //we're a friend.

      //By convention, only connect destination to point to source.
      //that is, destination's input is source's output.

      //Assume source is allocated, otherwise return false
      if (layers[source]->OutputIsNull())
      {
            cout<<"Warning: Pointing layer "<<destination<<"'s input to
";
            cout<<"layer "<<source<<"'s output, which is NULL."<<endl;
      }
```

```
      //return layers[destination]->SetInput(layers[source]->Output);
      return layers[destination]->SetInput(layerVecs[source]);

      //This will work only if we initialize to NULL in the constructor
      //for the layers.

      //we do this by passing NULL to our layer constructors.
}
//
.....................................................................
.........


//
.....Connect.........................................................
.........
bool NeuralNet::Connect(NLayer source, NLayer destination)
{
      return destination.SetInput(source.GetOutput());
}
//
.....................................................................
.........


//
.....TestConnections.................................................
.........
bool NeuralNet::TestConnections()
{
      //This function tests each layer to ensure that its input and output
      //are not NULL. It does not ensure that non-NULL values are valid
      bool retval=true;

      for (int i=0;i<numPatterns;i++)
      {
            if(layers[i]->InputIsNull())
                  cout<<"Input of layer "<<i<<" is NULL."<<endl;
            if(layers[i]->OutputIsNull())
                  cout<<"Output of layer "<<i<<" is NULL."<<endl;
            retval=retval&&(layers[i]->GetInput() && layers[i]-
>GetOutput());
      }

      if (numLayersizesSet!=numLayers)
      {
            cout<<numLayers-numLayersizesSet<<" of "<<numLayers<<" not
set."<<endl;
            cout<<"Layer test returned "<<retval<<"."<<endl;
            return false;
      }
```

```
      return retval;
}
//
.....................................................................
.........


//
.....SetLayerWeights.....................................................
.........
bool NeuralNet::SetLayerWeights(int thelayer, double** theWeights)
{
      //Warning. The layer class asumes it owns its weights. It will
      //dispose of the memory passed it when the destructor is called,
      //either explicitly or implicitly. This may cause a problem if
      //the weights are on the stack.

      //add error checking.

      return layers[thelayer]->SetWeights(theWeights);
}
//
.....................................................................
.........


//
.....Run.................................................................
.........
bool NeuralNet::Run(bool output)
{//runs through and evaluates each pattern


      for (whichPattern=0;whichPattern<numPatterns;whichPattern++)
      {//set which pattern
            layers[0]->SetInput(Input[whichPattern]);//set input layer
to pattern
            layers[numLayers-1]->SetOutput(Output[whichPattern]);//set
output layer

            for(int i=0;i<numLayers;i++)//I could make this an inline
function
                  layers[i]->Evaluate();
      }

      if(output)
      {
            cout<<"In";
            for(int i=0;i<inVecSize;i++)
                  cout<<"      ";
```

```
            cout<<" Out"<<endl;

            for (int i=0;i<numPatterns;i++)
            {
                  for(int j=0;j<inVecSize;j++)
                        printf("%1.3f ",Input[i][j]);
                        //cout<<Input[i][j]<<" ";
                  cout<<"== ";
                  for(int j=0;j<outVecSize;j++)
                        printf("%1.3f ",Output[i][j]);
                        //cout<<Output[i][j]<<" ";
            cout<<endl;
            }
      cout<<endl;
      }//output

      whichPattern=0;//reset for next time
      layers[0]->SetInput(Input[whichPattern]);//set input layer to
pattern
      layers[numLayers-1]->SetOutput(Output[whichPattern]);//set output
layer

      return true;
}
//
.....................................................................
.........


//
.....Run.....................................................................
.........
bool NeuralNet::Run(bool output, double **in,double **out, int num)
{     //Produce Output to Input for given non-training array

      //runs through and evaluates each pattern
      for (whichPattern=0;whichPattern<num;whichPattern++)
      {//set which pattern
            layers[0]->SetInput(in[whichPattern]);//set input layer to
pattern
            layers[numLayers-1]->SetOutput(out[whichPattern]);//set
output layer

            for(int i=0;i<numLayers;i++)//I could make this an inline
function
                  layers[i]->Evaluate();
      }

      if(output)
      {
```

```
            cout<<"In";
            for(int i=0;i<inVecSize;i++)
                  cout<<"      ";

            cout<<" Out"<<endl;

            for (int i=0;i<num;i++)
            {
                  for(int j=0;j<inVecSize;j++)
                        printf("%1.3f ",in[i][j]);
                        //cout<<Input[i][j]<<" ";
                  cout<<"== ";
                  for(int j=0;j<outVecSize;j++)
                        printf("%1.3f ",out[i][j]);
                        //cout<<Output[i][j]<<" ";
            cout<<endl;
            }
      cout<<endl;
      }//output

      whichPattern=0;//reset for next time
      layers[0]->SetInput(Input[whichPattern]);//set input layer to
pattern
      layers[numLayers-1]->SetOutput(Output[whichPattern]);//set output
layer

      return true;
}
//
.....................................................................
.........


