12-2011

Automating construction and selection of a neural network using stochastic optimization

Jason Lee Hurt
University of Nevada, Las Vegas

AUTOMATING CONSTRUCTION AND SELECTION OF A NEURAL NETWORK USING STOCHASTIC OPTIMIZATION

by

Jason Lee Hurt

Bachelor of Science (B.Sc.)
University of Nevada, Las Vegas
2004

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science Degree in Computer Science

School of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2011

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Jason Lee Hurt

entitled

Automating Construction and Selection of a Neural Network Using Stochastic Optimization

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Howard R. Hughes College of Engineering

Matt Pedersen, Committee Chair

Kazem Taghva, Committee Member

John Minor, Committee Member

Aly Said, Graduate College Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

December 2011

Abstract

An artificial neural network can be used to solve various statistical problems by approximating a function that provides a mapping from input to output data. No universal method exists for architecting an optimal neural network. Training one with a low error rate is often a manual process requiring the programmer to have specialized knowledge of the domain for the problem at hand.

A distributed architecture is proposed and implemented for generating a neural network capable of solving a particular problem without specialized knowledge of the problem domain. The only knowledge the application needs is a training set that the network will be trained with. The application uses a master-slave architecture to generate and select a neural network capable of solving a given problem.

Acknowledgements

I would like to thank my mom for showing me by example the value of higher education and for her help in editing this paper. I would like to thank my sister for getting a doctoral degree at the age of twenty-five which helped motivate me to finish my thesis work. I would like to thank Dr. Pedersen for helping me to research, write and publish before my thesis and for the time he spent reading many revisions of my writing.

Jason Lee Hurt

University of Nevada, Las Vegas
December 2011

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Certain problems lend themselves toward solutions that are probabilistic in nature and not always deterministic. These include facial recognition, voice recognition, image recognition, and clustering. An artificial neural network "ANN" or "NN" is one way to solve problems of this nature. From a high level, a neural network is a black box, that given an input pattern, produces an output pattern that is the network's best guess to be correct. It essentially estimates a function where the domain is the set of all input patterns and the range is the set of all output patterns. Typical neural network implementations train a neural network for a specific task, one at a time, sequentially. Preprocessing of the data for selecting an appropriate feature vector is tuned, as is selecting appropriate training and test data sets that give low error rates. The tuning of the structure of the neural network itself is often ignored. This research focuses on tuning those parameters automatically to find a neural network that performs well for a given training set and feature selection, as opposed to the more typical approach of focusing on feature extraction and representative training data.

NNGenerator is the software used to train multiple networks at a time and combine the results to generate a network that will attempt to adapt to the training set. This software is an attempt to mitigate the problem of choosing a neural network structure by using an iterative search algorithm similar to a genetic algorithm to find an optimal structure of a neural network for the features of the training set.

A problem is described to the software as a training set, which is simply a map of inputs (a vector of features) to actual output, which is a member of the solution space. Three problems are described in this thesis that were used to test the effectiveness of tuning the structure of a neural network with little focus on preprocessing to obtain feautures of the input space. The first is the problem of recognizing the *XOR* pattern commonly found in digital circuits. This is a simple example of a non-linear separable pattern. This problem was also used when constructing the *NNGenerator*, to test the functionality of various aspects of the software. The second problem is learning to play a common casino card game, blackjack. This problem is an example of training agents in games based on past events. The agent can learn to get better at a specific task or tasks, and can perform better than a hard-coded heuristic that does not adapt to events in the game. The third problem is to recognize a test set of handwritten characters based on features extracted from binary images of a disparate training set of handwritten characters. This problem is an example of pattern recognition that neural networks are often used in.

With every problem set, we hope to generate a neural network whose structure is demonstratively better than other structures trained with the same data. The goal is that as the search progresses, a progressively lower error rate than other structures that were trained for the same amount of iterations with the same training set will be observed.

Chapter 2

Neural Networks

2.1 Biological Neural Networks

The human nervous system consists in part of many neural cells that are interconnected and can communicate with each other via electrical pulses. These cells are called *neurons*, and they receive input from other neurons via *dendrites*, and they can send a signal to a single other neuron via an *axon*. A neuron can also receive signals in reaction to outside stimuli. Figure 2.1 is an illustration of a biological neuron.

A network of neurons stores information at the contact points between the neurons. These contact points are called *synapses* and provide a biological neural network with a memory. Figure 2.2 shows the synapses of a connection between two neurons.

Information is stored at the synapses, and the information for a particular neuron synapse can be viewed as a function f of the sum of the weighted edges that are input to the neuron. The edges in this case are the one or more axons connected to the neuron, and the weighted values correspond to the efficiency of the particular edge. When a neuron receives a signal from a group of axons, it will determine whether or not to send an output signal by applying f to the sum of the weighted value associated to each input axon. If the neuron does send a signal, this is known as *firing* or *excitation*. The efficiency of an edge can be changed through various biological mechanisms. The changing of the efficiencies of the edges over time can be viewed as learning. This oversimplification of biological neural networks is sufficient to draw parallels to the theory for artificial neural networks.

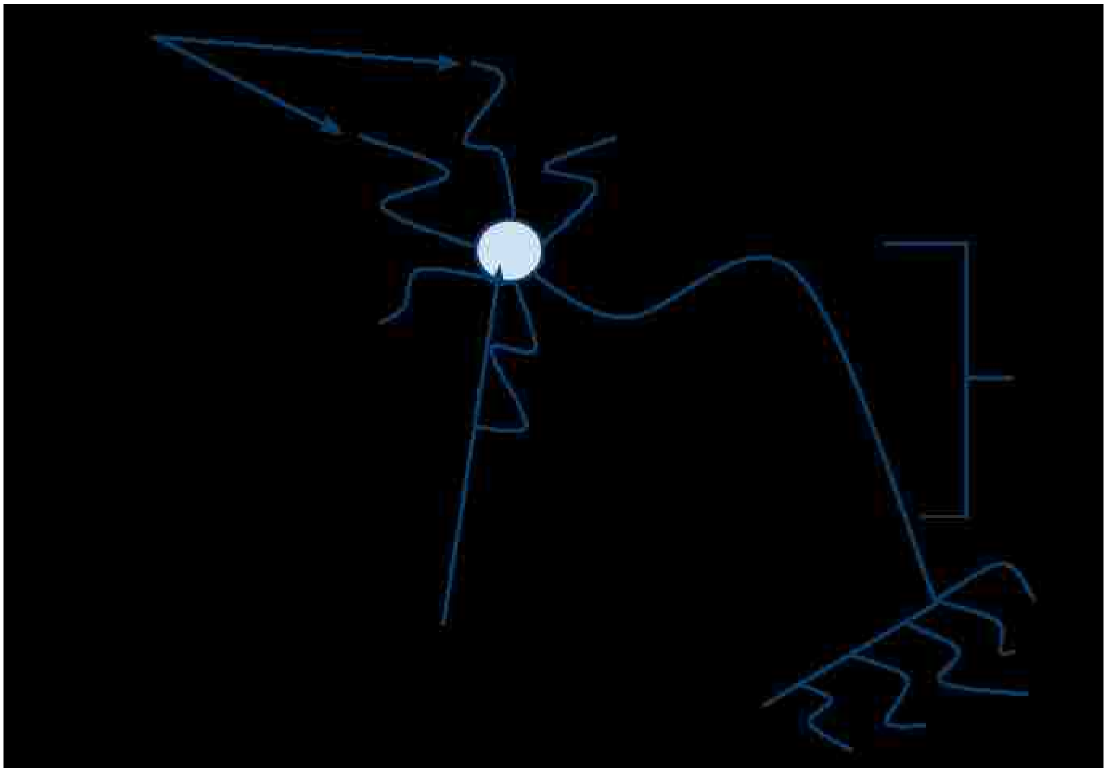


Figure 2.1: Biological Neuron.

2.2 Artifical Neural Networks

An artifical neural network or *ANN* is a way of processing information that is loosely based on the way the nervous system in a human brain functions. Neural networks have been used as solutions in pattern recognition problems such as optical character recognition (OCR) [AIDG95, Rog94], facial recognition [LGTB97], and decision making problems [Bax90, HMOR94]. The purpose of an artificial neural network is to provide a mapping from a set of input data to output data. In mathematical terms the goal is to map an n-dimensional real input $(x_1, x_2, \ldots, x_n)$ to an m-dimensional real output $(y_1, y_2, ..., y_m)$, approximating a function [Roj96, 29-30]:

$$F : R^n \to R^m$$

A neural network builds this mapping in an iterative fashion from training data consisting of known input/output pairs that are presented to it. The training inputs are a subset of the total possible inputs to the network, and assuming there is a function of input to output data, a neural network can be viewed as a black box that may approximate that function for us. Henceforth, the term neural network will be used for brevity, with the implicit assumption being that an artificial neural network is meant unless otherwise stated.

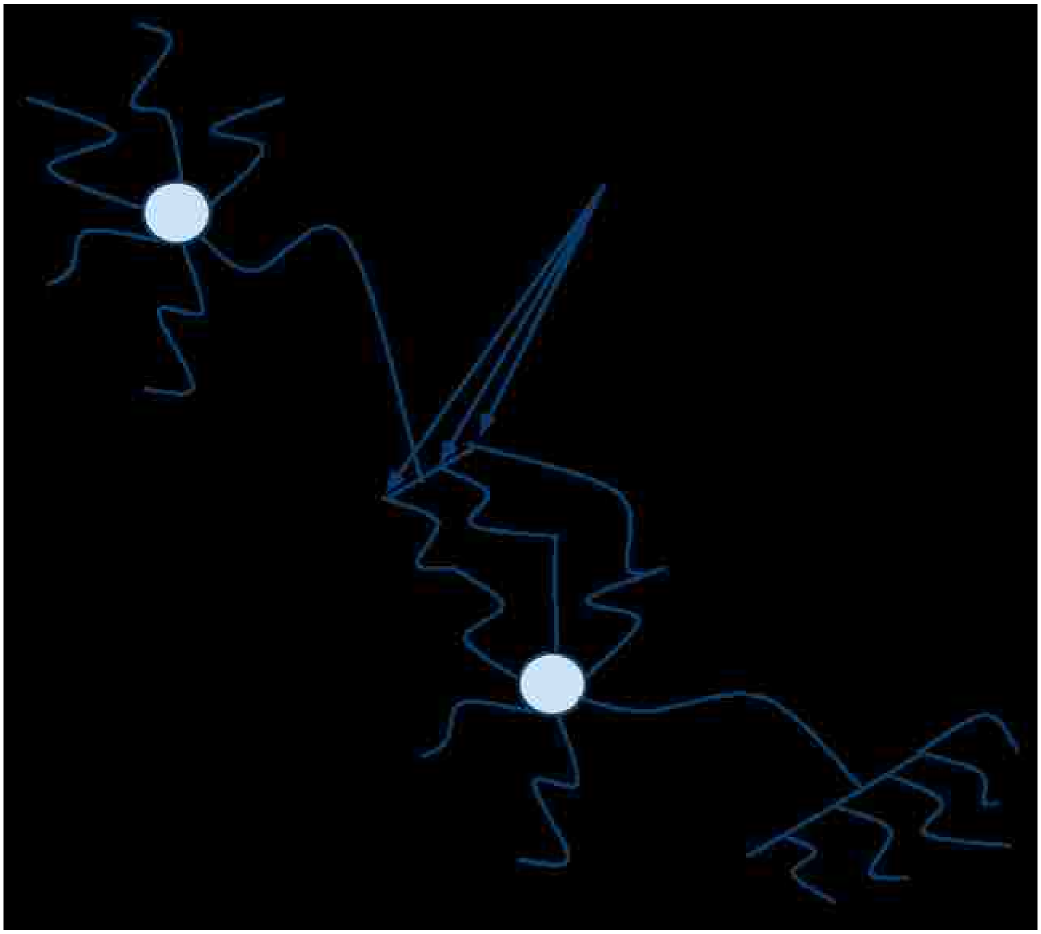


Figure 2.2: Synapses.

As with biological neural networks, the basic building blocks of artificial neural networks are neurons. Each neuron is connected to one or more others neurons in the network. Each neuron outputs a single value based on the evaluation of a function f of the sum of the values of the inputs to the neuron. This function is known as the activation function.

The activation function can be viewed as a composition of functions g and h. g is an aggregate *n-ary* function that takes n arguments and outputs a single value. Typically g is simply the summation function. h takes the output of g and produces a value that is the output of the neuron.

The output value of a neuron with unweighted edges is a function of its inputs, $f(x_1, x_2, \ldots, x_n)$ where x_i is the i^{th} of n inputs as shown in Figure 2.3.

The output value of a neuron with weighted edges is a function of its inputs and weight values of its input edges, $f(x_1 * w_1, x_2 * w_2, ..., x_n * w_n)$, where x_i is the i^{th} of n inputs and w_i is the i_{th} of the corresponding n input edges as shown in Figure 2.4.

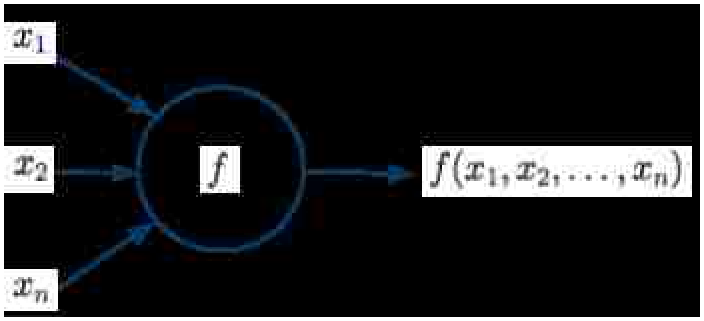


Figure 2.3: Artificial neuron.

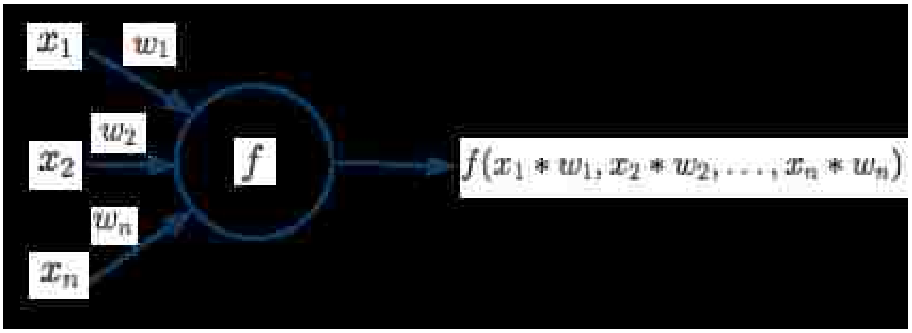


Figure 2.4: Weighted neuron.

Here we only consider neural networks with weighted edges. In general they are more computationally powerful than neural networks with unweighted edges. The weighted edges connecting the neurons are simply called weights, and the values associated with these weights can be similarly be adapted as they are with biological neural networks. In this way a neural network can learn how to process the information presented to it based on past experience. Unlike biological neural networks, we impose a restriction on the topology of the connections. A network is broken into layers, with one or more neurons belonging to a particular layer. The restriction depends on the type of neural network.

2.2.1 Perceptron

A simple example of a neural network is one with a single neuron known as a perceptron. The perceptron takes a vector of values as inputs and outputs a single value, 0 or 1:

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x > 0 \\ 0 & \text{else} \end{cases}$$

A perceptron is trained to learn by adjusting its weights after an input/output pair is presented to it.

Algorithm 2.1 describes an algorithm for perceptron training.

Algorithm 2.1 Perceptron training algorithm.

```
while all input/output pairs have not been classified correctly do
  Select a pair of training data, input $x_n$ and known output $y_n$
  Present the vector to the perceptron and compute $z = f(x_1 * w_1, x_2 * w_2, ..., x_n * w_n)$
  if $z <= 0$ and $y_n > 0$ then
    $w_{i+1} = w_i + x_n$
  else if $z > 0$ and $y_n <= 0$ then
    $w_{i+1} = w_i - x_n$
  else
    no adjustment
  end if
end while
```

Although the perceptron can be trained to classify patterns, there are sets of data that can be separated into two classes that a perceptron is not capable of learning to classify. A perceptron can only classify data sets that are linearly separable. The logic operators *AND*, *OR* and *XOR* provide a simple way of visualizing linear separability. Consider the *AND* operator with the truth table in Figure 2.1.

Figure 2.6 shows that the discrete input space of the *AND* operator can indeed be separated by a single straight line.

Now consider the *OR* operator with the truth table in 2.1.

Table 2.1: *OR* and *AND* truth table.

X	Y	$X \vee Y$	$X \wedge Y$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Figure 2.5 shows the separation of input space for the *OR* operator.

Now consider the *XOR* operator with the truth table in Figure 2.2.

Table 2.2: *XOR* truth table.

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

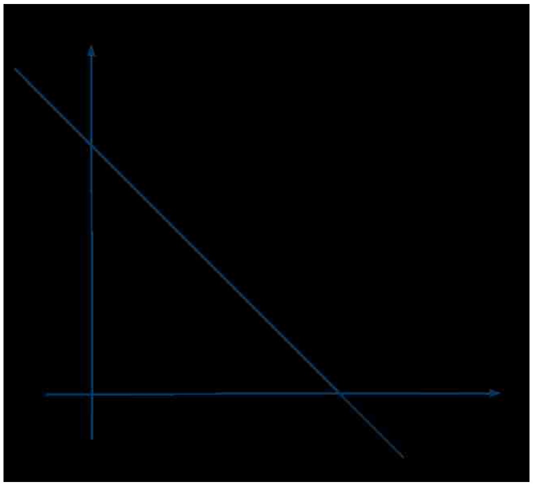


Figure 2.5: *OR* input space separation.

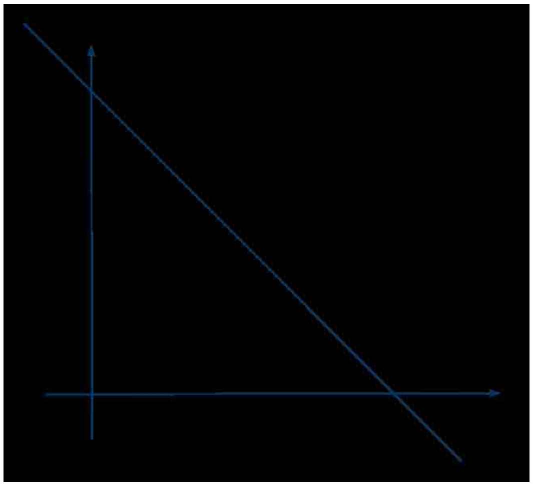


Figure 2.6: *AND* input space separation.

