The use of neural nets for classification of multi-variable systems has been found effective where conventional formula based systems have failed. Unfortunately, the back propagation neural net approach gives poor results when trained upon a data set that does not properly map the input space. This study created techniques for the auto-generation of additional training inputs to improve the abilities of the final neural net. The additional inputs were created from an evaluation of inadequacies in the initial trained network. These inadequacies are determined by the autonomous evaluation of the trained network’s output characteristics. The additional inputs are presented to the human expert for classification and then supplemented to the initial training set. Successful implementation of these methods would enable neural net programmers to spend less effort in the creation of training data sets, for inadequacies would be self extracted for clarification.
IMPROVEMENT OF CLASSIFICATION NEURAL NETWORKS THROUGH AUTO-GENERATION OF SUPPLEMENTAL TRAINING INPUTS

By

David K Windmueller

Thesis submitted to the Graduate Faculty of Christopher Newport University in partial fulfillment of the requirements for the degree of Master of Science in Applied Physics and Computer Science 2001

Approved:
David Hibler, Chairman
John Hardie
Lynn Lambert
Acknowledgments

I would like to thank my advisor, Dr. David Hibler for all of his help along the long road to completion. From the initial decision of a topic to putting together the final paper, Dr. Hibler has been a constant source of ideas and motivation. It has been a pleasure to work with such a learned person whose life has been shaped by his curiosity. I would like to thank my other committee members, Dr. Lynn Lambert and Dr. John Hardie. Their patience has been invaluable and I have enjoyed working together with them during my time at Christopher Newport University. A note of thanks goes to the system administration in the Computer Science department whose hard work is evident due to my lack of troubles during the whole process.

I would like to also express my appreciation to my friends and family who have been so supportive during the writing of this paper.\[1\] I would also like to thank my cat, Mr. Jack Kerouac, for his constant support throughout this whole process and helping me to see a misuse of a pointer in some of the back propagation code.

\[1\] This is as close as I will ever get to having a camera on me in the end zone after spiking a spheroid. I’d like to take this special opportunity to say...Hi Mom!
# Contents

**List of Tables**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>v</td>
</tr>
</tbody>
</table>

**List of Figures**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>vi</td>
</tr>
</tbody>
</table>

**Chapter 1 Introduction**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Statement of Problem</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Purpose of Study</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Research Questions</td>
<td>3</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Input Addition</td>
<td>3</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Sorting Methods</td>
<td>3</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Auto-Generation Methods</td>
<td>4</td>
</tr>
</tbody>
</table>

**Chapter 2 Review of Literature**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

**Chapter 3 Methodology**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Data</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Neural Network Construction</td>
<td>11</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Backpropagation Neural Network</td>
<td>11</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Data Extraction</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Training File Distortions</td>
<td>14</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Averaging Values</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Addition of Random Noise</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>Sorting</td>
<td>16</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Sorting Algorithms</td>
<td>16</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Batch Processing</td>
<td>16</td>
</tr>
<tr>
<td>3.5</td>
<td>User Classification</td>
<td>17</td>
</tr>
</tbody>
</table>