//
.....DumpNet.....................................................
.........
void NeuralNet::DumpNet()
{
      for(int i=0;i<numLayers;i++)
      {
            cout<<"Layer "<<i<<endl;
            layers[i]->DumpWeights();
            cout<<endl;
      }
}
//
.....................................................................
.........
```

```
//
·····PrintStats·····
·····
void NeuralNet::PrintStats(ostream& foo)
{

      //print out:network architecture
      //                              numlayers
      //                              units per layer
      //                              ????
      //                        weight range
      //                        bias range
      //                        random seed    --NO. external. Has nothing
to do with network
      //                        learning rate

      foo<<"-------Network Stats--------------------------------
"<<endl<<endl;
      //archetecture:
      foo<<"Network has "<<numLayers<<" processing layers:"<<endl;
      for (int i=0;i<numLayers;i++)
            foo<<"Layer "<<i+1<<" has "<<layers[i]->NumNodes()<<"
nodes."<<endl;

      //weight range:
      foo<<"Weights initialized to +-"<<weightRange<<endl;
      //bias range:
      foo<<"Bias initialized to +-"<<biasRange<<endl;
      //Learning Rate:
      foo<<"Learning Rate: "<<lrate<<endl;
      foo<<"-----------------------------------------------------
"<<endl<<endl;
}
```

B.1.7 backprop.cc

```
//backprop.cc
#include "backprop.h"
#include <iomanip.h>
#include <console.h>
#include <Events.h>//this is for 'press s to save' functionality.
#include <Sioux.h>


BPNet::BPNet(int nlayers, int* sizes, int insize, int outsize, int
numpats,
                  double** inv, double** outv, double** desv):
                  NeuralNet(nlayers,1,insize, outsize, numpats,inv,
outv, desv)
{
      //nlayers- number of processing layers
```

```
        //insize- number of input items(length of an input vector)
        //outsize- number of output items
        //numpats- number of patterns (number of input and output vectors
        //inv       - array of input vectors (size insize)
        //outv      - array of output vectors (size outsize)
        //desv      - array of desired vectors (size outsize)

        //backprop specific:
        //sizes     -size of each processing layer

        int i;

        //set up the links.
        SetLayerSize(0,insize, sizes[0]);//set the first processing layer
                                                                    //to take
insize inputs and have
                                                                    //sizes[0]
neurons

        for (i=1;i<nlayers-1;i++)//for each layer
              SetLayerSize(i,sizes[i-1],sizes[i]);

        SetLayerSize(nlayers-1,sizes[nlayers-2],outsize);//set the last
processing
                    //layer to take sizes[nlayers-2] inputs and have
outsize neurons.
                    //we could work this into the previous loop, but let's
do it this way.
                    //that means that sizes can be one value short. let's
say it shouldn't
                    //be; we always pass a size for _EACH processing layer.

        //set activation, which should be a private member??
        SetSimpleActivation(0,Sigmoid);

        for (i=0;i<nlayers-1;i++)//Connect up the layers
              Connect(i,i+1);


        //done

}

//
.....Sigmoid**.....................................................
.........
inline double BPNet::Sigmoid(const double in)
{
        return 1.0/(1.0+exp(-in));
}
```

```
//
.....................................................................
.........

bool BPNet::Learn(int cycles)
{
      int ncycles;
            //overrides class lrate:
                  //double lrate=0.1;
      int i,j,k,pattern;//counters
      double tss;

      EventRecord event;

      clock_t ptime, starttime;
      //cycles is number of times to cycle.
      starttime=clock();
      ptime=clock()+10*CLOCKS_PER_SEC;

      char c;
      for(ncycles=0;ncycles<cycles;ncycles++)//for each cycle
      {
            //if ((ncycles+1)%100==0)
            //    cout<<ncycles<<endl;

            if (clock()>=ptime)
            {
                  //if (kbhit())
                  if (GetNextEvent(mDownMask|mUpMask,&event))
                        SIOUXHandleOneEvent(&event);
                  if (GetNextEvent(keyDownMask,&event))
                  {
                        //if (event.modifiers&cmdKey)
                              //SIOUXHandleOneEvent(&event);
                        //else
                        {
                              c=event.message&charCodeMask;
                              //cout<<c<<endl;
                              if(c=='s')
                                    cout<<"I would save here."<<endl;
                              cout<<"Starting cycle "<<ncycles<<endl;
                              cout<<"  Estimate ";
                              cout<<(cycles-ncycles)*(double((ptime-
starttime)/CLOCKS_PER_SEC)/ncycles);
                              cout<<" seconds remaining."<<endl;
                        }//else
                  }
                  ptime=clock()+10*CLOCKS_PER_SEC;
            }
//          if (ncycles%500==0)
//                if (kbhit())
```

```
//                      cout<<"Starting cycle "<<ncycles<<endl;

            #ifdef VERBOSE_BP
                  tss=0;
            #endif

            for(pattern=0; pattern<numPatterns; pattern++)//for each
pattern
            {
                  #ifdef VERBOSE_BP
                  cout<<" Evaluating Pattern "<<pattern<<endl;
                  #endif

                  layers[0]->SetInput(Input[pattern]);//set input layer
to pattern
                  layers[numLayers-1]->SetOutput(Output[pattern]);//set
output layer

                  //evaluate the network, ie forward pass
                  for(i=0;i<numLayers;i++)
                        layers[i]->Evaluate();

//if((ncycles+1)%100==0)
//{   cout<<"In   Out"<<endl;
//    cout<<theNet.Input[pattern][0]<<theNet.Input[pattern][1]<<"
"<<theNet.Output[pattern][0]<<endl;
//}

                  //now the backward pass.