In Figure 2.7 notice how there is no way to draw a straight line to distinguish between the positive and negative classes; at least two lines must be used to separate the positive and negative classes.

If a perceptron is given a problem whose input space is linearly separable, it is guaranteed to find a solution in a finite number of steps [TK09]. In the worst case the number of iteration steps can grow exponentially [Roj96, 95-96]. There are better algorithms for finding solutions to linearly separable problems in polynomial time such as Karmarkar's polynomial time algorithm [Kar84].

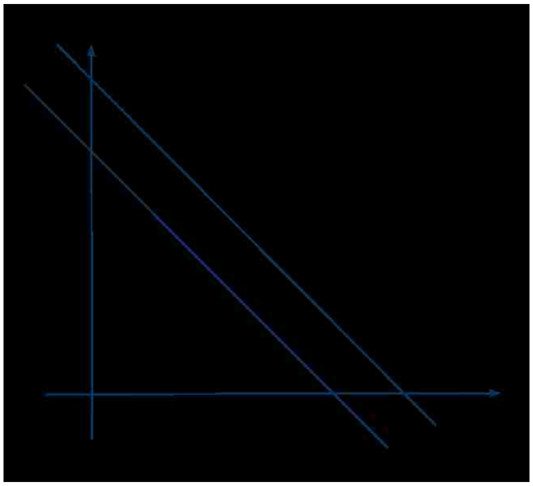


Figure 2.7: *XOR* input space separation.

2.2.2 Feed Forward Neural Networks

One way to solve a problem with a non-linearly separable input space is with a feed forward neural network. With feed forward neural networks, sets of neurons are grouped into layers and information flows from the input layer to output layer in one direction only. A neuron in layer n may only send signals to neurons in layer $n + 1$, and it may only receive signals from neurons in layer $n - 1$. The 0^{th} layer does not receive signals from other neurons and the output of the n^{th} layer can be viewed as the output of the network. Figure 2.8 shows an example of this topology.

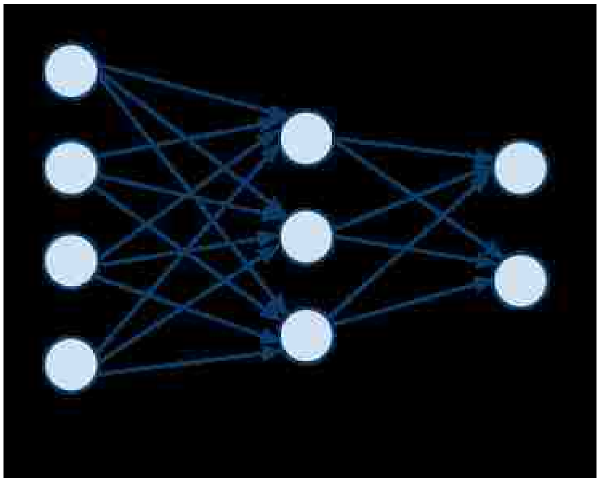


Figure 2.8: Feed forward network topology.

Notice that the edges are directed and indicate that the flow of information is from left to right and that there are no cycles. A network is feed forward if the connections between its neurons do not form a cycle.

Revisiting the *XOR* problem we want to label the input spaces and create a mapping of each input vector to a single label. In Figure 2.9 we label the three regions A, B and C.

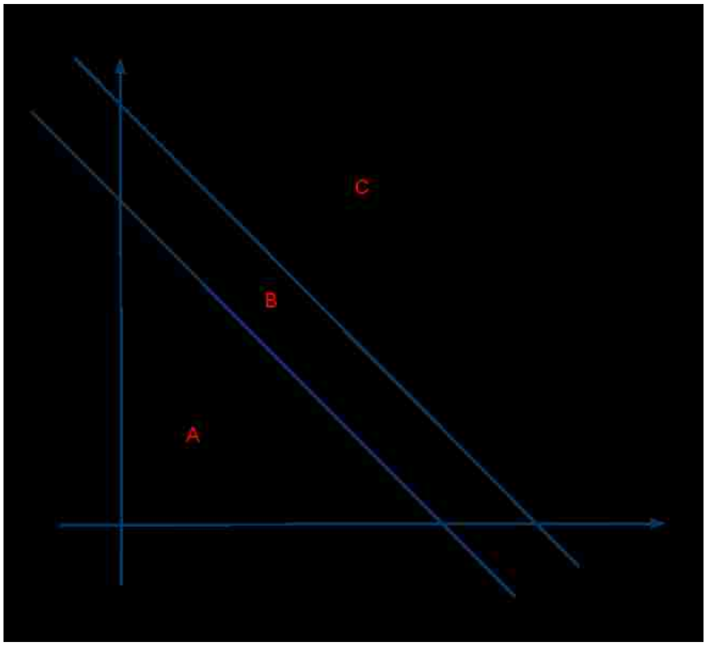


Figure 2.9: Labeled three regions of the *XOR* input space.

And we also want the network to remember which input vector x gets mapped to a particular label, as seen in Figure 2.3.

Table 2.3: Mapping of *XOR* input space to label.

x_1	x_2	*label*
0	0	A
0	1	B
1	0	B
1	1	C

The *XOR* problem can be solved with a feed forward neural network with one hidden layer. The neural network has three layers, the input layer, one middle layer, and the output layer. Figure 2.10 shows the structure of the network.

For the neuron function, $f(x) = x * \frac{1}{2}$ is used. The goal is to find a set of weights that satisfies the following equations for the four known inputs and outputs in the *XOR* truth table:

$$f^1(x) = (x_1 * w^1_{11} + x_2 * w^1_{21}) * \frac{1}{2}$$

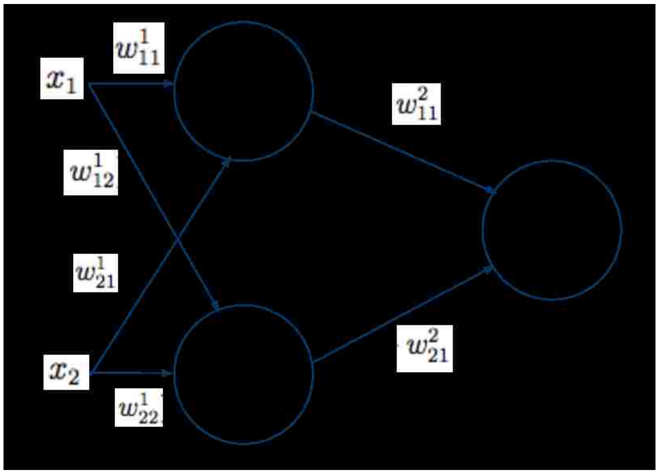


Figure 2.10: Weighted feed forward neural network architecture that is capable of solving the *XOR* problem.

$$f^2(x) = (x_1 * w^1_{12} + x_2 * w^1_{22}) * \frac{1}{2}$$

$$f(x) = (f^1(x) * w^2_{11} + f^2(x) * w^2_{21}) * \frac{1}{2}$$

Figure 2.11 shows a set of weights that solves the problem.

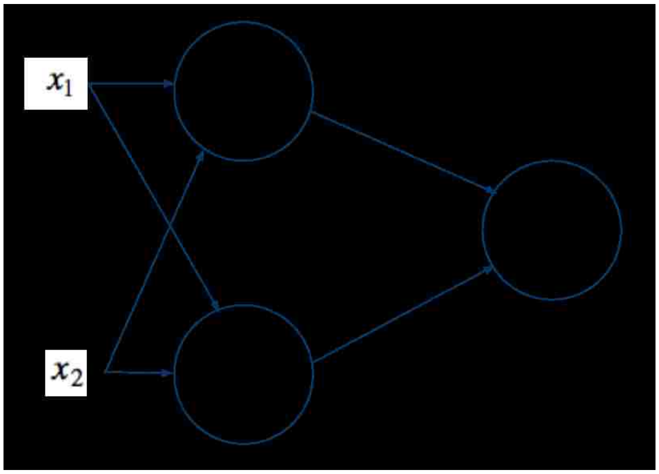


Figure 2.11: Weighted feed forward neural network that solves the *XOR* problem.

Feed forward networks provide a powerful computational model capable of correctly mapping a set of input vectors to a specific output, even for non-linearly separable input spaces. However, finding a set of weights that solves a problem becomes increasingly expensive as the arity of the feature vector expands, the number of nodes increases, and/or the number of layers increases. In addition, traditional methods of solving systems of equations do not work well with noisy data and missing or incomplete data, as is the case in most real world problems. To solve these issues, the backprogation algorithm can be used.

2.2.3 Backpropagation

The backpropagation algorithm is the most popular algorithm for finding a set of weights for a neural network to solve a particular problem. It uses a numerical method known as *gradient descent* to search for the minimum of the error function in weight space. Gradient descent, also called steepest descent, attempts to find the local minimum of a function. It starts at one point P_0 and moves from P_i and P_{i+1} by minimizing the line extending from P_i in the direction of $-\nabla f(P_i)$ [Wei].

For gradient descent to work properly, the function produced by the neural network must be differentiable at each point and continous. Since the value of a network can be seen as a composition of the functions of its neurons, typically the activation function is sigmoidal and it also known as a sqaushing function, since it has the ability to squashes all output between two real values [Mit97]. The most commonly used sigmoidal function in neural networks is the logisitic function, defined as: $P(t) = \frac{1}{1+e^{-t}}$ Figure 2.12 shows the graph of the logistic function on a cartesian coordinate system.

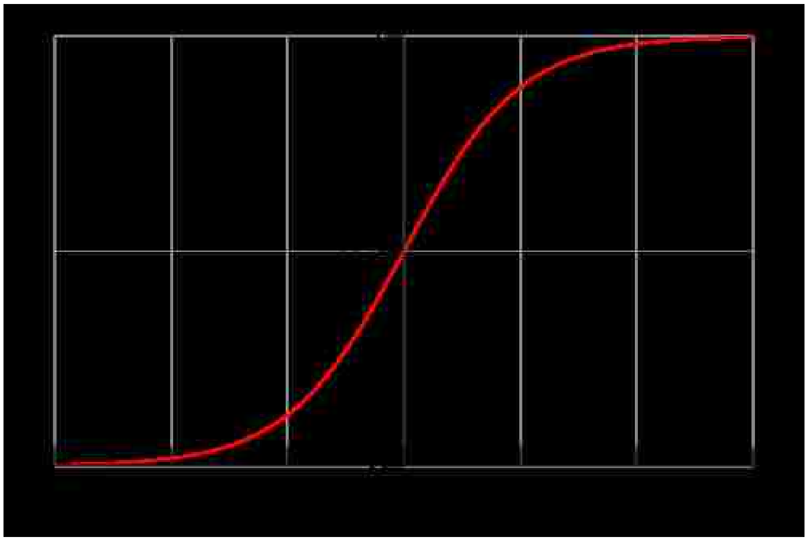


Figure 2.12: Logistic function.

The *NNGenerator* software uses a modifed version of the backpropagation algorithm described in [Roj96, 166-170]. An outline of the algorithm is given in Algorithm2.2. The main difference is that it keeps track of the error rate for every input presented to it. It does so by keeping a map of each input to the current error rate, which defaults to one-hundred percent for inputs that have not yet been presented to the neural network. Here the effectiveness of a neural network is found by calculating its error rate for all inputs it has been trained with. A low error rate means the network is good at estimating the problem/solution function. Finding a neural network with a low error rate for a particular problem presents a number of challenges:

1. Choosing which features from the data to use to form inputs of a training set.

2. How to represent the features of a data set.

3. Deciding how many hidden layers the neural network will have.

4. Deciding how many nodes in each hidden layer should be used.

5. Deciding the activation function to use for each neuron.

The combination of number of hidden layers, number of nodes per hidden layer, and activation function of each neuron determines the network structure. With the *NNGenerator* software, the structure of a neural network, including the type of activation functions at each hidden node, the number of nodes per hidden layer, the number of hidden layers, the constant that defines step length correction at each step, and the constant that defines the momentum factor for help in preventing oscillation during learning, may all be bounded by the user. The software will work within those bounds over a large search space to find a neural network structure and accompanying weights that outperformed all others in the search.

Algorithm 2.2 Backpropagation training algorithm.

```
while error rate is sufficiently small or training data has been exhausted do
    Select a pair of training data, input $o^n$ and known output $t_n$
    Present the vector to the neural network and compute the output vector $o_m$ for the output nodes in a feed forward fashion.
    Calculate the backpropagated error for the output layer:
        $\delta_j = o_j(1 - o_j)(o_j - t_j)$
    Calculate the first set of partial derivatives for the error:
        $\frac{\partial E}{\partial w_{ij}} = [o_j(1 - o_j)(o_j - t_j)]o_i = \delta_j o_i$
    In a similar manner, calculate the backpropagated errors and partial derivatives for the hidden layers, starting with the $j^{th}$ layer connected to the output layer, and continuing to the $j - 1^{th}$ layer until all layers have been exhausted.
    Use the partial deriviates to update the weights of the neural network in the negative gradient direction where $\gamma$ is a constant that defines the step length of correction:
        $\delta w_{ij} = -\gamma o_i \delta_j$
end while
```

Chapter 3

Genetic Algorithms

Genetic algorithms are a form of stochastic optimization [HS03] commonly used in learning problems [DPAM02, MSV93, DH95]. They are search algorithms that search over large spaces for generally good solutions to problems. The search space for these problems is typically large enough to be computationally inefficient to compute through enumeration methods. They can also be used when a function is discontinuous and cannot be solved with methods requiring the derivative at points in the space to be found, such as with neural networks.

3.1 Encoding

Genetic algorithms start with a random sample of possible encoded representations for a problem. The encoding is typically a binary string of a fixed length, though it can also be represented using more complex data structures. These encodings are called *chromosomes* and are analogous to biological chromosomes that are found in cells.

3.2 Operations

A new generation of samples is created by applying simple operations to the current generation of samples. Three of the commonly used operations are crossover, reproduction, and mutation [Gol89, 62-65]. The ideas behind these operations are similar to natural selection mechanisms that occur during evolution, hence the name genetic algorithm.

3.2.1 Crossover

Crossover combines two samples in the population at some randomly selected index called the cross site. For example, the string 00110 crossed with the string 10001 at index 2 will produce the strings 10110 and 00001 as shown in Figure 3.1.

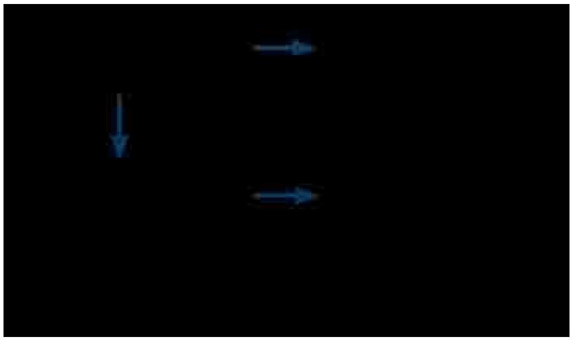


Figure 3.1: Crossover on two binary strings.

The crossover in the *NNGenerator* software crosses over two neural network structures in a similar manner, it starts by choosing a random cross site to split the structures at. It proceeds by creating two new children C and D, one with the first part structure A crossed with the second part of structure B, and a second with the first part of structure B crossed with the second part of structure A, as shown in Figure 3.2.

3.2.2 Reproduction

Reproduction will copy the sample over to the new population based on some probability depending on the sample's fitness. In this way the fittest samples have the best chance of survival as in nature with natural selection.

3.2.3 Mutation

During mutation there is a low probability that a part of a sample in the population will be changed in some small way. If a binary encoded string 0100 were selected for example it may become 0101 as shown in Figure 3.3.