**Chapter 4 Results**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Normal Processing</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Random Noise Auto-Generation</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>Average Input Auto-Generation</td>
<td>23</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Classified Input Patterns From Random Noise Generation . . . . . . . . . . 21
4.2 Classified Input Patterns From Random Noise Generation . . . . . . . . . . 25
4.3 Classified Input Patterns From Average Input Generation . . . . . . . . . . . 25
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic Neural Network</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Random Pixel Display of Training Data</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Classification Software</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Normal Neural Network Training</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Neural Network Training with 10% Noise Entries: Method 1</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>Neural Network Training with 20% Noise Entries: Method 1</td>
<td>23</td>
</tr>
<tr>
<td>4.4</td>
<td>Neural Network Training with 30% Noise Entries: Method 1</td>
<td>24</td>
</tr>
<tr>
<td>4.5</td>
<td>Neural Network Training with 10% Noise Entries: Method 2</td>
<td>26</td>
</tr>
<tr>
<td>4.6</td>
<td>Neural Network Training with 20% Noise Entries: Method 2</td>
<td>27</td>
</tr>
<tr>
<td>4.7</td>
<td>Neural Network Training with 30% Noise Entries: Method 2</td>
<td>28</td>
</tr>
<tr>
<td>4.8</td>
<td>Neural Network Training with Averaged Input Patterns</td>
<td>29</td>
</tr>
<tr>
<td>4.9</td>
<td>Visual Representation of Extracted Seven Digit</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Neural Network Training with no additional input patterns</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Neural Network Training 10% Noise Entries: Method 2</td>
<td>33</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The back propagation neural network is a powerful software tool that is used to control or classify systems that are not easily described by mathematical formulae. The artificial neural network has the ability to train itself from a set of input/output pairs and then extrapolate its findings on input patterns that it has never seen before. This makes an effective approach on systems that have relationships that are difficult to accurately describe with a set of rules. The properly trained neural network is also able to successfully classify non-precise data input patterns that deviate from the original training set. As with most systems, the quality of the output is determined by the quality of the input. The classification abilities of the final neural net are dictated by the quality of the training set.

The normal process for training a back propagation neural network involves a number of steps. The programmer needs to take the initial set of input/output data and separate it into two groups. The first set is used as a training set that shapes the internal weights. The second set of data is processed by the trained net as an independent check of its accuracy for the network has never seen any of this data. The output of the trained net is compared to the desired outputs of the test set as fair indication of error.
1.1 Statement of Problem

After finding that the network is unable to properly classify the test set, the programmer can continue to train the network on the initial data in hopes of lowering the test set data error. Another option is to find the specific representations of input patterns where the net is failing and add more examples to the training set. These specific examples would help to clarify classifications in the problem areas.

Currently, this iterative process is a time consuming task, which limits the ability of the neural net to the skills of the user to create and improve the initial training set. Also data sets can be too complicated to allow the programmer to visualize the needed input patterns to improve classification abilities.

1.2 Purpose of Study

This study will create and evaluate methods for the auto-generation of input patterns for improving the overall quality of the final training set. The created patterns will not be generated from deficiencies discovered from the use of the test set, rather they will be found from the analysis of the trained network itself. The input patterns will be created directly from the neural network or from data created by running alterations of the original test set through the network and selecting choice patterns by a sorting process. These input patterns will be added to the original training set and the network will be trained further.

The goal of this study is to compare the quality of various methods the researcher wrote for the auto-generation of input data for a classification neural network. The improvements found might lead to a change in the requirements for creating a successful training data set. With the ability to find weak areas after training, the neural net system can tolerate an initial data set of lesser quality.
1.3 Research Questions

To achieve the goal of this study, a number of research questions must be answered. The first is a simple test to find if the addition of new data pairs will actually improve the total abilities of the network. The next question deals with the effectiveness of various methods for selecting examples from a large generated set. The final question compares the abilities of a number of auto-generation techniques to find which offers the greatest improvements to the overall performance of the network. To answer each of these question, I wrote software for each process in the auto-generation method.

1.3.1 Will the addition of generated input data improve the classification abilities of the original neural net?

The answer to this question leads to two courses of research. If the response of a fully trained network is not improved by the introduction of new data, comparing the various auto-generation methods would be difficult. If this is found to be true, the net could be trained significantly less on the original data and then the auto-generation techniques would be applied. This technique would investigate if the total number of training iterations could be lowered when compared to the full training of network with the initial test set.

1.3.2 Which method of sorting auto-generated data will provide better results?

A number of the auto-generation methods that were created for this research provide a greater number of input patterns than is desired by the human expert for evaluation. To minimize the burden on the human, various methods will be used to select the ones that will possibly have the greatest positive effect in reducing the classification error. The various
selections of the sorting routines will be used and compared for output characteristics.