                  //compute output-layer error
                  #ifdef VERBOSE_BP
                        cout<<"  Computing output-layer error"<<endl;
                  #endif
                  for (i=0;i<layers[numLayers-1]->NumNodes();i++)//for
each output neuron
                  {
                        //The error term for the neuron is the difference
between the output and
                        //desired/target, ie the output of the layer's
ith neuron - the desired
                        //pattern's ith member.
                        //theNet.layers[numLayers-1]->ErrorTerm[i]=
                        //
theNet.layers[numLayers-1]->Output[i]
                        //                                          -
theNet.Desired[pattern][i];//d=o-t


                              //d=t-o
                        layers[numLayers-1]->SetETElement(i,
```

```
                                        (Desired[pattern][i] -
layers[numLayers-1]->GetOutElement(i))

);
                        //Multiply the difference by 1-o...
                        layers[numLayers-1]->SetETElement(i,//d=(1-o)(t-
o)

(layers[numLayers-1]->GetETElement(i) *
                                                                (1.0-
layers[numLayers-1]->GetOutElement(i))
                                                                ));

                        //Multiply by o, ie the output
                        layers[numLayers-1]->SetETElement(i,//d=o(1-
o)(t-o)

(layers[numLayers-1]->GetETElement(i)*

layers[numLayers-1]->GetOutElement(i) )

);

//if (/*((pattern==0)||(pattern==3))&&*/((ncycles+1)%100==0))
//    cout<<"Pattern "<<pattern<<" Error: "<<layers[numLayers-1]-
>GetETElement(i)<<endl;


                }//output-layer error

                #ifdef VERBOSE_BP
                cout<<"  Computing hidden layer errors"<<endl;
                #endif
                //now compute error for each hidden layer. This
corresponds to any layer with
                //output, ie any and all layerVecs except the last one.
Do in reverse.

                for(i=numLayers-2;i>=0;i--)//for each hidden layer
                {
                      #ifdef VERBOSE_BP
                      cout<<"   Layer "<<i<<endl;
                      #endif
                      //calculate the last layer's error*Wts and place
in current layer's
                      //error term. (W2d)


                      layers[i+1]->BackPass(layers[i+1]->GetET(),

layers[i]->GetET());
```

```
                        for(j=0;j<layers[i]->NumNodes();j++)//for each
neuron in the current
                        {
            //(ith) layer

                              //Multiply error by 1 minus this layer's
output ((1-h)W2d)
                              double temp=(1.0-layers[i]-
>GetOutElement(j));
                              layers[i]->SetETElement(j,layers[i]-
>GetETElement(j)*temp);

                              //Multiply by this layer's output (h(1-
h)W2d)
                              temp=layers[i]->GetOutElement(j);

                              layers[i]->SetETElement(j,layers[i]-
>GetETElement(j)*temp);

                        }//for each neuron

                  }//hidden-layer errors

                  #ifdef VERBOSE_BP
                  cout<<"  Updating Weights"<<endl;
                  #endif

                  //update all weights, starting with last, working to
first.
                  //implement momentum here later.
                  //update Bias too
                  for(i=numLayers-1;i>=0;i--)//for each layer
                  {
                        for(j=0;j<layers[i]->NumIn();j++)
                              for(k=0;k<layers[i]->NumNodes();k++)
                              {
                                    double temp2;
                                    if((i-1)<0)
                                          temp2=Input[pattern][j];
                                    else
                                          temp2=layers[i]-
>GetInElement(j);
                                    temp2*=layers[i]->GetETElement(k);
                                    temp2*=lrate;
                                    layers[i]-
>SetWeightElement(j,k,(temp2+

layers[i]->GetWeightElement(j,k) )
```

```
      );
                              }
                        //update Bias

                        for(k=0;k<layers[i]->NumNodes();k++)
                              layers[i]->SetBiasElement(k,(layers[i]-
>GetBiasElement(k)+

(lrate*layers[i]->GetETElement(k)) )
                                                                  );

                  }//for each layer - weight update

                  #ifdef VERBOSE_BP
                        cout<<"  Completed Weight Update"<<endl;

                        for (i=0;i<layers[numLayers-1]->NumNodes();i++)
                              tss+=layers[numLayers-1]->GetETElement(i)*
                                                      layers[numLayers-
1]->GetETElement(i);
                  #endif

//          cout<<"tss: "<<tss<<endl;

            }//for pattern
            layers[0]->SetInput(Input[pattern]);//set input layer to
pattern
            layers[numLayers-1]->SetOutput(Output[pattern]);//set output
layer

            //update tss

//          if ((ncycles+1)%100==0)
//          {
//                cout.setf(ios_base::fixed,ios_base::floatfield);
//                cout<<"TSS: "<<setprecision(20)<<tss<<endl;
//                cout.setf(0, ios_base::floatfield);
//                //
cout<<setiosflags(ios::fixed)<<setprecision(30)<<tss<<endl;
//                //cout<<setiosflags(ios::scientific);
//                //printf("%2.30f\n",tss);
//          }
            #ifdef VERBOSE_BP
            cout<<" Completed pattern"<<endl;
            #endif
      }//for cycles
      return false;
}

void BPNet::PrintStats(ostream& foo)
```

```
{
      NeuralNet::PrintStats(foo);
}
```

B.2 Evolutionary System and Supporting System Code

B.2.1 genetic.h