The purpose of the mutation operator is to help the large search to not get stuck in a local minima. In the *NNGenerator* software, mutating the data structures was not straightforward to implement. To help introduce a similar random nature into the search, the software generates a random neural network structure for two to five percent of each new population. These are generated within the

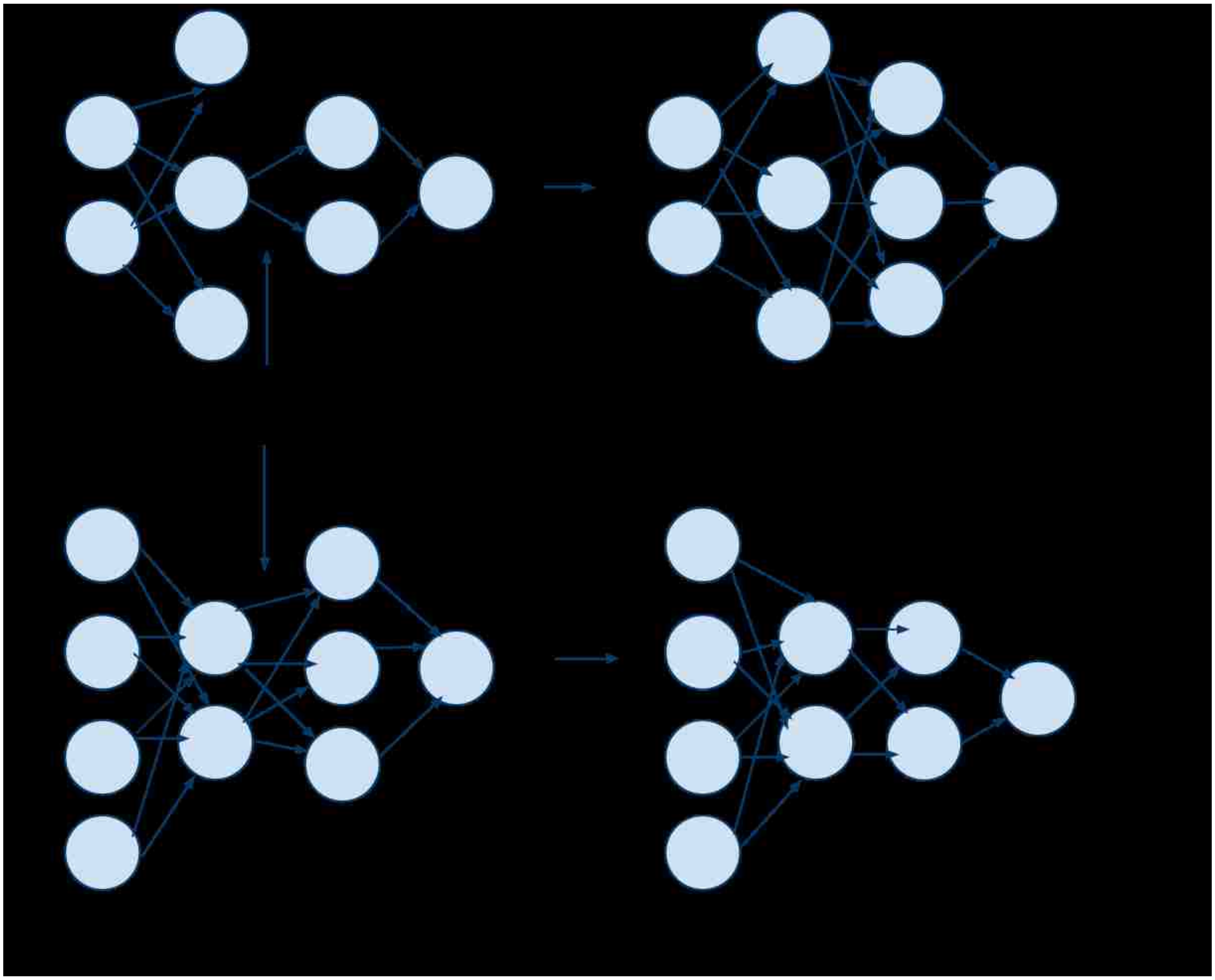


Figure 3.2: Crossover of neural network structures

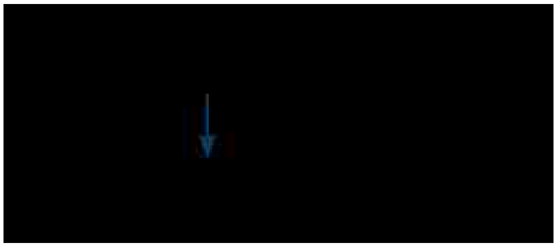


Figure 3.3: Mutation on a binary string.

same bounds as the initial population, so the structures respect the user's specified bounds on the number of nodes per layer and maximum number of layers.

3.3 Fitness Function

At each iteration of a genetic algorithm, a function that evaluates the fitness of each sample is applied to every sample. In the *NNGenerator* software, the root mean squared or *rms* training error of a neural network is the only parameter used to judge the fitness of each neural network. The fittest samples have the best chance of being considered for the crossover, reproduction, and mutation operations. A simple way to select the fittest samples is to order them in descending fashion from highest fitness to lowest fitness, then remove a number of the least fit from the population.

3.3.1 Roulette Wheel Selection

In the *NNGenerator* software, each chromosome has a probability of surviving proportional to its fitness compared to the others in the same generation. So the least fit chromosomes have the least chance of reproduction and crossover when forming the next population of samples. This algorithm is known as roulette wheel selection [Gol89, 237]. Each item along the roulette wheel is a separate chromosome and the probability of selecting a chromosome is directly proportional to its fitness.

For example, given eight samples labeled: A, B, C, D, E, F, G, H, and respective fitness proportional to the entire population's fitness: $5, 23, 10, 10, 5, 37, 5, 5$, then the sample F will have the best chance of reproducing itself and crossing over with another sample at 37%, followed by sample B with a chance of 23%, and so on. Figure 3.4 shows a pie chart representing the described probability distribution.

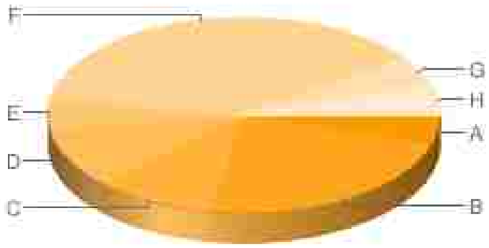


Figure 3.4: Roulette wheel selection example.

Chapter 4

Architecture

4.1 Master/Slave Pattern

From a high level, the *NNGenerator* software is modeled on the popular master/slave pattern. This is a pattern that is meant to be used when breaking up a particular task into many smaller tasks and doing the work in parallel. A single master is responsible for delegating the work to the slaves and collecting and combining the results into a solution. These smaller tasks may not necessarily work at the same rate. This can be due either to external issues outside of the programs control, such as high CPU load, or to the nature of the data or problem at hand.

4.1.1 Computing the Mandelbrot Set

An example of this pattern in action is as a solution to calculating the Mandelbrot set. The Mandelbrot set consists of a set of points in the complex plane. The boundary of the set forms a fractal. Take the set of functions $f(x) = x^2 + C$ where C is a constant complex number and $x \in \mathbb{R}$ For a particular complex number C, C is a member of the Mandelbrot set if the recurrence relation generated by repeatedly applying f to increasingly large values of x, starting with $x = 0$ diverges to infinity. A two-dimensional binary image of the Mandelbrot set can be generated in order to visualize the set using the real and imaginary parts of a set of complex points on a bounded two-dimensional plane. Black pixels will represent complex numbers that are members of the set and white pixlels will represent complex numbers that are not members of the set. Since we have to evaluate a recurrence for each pixel, the runtime of the generation of an image is $O(rNM)$ where r is a constant representing number of recurrence patterns that are evaluated, N is the width of the image and M is the height of the image. This algorithm is CPU intensive and grows polynomially as the size of the image grows. An interesting property of generating this image is that each pixel i, j of the image can be calculated *independently* of the other pixels. This is because each complex number C can be tested for membership of the Mandelbrot set regardless of all other complex numbers membership status.

Such a problem is called *embarassingly parallel* and is a prime candidate for using a master/slave architecture in a shared or distributed memory setting for speedup. The master will split the image up into smaller images, and it will give each slave a small image region to calculate. The slaves will calculate these smaller images in parallel and send the results back to the master. When the master has received all of the smaller images, it will assemble them into the resultant image.

Consider a distributed environment with five nodes, one master node and four slave nodes, and an image of size 256 x 256. The algorithm begins with the master assigning the slaves a region of the image to calculate. Each slave will get assigned one of 4 regions of size 128 x 128 as shown in Figure 4.1.

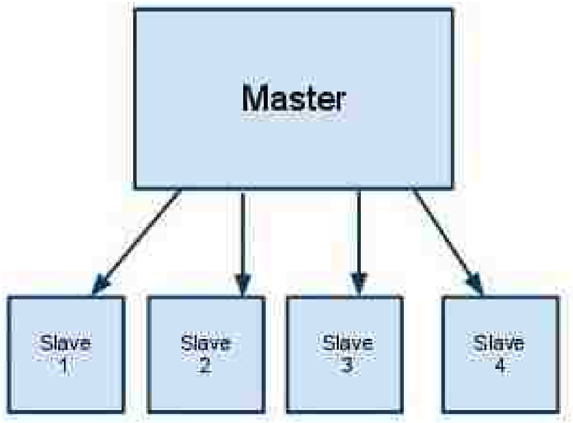


Figure 4.1: Distribution of image regions from master to slave.

Each slave calculates its region of the image and sends the resultant sub-image to the master node as shown in Figure 4.2.

The master then assembles the resultant sub-images into the resultant image as shown in Figure 4.3.

4.1.2 Training Neural Networks

The *NNGenerator* consists of two modules; one of these is the master and runs on a single computer with a display and keyboard so that it can accept user input. The second of these is the slave module which is run multiple times depending on the environment. The master and slaves communicate with each other via network sockets, and so the slaves can run in a hybrid distributed environment

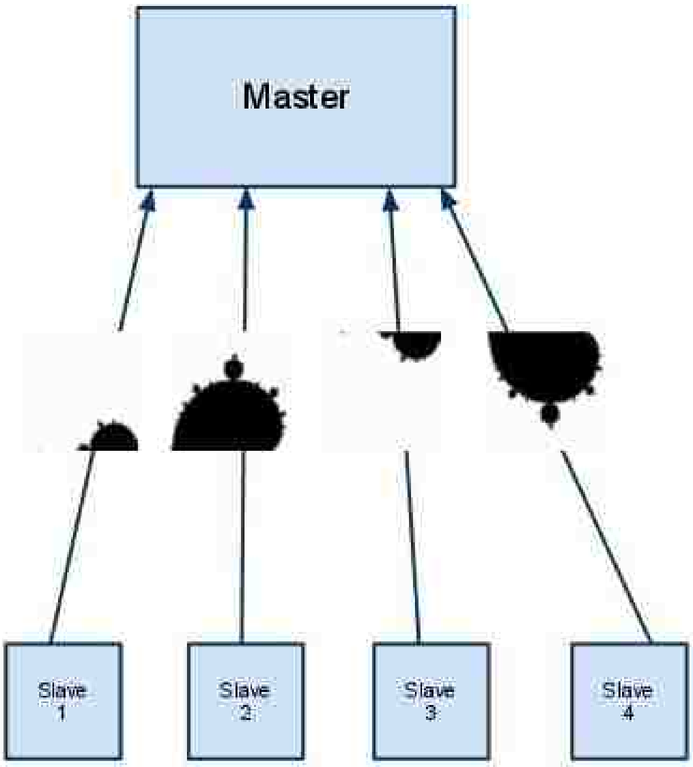


Figure 4.2: Slaves sending fragmented pieces of the complete image region to the master.

as one or more instances on a single box with a network connection. The slaves do not communicate with each other and are not aware of each other's presence. In fact, there does not have to be more than one slave for the software to function. In the case of only a single slave, the inherent parallelism of the algorithm is not exploited but a solution will still be found, albeit rather slowly.

Each slave is a trainer of neural networks and the master, as the implementor of the genetic algorithm, is responsible for overseeing and coordinating the search. The master starts by generating a number of random neural network structures based on the inputs given by the user. A typical structure may look like Figure 4.4.

This is an example randomly generated structure by the software. It can be used for a data set with two inputs and one output. There is one hidden layer with six nodes, the last node of a hidden layer is always a bias node whose value is always equal to 1.0. There are three input nodes because the last input node is always a bias node. In general, for an $N \times M$ data set, there will be $N + 1$ input nodes and M output nodes. The upper bound on the number of generated layers is four in this case, and the upper bound for the number of nodes per layer, except for the input and output layers which are fixed, is five. Notice the connectivity of the structure, each node in layer N

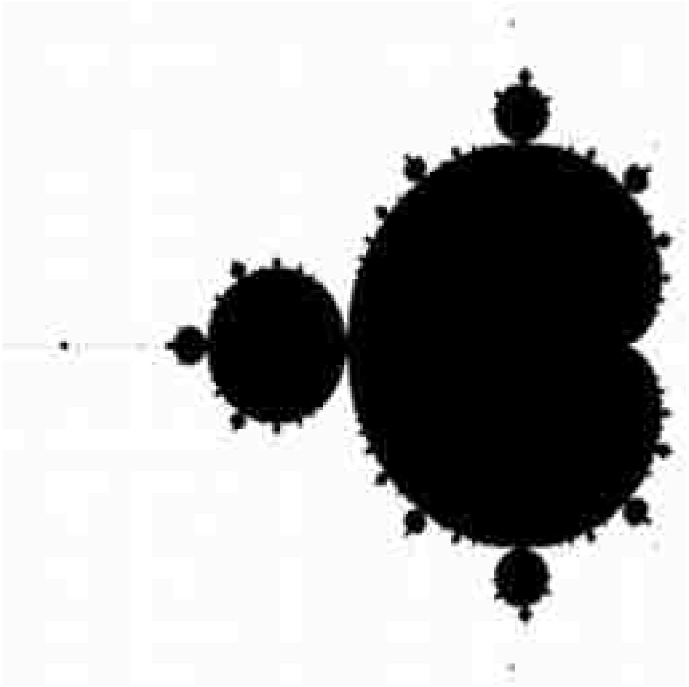


Figure 4.3: Master assembles resultant image.

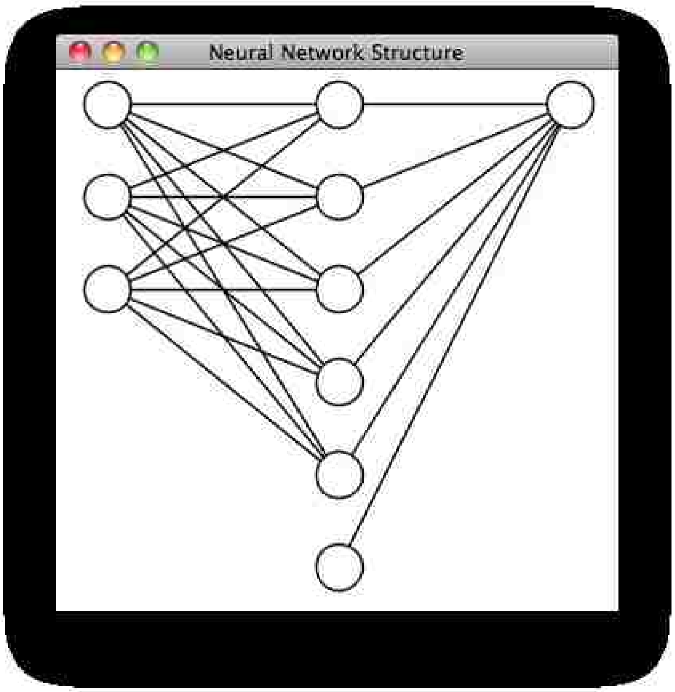


Figure 4.4: Structure of a neural network in *NNGenerator*.

is connected to every other node in layer $N + 1$.

The slave nodes understand how to train these structures against various data sets. After generating a set of random structures constituting the first population, the master then puts these structures into messages that also contains a type field that indicates to a slave which data set to train the structure with as shown in Figure 4.5.

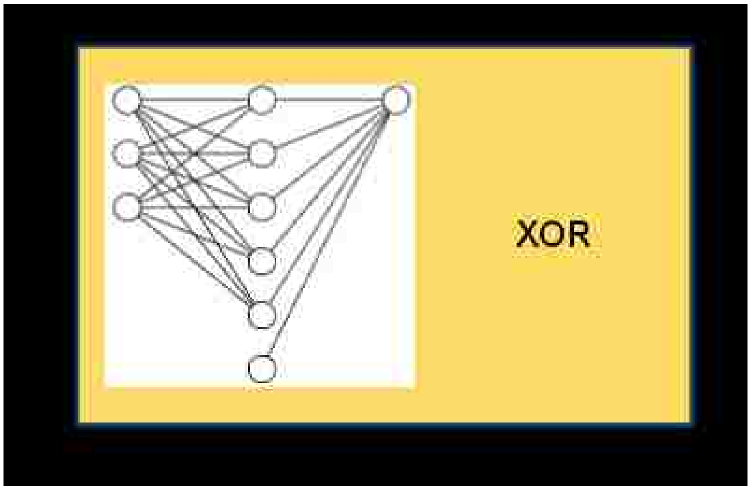


Figure 4.5: Sample XOR training message.

It pushes each of these structures to a message queue that holds training messages that have not yet been processed by a slave as shown in figure 4.6.

The slaves on the network consume messages from the queue in first in, first out fashion as shown in Figure 4.7.

Each slave trains the neural network structure it consumed from the queue starting with a random set of initial weights and a training data set common across all slaves. Upon finishing training, each slave has computed a weight matrix for the neural network structure and data set. The final combined structure may look like Figure 4.8.

The numbers along each weight indicate the weight value of each input node to its corresponding output node.

Each member of the population are encoded as Clojure data structures called struct maps. These struct maps contain information about the structure and fitness of a trained neural network. The specific information is the number of hidden layers, the activation function and derivative of the activation function at each hidden node, the number of nodes at each layer, the connectivity of the nodes, the α and γ constants of the network, and the training error of the network. The α constant is the momentum factor for help in preventing oscillation during learning. The γ constant is a learning constant that defines step length of correction at each step of the training. The slave places the

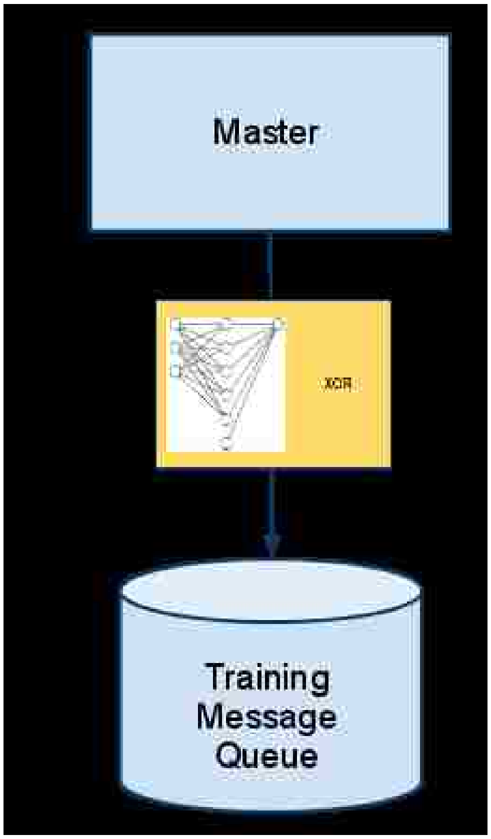


Figure 4.6: Training message queue.

resultant neural network structure along with the other information into a training result message as shown in Figure 4.9.

Afterwards, the slave enqueues the message to a resultant message queue for the master to process as shown in Figure 4.10.