1.3.3 Which method of creating auto-generated data will provide the best inputs for the improvement of the net?

This is the fundamental question of the research. The evaluation of the auto-generation methods leads to a greater understanding of data set construction techniques for neural applications. Also it indicates a level of practicality of various methods. The research demonstrates the amount of accuracy that is gained with each method versus the amount of extra effort that is needed from both the human and the computer.
Chapter 2

Review of Literature

The Artificial Neural Network is a software configuration used to emulate a function that learns. There are many different types that are suited for a variety of tasks. The back propagation neural network is well designed for classification applications. The software mathematically produces one or many outputs from an input of one or many values. The network design allows for many internal variables to be adjusted with training to facilitate classification of many different input patterns. The internals of the artificial neural net dictate the method of programming needed to allow the structure to learn.

The foundation of the network is the neuron unit. The neuron itself is similar to its biological counterpart in a number of ways. The first is its input and output structure. Both neurons have a variable number of inputs and a solitary output. The premise of its operation is that all of the inputs are collected and evaluated. The evaluation determines what to send as an output from the individual neuron. The evaluation process of the software and that of the biological neurons share the concept of thresholds. The output of the neuron will be fired if the sum of the inputs is greater than a set threshold. In the software neuron, the amount that each input into the neuron adds to the sum of the function is dependent
on a weighted multiplier. [Ska96]

These neurons are linked together to create a structure that accepts the numerical inputs and gives an appropriate output after each layer performs their sequential calculations. The conversion of the inputs into outputs is called forward propagation. A generic view of a complete neural network is shown in Figure 2.1. The first layer is called the input layer and this is where the input pattern is shown to the network. Each value of the pattern is multiplied by a separate weight for each node of the next layer. This layer is considered the hidden layer. The values of the modified inputs are summed up in each neuron and the neuron will send a high value if the sum is greater than the threshold or bias value. The output of the each middle layer node is sent to each outer layer node after being multiplied by its appropriate weight. The operation of the output node is exactly like the middle layer nodes.

![Figure 2.1: Basic Neural Network](image)

To train a neural network, the multipliers are varied using a gradient descent technique.
to minimize error. After the input pattern has been forward propagated through each layer, an error calculation is performed which computes a quantitative indication of which nodes are not performing properly to get the desired output for the specific input pattern. The errors are back propagated to each layer using the chain rule and then a weight adjustment strategy is used to train the network for better performance. The training process can be quite time consuming, but once a neural net is trained, the forward propagation calculations are relatively fast.

There are many obstacles when using a neural network. Among those are layer geometry, squashing function selection, and training procedures. A knowledge of these concepts will save many iterations when changing the net construction while trying to optimize its performance. The network needs to be designed to fit the task that it is intending to do.

One of the most important features is layer geometry. The actual number of neurons for each layer is crucial for if there are too many internal nodes the network will become quite specialized and not be able to extrapolate very well to data it has never seen. If there are too few nodes, the network will never be able to lower its overall error. Another important part when considering layer geometry is how the system will see the input data.

The United States Postal Service uses neural networks to recognize zip code data [Win92]. The network is designed in stages in which a smaller area of the digit is scanned and sent to separate neural nets. The output of each of the smaller networks is sent to another layer of networks which leads to the final network. By breaking up the image parsing into a number of areas, it lowers the total number of weights needed and thus lowers training time. These specialized area convey information only about the immediate area around it and not the whole character [HKP91].

A number of improvements to the back propagation techniques have been the goal of different research projects. These projects have focused on many elements of the network,
including the evaluation function itself, variations of network construction and also training parameters such as momentum. Pertaining to the construction of the input data set, some work has been performed studying the effect of the selective introduction of input pairs for training to the network in hopes of maintaining the network’s generalization abilities. [1134]

The learning rate is another element that is important for proper training. Due to use of gradient descent, the system may possibly settle on a local minima of the error function and not the best possible solution. Using a dynamic learning rate, the system may be able to skip over smaller local minima and find a position with a lower error level.