```
//genetic.h

#include "backprop.h"

const double MutationRate=.001;
const double MutationRange = 1.0;

typedef double Gene;

typedef Gene* Chromosome;

typedef Chromosome* Genome;

typedef Genome* GenePool;

Genome Encode(BPNet &theNet);

GenePool EncodePopulation(BPNet** &theNets, int popSize);

BPNet** DecodePopulation(GenePool allGenes, int popSize, ostream&
filerecord);

bool Decode(BPNet &theNet, Genome theGenes);

void DestroyPop(GenePool prevgen, BPNet** &theNets, int popSize);
void DestroyPop(GenePool &prevgen, int* sizes, int popSize);


bool Cross(Chromosome x, Chromosome y);

bool Mate(Genome x, Genome y);

inline void Mutate(Chromosome x, Chromosome y);

Genome CopyGenome(const Genome& src, BPNet* net);
```

B.2.2 netparms.h

```
//netparms.h
```

```
//#pragma once
/*
#ifndef NETPARMS
#define NETPARMS
*/

extern int numPats,numIn,numOut;
extern int trainPats, testPats;

extern double ** trainIn;
extern double ** trainOut;
extern double ** trainTarget;

//#endif
```

B.2.3 genetic.cc

```
//genetic.cc
#include "genetic.h"
#include <math.h>
#include "netparms.h"

//
.....................................................................
.........


//
.....Encode.....................................................
.........

Genome Encode(BPNet &theNet)
{
      //encode each layer of the neural net into a 1-D
      //array of doubles. Return an array of those arrays.

      Genome retval;
      int size,i,j,whichLayer,genepoint;
      //double temp;


      retval=new Chromosome[theNet.GetNumLayers()];

      for (i=0;i<theNet.GetNumLayers();i++)
      {
            //let's store the size of the gene in the first (0th) element
            size=theNet.GetLayerSize(i)*theNet.GetLayerInSize(i);
            retval[i]=new Gene[size+1];
            retval[i][0]=size;

            //for (j=0;j<=size;j++);
```

```
            //    retval[i][j]=0.0;

            //ie, the first element of the array is a number which tells
            //how many _more numbers are in the array. Eg, the string
"abcd"
            //would be stored as "4abcd". So to loop, we start at 1 and
            //go to <=4
      }
      //ok, because there's no way to know, I've decided to do this in
      //column-major order. That is, go down the columns first, and read
      //into the array.

      //the number of rows = the number of inputs to the layer. The
      //number of hidden units/outputs corresponds to the number of
      //columns

      //for each Layer

      for (whichLayer=0,genepoint=1;
                  whichLayer<theNet.GetNumLayers();
                                    whichLayer++,genepoint=1)
      {
            //for each column/node:
            for(j=0;j<theNet.GetLayerSize(whichLayer);j++)
            {
                  //for each row/input

for(i=0;i<theNet.GetLayerInSize(whichLayer);i++,genepoint++)
                  {
                        //temp=theNet.GetLayerWeight(whichLayer,i,j);

retval[whichLayer][genepoint]=theNet.GetLayerWeight(whichLayer,i,j);
                        //retval[whichLayer][genepoint]=temp;
                        //note: instead of genepoint I could use i+j+1
                  }
            }
      }


      return retval;
}

//
.....................................................................
.........


//
·····EndodePopulation·····.....................................................
.........
```

```
GenePool EncodePopulation(BPNet** &theNets, int popSize)
{

      //This function takes an array of networks, and a population size
and returns
      //its genepool
      GenePool genetics;

      int i;


      genetics=new Genome[popSize];

      for (i=0;i<popSize;i++)
            if (i==12)
                  genetics[i]=Encode(*theNets[i]);
            else
                  genetics[i]=Encode(*theNets[i]);

      return(genetics);
}
//
.....................................................................
.........


//
.....Decode.....................................................
.........
bool Decode(BPNet &theNet, Genome theGenes)
{
      int whichLayer, genepoint, i, j;


      for (whichLayer=0,genepoint=1;
                  whichLayer<theNet.GetNumLayers();
                                    whichLayer++,genepoint=1)
            //for each column/node:
            for(j=0;j<theNet.GetLayerSize(whichLayer);j++)
                  //for each row/input

for(i=0;i<theNet.GetLayerInSize(whichLayer);i++,genepoint++)

theNet.SetLayerWeight(whichLayer,i,j,theGenes[whichLayer][genepoint]);
                        //I could use SetWeights here, but then I'd have
to delete the already
                        //allocated memory, and it just seems kind of
messy and foolish.
      return true;
}
```

```
//
.....
.........


//
.....DecodePopulation.....
.........
BPNet** DecodePopulation(GenePool allGenes, int popSize, ostream&
filerecord)
{
      BPNet** retval;
      int layersizes[2]={1,2};//This array will be passed to the BP
constructor.
                                                //We'll change the 0th
element every time.
      int i;

      //allocate the array of bpnet pointers
      retval=new BPNet*[popSize];

      for(i=0;i<popSize;i++)
      {
            layersizes[0]=allGenes[i][0][0]/30.0;//take the ith
network's

//first layer's size, and

//divide by the number of inputs

//to get the number of hidden units.
            retval[i]=new
BPNet(2,layersizes,numIn,numOut,trainPats,trainIn,trainOut,
                        trainTarget);

            //set all the weights from the correct chromosome.
            Decode(*(retval[i]), allGenes[i]);

            //dump it.
            //filerecord<<"Network "<<i<<endl;
            //retval[i]->PrintStats(filerecord);
            //filerecord<<endl;


      }

      return retval;
}
```

```
//
.....
.........