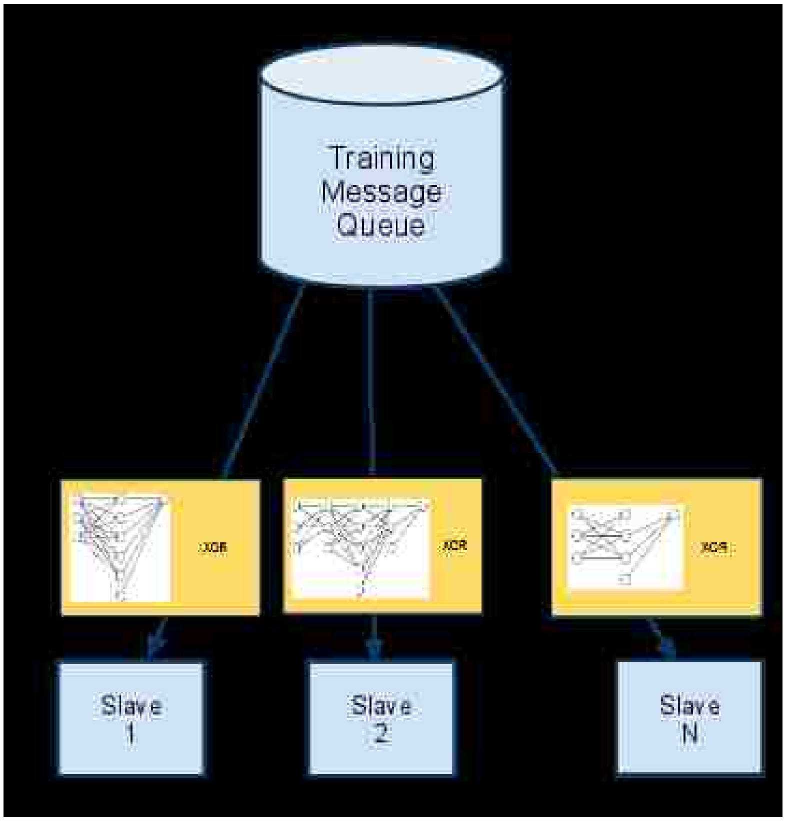


Figure 4.7: Slaves consuming training messages.

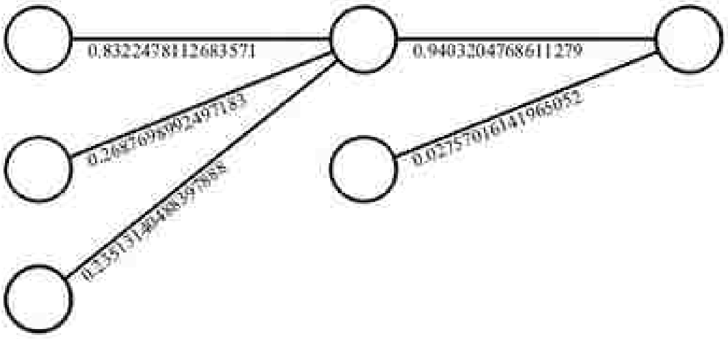


Figure 4.8: Neural network generated by *NNGenerator* that solves the XOR problem.

The master consumes these messages from the queue in a first in, first out fashion as shown in Figure 4.11.

Messages are stored in memory until the master has gathered all of the current populations' resultant messages. The size of each population is specified by the user before the algorithm starts. The master then performs reproduction, crossover, and mutation operations to the network structures to

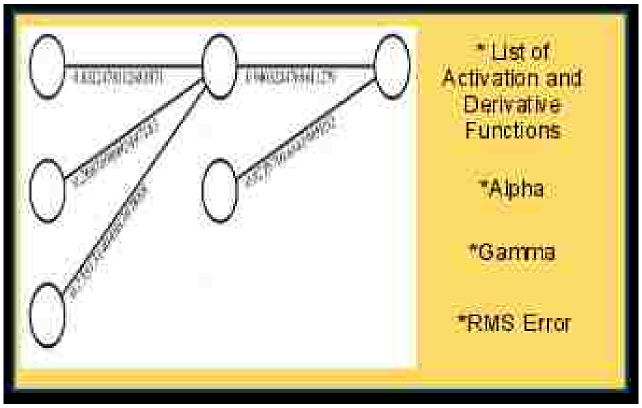


Figure 4.9: Sample *NNGenerator* training result message.

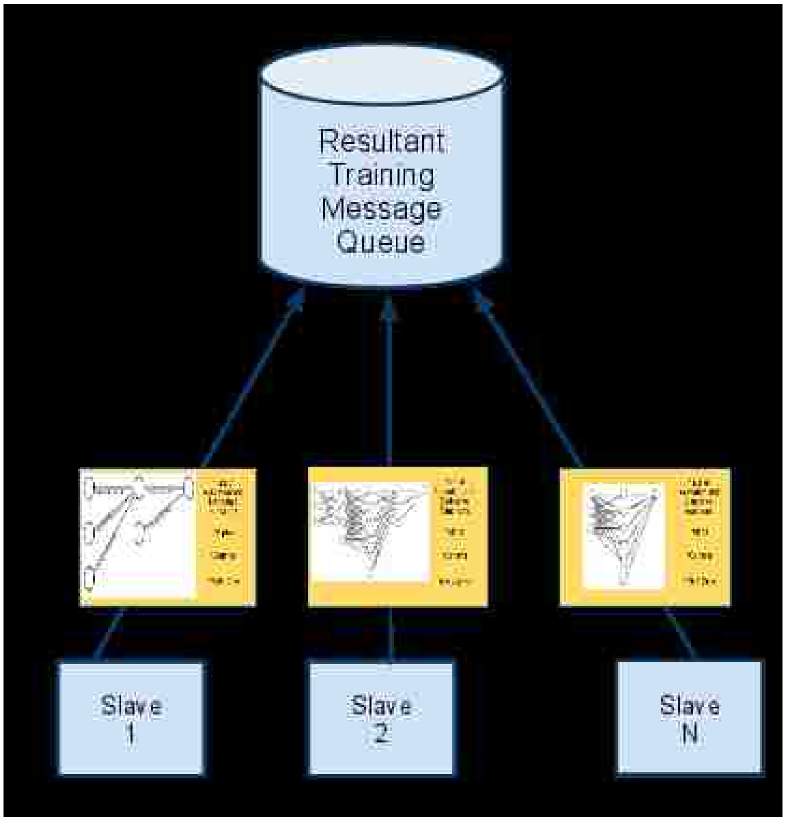


Figure 4.10: Training result message queue.

create a new population of structures for the slaves to train. The master discards the bottom half of the population sorted in order of fitness; so the least fit of the population. The remaining results consist of the possible parents for the new generation. A number of children are generated from two parents by applying crossover and reproduction to the possible parents using a roulette wheel based

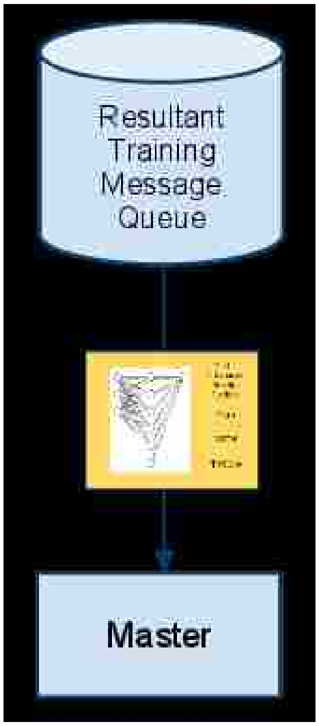


Figure 4.11: Master consuming training result message.

selection for each set of parents. The probability of selecting a parent is determined based on the fitness of the parent in comparison to the fitness of all other parents. This way the fitter of any two parents has a higher probability of getting selection for reproduction and/or breeding.

The process continues in this manner until a certain number of generations have been bred. This number is specified by the user. The fittest network of the last generation is selected as the neural network most capable of solving the particular problem.

Chapter 5

Implementation

This section describes the tools used to create the *NNGenerator* software. The language, libraries, and runtime used are described as well as the motivation for each. A few of the data structures used are also discussed.

5.1 Clojure

The application is written in Clojure [Hicb] which is a LISP [SG93] implementation for the Java Virtual Machine [Orac], or JVM. Clojure is fully interoperable with Java [GJSB05] and so the parts of the application that interface with existing Java libraries are also written in Clojure. Programmers that have used languages in the LISP family may be familiar with some concepts that are present in Clojure, such as functions as arguments, dynamic typing and lazy evaluation. Other features present in Clojure used in *NNGenerator* are interesting enough to be worth mentioning here also, including multimethods and software transaction memory.

5.1.1 Functions as Data Types

In functional languages, functions are first class data types; they can be passed to other functions as arguments, thereby allowing specific behavior not related to the calling function to be coded separately from the calling function. Object oriented languages accomplish similar behavior with polymorphism which is facilitated through inheritance. Having functions as first class data types in a language is more flexible than inheritance because the runtime only needs to check a function's signature and does not need to examine the type hierarchy that a function belongs to.

5.1.2 Closures

The name for Clojure is based on the term *closure*, which describes a function that has access to all variables bound in the closure's current scope. In the case of a closure, in addition to a function definition, pieces of state that are not in global scope can be optionally bound. In this a function can be created that uses state that is hidden from the function that invokes it. A simple example of this concept is the following function *incrementAndGet*. It keeps track of a variable to increment and return without using global state:

```
(let [i (ref 0)]
  (defn incrementAndGet []
     (dosync (ref-set i (inc @i))) @i))
```

Here is the example output from calling this multiple times in a Clojure REPL:

```
 user=> (incrementAndGet)
1
user=> (incrementAndGet)
2
user=> (incrementAndGet)
3
```

5.1.3 Dynamic Typing

Java is a statically typed language; its function arguments are checked at compile time and the programmer must explicitly mark all object references with a type declaration. This can help to catch errors in calling functions at compile time rather than at run time, but the code becomes littered with types, and changing the structure of a type or adding functionality to an object is difficult.

Clojure is a dynamically typed language; it determines the actual type of method parameters at runtime, and it accomplishes this via the Java reflection Library [Orab]. In the rare case that reflection is causing performance issues, such as a tight CPU bound loop, type hints can be provided for variables. In Java 7, a new bytecode instruction called `invokedynamic` [Ros09] has been added. This instruction allows dynamic languages on the JVM to have native support for dynamic typing without using the Reflection library. This instruction is similar to the `invokevirtual` instruction which is used whenever a virtual method is called in Java. By default all methods in Java are virtual unless marked as private or static. This is because static and private methods cannot be overridden in a subclass, so there is no need to lookup their location at runtime since it can be hardcoded at compile time. The addition of the `invokevirtual` instruction will allow future Clojure implementations to see performance improvements as opposed to using reflection, which is much slower in comparison.

5.1.4 Lazy Evaluation

Clojure supports lazy evaluation, meaning that program statements are not evaluated line by line as they are with imperative languages. The result of any particular computation is not actually computed until the exact moment it is needed. The result may be needed in order to evaluate the result of another computation. It may also be needed in a method that has side effects, such as writing data to a socket or writing some data to a video buffer, or in the case of Clojure/Java interop where the Java functions are not typically pure and need to be executed to force some side effect. This lack of pureness in Java functions is typical in object oriented programs and means that most methods are not idempotent. In other words, calling the same set of functions repeatedly with the same arguments does not necessarily produce the same result. This makes reasoning about the code particularly hard. The result may never actually be needed in the course of the program and in this case it will never be calculated. Special syntax is provided for when it is necessary for the programmer to force evaluation. The following in a snippet from *NNGenerator* that uses the *doall* keyword to force the side effect of the *rmdir* Java method:

```
(defn rmdir [dir]
 (if (.isDirectory dir)
     (doall (map rmdir (.listFiles dir))))
 (.delete dir))
```

Lazy evaluation allows Clojure to represent things such as infinite sequences, which are otherwise very hard to represent in imperative languages. The following defines the infinite set of positive even integers:

```
(def evens (iterate (fn [x] (+ 2 x)) 0))
```

Obviously realizing this full set at once is not possible because it is an infinite set and the machine has a finite memory. Parts of this set can be realized in smaller pieces. Here is some output of using the *take* method to fetch pieces of the set:

```
user=> (take 4 evens)
(0 2 4 6)
user=> (take 10 evens)
```

```
(0 2 4 6 8 10 12 14 16 18)
user=> (take 15 evens)
(0 2 4 6 8 10 12 14 16 18 20 22 24 26 28)
```

A particular downside of laziness is that it makes debugging harder. While stepping through code line by line, some expressions may not ever be calculated even though the code for the expression has already been seen by the runtime.

Another interesting case of lazy evaluation is that it is possible to blow the call stack if the level of unevaluated expressions gets too deep. While writing the *NNGenerator* software this happened in the Slave module for large training sets. The matrix multiplication calls were not being evaluated since the result was not needed until the training of the entire neural network was complete. To circumvent this issue, the *doall* is used in the *matrixAdd* method:

```
(defn matrixAdd [matrixA matrixB]
  (if (and (not (empty? matrixA)) (not (empty? matrixB)))
    (conj
      (matrixAdd (rest matrixA) (rest matrixB))
      (doall (map + (first matrixA) (first matrixB))))))
```

5.1.5 Multimethods

Many object oriented languages such as Java and C# support dynamic method dispatch through a concept called polymorphism. The runtime selects an appropriate method to call for an object based on the runtime type of the object. One reason for this is to be able to pass functionality into methods via interface type declarations because the languages do not support functions as first class objects. Clojure goes beyond polymorphism and offers a more general concept of dynamic method dispatch in which the programmer defines the dispatch function as opposed to always using the runtime type of a particular object.

The following uses polymorphism in Java:

```
interface Foo {
        String foo();
}

class FooBar implements Foo {
        public String foo { return ”foobar”; }
}
```

```
class FooCat implements Foo {
        public String foo { return "foocat"; }
}
```

To implement polymorphism in Clojure using the same Java types:

```
(defmulti foo (fn [obj] (.getName (.getClass obj))))

(defmethod foo "FooBar" [obj] "foobar")

(defmethod foo "FooCat" [obj] "foocat")
```

The *defmulti* macro is used to define the name of the function, in this case *foo*. It also defines the dispatch function for the argument or arguments that gets passed to the function. In this case, a single argument will be passed to the function, and the classname of the argument will be returned by the dispatch function. The *defmethod* macros create and install two new methods of multimethod *foo* associated with the dispatch-values *FooBar* and *FooCat*.

The following is a switch statement in Java:

```
MyCalendar cal = new MyCalendar();
switch (cal.getMonth()) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
            case 8:  System.out.println("August"); break;
            case 9:  System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Invalid␣month.");break;
        }
```

This can be broken up into many methods using a multimethod in Clojure:

```
(defmulti getMonthString (fn [cal] (.getMonth cal)))

(defmethod getMonthString 1 [cal] "January")
(defmethod getMonthString 2 [cal] "February")
(defmethod getMonthString 3 [cal] "March")
(defmethod getMonthString 4 [cal] "April")
(defmethod getMonthString 5 [cal] "May")
(defmethod getMonthString 6 [cal] "June")
(defmethod getMonthString 7 [cal] "July")
(defmethod getMonthString 8 [cal] "August")
(defmethod getMonthString 9 [cal] "September")
(defmethod getMonthString 10 [cal] "October")
(defmethod getMonthString 11 [cal] "November")
(defmethod getMonthString 12 [cal] "December")
(defmethod getMonthString :default [cal] "Invalid␣month.")
```

If simply writing a String is all that a switch statement does, the advantage of breaking it up into many methods is not noticeable. For complex logic that would typically be handled with a long else if or a switch statement, multimethods help keep code clean and more readable.

5.1.6 Simplicity and Verbosity

Functional programs take the programmer farther away from the hardware than non-functional ones. Consider the following simple example when asked to write a function that takes a list of integers as its input and outputs a list of integers of the same arity whose values are the incremented value of the corresponding value of the original list. For example, given the input (-1 -2 -3 1 2 3), the output is (0 -1 -2 2 3 4). If asked to write this in C, Java, Pascal, or any other non-functional language, a typical implementation might look like the following Java code:

```
int[] increment(int[] x) {
        int[] y = new int[x.length];
        for(int i = 0; i < x.length; i++) {
          y[i]=x[i]+1;
         }
         return y;
}
```

A typical Clojure implementation may look like the following:

```
(defn inc_func [x] (map inc x))
```

Syntactically, the Clojure version is much less verbose than the Java one. Ignoring syntactical differences, one piece of code in the Java version that we see is uninteresting is the declaration, checking, and incrementing of the variable `i`. Here we must explicitly create and use a separate piece of state for indexing into the arrays. This temporary state is unnecessary and can be replaced with Clojure's *map* function. The *map* function returns a lazy sequence consisting of the result of applying *f* to the set of first items of each collection, followed by applying *f* to the set of second items in each collection, until any one of the collections is exhausted. *map* is a n-ary function, and when called with a single list, it applies a function *f* to every item in the list and returns the result. There is no temporary state to be able to index into the list. Although it can become natural to create and use temporary state in performing an operation on each item in a list of items, it is usually not necessary to use the state in the computation, and it clutters the solution.

Another uninteresting piece of code in the Java version is the line where `y` is defined. This declaration is really just an artifact of how von Neumann based architectures work. In order to write this function without making changes to the original list, a new list of the same size must first be created, making the code unnecessarily verbose.

5.1.7 State

Many functional languages are called pure functional languages. They are ”pure” in the sense that every function takes some state and returns some state and does not modify any global state. In fact there is no global state. This makes reasoning about code and verifying certain properties much simpler than a procedural or object oriented language where state is everywhere and functions can do whatever they want with that state [Wel02]. In fact, these languages do not even enforce correctness when working with state from multiple threads; programmers are required to design correct solutions to these problems themselves. All of the data types native to Clojure support the concept of purity. When you perform operations on lists for example, the results of those operations are actually new lists and the original one passed in remains unchanged. This has the same semantics as call by value with performance comparable to call by reference without the danger of destroying the reference. In some LISP implementations this causes performance problems since lists are getting copied a lot. Clojure uses a much more efficient way of doing this, which underneath the hood shares the data structures [Hica] and so can give the same performance guarantees as using mutable data structures in Java without the added complexity of mutable state.

Another problem with state and object oriented languages is misuse of mutability. By default, member variables are mutable unless declared as final by the programmer. Even the final modifier only enforces that the referenced object cannot be changed after being set in the constructor; it does not enforce any immutability with respect to the object's data members. In Clojure, the opposite is true, every data definition is immutable by default unless it is explicitly made mutable by the programmer. When defining a Java data type, it cannot enforce anything with regard to mutating that object's state, and this is one thing to watch out for when using the Clojure/Java interop feature. Although lack of global and mutable state can be nice for theorem proving and code reasoning, in the real world programs that do something useful usually have some mutable state somewhere and are not just collections of pure functions and immutable data.