All of these factors are to be considered if trying to find the optimum neural network for a particular application. This research is using a simple network construction that is not optimized in hopes of making any performance changes more drastic.
Chapter 3

Methodology

This study involved processing a number of steps to answer the research questions. The first item involved data selection for the neural network. The data was used to train the net and also act as a starting point for some of the auto-generation techniques. The neural network software was created using a basic back propagation template. This software was supplemented with extra features that were used for pattern extraction after the network has learned the characteristics of the training data.

The trained network was also used to process the other forms of auto-generated inputs. The output of the network’s processing was sent through a sorter routine. This sorting process selected a limited number of input patterns that possibly demonstrate a weakness in the neural net’s classification abilities. The subset of input patterns was presented to the user for expert classification. These classified lines of data supplement the original data set and the network was trained further. The whole process was monitored by recording the error rate of the neural network with the training and testing set of data.
3.1 Data

To help demonstrate the effectiveness of auto-generated input patterns, a data set was needed that would be challenging for a basic neural network structure. The data was acquired from a Turkish database [Kay95] with a variety of hand-written representations of the 10 numeric digits. The representations include such variations as crosses in sevens and the different styles of fours. Each digit is constructed of a 32x32 bitmap that is viewed as an 8x8 grid of squares. Each of the squares that make up the grid are a 4x4 space thus they have 16 pixels each. They are represented in the datafile by a number from 0 to 16 which indicates how many pixels in that particular square are on or darkened. The actual location of darkened pixels within the square is unknown. The final number of the data line is the actual digit (0-9) represented by the previous sixty-four numbers. The optdigits.tra file which contains 3823 examples was used for training. With 1797 examples, optdigits.tes file was used for independent checking of how well the neural net could respond to characters that it was not trained on.

Since the exact location of the pixels is not known inside of each square, the data is translated into a visual representation using two methods. The first method uses random placement of pixels in the square up to the number that is needed. An example of this can be found in Figure 3.1. The second method uses gray scale values for the whole square. This method is described in the User Classification portion (Section 3.5).

A small program, adaptfile.c, modifies the format of the data so it can be used by the neural net software (See Appendix A.2.1). Two modifications to the original datafile were needed to allow the neural network software to read it. These modifications replaced the comma delineator with a space and changed the digit classification from a single digit from 0-9 into the ten separate desired outputs from the network, thus a classification of seven is
3.2 Neural Network Construction

3.2.1 Backpropagation Neural Network

The neural network is composed of a 3 layer structure with an input layer of 64 nodes. The 64 inputs correspond to the total number of squares that construct a graphic of an input character. The middle layer is made of 28 hidden nodes and the final layer has 10 output nodes. Each of the outputs corresponds to a digit ranging from 0-9. The number of middle layer nodes was selected arbitrarily to allow for enough flexibility to enable learning, but not too many to prevent proper extrapolation. This number was loosely based upon having
approximately half of the difference of nodes between the input and output layer.

The code to construct the network is mainly located in the layer.cc and layer.h files. This was based upon the source code found in work by Rao and Rao [RR99]. This code was altered to properly perform back propagation using standard equations and allow for various extra functions that are used throughout this research. These changes included the addition of a bias node for each neuron and adding data extraction abilities. Another change was the implementation of more standard formulas for weight adjustment that are suggested by Winston [Win92]. The following formulas are used for neural net processing in this research.