//
.....DestroyPopulation.....
.........
void DestroyPop(GenePool prevgen, BPNet** &theNets, int popSize)
{

      int i,j;

      for (i=0;i<popSize;i++)
      {
            for (j=0;j<theNets[i]->GetNumLayers();j++)
                  delete[] prevgen[i][j];

            delete[] prevgen[i];
      }


      delete[] prevgen;

}

//
.....
.........

//
.....DestroyPopulation.....
.........
void DestroyPop(GenePool &prevgen, int* sizes, int popSize)
{

      int i,j;

      for (i=0;i<popSize;i++)
      {
            for (j=0;j<sizes[i];j++)
                  delete[] prevgen[i][j];

            delete[] prevgen[i];
      }


      delete[] prevgen;
      prevgen=NULL;

}
```

```
//
.....
.........

//
.....Cross.....
.........

bool Cross(Chromosome x, Chromosome y)
{
      //Now, since chromosomes have their size as the 0th element,
      //our life is made easier.

      //Let's declare a couple of pointers to make life easier.

      Chromosome smaller, bigger;
      int cut, i;

      double temp;

      //First, determine which is smaller
      if(x[0]<y[0])
      {
            smaller=x;
            bigger=y;
      }
      else
      {
            smaller=y;
            bigger=x;
      }

      //now, generate a cut point, which is a number between 1 and
Smaller[0]
      //Discreen uniform = (top+1)*rand+bottom

      cut= floor( (smaller[0]) * ((double)rand()/RAND_MAX) + 1);
            //I decided that since same-size arrays can cross, I want to
enforce
            //at least one exchanged gene, hence smaller[0] is the top.

      //now, take the cut stuff our of the bigger, and put it in temp.
      //Simultaneously copy from smaller to bigger.

      for(i=1;i<=cut;i++)
      {
            temp=bigger[i];
            bigger[i]=smaller[i];
            smaller[i]=temp;
      }
```

```
	return true;
}

//
.....
.........


//
.....Mate.....
.........
bool Mate(Genome x, Genome y)
{
	//this function will be changed depending on what type of mating we
want
	//to do. For now, let's select which layer to cross over randomly.

	//We may later want to consider inter-layer mating.

	int which;
	double coin;

	//generate random [0,1] continuous
	coin=(double)rand()/RAND_MAX;

	which=( (coin<=.5)?0:1 );

	Cross(x[which], y[which]);

	Mutate(x[which], y[which]);

	return true;

}

//
.....
.........


//
.....Mutate.....
.........
inline void Mutate(Chromosome x, Chromosome y)
{
	//mutate the two chromosomes based on Binomial Variate.

	int i;
```

```
      for (i=1;i<=x[0];i++)
            if((double)rand()/RAND_MAX<=MutationRate)
                  x[i] = (MutationRange- -MutationRange)*
                                    (double(rand())/RAND_MAX)+ -
MutationRange;

      for (i=1;i<=y[0];i++)
            if((double)rand()/RAND_MAX<=MutationRate)
                  y[i] = (MutationRange- -MutationRange)*
                                    (double(rand())/RAND_MAX)+ -
MutationRange;

}
//
.....................................................................
.........


//
.....CopyGenome.....................................................
.........

Genome CopyGenome(const Genome& src, BPNet* net)
{     //passing pointer to bpnet, so no destructor called.

      //makes a copy of src
      Genome dest;
      int i;

      dest=new Chromosome[net->GetNumLayers()];

      for(i=0;i<net->GetNumLayers();i++)
      {
            dest[i]=new Gene[(int)(src[i][0]+1)];

            memcpy(dest[i],src[i],(src[i][0]+1)*sizeof(double));
            //for(j=0;j<=src[i][0];j++)
            //    dest[i][j]=src[i][j];
      }

      return dest;
}
```

B.2.4 main.cc

```
#include <iostream>
#include "backprop.h"
#include "genetic.h"
#include <time.h>
#include <console.h>
```

```
#include <fstream.h>

using namespace std;//introduces namespace std

//Useful Functions
void PrintBanner(ostream& out=cout);
inline void NextDigit(ifstream& thestream);
int EvaluateBinary (int &unclass, double tolerance, int numPats, int
vecSize,
                    double **Output, double **Target);
double CalcTSS(int numPats, int vecSize, double **Output, double
**Target);
void NetSet(BPNet** &array, int howmany, ofstream& filerecord);

//-----------------

//Variables

int numPats,numIn,numOut;
int trainPats, testPats;

int layersizes[2]={4,2};//last one must equal numOut

double ** inArray=NULL;
double ** outArray=NULL;
double ** targetArray=NULL;

double ** trainIn=NULL;
double ** trainOut=NULL;
double ** trainTarget=NULL;


double ** testIn=NULL;
double ** testOut=NULL;
double ** testTarget=NULL;


//misc control stuff
unsigned int seed=6458; //1574
int split=0;
int epochsPerCycle=250;
int numGens=50;
int eachgen;//a counter
ofstream outfile("neural.out");
bool continuance=true;
int continuers;
double contbot=.9;
double conttop=.97;

//population control
int popSize=100;  //size of population, ie number of networks
```