Threads

Clojure fully acknowledges the fact that mutable state is needed somewhere in most real world applications, however, it does not use the problematic thread model to provide for reading and writing mutable state. Here we briefly examine a few of the problems with using threads. For a more thorough inspection refer to [PGB$^+$05].

Problems can arise if one decides to edit a list inline either for terseness or to save memory and therefore compromise on the "without making changes to the original list" constraint:

```
int[] incrementInPlace(int[] x) {
        for(int i = 0; i < x.length; i++) {
          x[i]++;
         }
         return x;
}
```

For single threaded programs, there is no issue; in a multi-threaded environment, though, consider how the above code causes issues even with just two threads calling it at the same time. If thread A attempts to access list at the same time as thread B is modified the list, then the result that thread A computes is also invalid. It gets worse; the thread solution to this is to use a monitor or mutual exclusion block to ensure that no two threads can call the function at the same time. In Java, the monitor can be implicit by using the method modifier `synchronized` to the method definition. If the method is static, the monitor is implicitly the singleton instance of an object's *getClass* method:

```
static synchronized void foo() { }
```

If the method is a non-static method of a class, the monitor is a particular instance of the class and is shared among all non-static synchronized methods of a particular object:

```
synchronized void foo() { }
```

You can also declare arbitrary objects and use them as monitors:

```
Object x = new Object();
synchronized (x) {
 //holding monitor of x
}
```

In C#, the monitor is a library:

```
System.Object obj = (System.Object)x;
System.Threading.Monitor.Enter(obj);
try
{
    DoSomething();
}
finally
{
    System.Threading.Monitor.Exit(obj);
}
```

With some syntactic sugar:

```
lock (x)
{
    DoSomething();
}
```

Whether built into the language or provided as a library, locks create problems [Lee06]. Consider what happens when thread A acquires a lock to object X and then waits for a lock on object Y to be released, then gets interrupted, then thread B, which is holding a lock on object Y, gets switched in and immediately begins waiting on the lock to object X, deadlock. This makes it hard to create robust multithreaded API's, since the client code may acquire the locks out of order not knowing the lock ordering rules of the API.

Another problem with locks is that, when used in non-functional language, they cause readers to block readers and writers, and writers to block readers and writers. All of this blocking leads to performance problems when many threads are reading or writing some shared state.

Software Transactional Memory

In addition to the desirable properties offered by functional languages which including composability of functions and lack of shared state, Clojure provides a model for concurrency that is easier to understand, implement, and verify than that of Java threads. Software transactional memory, or STM, is a pattern that is used for providing transactional memory in a distributed shared memory [ST97]. STM in Clojure is completely non-blocking. Readers do not block readers, and writers do not block readers. Readers read the value of a reference at the time requested; in other words, the latest committed value. Writers do not block writers, and readers do not block writers. Many writers can run in parallel, and, before they commit their transactions, they check the value of the shared state they are going to write to. If the value has changed since the start of the transaction, the writer does its work again with the new value of the shared state. In this way, writers do not block each other, and high contention to write a particular value results in writers repeating their work more than once. In Java, this would prove difficult as most functions are not pure and cannot be run repeatedly in transactions blindly.

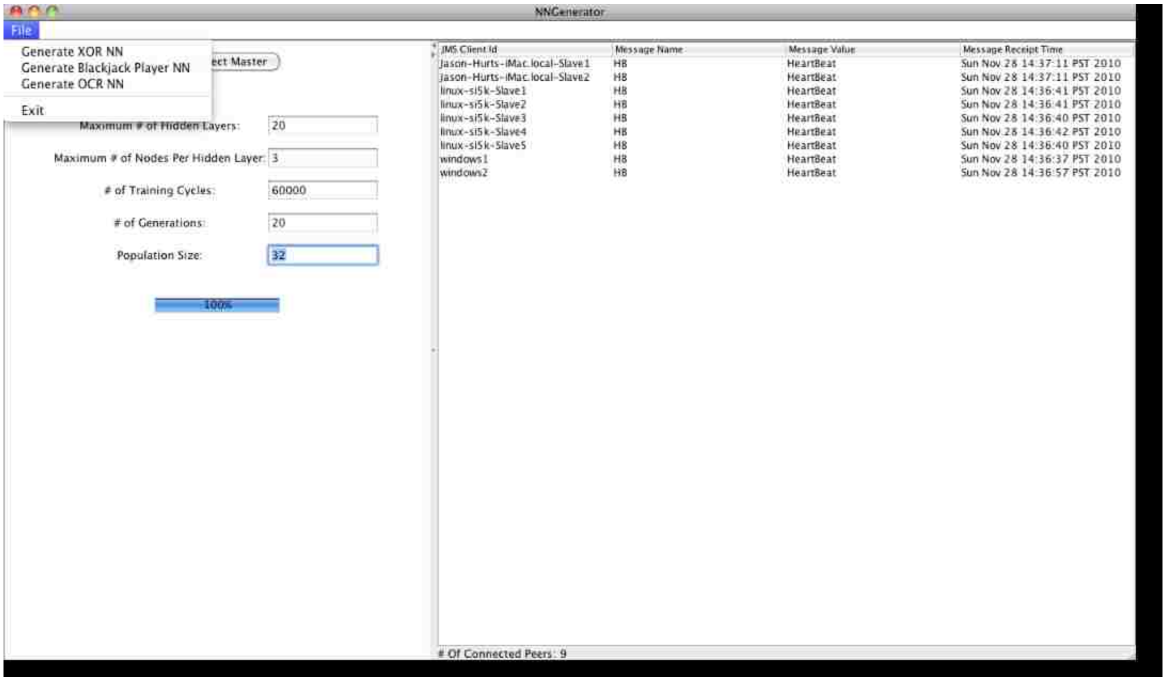
Another benefit of STM in Clojure is that nested transactions are handled without other functions knowing the order and nature of the transactions. As previously described, using the thread model in Java, locks shared among classes must be acquired in a certain order to avoid deadlocking multiple threads. The correct use and/or the knowledge of this ordering is often missing from programs. Providing the ordering couples classes together in a way that is not necessary with Clojure's STM.

Clojure combines many of the benefits of functional programming with the power of the JVM to allow for a very powerful tool for creating a distributed and multithreaded application such as this one.

5.2 User Interface

When the master module is started, a graphical user interface, or GUI, is displayed that allows for controlling various parameters of the application. Figure 5.1 shows this user interface when run on OS X Snow Leopard. A similar interface will appear when run on a Linux or Windows operating system.

The table to the right shows how many slaves are connected, the hostname followed by a global counter for each host, and the last time a heartbeat message was received from the slave. The screenshot shows there are 9 slaves connected to the network. Two of them are from a dual core



Maximum # of Hidden Layers: 20

Maximum # of Nodes Per Hidden Layer: 3

# of Training Cycles: 60000

# of Generations: 20

Population Size: 32

JMS Client Id	Message Name	Message Value	Message Receipt Time
Jason-Hurts-iMac.local-Slave1	HB	HeartBeat	Sun Nov 28 14:37:11 PST 2010
Jason-Hurts-iMac.local-Slave2	HB	HeartBeat	Sun Nov 28 14:37:11 PST 2010
linux-si5k-Slave1	HB	HeartBeat	Sun Nov 28 14:36:41 PST 2010
linux-si5k-Slave2	HB	HeartBeat	Sun Nov 28 14:36:41 PST 2010
linux-si5k-Slave3	HB	HeartBeat	Sun Nov 28 14:36:40 PST 2010
linux-si5k-Slave4	HB	HeartBeat	Sun Nov 28 14:36:42 PST 2010
linux-si5k-Slave5	HB	HeartBeat	Sun Nov 28 14:36:40 PST 2010
windows1	HB	HeartBeat	Sun Nov 28 14:36:37 PST 2010
windows2	HB	HeartBeat	Sun Nov 28 14:36:57 PST 2010

Of Connected Peers: 9

Figure 5.1: *NNGenerator* Software Graphical User Interface.

iMac running OS X 10.6, two of them are from a dual core Intel running Windows 7, and five of them are from a quad core Intel running SuSE Linux 11.0.

One of the parameters the user can enter is the population size, which does not have to match the number of actual network slaves; it is the number of networks to train at each step. So if you have only eight slaves and you enter sixteen, then each slave will train two neural networks at each iteration of the search algorithm. Another parameter is the number of generations to train. As you increase this parameter, the time to complete the algorithm increases linearly. Another parameter sets the amount of iterations a slave should train a single neural network. This is a constant for all slaves so that each generated neural network has been given a fair chance at training its weights. The remaining parameters place upper bounds on the neural network structures themselves. A maximum number of hidden layers parameter sets an upper bound of the number of hidden layers for a neural network. A maximum number of nodes per layer parameter sets an upper bound on the number of nodes for a hidden layer. The number of input and output nodes depends on the map of input vectors to outputs used during training and cannot be changed by the user. Tweaking the number of iterations to train a network, the number of nodes per layer and number of layers can have a substantial effect on the time it takes for the algorithm to complete. To get the output of each hidden layer and the output layer is on the order of NM, or the product of the weight matrix between the previous layer and the current layer and the nodes in that layer. As the number of nodes increases, the time to complete the operation for a layer increases.

The user interface is written using the Java Swing library [Orad]. This library comes bundled with the Java Runtime Environment and allows creating cross-platform GUI's that look native to the OS or can be skinned in a customizable way. Swing uses a single threaded model and embraces the Model-View-Controller or MVC design pattern [mvc88]. This pattern keeps the view code separate from the model code, and the view can update the model only though the controller. In Swing, the controllers are action listeners which are fired based on events such as a mouse click or keyboard input. The action listeners all spawn a new thread to do their work, and later the UI may be updated asynchronously on the main GUI thread called the *Event Dispatch Thread* or EDT. Unfortunately, the library cannot enforce this behavior, it is up to the programmer to keep this pattern in mind. If work is done on the EDT that should be done in a background thread, the user interface will become slow and unresponsive.

5.3 Java Messaging Service

Although the concurrency features of Clojure are nice, they are limited in that they only provide support for shared memory concurrency on a single JVM. The distributed architecture uses a messaging model that is facilitated via the Java Messaging Service or JMS API [Oraa, TS02]. JMS is a Java API that provides applications with the ability to communicate via objects called messages. A program that uses JMS to produce and/or consume messages is called a JMS client. Message passing in JMS uses an asynchronous send in which a producer produces a message and does not block waiting for it to be consumed, though it does block waiting for the message broker to confirm receipt. On the consumer side, receiving a message can either block until it has a message, or it can process messages asynchronously through an event listener. In the latter case, an event will fire for every single message that the client receives. JMS is also reliable, it ensures that every message successfully produced by a JMS client gets consumed by a subscriber if one is available.

There are two available approaches to messaging in JMS. The first is a publish/subscribe model in which JMS clients publish messages to queues called *topics*. A topic can have one or more subscribers and will send all messages in the topic to all current subscribers, meaning that a particular message may get processed more than once. A client must be subscribed to a topic before it can receive messages from that topic. This implies an ordering on the time a publisher publishes a message to a topic, and the time a subscriber consumes that message from the topic; the publish must happen first. Figure 5.2 is an illustrative example of the publish/subscribe model in JMS.

The second approach is a point-to-point model which is very similar to the publish/subscribe approach. With this approach, messages are queued to named queues. Each message will only get

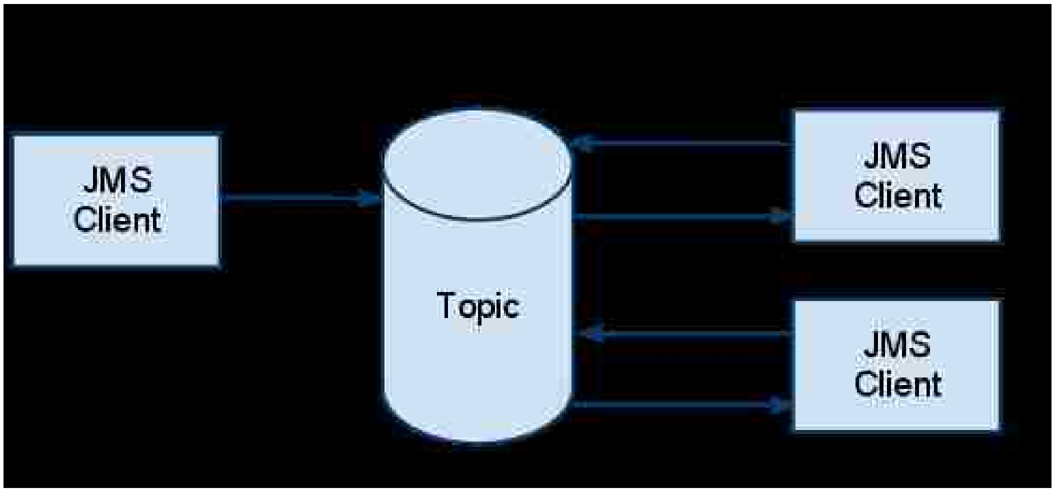


Figure 5.2: JMS publish/subscribe model.

consumed once by a single JMS client. There is no ordering on the when a client consumes a message from a queue and when a client publishes that message to the queue. This is not unlike languages that use tuple spaces such as Linda [Gel89]. Figure 5.3 shows how the point-to-point model works in JMS.

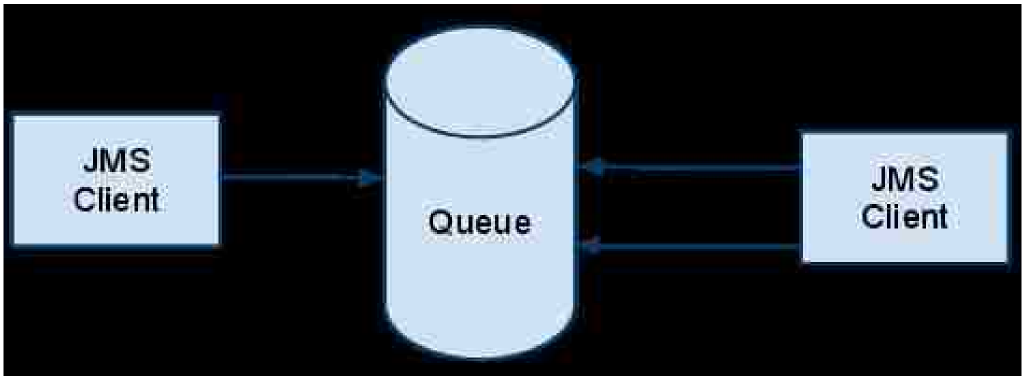


Figure 5.3: JMS point-to-point model.

The *NNGenerator* software uses the point-to-point model. We do not want either training messages or training result messages to ever be processed more than once. The messaging model is simple, and as described in the architecture, there are two message queues. One queue is for the single master node to write to and for the slaves to read from, and the other is for the entire set of slaves to write to and for the master to read from. As soon as a master or slave node sends a message, it does not block waiting for a response.

JMS message queues and topics are provided via JMS brokers. All message queueing and dequeing is handled through one or more JMS brokers. These brokers behave like proxies and have logic for making sure that messages get sent and received. In a production environment, these brokers are usually replicated in case one of them fails. The *NNGenerator* software expectes a single JMS broker to be available to the master and slaves nodes. Implementors of the brokering services come in many different implementations. An implementation must implement the JMS specification for a broker in order to work with JMS clients. During testing and while collecting results we are using a message broker provided by Apache ActiveMQ [Fou]. The software could just as easily use another JMS broker to connect to. The ActiveMQ broker listens on a single TCP socket for messages from other nodes on the network. Nodes never have to talk directly to each other on the network; they only need to be able to connect to the broker. This allows the master and the slaves to operate on different networks and they never have to communicate directly.

The *NNGenerator* master and slave JMS clients both use asynchronous receives. An event listener is wired up for both the slaves and master for when messages are received. The slave will sit idle until it has a training message to consume, at which point it will become CPU bound while it trains a neural network. After it finishes training, it will sit idle again until another training message comes in. The master also has a thread that sits idle and waits for result training messages in the same manner. The master receives a message for each neural network that a slaves trains. It also receives heartbeat messages from slaves to determine when slaves disconnect, or might have run into some other problem that will prevent them from being able to train a network. Clojure multimethods are used to process incoming messages based on the type of the message. Slaves receive a different message type for each different problem. A problem is specified by its training data set. The type allows the slave to select the training data set to use for training the network. For example, the XOR input output map will be selected when a slave receives a *TRAIN-XOR* message. The training sets are distributed along with the slave jars to keep the master from having to send the data set over the network. The data can get very large in the case of binary formats such as images and sound. As stated in the architecture chapter, the algorithm can continue as long as one slave on the network is still able to receive messages. This is undesirable, however, as the algorithm is meant to fully exploit the inherent parallelism of genetic algorithms by having a one to one mapping between the number of slaves on the network and the size of the population.

5.4 Data Structures

The neural network structures that are generated and bred are all backpropagation neural networks with at least one hidden layer. They are encoded as strings representing serialized maps in Clojure. These strings are relatively small compared to serialized Java objects and are also human readable since they are just Clojure data. The serializing and deserializing functions are also simple compared to what would be needed to serialize a Java object. Here is serialization:

```
(defn serialize [x]
  (binding [*print-dup* true] (pr-str x)))
```

and here is deserialization:

```
(defn deserialize [x]
  (let [r (new PushbackReader (new StringReader x))]
    (read r)))
```

Chapter 6

XOR Problem

The test problem that was used while building the application is a simple *XOR* problem. The input/output map is provided to the program as the following Clojure map:

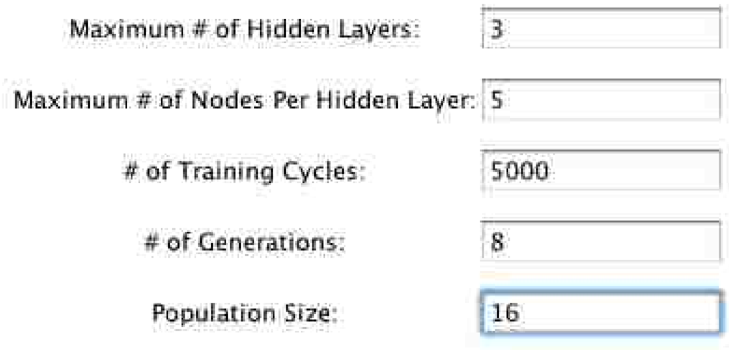
```
(def XOR-table {[-1 -1] [-1]
                [-1 1] [1]
                [1 -1] [1]
                [1 1] [-1]})
```

The networks are trained with random samples from the map.