\[ o_j = f\left(\sum_i (w_{i\rightarrow j} o_i) + b_j\right) \]  
(3.1)

where

- \( o_j \) = Output of the node \( j \) in layer \( j \)
- \( w_{i\rightarrow j} \) = Individual weight between each layer \( i \) and \( j \) node
- \( o_i \) = Output of node \( i \) in layer \( i \)
- \( b_j \) = Bias for node \( j \) in layer \( j \)
- \( f(x) = \frac{1}{1+e^{-x}} \)

\[ \beta_z = d_z - o_z \]  
(3.2)

where

- \( \beta_z \) = Output Error for Output Layer node \( z \)
- \( d_z \) = Desired Output for Output Layer node \( z \)
- \( o_z \) = Actual Output for Output Layer node \( z \)

\[ \beta_j = \sum_k w_{j\rightarrow k} o_k (1 - o_k) \beta_k \]  
(3.3)
where

\[ \beta_j = \text{Output Error for Middle Layer node } j \]
\[ w_{j \rightarrow k} = \text{Individual weight between each layer } j \text{ and } k \text{ node} \]
\[ o_k = \text{Output of the node } k \text{ in layer } k \]
\[ \beta_k = \text{Output Error for Layer node } k \]

\[ \Delta w_{i \rightarrow j} = r o_i o_j (1 - o_j) \beta_j \]  \hspace{1cm} (3.4)

where

\[ \Delta w_{i \rightarrow j} = \text{Change to the individual weight} \]
\[ r = \text{Learning Rate} \]
\[ o_i = \text{Output of the node } i \text{ in layer } i \]
\[ o_j = \text{Output of the node } j \text{ in layer } j \]
\[ \beta_j = \text{Output Error for Layer node } j \]

The code uses an object oriented design to allow for the various types of layers to share similar functionality when needed. The source code for the network and layer classes can be found in layer.h (Appendix A.1.1) and layer.cc (Appendix A.1.2). This software contains the network class which contains a number of instantiations of layer objects. There are three types of layer classes, input, output, and a middle layer. The inputs and outputs of the layers are interconnected by the use of pointers to the various data structures.

The main program that uses the network object is backprop.cc (Appendix A.1.3). This code instantiates the network and uses a text-based menu to select the proper mode of operation. The network can be trained or just process a data file using stored weights. One feature was added to the base code to allow for command line processing of data with no user interaction. This allows for batch processing of large data files (Section 3.4.2).

3.2.2 Data Extraction

One method of auto-generating data is the direct use of the trained neural network itself. The software was modified to allow for a data extraction mode. This mode creates
an extra layer with one node before the normal $64 \rightarrow 28 \rightarrow 10$ node structure. This node has a fixed output of 1 and the bias node has been zeroed. Each line of input data with the 64 values is placed in the weights between the additional layer and the old input layer. The expected value is a series of ten values that are the desired outputs. The input pattern is processed and the error for the various layers are calculated. The back propagation method is applied except the only weights that are changed are input values which correlate to a regular input pattern. The network is ‘trained’ until the input values give the desired outputs from the network.

For auto-generation, a series of desired values were picked as confusing outputs so that when the extracted input patterns were classified, they would help to add clarity to the training file. The confusing outputs had values that contained two outputs with .5 values, indicating that the network could not differentiate between the two characters.

### 3.3 Training File Distortions

Another method of creating auto-generated data is to alter the original training data set and process it through the partially trained neural network. The output of the neural network would be put through sorting software to select the values that would most likely help the training process.

#### 3.3.1 Averaging Values

A simple method of generating new input patterns is creating average values of each pair of the original data set. The software to do this is found in Appendix A.2.2. From a set of $n$ input/output pairs, this method creates $\frac{1}{2}(n(n - 1))$ new input patterns for processing through the neural network. This software creates many different characters, many of which
are not expected to be classified by the network.

### 3.3.2 Addition of Random Noise

Another method of creating new input patterns from the original training file is to add noise to the original data. `Addnoise.c` (Appendix A.2.3) is used to add a fixed percentage of random noise to each input pattern of the training file. The noise percentage is entered by the user and the program will translate this as a certain number of bits to flip.

Since there are 64 squares with 0-16 bits, there are 1024 possible changes that can occur. A 10% noise would be 102 bits would change state.