```
                                                    //MUST BE EVEN NUMBER
BPNet **Population;//The population. [Array of Networks]

GenePool theGenes=NULL, newGenes=NULL;//array of genomes for each netork;
int* genomeSize;//this array is to store the size of the genome,
                                  //ie how many chromosomes for a particular
individual


//........MAIN.....................................
int main(int argc, char*argv[])
{
      argc=ccommand(&argv);
      if (argc<2)
            return 1;

      int i; //a useful counter that doesn't %^&* up the debugger.


      clock_t TStart,TEnd,TTime;
      TStart=clock();
      srand(seed);

      PrintBanner();
      PrintBanner(outfile);

//reading in data file

      if (argc==3)
            split=atoi(argv[2]);
      ifstream infile(argv[1]);

      if(!infile.is_open())
      {
            cout<<"error opening file."<<endl;
            return 1;
      }

      infile>>numPats;
      NextDigit(infile);
      infile>>numIn;
      NextDigit(infile);
      infile>>numOut;
      NextDigit(infile);

      cout<<numPats<<" patterns, "<<numIn<<" inputs, and "<<numOut<<"
outputs."<<endl;
      outfile<<numPats<<" patterns, "<<numIn<<" inputs, and "<<numOut<<"
outputs."<<endl;

      //allocate array sizes, first dimension
```

```
        inArray=new double*[numPats];
        outArray=new double*[numPats];
        targetArray=new double*[numPats];


        //allocate pattern lengths, second dimension
        for (i=0;i<numPats;i++)
        {
              inArray[i]=new double[numIn];
        }
        for (i=0;i<numPats;i++)
              outArray[i]=new double[numOut];
        for (i=0;i<numPats;i++)
              targetArray[i]=new double[numOut];

        NextDigit(infile);

        //Read in
        cout<<"Reading in..."<<endl<<endl;
        for (i=0;i<numPats;i++)
        {
              for(int j=0;j<numIn;j++)
              {
                    infile>>inArray[i][j];
                    NextDigit(infile);
              }
              for(int j=0;j<numOut;j++)
              {
                    infile>>targetArray[i][j];
                    NextDigit(infile);
              }
        }
        cout<<"Finished Reading."<<endl<<endl;
//finished reading in data file

//do split of data file.
        if (split)
        {
              cout<<"Splitting..."<<endl;
              trainPats=split;
              testPats=numPats-split;

              //split them...
              cout<<"  Allocating"<<endl;
              //first allocate array sizes, first dimension
              trainIn=new double*[trainPats];
              trainOut=new double*[trainPats];
              trainTarget=new double*[trainPats];

              testIn=new double*[testPats];
              testOut=new double*[testPats];
```

```
            testTarget=new double*[testPats];


            //allocate pattern lengths and copy patterns
            cout<<"  Copying training input"<<endl;
            for (i=0;i<trainPats;i++)
            {
                  trainIn[i]=new double[numIn];
                  for(int j=0;j<numIn;j++)
                        trainIn[i][j]=inArray[i][j];
            }
            cout<<"  allocating training output"<<endl;
            for (i=0;i<trainPats;i++)
            {
                  trainOut[i]=new double[numOut];
            }
            cout<<"  Copying training target"<<endl;
            for (i=0;i<trainPats;i++)
            {
                  trainTarget[i]=new double[numOut];
                  for(int j=0;j<numOut;j++)
                        trainTarget[i][j]=targetArray[i][j];
            }
            cout<<"  Copying testing input"<<endl;
            for (i=0;i<testPats;i++)
            {
                  testIn[i]=new double[numIn];
                  for(int j=0;j<numIn;j++)
                        testIn[i][j]=inArray[i+trainPats][j];
            }
            cout<<"  allocating testing output"<<endl;
            for (i=0;i<testPats;i++)
            {
                  testOut[i]=new double[numOut];
            }
            cout<<"  Copying testing target"<<endl<<endl;
            for (i=0;i<testPats;i++)
            {
                  testTarget[i]=new double[numOut];
                  for(int j=0;j<numOut;j++)
                        testTarget[i][j]=targetArray[i+trainPats][j];
            }



            cout<<"Split off first "<<split<<" patterns to train,"<<endl;
            cout<<numPats-split<<" to test.\n"<<endl;
            outfile<<"Split off first "<<split<<" patterns to
train,"<<endl;
            outfile<<numPats-split<<" to test.\n"<<endl;
```

```
	}

	else
	{
		cout<<"No split\n"<<endl;
		outfile<<"No split\n"<<endl;
		testIn=trainIn=inArray;
		testOut=trainOut=outArray;
		testTarget=trainTarget=targetArray;
		testPats=trainPats=numPats;


	}


//----End of File stuff-------------------------

	//variables

	int testCorrect, testClassified, testUnclassified;
	double* testFitList= new double[popSize];//array of fitnesses
	double testTotalFitness=0;//total fitness of test set.
	int   nonzeros;

	double roulette;
	double cumTotal;//cumulative total for roulette wheel.