6.1 Results

6.1.1 First Result Set

Table 6.1 shows the results for running the *NNGenerator* software with the following parameters:



Maximum # of Hidden Layers:	3
Maximum # of Nodes Per Hidden Layer:	5
# of Training Cycles:	5000
# of Generations:	8
Population Size:	16

Table 6.1: *XOR* Trainer Test Results 1.

Generation	Lowest RMS Error	Average RMS Error
1	6.749198082231432E-28	0.05717612521054472
2	1.6652497156304906E-22	0.011905249628373422
3	3.887822085290851E-26	4.5259383247370363E-7
4	5.3155295284848575E-21	2.3281523385011523E-4
5	5.372827720630989E-59	0.007860022032154978
6	2.785665071561689E-29	0.12405979617855686
7	2.737285346503349E-23	0.1892872997753143
8	1.2154326714572542E-63	0.18667527335585038

The software generated the neural network shown in Figure 6.1.

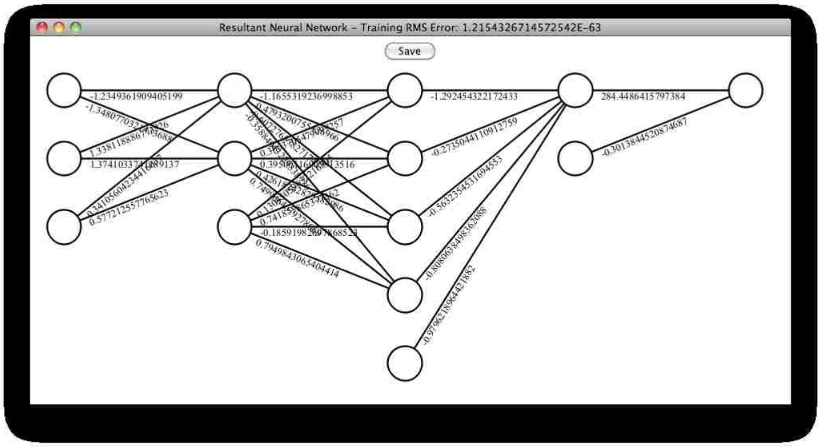
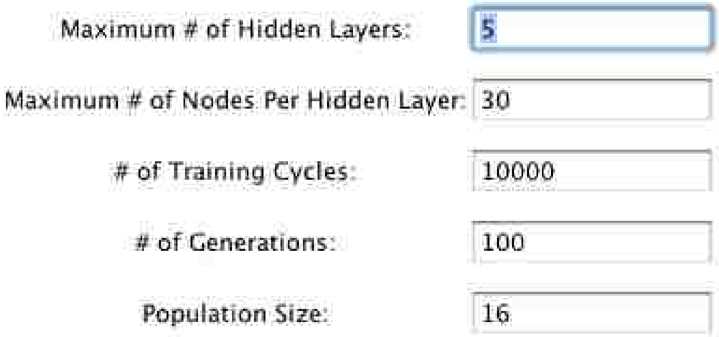


Figure 6.1: Resultant neural network for first *XOR* trainer result set.

6.1.2 Second Result Set

Table 6.2 shows the results for running the *NNGenerator* software with the following parameters:



Maximum # of Hidden Layers: 5

Maximum # of Nodes Per Hidden Layer: 30

# of Training Cycles: 10000

# of Generations: 100

Population Size: 16

Table 6.2: *XOR* Trainer Test Results 2.

Generation	Lowest RMS Error	Average RMS Error
1	2.5054577337217813E-25	7.306048546794003E-6
2	1.3866695599587982E-28	3.3460553362913697E-20
3	1.0871489350077044E-29	1.566204032546096E-21
4	7.457200744667331E-29	4.705563879914948E-11
5	4.438190128677072E-53	1.419312648141364E-21
6	9.880180029664589E-46	1.7111405648511735E-8
7	1.2325184417509783E-30	1.6042524538056823E-11
8	1.4626339857138885E-27	6.712774782870025E-12
9	1.889609509681072E-30	1.521749396876346E-9
10	1.3634966915758752E-33	1.5937700670241007E-10
11	2.6011938055248963E-30	5.742581610923849E-12
12	2.1154096662283722E-27	3.4111424869786424E-11
13	1.040421067398336E-25	1.087397588423917E-11
14	1.863208120020399E-26	3.1986877746298507E-9
15	1.2696810965157856E-34	1.8087324593907032E-11
16	7.425113805196876E-31	7.489259430539917E-12
17	1.130774542447281E-32	4.94706855516273E-12
18	1.5538009240947378E-29	1.783129695012996E-10
19	6.257678976796708E-37	4.457178593375146E-11
20	4.206190914291976E-28	7.574632975943686E-9
21	1.7054132911551076E-25	1.3174383099546557E-10
22	3.4438143119862094E-29	5.852895369947579E-7

23	2.1320443017741385E-29	1.363858203279037E-9
24	3.507923759549089E-23	3.7802210639103527E-11
25	1.6624417039146802E-34	1.589033712660971E-6
26	1.0717512757427784E-34	1.7692135458311465E-10
27	3.119804464516293E-47	6.606650434571091E-7
28	4.3780840755578434E-35	4.033116569745029E-12
29	6.309267783614088E-41	1.4425621254468801E-11
30	1.7080256423619204E-38	3.565016064982958E-12
31	1.2406943383084812E-41	2.6223417933475996E-12
32	1.766760416476497E-31	1.4927307016594083E-7
33	2.2982642291056944E-37	1.0593184690594624E-11
34	1.2109701276932383E-27	1.0066696433031767E-6
35	2.62676195083333E-29	5.802242694634442E-11
36	7.29512571536478E-25	1.201484738365783E-11
37	5.29765521394315E-41	3.091446648820336E-12
38	1.3582567589713704E-39	6.368793201739928E-12
39	4.21517535679883E-39	3.218907087442485E-9
40	5.5940177088011746E-27	2.9139084880615986E-11
41	1.2638630346285576E-26	1.2010070663605773E-7
42	1.4717817274666266E-49	2.646020640251503E-12
43	1.628304840062059E-48	2.600339181182816E-12
44	2.359295819405455E-40	5.593464941213629E-13
45	3.5354224829021327E-34	3.572130163274878E-11
46	1.0899580456980869E-43	1.830531785227526E-12
47	5.4557045941430794E-48	2.1749658514322455E-12
48	1.1572427749164428E-37	2.4847863730405124E-12
49	1.5120086121347679E-34	9.5679993576262E-13
50	2.8714558184013917E-27	7.019401364503659E-12
51	9.376308302862078E-39	9.115604418703448E-13
52	1.4950316868745454E-26	2.1511512209630587E-8
53	2.7397806035973195E-27	1.4966826339695197E-8
54	3.3607520640408625E-33	1.4261670845673156E-8

55	2.341358052886801E-25	5.83192063995526E-12
56	1.8268831620392305E-29	6.536166923975071E-11
57	3.6088254267876384E-22	1.6360408104704007E-11
58	3.233787670137434E-26	5.941969350364327E-12
59	4.816863814274461E-28	3.472295729616766E-10
60	4.1449988755171495E-24	8.541371100340901E-11
61	2.7344054988979634E-54	1.877341508736524E-16
62	1.890955555743946E-37	1.5985221490921967E-12
63	7.333721252753549E-37	7.389430914285899E-11
64	3.034446173726286E-53	6.682432218587649E-9
65	7.596454196607839E-65	2.6165515609367735E-10
66	3.2285543292592315E-39	3.0418566647132515E-11
67	8.608885138783913E-44	3.144721509624129E-9
68	4.215463433161572E-37	5.000405208006925E-12
69	3.6712526174074825E-39	1.8021116189820054E-11
70	3.272165084913714E-42	1.3884046503245164E-7
71	5.050771981831241E-38	2.4287373567393382E-12
72	7.626825647336907E-28	3.243008579555274E-8
73	6.174205148205196E-29	1.798845882265808E-11
74	2.9232226919095886E-28	9.829923007268686E-11
75	2.6064511402455526E-18	2.188827197125955E-9
76	1.5772613482836327E-16	4.867564735682575E-7
77	1.0003200793065999E-24	2.257414831458416E-10
78	4.8922473781801076E-27	5.708921055194993E-10
79	9.710800829327612E-25	3.7781006460765763E-10
80	4.591293727902238E-26	6.961953626054578E-17
81	2.1816492647895572E-24	9.868108873282733E-16
82	8.780219090332052E-25	8.943730967857937E-19
83	8.287969885478218E-29	1.715005282042369E-11
84	2.9353514283273056E-27	1.308911336428805E-19
85	7.194726999976752E-26	5.367242065729736E-14
86	3.940537715282238E-26	1.0997773068358775E-19

87	4.3750423018779045E-26	1.716479369944157E-12
88	7.928101401277392E-27	1.3503912231590541E-13
89	9.525988468609332E-28	5.640291554308247E-14
90	1.2939290997885318E-25	3.33095330120416E-12
91	1.7927307805405478E-26	8.421661830501178E-13
92	8.167126255556022E-25	4.6303226655305065E-9
93	1.0452091449814626E-25	0.02929982143704606
94	2.0377702061864192E-25	2.422528592468592E-19
95	8.697191480061614E-29	7.908968688024457E-19
96	8.788543545034392E-25	6.986436200064787E-18
97	1.6367305783044875E-25	3.177439957111431E-18
98	4.930380657631319E-30	1.3585826061122317E-18
99	1.9568680830137355E-26	4.608799951673015E-20
100	2.3863042382935022E-27	1.9904898035610088E-8

The software generated the neural network shown in Figure 6.2.

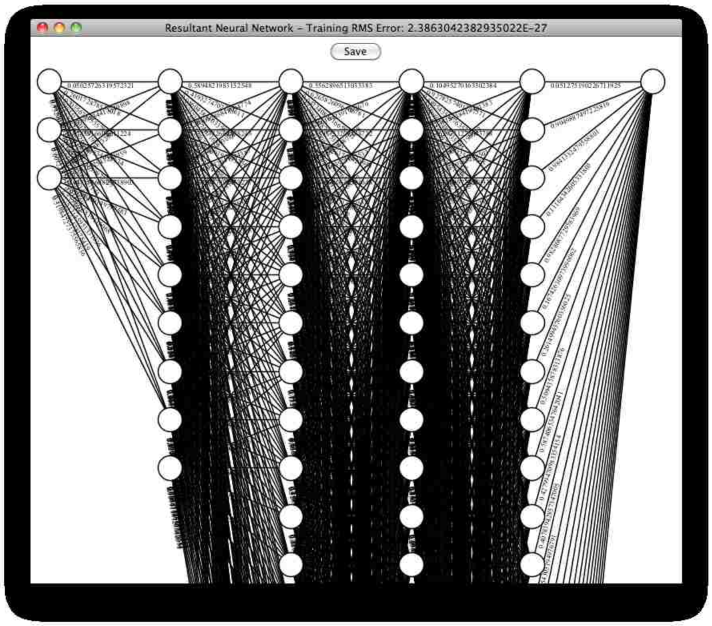

Resultant Neural Network – Training RMS Error: 2.3863042382935022E–27
Save

Figure 6.2: Resultant neural network for the second *XOR* trainer result set.

The fittest neural network was actually found in the $65th$ generation with a RMS error of $7.596454196607839 \times 10^{-65}$. The best overall generation is the second generation with an average RMS error of $3.3460553362913697 \times 10^{-65}$. This test run did not indicate that the neural networks were getting fitter with each generation.

Chapter 7

OCR Trainer

Another one of the problems used to test the program with is an optical character recognition problem, a technique used for converting handwritten text to a digital format. In this case the characters to be recognized are the arithmetic numerals $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

The training and data sets are from the MNIST database[LeC]. The characters used for training the classifier are a set of a subset of 60,000 samples of handwritten numerals of 30,000 patterns from SD-3, and 30,000 patterns from SD-1 from the NIST character set. The characters used for training the classifier are a set of a subset of 10,000 samples of handwritten numerals of 5,000 patterns from SD-3, and 5,000 patterns from SD-1 from the NIST character set. The intersection of the training set and test set is null as they are disjoint.

The training and test sets are comprised of 2 files each, a binary file containing the image data for the numerals, and a label file containing the correct output of the image to a digital format. The test image file is 45MB and so to save time in the feature extraction step, the input/output map for the set is preprocessed once and stored as a Clojure data structure. For training, the 45 MB image file and 60K label file is represented as a 5.6 MB input/output file. The input consists of extracting a binary string of length 16 that represents a threshold of pixel counts in the input image. The string is constructed of taking a 4x4 piece of the image row-wise. If the average pixel value is greater than a certain threshold, the value is 1, otherwise the value is 0. The output is a string of length 4 that represents the digital numeral as a binary string. The values 10,11,12,13,14,15 are never fed to the neural network during training.

When a slave starts the OCR trainer, it first reads the training file from disk and deserializes it into a Clojure data structure that the backpropagation function uses to train a neural network. Once this structure is loaded into memory, training begins. After training for the specified number of iterations, the slave posts a message to the master containing the results. If the master posts a message back, it will be another train OCR message and the slave will repeat the process. If not, the master has finished breeding and will display the resultant neural network structure. This structure can be saved and used to test the neural network against the test data set.

A single network is be generated by presenting the application with a training set, then a test data set that is disjoint from the training set will be used to test the accuracy of the network.

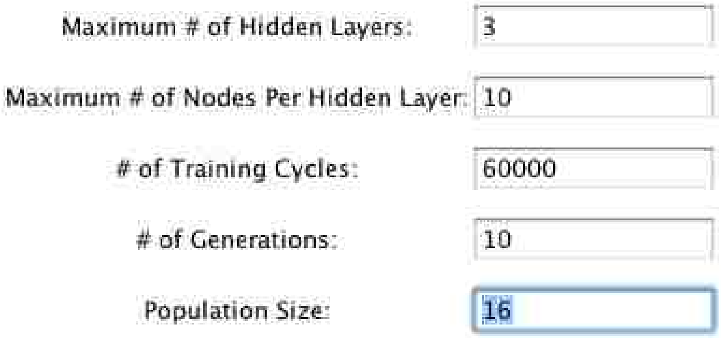
7.1 Results

7.1.1 Result Set 1

Table 7.1: OCR Trainer Test Results 1.

Generation	Lowest RMS Error	Average RMS Error
1	4.3502045626631517E-14	0.016356935630774453
2	2.0285182716361427E-8	0.010798833096735282
3	5.016625900942483E-12	0.009313208954575207
4	9.654223812029E-12	0.009764548655503679
5	8.822783538883067E-12	0.007621033973619068
6	6.432802850701297E-6	0.007046347953596382
7	5.3697890904199377E-14	0.013685965230928254
8	3.949459206373761E-10	0.00832293977963215
9	1.9570865437930458E-24	0.0077764095579861804
10	2.506513021332053E-6	0.010869428910938178

Table 7.1 shows the results of running the *NNGenerator* software with the following parameters:



Maximum # of Hidden Layers: 3

Maximum # of Nodes Per Hidden Layer: 10

# of Training Cycles: 60000

# of Generations: 10

Population Size: 16

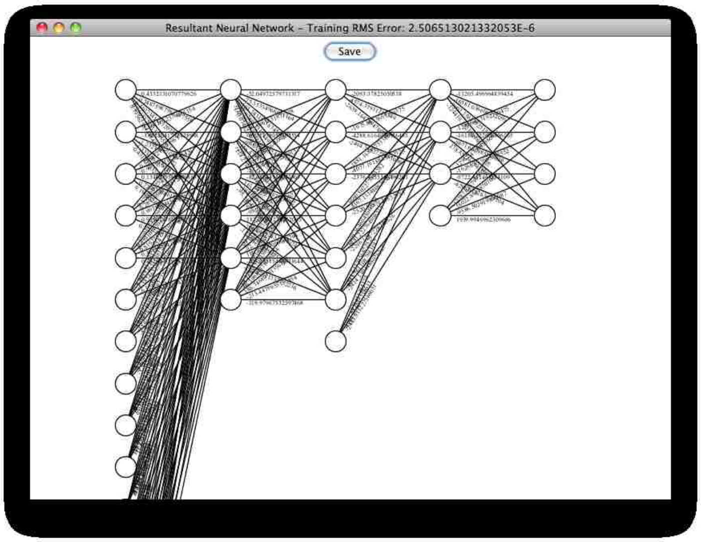

Resultant Neural Network – Training RMS Error: 2.506513021332053E-6
Save

Figure 7.1: Resultant neural network for first OCR result set.

A screenshot of the resultant network is shown in Figure 7.1.

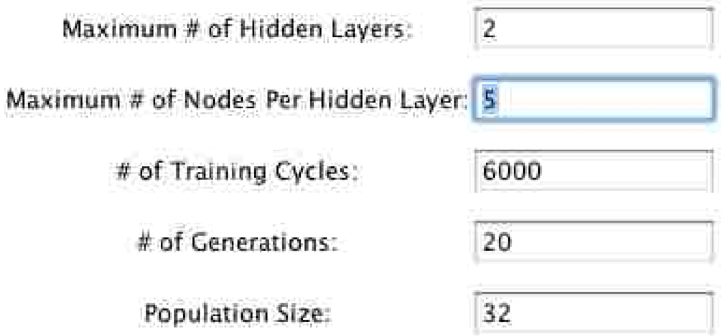
The resultant neural network was unable to converge and gave the following results when run against the test set and only classifed 1,135 of the 10,000 test characters correctly.

7.1.2 Result Set 2

Table 7.2: OCR Trainer Test Results 2.