The original information is not so specific that you can simply change the state of a particular bit so the software narrows down the changes to each of the 64 squares with each one having no more than 16 changes. Once the number of changes are determined per square, the type of change is determined (darker or lighter) thus the change will be \( \pm 16 \).

When added to the original number of pixels that are darkened, the end result will range from -16 to +32.

To determine the new value of pixels to be darkened, the following rules are used:

\[
x = \begin{cases} 
-x, & \text{if } x < 0 \\
x, & \text{if } 0 \leq x \leq 16 \\
32 - x, & \text{if } x > 16 
\end{cases}
\]

These formulas assure a smooth transition for any amount of change from any starting point of bits in that particular square.
3.4 Sorting

Some of the auto-generation techniques produce more input patterns than are desired for human classification. I had selected one hundred as the maximum number of entries to be classified. This was approximately 3% of the size of the original training set. The sorting process would take the generated input patterns and send them through the partially trained neural net. The sorter would look at the output file of the neural net and give the top one hundred input patterns according to a certain sorting algorithm. The basic source code for the sorting library can be found in Appendix A.3.1 with Sort.h and Sort.cc. Sortmain.cc uses these objects to sort one file.

3.4.1 Sorting Algorithms

There were two algorithms that were used to sort the output file in different manners to produce input patterns for human classification. The first one ranked the outputs of the neural net by their second highest classification value. Having a high second value indicates that the neural net was having some trouble differentiating between the highest classification and second highest classification value. The second method simply ranked the outputs by their highest value output values. The selected ones were those that had the smallest high value in the set. This indicates the neural net’s lack of ability to classify the entry at all. Unfortunately, this method would not be useful on auto-generation techniques that create many unclassifiable input patterns such as the input averaging method.

3.4.2 Batch Processing

When an auto-generation technique produces a large number of possible input patterns, it was not practical to write software to sort extremely large numbers of results simultaneously.
Batchsort.cc (Appendix A.3.1) was created to process the created input patterns in smaller groups while keeping the highest input values according to the sorting process after each iteration. This method was able to sort any size of input files if given enough time.

### 3.5 User Classification

Once the auto-generation of input patterns has been processed, the human expert needed a visual method for classifying the data. The TCL/TK program classifydata.tcl (Appendix A.3.2) was written to create the necessary outputs for neural net training from auto-generated data and the user classification. The program reads in the input file and shows the information visually. Since the exact location of the pixels within the square are not known, the digits have been represented by two methods. The first method uses random placement of pixels in the square up to the number that is needed. The second method uses gray scale values for the whole square. (See Figure 3.2)

The user can select a digit that the image represents or unknown. If the image is unknown, it is discarded. If it is able to be classified, the original pattern and its classification are send to standard output. This is directed to a file for addition to the training set for further training.
Figure 3.2: Classification Software
Chapter 4

Results

4.1 Normal Processing

The neural net was first trained with normal back propagation techniques for 300 iterations. Figure 4.1 displays the progress of the training and also how well the neural net performed on the test set at the same time. The error percentage was based upon how many characters the network correctly identified using the following basis:

- The correct entry had to have an output of greater than 0.95 (Desired output was 1.0)
- All of the incorrect entries had to have a value of less than 0.05 (Desired output was 0.0)

If the character violated either of these criteria, the classification was considered incorrect for that character. This technique gives a maximum error total of .5 for the entire desired output. This error calculation was used to accentuate the improvements of the neural network when adding additional input patterns and is only used for the graph output.
The error calculations that are performed for back propagation are found in Equations 3.1 – 3.4 (page 12).