	//Set up the networks
	NetSet(Population,popSize,outfile);
	theGenes=EncodePopulation(Population,popSize);

	genomeSize=new int[popSize];

	cout<<"Starting runs for "<<popSize<<" networks."<<endl;
	cout<<numGens<<" generations."<<endl;
	cout<<epochsPerCycle<<" epochs of training per set."<<endl<<endl;
	outfile<<endl<<endl;
	outfile<<"Starting runs for "<<popSize<<" networks."<<endl;
	outfile<<numGens<<" generations."<<endl;
	outfile<<epochsPerCycle<<" epochs of training per
set."<<endl<<endl;

	cout<<"Distributed Learning"<<endl;
	outfile<<"Distributed Learning"<<endl;

	if (continuance)
	{
		cout<<"Continued learning range: "<<contbot<<" to
"<<conttop<<endl;
		outfile<<"Continued learning range: "<<contbot<<" to
"<<conttop<<endl;
```

```
	}

	clock_t genStart, genEnd, genTime;

	for(eachgen=0;eachgen<numGens;eachgen++)
	{
		genStart=clock();

		cout<<"Generation "<<eachgen<<endl;
		outfile<<"Generation "<<eachgen<<endl;

		testTotalFitness=0;
		nonzeros=0;

		cout<<"Network:\t"<<"\tFitness:\tEpochs:"<<endl;
		outfile<<"Network:\t"<<"\tFitness:\tEpochs:"<<endl;

		for (int whichnet=0;whichnet<popSize;whichnet++)//for each
network
		{
			epochsPerCycle=(500.0-0.0)*(double(rand())/
RAND_MAX)+0.0;

			Population[whichnet]->Learn(epochsPerCycle);


			//Run the member on the training and test data.
			Population[whichnet]->Run(false, testIn, testOut,
testPats);
			//Population[whichnet]->Run(false, trainIn, trainOut,
trainPats);

			//evaluate performance on the test set.
			testCorrect=EvaluateBinary(testUnclassified,.1,
testPats,numOut,testOut, testTarget);
			testClassified=testPats-testUnclassified;
			testTotalFitness+=testFitList[whichnet]=testCorrect/
(double)testPats;


cout<<"\t"<<whichnet<<"\t\t\t"<<testFitList[whichnet]<<"\t";

outfile<<"\t"<<whichnet<<"\t\t\t"<<testFitList[whichnet]<<"\t";
			if(testFitList[whichnet]==0)
			{
				cout<<"\t\t";
				outfile<<"\t\t";
			}
			else
				nonzeros++;
```

```
                cout<<epochsPerCycle<<endl;
                outfile<<epochsPerCycle<<endl;

            }//for each network

            //ok, we've trained each network, and got the total fitness.
Now we...

            //output total fitness:
            cout<<endl<<"Total fitness: "<<"\t"<<testTotalFitness<<endl;
            outfile<<endl<<"Total fitness:
"<<"\t"<<testTotalFitness<<endl;
            cout<<endl<<"Real average: "<<"\t"<<testTotalFitness/
popSize<<endl;
            outfile<<endl<<"Real average: "<<"\t"<<testTotalFitness/
popSize<<endl;
            cout<<endl<<"Miss-0 avg: "<<"\t"<<testTotalFitness/
nonzeros<<endl<<endl;
            outfile<<endl<<"Miss-0 avg: "<<"\t"<<testTotalFitness/
nonzeros<<endl<<endl;


            if (eachgen+1==numGens)
                  break;


            //allocate the new genes

            newGenes= new Genome[popSize];
            continuers=0;
            cout<<"starting continuance"<<endl;
            if(continuance)
            {
                  continuers=0;
                  for(i=0;i<popSize;i++)//for every memmber in the
population
                  {     //generate a continuous random number from
contbot to conttop
                        roulette=(conttop-contbot)*(double(rand())/
RAND_MAX)+contbot;

                        //Now, if our fitness is greater than that number,
continue:
                        if(roulette<testFitList[i])
                        {

newGenes[continuers]=Encode(*Population[i]);
                              genomeSize[continuers++]=Population[i]-
>GetNumLayers();
                        }
                  }
```

```
            }
            cout<<"ending continuance"<<endl;


            //For each member in the population member, spin the roulette
wheel
            //and place that member of the genePool into a new population.

            for(i=continuers;i<popSize;i++)
            {
                  //Now we spin our roulette wheel. Generate a number from
0 to TotalFitness.
                  //note that we have to be careful of the ends of the
range.
                  roulette=(testTotalFitness-0.0)*(double(rand())/
RAND_MAX)+ 0.0;
                  cumTotal=0.0;
                  for(int k=0;k<popSize;k++)//for each net/pop member
                  {

                        if(roulette==testTotalFitness)
                        {

newGenes[i]=CopyGenome(theGenes[k],Population[k]);
                              genomeSize[i]=Population[k]-
>GetNumLayers();
                              break;
                        }

                        cumTotal+=testFitList[k];

                        if(roulette<cumTotal)
                        {

newGenes[i]=CopyGenome(theGenes[k],Population[k]);
                              genomeSize[i]=Population[k]-
>GetNumLayers();
                              break;
                        }
                  }//for k



            }//for i, each new pop member.