Generation	Lowest RMS Error	Average RMS Error
1	7.891880999221867E-8	0.0173904761395515
2	2.013460555515104E-4	0.013774106331458341
3	1.277980970409587E-4	0.012982988605805981
4	1.2173258203796399E-5	0.00992557079845578
5	1.1493958742463188E-13	0.009978878509184833
6	3.4014167363083914E-12	0.00936778989360004
7	7.754248857947593E-9	0.01387680402589895
8	1.2555831561710939E-8	0.012832542111672472
9	4.3633615514969336E-8	0.009243350832757954
10	1.1771243219151047E-15	0.008222115616230612
11	4.176121633494215E-15	0.009546788434460465
12	1.2740827726504187E-14	0.008903159909956734
13	4.237583140121324E-15	0.01073538313750781
14	4.109851463789817E-15	0.008514012583273027
15	1.9732404636638029E-16	0.004604095778052103
16	4.117632915636444E-9	0.008408504165641664
17	6.476978089350002E-18	0.007001315851873932
18	6.292739877595802E-11	0.010569325971334793
19	1.5056421055145552E-15	0.007515515537989361
20	5.193350484983424E-18	0.009144290171367774

Table 7.2 shows the results of running the *NNGenerator* software with the following parameters:



Maximum # of Hidden Layers: 2

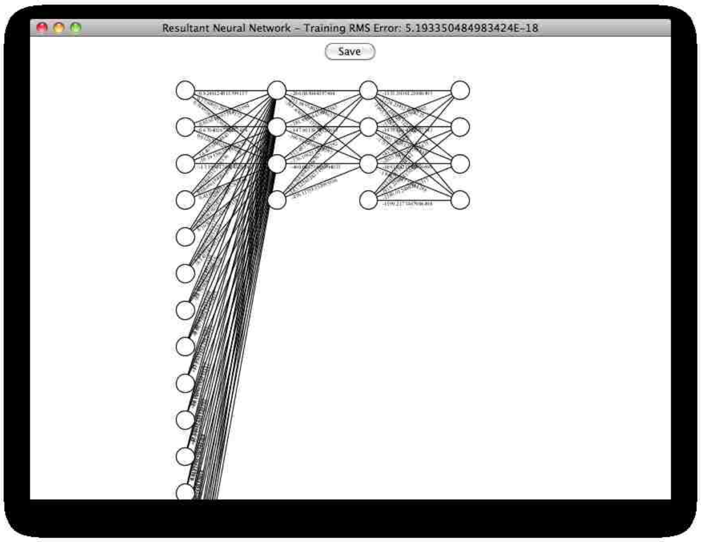
Maximum # of Nodes Per Hidden Layer: 5

# of Training Cycles: 6000

# of Generations: 20

Population Size: 32

A screenshot of the resultant network is shown in Figure 7.2.



Resultant Neural Network – Training RMS Error: 5.193350484983424E-18

Figure 7.2: Resultant neural network for second OCR result set.

The resultant neural network was able to classify 3,980 of the 10,000 test images correctly.

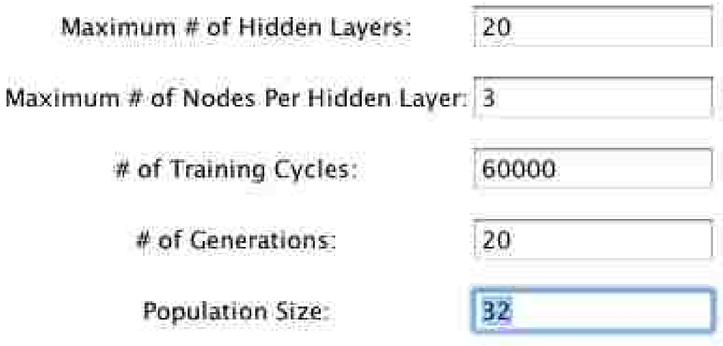
7.1.3 Result Set 3

Table 7.3: OCR Trainer Test Results 3.

Generation	Lowest RMS Error	Average RMS Error
1	7.989291092433633E-5	0.019821599105480103
2	4.004050993350956E-11	0.00781206556488213
3	2.953372921182323E-8	0.011670288713386441
4	3.561004565825299E-10	0.013758225968169701

5	4.969952505241766E-12	0.012584144555919543
6	9.620313006394983E-13	0.010457804379950851
7	1.3588012614107163E-8	0.01021952393825524
8	3.1156755853397372E-12	0.00900057086660724
9	2.900200837302653E-13	0.010918656380408584
10	1.0123949301175608E-12	0.008882579007562777
11	3.272790896905573E-11	0.009509713284177332
12	2.809366733938291E-11	0.009178001985773981
13	1.6355062166394745E-6	0.01386270936376654
14	6.541201436743673E-14	0.009129671062543086
15	2.3385156478767894E-11	0.012915654815698315
16	1.9655423855241817E-11	0.010588685426136333
17	1.6125954810677866E-13	0.009769752475599788
18	7.890250832783692E-15	0.0092041872140139
19	3.2510336758984747E-12	0.00768497129908572
20	1.561943265362401E-15	0.0086760219170695

Table 7.3 shows the results of running the *NNGenerator* software with the following parameters:



Maximum # of Hidden Layers: 20

Maximum # of Nodes Per Hidden Layer: 3

# of Training Cycles: 60000

# of Generations: 20

Population Size: 32

A screenshot of the resultant network is shown in Figure 7.3.

The resultant neural network was able to classify 3,135 of the 10,000 test images correctly.

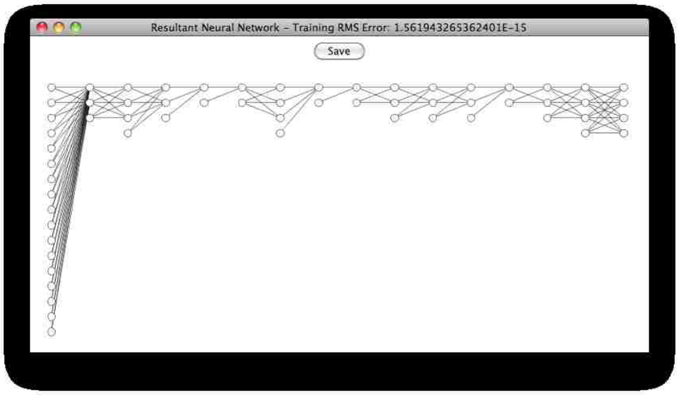

Resultant Neural Network – Training RMS Error: 1.561943265362401E-15
Save

Figure 7.3: Resultant neural network for third OCR result set.

7.2 Evaluation

All three test runs gave admittedly poor results, with the third (and best) neural network only able to classify about 40% of the test characters correctly. This is probably due to a poor selection of features for recognizing the characters. The RMS error did get better over time for the second and third result sets. For the first result set, the RMS error over each generation indicates no trend in either direction.

Chapter 8

Blackjack Trainer

One of the problems used to test the program is a blackjack player. Blackjack is a card game whose basic premise to get a hand value that is closer to 21 than that of the dealer, without going over 21. Each player plays against the dealer and does not compare hands with other players. The players can only see the dealers top, or second dealt, card. The value of an Ace can count as either 1 or 11. The cards 2 through 10 are valued at their face value. The Jack, Queen, and King are all valued at 10. The players and dealer are initially dealt 2 cards each, and a player can choose to either hit one or more times in succession or to stay. Each time a player hits, the dealer deals him another card. When a player stays, it is either the next player's turn or the dealer's turn in the case all players have played. There are a few variations of rules in casinos that determine when a dealer should hit or stay. The most common, and the one used in this program, is that the dealer must hit until the combined value of his cards is greater than or equal to 17. There are also more variations on the options a player has in addition to hitting and staying, such as doubling down and splitting pairs. For simplicity of the trainer, these variations are not included in the blackjack trainer.

The input consists of a binary string of length nine. The first five digits represent the value of the player's hand. The last four digits represent the value of the card the dealer is showing. The value of the dealer's hand is unknown to the player in an actual blackjack game, so the dealer's first card is not used in the input. For example, if the players hand totals thirteen and the dealer's top card is a three, the input string would be: 011010011.

The output pairs are found by playing simulated hands between a dealer and a single player. The output is 1 if hitting resulted in the player winning, -1 if staying resulted in a win, and 0 otherwise. The output is determined by using the dealer's rule of continuing to hit until the value of the hand is greater than or equal to 17. Note that this is not always the optimal strategy, therefore the data used during training does not always train the network with the optimal strategy.

The simulator used during training deals cards out of a shuffled deck for each hand. Because this shuffling is random, each slave will train its neural network structure with a different input/output map than every other slave. The chance that every possible combination of player's hand and dealer's top card will be considered increases as the as the number of training iterations increases.

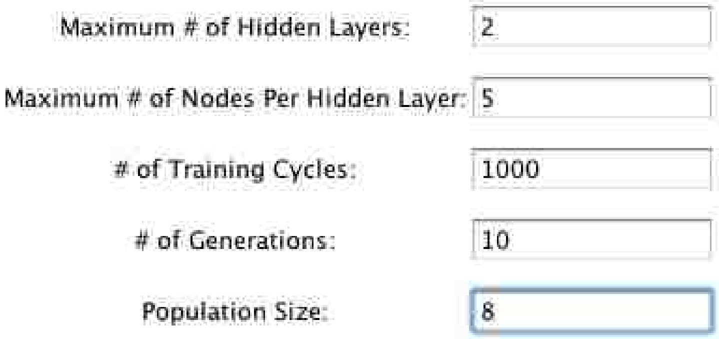
The trainer also has two simulators for testing a neural network. One is a simulator that will play a specified number of games where the player uses dealer's rules of hitting. The second simulator also runs for a specified number of games using the output of a neural network to determine whether or not to hit. These two simulators are used to determine if a neural network solution can do better than dealer's rules for hitting.

The following results are compared against playing 50,000 simulated games by dealer rules. When playing by these rules, the player won 20,309 of the games, tied 9,107 of the games, and lost 20,584 games.

8.1 Results

8.1.1 Result Set 1

Table 8.1 shows the results of running the *NNGenerator* software with the following parameters:



Maximum # of Hidden Layers: 2

Maximum # of Nodes Per Hidden Layer: 5

# of Training Cycles: 1000

# of Generations: 10

Population Size: 8

Table 8.1: Blackjack Trainer Test Results 1.

Generation	Lowest RMS Error	Average RMS Error
1	0.0058351248061235105	0.008098956382062575
2	0.005124453989806542	0.008788577240515025
3	3.112780421164862E-4	0.006815992145352396
4	0.001878277865251978	0.00615390146502372
5	0.0033683109943283512	0.005027418011111019

6	7.906376846118689E-26	0.0055728544464406365
7	5.0558238025193206E-20	0.0013660235242617068
8	2.669238353451669E-13	0.0019471455135151232
9	7.581691939386478E-23	0.0013389815986515853
10	5.125102775087759E-25	0.001963155659988856

A screenshot of the resultant network is shown in Figure 8.1.

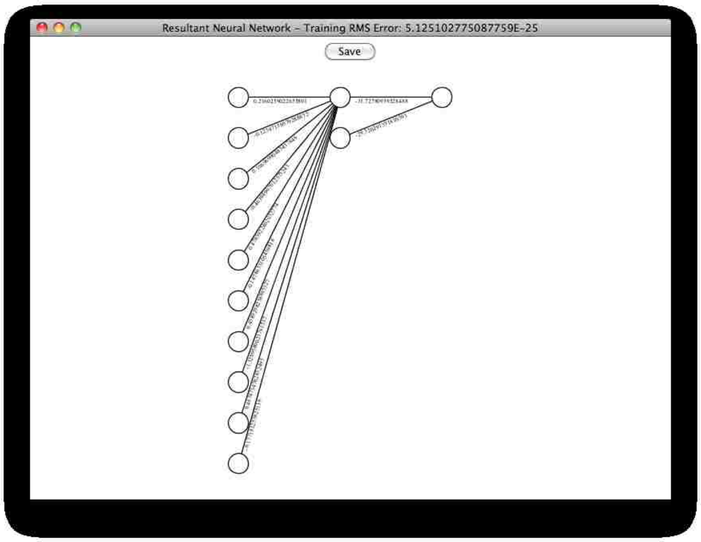
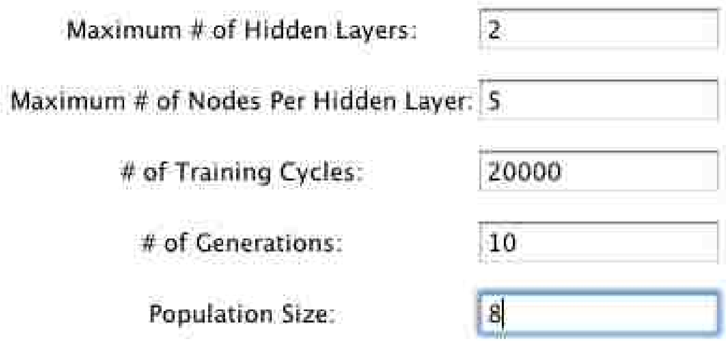

Resultant Neural Network – Training RMS Error: 5.125102775087759E-25
Save

Figure 8.1: Resultant neural network for first blackjack trainer result set.

When run in a simulation of 50,000 games, this neural network won 18,508 games, tied 1,021 games, and lost 30,471 games.

8.1.2 Result Set 2

Table 8.2 shows the results of running the *NNGenerator* software with the following parameters:



Maximum # of Hidden Layers: 2

Maximum # of Nodes Per Hidden Layer: 5

# of Training Cycles: 20000

# of Generations: 10

Population Size: 8

Table 8.2: Blackjack Trainer Test Results 2.

Generation	Lowest RMS Error	Average RMS Error
1	0.0058344177408309154	0.00859211658180253
2	0.0012314974802662725	0.006396436601464739
3	1.832749859906455E-40	0.005518007349382453
4	1.016123779257207E-37	0.00350240342538391
5	1.8813030044936772E-44	0.0029734361621361685
6	5.424008130994838E-57	0.0029949668046492634
7	8.927587203600315E-21	0.0029488302211954707
8	4.066574120048379E-31	0.004020245969844272
9	1.0725743127877895E-18	0.0024621419830630643
10	1.1626319699415398E-18	0.0022126537564833263

A screenshot of the resultant network is shown in Figure 8.2.

When run in a simulation of 50,000 games, this neural network won 19,329 times, tied 2,442 times, and lost 28,229 times.

8.1.3 Result Set 3

Table 8.3 shows the results of running the *NNGenerator* software with the following parameters:

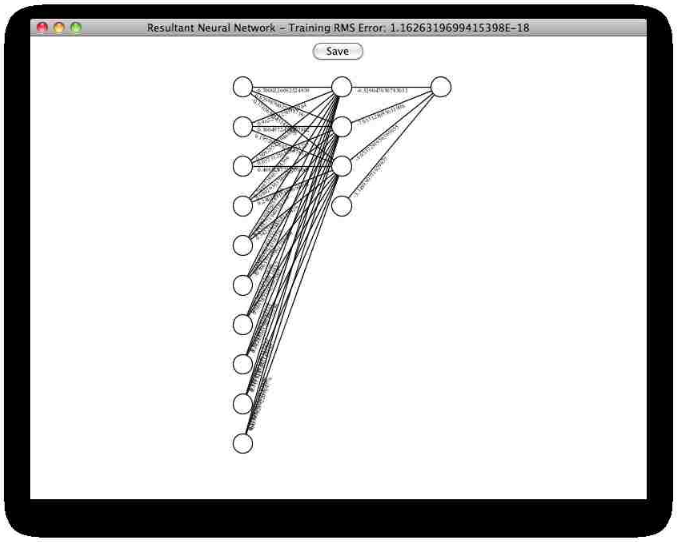
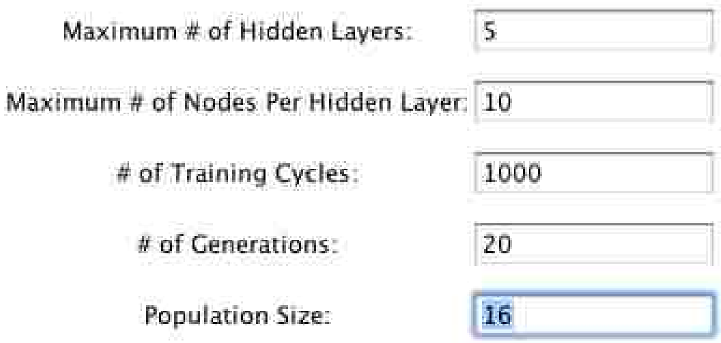

Resultant Neural Network – Training RMS Error: 1.1626319699415398E-18

Figure 8.2: Resultant neural network for second blackjack trainer result set.


Maximum # of Hidden Layers: 5

Maximum # of Nodes Per Hidden Layer: 10

# of Training Cycles: 1000

# of Generations: 20

Population Size: 16

Table 8.3: Blackjack Trainer Test Results 3.

Generation	Lowest RMS Error	Average RMS Error
1	2.3423043495161635E-26	0.006191972674194506
2	7.713975827865027E-49	6.838882427375859E-4
3	5.721639714233759E-57	5.002852391625394E-4
4	3.6604008616284524E-85	9.243851674850428E-4
5	1.3564129229122493E-50	0.002334027724635785
6	7.24117696366482E-75	4.0741007962774844E-4
7	9.677630945830647E-51	0.001490403899462393
8	3.1960667016232624E-39	0.0013689573037240581
9	2.0188362353764715E-22	0.0017899872694982126

10	2.687682804584388E-38	0.0017233290574511732
11	1.00056337236465E-31	7.372035015047415E-4
12	6.273404585352309E-26	0.001600616607656577
13	7.411613627765053E-32	0.0012268277057698822
14	6.255386051894567E-33	8.57762034135896E-4
15	1.026430750315555E-17	5.413214621155046E-4
16	6.318718192501435E-14	0.003901462810525832
17	1.4977732263195537E-13	0.002312616310845903
18	4.729569714675701E-17	0.0021665634257936285
19	2.4822739049956817E-14	0.0014517697603192259
20	3.640692567568787E-8	0.004668787663319438

A screenshot of the resultant network is shown in Figure 8.3.