4.2 Random Noise Auto-Generation

The first auto-generation technique attempted was the addition of random noise to the original data set. The neural net was first trained through 150 iterations with the training file alone. Three different percentages of noise were applied to the training set and then each noisy version of the training set was run through the neural net. The first sorting
algorithm was applied to the output of the neural net. This sorting routine ranks the input lines by the value of the second highest output. The sorting program listed 100 input values for the classification software. I classified the input value as well as possible and concatenated the new input/output pairs with the initial training set. The network was then trained another 150 iterations. Table 4.1 shows the amounts of input patterns that could be classified from the first sorting method.

<table>
<thead>
<tr>
<th>Percentage of Noise Added</th>
<th>Number of Input Patterns</th>
<th>Number Classified</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>20%</td>
<td>100</td>
<td>84</td>
</tr>
<tr>
<td>30%</td>
<td>100</td>
<td>60</td>
</tr>
</tbody>
</table>

Figure 4.2 through Figure 4.4 show the training of the neural network with the additional input / output pairs added at the 150 iteration mark. There is very little change of the network’s ability with these additional input patterns. With the addition of the 60 input / output pairs from the group that had 30% noise added, the network notably performed worse initially and trained close to the level of other data sets.

The network demonstrated its most difficult differentiation through the input patterns that were presented to the human classifier from the sorting routine. Even with the European style of hand writing, most of the classifications were the digit 7 which was confused with the digit 2. Also the difference between the digit 1 and the digit 2 can be confusing if the 1 is drawn with a large base and cap. As the data got noisier, all digits were represented.

The next attempt was to try running the noisy data with the second sorting method. This method ranks the patterns by the value of their highest output. The patterns selected are those with the lowest high output value. This method attempts to find patterns that
Figure 4.2: Neural Network Training with 10% Noise Entries: Method 1

150 iterations with no additional input patterns and 150 iterations with additional input/output pairs (Second Highest Output Sorting Method)

the system is having trouble categorizing as any character as opposed to the first sorting method which detects if the network is demonstrating some trouble differentiating between two different classifications. Figure 4.5 through Figure 4.7 shows the results of adding classified pairs from this sorting method at the various random levels. This sorting method found many more input patterns that were unrecognizable as shown by Table 4.2. Even though the number of classification pairs were smaller than those produced by method one, their introduction to the training set produced a more obvious disturbance in the training
errors initially. After 150 iterations, these input pairs were found to slightly improve the classification abilities of the network with the test file in the case of the 10% random noise.

### 4.3 Average Input Auto-Generation

This method uses the input file to create average values from every possible pairing of input lines from the training set. With over three thousand entries in the training set, this effort took a considerable amount of time to generate and produced over seven million entries for
processing. After running all of these values through the batch sorting software which also took quite a while, the 100 entries were selected and classified. The new entries were added to the training set and the system was trained for another 150 iterations. The results of this process can be found in Figure 4.3.

The second method of sorting which simply selects characters with the lowest high output value was used on the averaged data. Since this method produces a lot of undesirable characters, the top 100 characters were difficult to classify and produced only 2 input/output pairs.
Table 4.2: Classified Input Patterns From Random Noise Generation

<table>
<thead>
<tr>
<th>Percentage of Noise Added</th>
<th>Number of Input Patterns</th>
<th>Number Classified</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>20%</td>
<td>100</td>
<td>56</td>
</tr>
<tr>
<td>30%</td>
<td>100</td>
<td>28</td>
</tr>
</tbody>
</table>

pairs. I did not attempt to train the network with the addition of these two characters.

Table 4.3: Classified Input Patterns From Average Input Generation

<table>
<thead>
<tr>
<th>Method of Sorting</th>
<th>Number of Input Patterns</th>
<th>Number Classified</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd Highest Output</td>
<td>100</td>
<td>96</td>
</tr>
<tr>
<td>Lowest High Value</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>

4.4 Inputs From Data Extraction

The data extraction portion was one of the most difficult tasks of this research and provided the least amount of data. Many modifications to the basic neural network software were made to be able to perform this task. These modifications included creating separate internal calculation functions for the new layer that was used for weight manipulation of the input pattern and automatic network geometry changes upon mode selection.