            //mate every pair, delete the old networks while we're about
it.
            for(i=0;i<popSize;i+=2)
            {
                  Mate(newGenes[i], newGenes[i+1]);
                  delete Population[i];
```

```
                delete Population[i+1];
            }

            delete[] Population;

            //create new networks
            Population=DecodePopulation(newGenes, popSize, outfile);

            //after cross over etc, delete theGenes before we delete the
old networks.
            DestroyPop(theGenes,genomeSize,popSize);
            theGenes=newGenes;
            newGenes=NULL;

            cout<<"Continued nets:\t"<<continuers<<endl<<endl;
            outfile<<"Continued nets:\t"<<continuers<<endl<<endl;

            outfile<<"Gen "<<eachgen+1<<" nets."<<endl;
            for(i=0;i<popSize;i++)
            {
                  outfile<<"Network "<<i<<endl;
                  Population[i]->PrintStats(outfile);
                  outfile<<endl;
            }

            cout<<endl;
            genEnd=clock();
            genTime=(genEnd-genStart)/((double) CLOCKS_PER_SEC);
            cout << endl << "Generation elapsed run time: "
                   << genTime<<" s, or " <<(double)(genTime/60.0)<<"
min."<< endl;

            outfile << endl << "Generation elapsed run time: "
                        << genTime<<" s, or " <<(double)(genTime/60.0)<<"
min."<< endl;


      }//for each generation

      for(i=0;i<popSize;i++)
      {
            outfile<<"Network "<<i<<endl;
            Population[i]->PrintStats(outfile);
            outfile<<endl;
      }

      TEnd=clock();
      TTime=(TEnd - TStart) / ((double) CLOCKS_PER_SEC);

      cout << endl << "Total elapsed run time: "
             << TTime<<" s, or " <<(double)(TTime/60.0)<<" min."<< endl;
```

```
      outfile << endl << "Total elapsed run time: "
            << TTime<<" s, or " <<(double)(TTime/60.0)<<" min."<< endl;


      outfile.close();
      return 0;
}
//
.....................................................................
.........


//
.....NextDigit.....................................................
.........

inline void NextDigit(ifstream& thestream)
{
      while (!isdigit(thestream.peek()) && thestream.peek()!=EOF)
            thestream.ignore();
}
//
.....................................................................
.........


//
.....EvaluateBinary.....................................................
.........

int EvaluateBinary (int &unclass, double tolerance, int numPats, int
vecSize,
                  double **Output, double **Target)
{
      //returns then number of successful
      //classifications within tolerance

      bool good;//tells whether the current pattern is good or not.

      double up,down;//lower and upper bounds

      int numgood=0;
      unclass=0;  //number unclassified;
      up=1.0-tolerance;
      down=0.0+tolerance;

      for (int i=0;i<numPats;i++)//for every pattern
      {
            good=true;
```

```
            for (int j=0;j<vecSize;j++)//for each element of the current
pattern
            {
                  if (Output[i][j]<up && Output[i][j]>down)
                  {
                        good=false;
                        unclass++;
                        break;
                  }
                  if (Target[i][j]==1.0 && Output[i][j]<down)
                  {
                        good=false;
                        break;
                  }
                  if (Target[i][j]==0.0 && Output[i][j]>up)
                  {
                        good=false;
                        break;
                  }
            }//for each element

            if (good==true)   //if we got here because the pattern was OK,
                  numgood++;        //increment the number of correct
patterns.

      }//for each pattern

      return numgood;
}

//
.....................................................................
.........


//
.....CalcTSS.....................................................
.........

double CalcTSS(int numPats, int vecSize, double **Output, double **Target)
{
      int i,j;
      double tss=0;
      double TSS=0;
      double out;

      for(i=0;i<numPats;i++)//for each pattern
            for(j=0;j<vecSize;j++)
            {
                  out=Output[i][j];
                  tss=Target[i][0]-out;//t-o
```

```
                  //tss*=1-out;           //1-o
                  //tss*=out;             //o
                  tss*=tss;
                  TSS+=tss;
            }
      return TSS;
}
//
.....................................................................
.........


//
.....PrintBanner.....................................................
.........
void PrintBanner(ostream& out)
{
      time_t thetime;

      thetime=time(NULL);

      out<<ctime(&thetime)<<endl;
      //Random Seed:
      out<<"Random Seed: "<<seed<<endl<<endl;


}
//
.....................................................................
.........


//
.....NetSet..........................................................
.........
void NetSet(BPNet** &array, int howmany, ofstream& filerecord)
{

      int HUs[20]={1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10};
            //the number of hidden units in each of the networks. Later,
this
            //can be generated randomly.
      int layersizes[2]={1,2};//This array will be passed to the BP
constructor.
                                                //We'll change the 0th
element every time.
      clock_t clockSeed;
      unsigned int aseed;

      array=new BPNet*[howmany];
```

```
	clockSeed=clock();
	aseed=(unsigned int)clockSeed;

	//aseed=9040343;
	//aseed=7793;
	//aseed=24238324;
	cout<<"Seed in Netset: "<<aseed<<endl<<endl;
	filerecord<<"Seed in Netset: "<<aseed<<endl<<endl;

	for(int i=0;i<howmany;i++)
	{
		filerecord<<"Network "<<i<<endl;
		filerecord<<"Seed  = "<<aseed<<endl;
		srand(aseed++);

		//create the network..
		//bad bad bad. I'm using globals. now I've put them in the
netparms.h header.
		layersizes[0]=HUs[i%20];
		array[i]=new
BPNet(2,layersizes,numIn,numOut,trainPats,trainIn,trainOut,
trainTarget);
		//dump it.
		array[i]->PrintStats(filerecord);
		filerecord<<endl;

	}//for

	filerecord<<"Final Random Seed  = "<<aseed<<endl<<endl;
	cout<<"Final Random Seed  = "<<aseed<<endl<<endl;
	srand(aseed++);
}
```