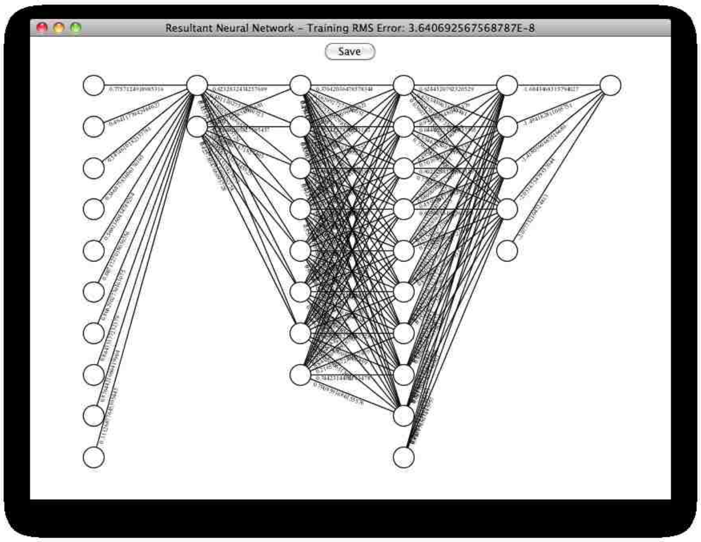
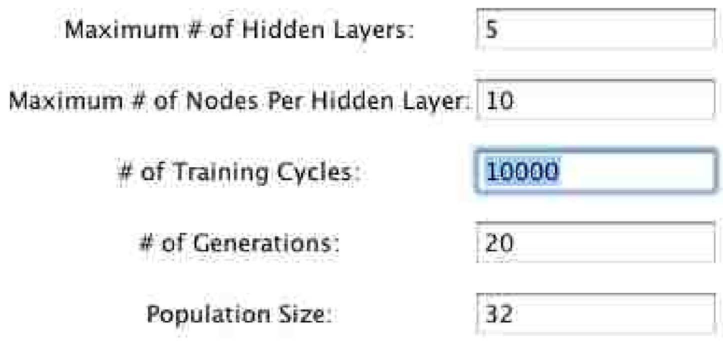

Resultant Neural Network – Training RMS Error: 3.640692567568787E-8
Save

Figure 8.3: Resultant neural network for third blackjack trainer result set.

When run in a simulation of 50,000 games, this neural network won 19,228 games, tied 2,422 games, and lost 28,350 games.

8.1.4 Result Set 4

Table 8.4 shows the results of running the *NNGenerator* software with the following parameters:



Maximum # of Hidden Layers: 5

Maximum # of Nodes Per Hidden Layer: 10

# of Training Cycles: 10000

# of Generations: 20

Population Size: 32

Table 8.4: Blackjack Trainer Test Results 4.

Generation	Lowest RMS Error	Average RMS Error
1	3.1979624169420553E-54	0.008392887198522927
2	3.30216753382502E-43	0.0028339109048548337
3	4.137248662374247E-24	0.002393305467378612
4	2.0628888009514347E-43	0.0027378438647991917
5	3.9401305363732547E-66	0.0019394102403545153
6	1.1489214206448965E-125	0.0019468026470207734
7	5.464385832218055E-75	0.0015102479284637865
8	5.398728180968858E-62	0.0018208462575754194
9	3.992851654112311E-119	3.447240940886299E-4
10	2.0077849357581424E-98	0.0019742919779239016
11	1.0016231392808343E-79	4.1683002715808907E-4
12	1.2092205492144992E-77	2.64295023685051E-4
13	3.0134752103069107E-72	5.718378452389642E-4
14	8.708137045296341E-73	0.002023881777690861
15	3.5891639877316514E-82	8.135181476220314E-4
16	2.8940654582081857E-81	0.0014602246841304558
17	3.4158309414591465E-61	0.0015632776592233925
18	6.919913252262711E-58	0.0015526383370014752
19	1.6585856333254463E-53	0.0015046932600912208
20	2.43435766306799E-44	0.0014879965817727319

A screenshot of the resultant network is shown in Figure 8.4.

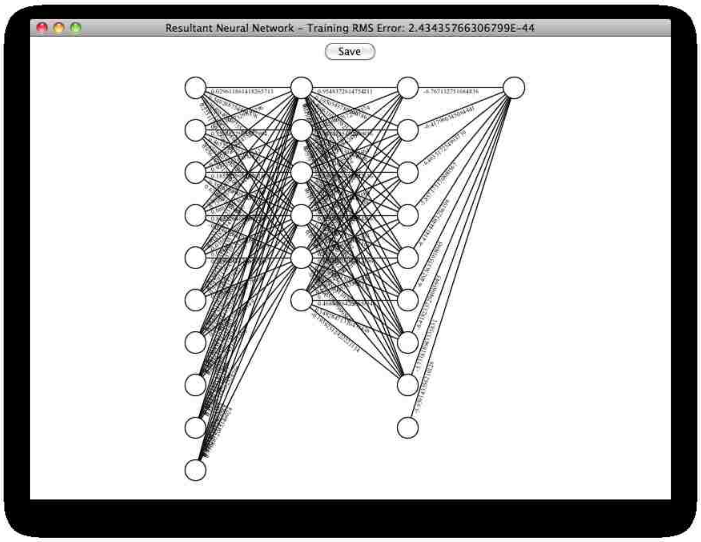

Resultant Neural Network – Training RMS Error: 2.43435766306799E-44
Save

Figure 8.4: Resultant neural network for fourth blackjack trainer result set.

When run in a simulation of 50,000 games, this neural network won 19,259 times, tied 2,515 times, and lost 28,226 games.

8.2 Evaluation

The software was not able to beat a player who plays by dealer rules in any of the four test runs. The first test run had the worst performance with only a 37% win rate. The last three result sets had marginally better win rates, at an average of about 38%. The fact that all of the result sets produce a very similar win rate, even though the network structures are not the same, indicates that the data used to train the network would need to be improved in order to achieve a win rate of over 38%. None of the result sets indicated that the RMS error gets lower with each generation; in fact the trend appears seemingly random.

8.3 Conclusion

Examining the blackjack and OCR result sets leads to some interesting observations. The first is that the generated neural networks with low RMS errors did not necessarily correspond to neural networks that performed well on the test data sets. This is partly due to a weakness in the fitness function, since it only uses the RMS error to determine fitness. It is also due to a poor selection of feature selection, particularly in the OCR trainer.

When determining if the genetic algorithm breeds better neural networks with each successive generation, in the case of the OCR trainer it did. However, in the case of the blackjack trainer, the RMS errors did not appear to trend in any direction with each successive generation.

The results refute the original hypothesis that a neural network structure that performs well can be generated without focusing on the preprocessing and feature vector selection of an input/output data set. The very simplistic features pulled in the OCR set were not enough to even get to a classification success rate of 50%, much less the usually desired high ninety percent ranges. The blackjack features were much better in comparison, however the results indicate that there was a limit to how well a neural network could perform in a simulated game with the features, namely around a 38% win rate. That limit was not able to be exceeded no matter what structure the *NNGenerator* software generated.

Chapter 9

Related Work

Research regarding combining the artificial intelligence methods of genetic algorithms has been done in the past. A brief overview of some of these methods is given here. Note that the terms *architecture*, *topology*, and *structure* are all used interchangeably depending on the author.

9.1 Training Neural Networks with Genetic Algorithms

The most common technique of combining these two powerful learning mechanisms is to use a genetic algorithm to train a neural network with a fixed architecture rather than the standard backpropagation technique.

[MD89] used genetic algorithms to train a neural network for the problem of classifying sonar images. Their approach was to use a genetic algorithm instead of backpropagation to train the weights for a neural network. The chromosomes are the weights of the network encoded as a list of real numbers. Special operators are used for adapting the weights. Their results show that they were able to get better training results using a genetic algorithm for training when compared to standard backpropagation.

9.2 Learning Neural Network Topologies with Genetic Algorithms

A less commonly used method of combining genetic algorithms and neural networks is to use a genetic algorithm to find a neural network topology for a given problem. This is the technique used by the *NNGenerator* software. Other such methods are described briefly here.

[WSB90] used genetic algorithms to search for topologies that were better at learning than feed forward neural networks. The topologies they considered are not layered, are allowed to contain cycles, and can have connections from any node to any other node. They used a 2-bit adder to to show that topologies exist that can learn to add much faster than a feed forward neural network. Figure 9.1 shows the topologies they used. They also ran into the same problem that the *NNGenerator* software has; namely that the time to compute backpropagation is prohibitive when attempting to consider a large number of architectures.

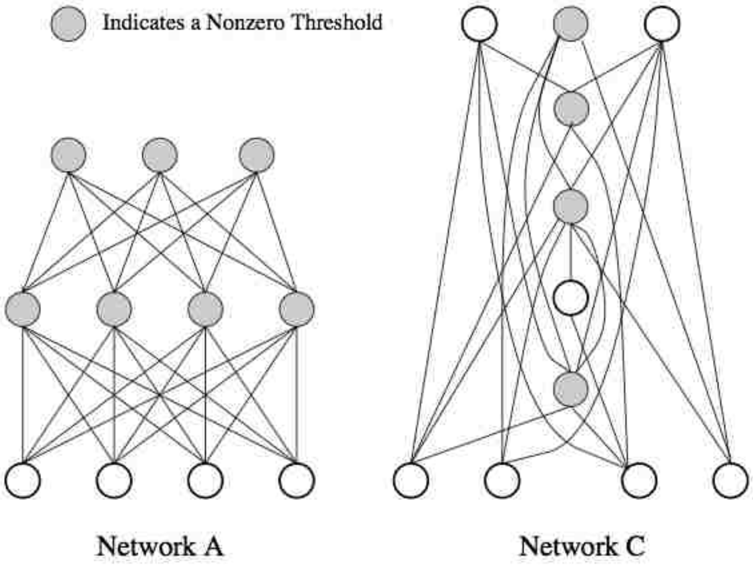


Figure 9.1: Neural Network Topologies for the 2-bit adder problem [WSB90].

[ZM93] also used genetic algorithms to find optimal neural network architectures. Their algorithm is different, however, because it also uses a genetic algorithm for weight training rather than the more common backpropagation method. They allow partial connectivity, and, like [WSB90], allow connectivity between any two nodes. They represent each neural network as a set of N tree structures. Here the weights are binary so that a less expensive hill climbing method can be used since gradient descent is computationally expensive. The fitness function used employs Occam's razor [Dom99] to construct a neural network with minimal complexity. They define the complexity of a neural network in terms of the weights, minimizing the number of weights and the size of each weight. The stopping condition uses both an acceptable fitness function as well as a maximum number of iterations. They used the top 20% of each population for mating. They tested their algorithm in the 4-input parity problem [SA92] and they were able to generate the minimal solution for the problem with their algorithm.

[Kit90] also used genetic algorithms for finding neural network architectures. There approach was different from [WSB90] in that they propose a grammatical encoding of the network architecture rather than a direct mapping of the network architecture into a chromosome. This helps their algorithm to scale better for large networks. They encode the structure as a using an L-system [RS80] which is a mathematical theory commonly used for modeling biological plant development. The encoding they used greatly reduced the length of the chromosomes that represent a neural network architecture. The problem they used was N-X-N encoder/decoder problem with N of length 4 and 8. They were able to show that their L-system encoding performed better on the problem sets than a direct encoding of the architecture.

Chapter 10

Future Work

Although a great deal of time was spent writing the *NNGenerator* software, there is plenty of room for improving upon it.

10.1 Improving the Fitness Function

The fitness function used is simply the RMS error of each neural network. This proved to not be a very good fitness function; there were some cases where the calculated RMS error was low but the neural network could not converge. There were also observations of neural networks that performed better on the test data sets than other neural networks with lower RMS errors. Similar to having a pluggable function that determines the training set for each problem, the fitness function could be abstracted out, allowing any arbitrary fitness function to be used for a particular problem. Perhaps a better fitness function would be one that evaluates a given neural network architecture and weight set against a test data set.

10.2 Speeding Up Backpropagation

The strength of a genetic algorithm cannot be exploited for very small numbers of generations, such as the ones used for the result sets in this paper. The software was meant to run on a large cluster, and when that cluster became unavailable, the software was run on commodity hardware available to me in my own home. However, there are platforms available that provide for large scale, massively parallel computations at a very small cost. One of particular interest is NVIDIA's CUDA platform [Cor, San10]. This platform allows the programmer direct access to the grapics processing unit, or GPU, available on any modern NVIDIA card. The latest Tesla cards boasts:"Delivers up to

515 Gigaflops of double-precision peak performance in each GPU, enabling a single workstation to deliver a Teraflop or more of performance. Single precision peak performance is over a Teraflop per GPU." Backpropagation could be implemented in C using the CUDA toolkit, then wrapped by the Java Native Interface, or JNI [Lia99]. In this way, one or more commodity computers with CUDA enabled GPU's could run the backpropagation algorithm quickly, then JMS could be used to gather the results and give back to the master for breeding.

10.3 Reducing the Encoding Size

The software encodes the chromosomes as Clojure data structures that directly represent a neural network's architecture. Perhaps a more compact encoding, such as the one used in [Kit90] could be used to get a fixed length binary string that is not too large. If this were done, the specialized genetic operations could be replaced with the more generic ones that operation on binary strings. This may help the genetic algorithm find fitter neural networks.

Bibliography

[AIDG95] Hadar I. Avi-Itzhak, Thanh A. Diep, and Harry Garland. High accuracy optical character recognition using neural networks with centroid dithering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:218–224, 1995.

[Bax90] William G. Baxt. Use of an artificial neural network for data analysis in clinical decision-making: The diagnosis of acute coronary occlusion. *Neural Computation*, 2(4):480–489, 1990.

[Cor] NVIDIA Corporation. ”what is cuda?”. `http://www.nvidia.com/object/what_is_cuda_new.html`.

[DH95] D. M. Deaven and K. M. Ho. Molecular geometry optimization with a genetic algorithm. *Phys. Rev. Lett.*, 75(2):288–291, Jul 1995.

[Dom99] Pedro Domingos. The role of occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3:409–425, 1999.

[DPAM02] K Deb, A Pratap, S Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions*, 6(2):182 – 197, 2002.

[Fou] Apache Software Foundation. ActiveMQ Homepage. `http://activemq.apache.org/`.

[Gel89] David Gelernter. Multiple tuple spaces in linda. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer Berlin / Heidelberg, 1989.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[Hica] Rick Hickey. Clojure data structures - part 1. `http://blip.tv/file/707974`.

[Hicb] Rick Hickey. Clojure homepage. `http://clojure.org/`.

[HMOR94] Tim Hill, Leorey Marquez, Marcus O'Connor, and William Remus. Artificial neural network models for forecasting and decision making. *International Journal of Forecasting*, 10(1):5 – 15, 1994.

[HS03] Daniel Heyman and Matthew Sobel. *Stochastic Models in Operations Research, Vol. II: Stochastic Optimization*. Dover Publications, 2003.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984. 10.1007/BF02579150.

[Kit90] Hiroaki Kitano. Designing Neural Networks Using Genetic Algorithms with Graph Generation System. *Complex Systems Journal*, 4:461–476, 1990.

[LeC] Yann LeCun. NEC Research Institute The MNIST Database of handwritten digits. `http://yann.lecun.com/exdb/mnist/index.html`.

[Lee06] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[LGTB97] S. Lawrence, C.L. Giles, Ah Chung Tsoi, and A.D. Back. Face recognition: a convolutional neural-network approach. *Neural Networks, IEEE Transactions*, 8:98–113, 1997.

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall, 1999.

[MD89] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 1*, pages 762–767, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[MSV93] Heinz Mhlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

[mvc88] A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[Oraa] Oracle. Java message service specification. `http://www.oracle.com/technetwork/java/index-jsp-142945.html`.

[Orab] Oracle. Java reflection javadoc. `http://download.oracle.com/docs/cd/E17409_01/javase/6/docs/api/java/lang/reflect/package-summary.html`.

[Orac] Oracle. The java virtual machine specification. `http://java.sun.com/docs/books/jvms/`.

[Orad] Oracle. Swing javadoc. `http://download.oracle.com/javase/6/docs/api/javax/swing/package-frame.html`.

[PGB$^+$05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[Rog94] Galina Rogova. Combining the results of several neural network classifiers. *Neural Networks*, 7(5):777 – 781, 1994.

[Roj96] R. Rojas. *Neural Networks. A Systematic Introduction*. Springer-Verlag, 1996.

[Ros09] John R. Rose. Bytecodes meet combinators: invokedynamic on the jvm. *VMIL '09 Workshop at OOPSLA*, 2009.

[RS80] Grzegorz Rozenberg and Arto Salomaa. *Mathematical Theory of L Systems.* Academic Press, Inc., Orlando, FL, USA, 1980.

[SA92] David G. Stork and James D. Allen. How to solve the n-bit parity problem with two hidden units. *Neural Networks*, 5(6):923 – 926, 1992.

[San10] Jason Sanders. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 2010.

[SG93] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of lisp. *SIGPLAN Not.*, 28:231–270, March 1993.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997. 10.1007/s004460050028.

[TK09] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Fourth Edition.* Academic Press, 2009.

[TS02] Shaun Terry and Terry Shawn. *Enterprise JMS Programming.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[Wei] Eric W. Weisstein. "method of steepest descent." from mathworld–a wolfram web resource.". `http://mathworld.wolfram.com/MethodofSteepestDescent.html`.

[Wel02] Peter H. Welch. Process oriented design for java: Concurrency for all. *ICCS 02: Proceedings of the International Conference on Computational Science-Part II*, 8:687, 2002.

[WSB90] D Whitley, T Starkweather, and C Bogart. Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing*, 14(3):347 – 361, 1990.

[ZM93] B Zhang and H Muhlenbein. Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex Systems*, 7(3):199 – 220, 1993.

Vita

Graduate College
University of Nevada, Las Vegas

Jason Lee Hurt

Degrees:

Bachelor of Science in Computer Science 2004
University of Nevada Las Vegas

Thesis Title: Automating Construction and Selection of a Neural Network Using Stochastic Optimization

Thesis Examination Committee:

Chairperson, Dr. Jan Bækguard Pedersen, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Committee Member, Dr. Kazem Taghva, Ph.D.
Graduate Faculty Representative, Dr. Aly Said, Ph.D.