After this programming was performed successfully, I was able to train a basic pattern to give a desired output on a number of desired input patterns. The first set of desired input patterns that were used consisted of an initial position of all 64 squares set to the middle value of 8 and a desired output of 1 for each digit. Each of these represented a pure digit. After 10000 iterations of each pattern, it was found that some patterns had trained down to less than 1% error for all of the outputs. Not all digits were able to be generated. Upon
150 iterations with no additional input patterns and 150 iterations with additional input/output pairs (Lowest High Output Sorting Method).

Figure 4.5: Neural Network Training with 10% Noise Entries: Method 2

Although the output of the data indicated that the neural network had definitely classified the input pattern as a certain number, it could not be detected visually. Figure 4.9 displays the visual output of an extracted seven output pattern. The regular test file was ran through the neural net to ensure that the network was working properly and its classification abilities were as expected. The extracted output pattern was run through the classifydata.tcl program, an unfortunate outcome was found with the visual indication of the data.
neural network through the same mechanism as the test file. It produced the same positive identification of the digit “7” and all other outputs were close to zero.

I attempted to extract usable data by changing the initial pattern that the input weights were started at. The input patterns were changed to high values and low values. This produced different designs that still were classified as the various digits, but were just as unrecognizable as the data constructed with the moderate values. The final attempt was to create a generic character that was an amalgamation of all of the digits. This template...
pattern was not as successful at creating input patterns that produced the desired outputs as starting from a solid color.

To investigate this extraction process further, I placed some of the characters with their proper classification from the test file into the desired output file. The network classified most of the characters properly and proceeded to change the input characters slightly to better match the desire output classification. The network seemed to change some of the characters to the minimum information to characterize the digit.
Figure 4.8: Neural Network Training with Averaged Input Patterns

150 iterations with no additional input patterns and 150 iterations with additional input/output pairs. (Second Highest Output Sorting Method)

The extraction method was not used to create the input patterns from confused outputs due to the fact that the input patterns that were created using ‘non-confusing’ outputs were difficult enough to make classification impossible.
Figure 4.9: Visual Representation of Extracted Seven Digit
Chapter 5

Conclusions

This research concludes after the testing of various auto-generation techniques. Overall, no method seemed to dramatically improve the neural network’s classification capabilities. There are many possible reasons for the lack of change that might be the subject of further research. These may be related to a variety of fixed parameters in this study such as network geometry, data set, and classification techniques. These are described fully by answering the initial research questions.

5.1 Input Pattern Addition

In answering the first research question, I found that there was no substantial proof that the methods of auto-generation have aided the training process of the neural network. There was only one case in which one could see an improvement through the introduction of additional input patterns. This was the method that added a slight bit of random noise (10%) and used the second method of sorting (Lowest High Value). This method and a comparison run of with no additional input patterns was performed for 1000 iterations. It
was noted that the network with the additional input patterns performed slightly better (1.2%) on the test file even though the training error was greater. (Figure 5.1 and 5.2)

![Neural Network Training with no additional input patterns](image)

Figure 5.1: Neural Network Training with no additional input patterns

1000 iterations

This seems to validate the concept that adding data with some noise or jitter on top of the original training set will enable the neural net to extrapolate better. [HK92]

5.2 Sorting Methods

The second research question asks to compare the various sorting methods. The two methods showed different characteristics that indicated that their use should be limited by the
Figure 5.2: Neural Network Training 10% Noise Entries : Method 2
150 iterations with no additional input patterns and 850 iterations with additional input/output pairs (Lowest High Output Sorting Method)

108 particular method of auto-generation that they are sorting. It was obvious that the Lowest High Method was inappropriate to use in a case where many entries would be created that had no real meaning. The averaged data method is an excellent example of this. Another concept to think about is what type of classification problems is the sorting method detecting. The highest second value method detected areas where two digits had similar classifications. The predominant characters that was represented to the human classifier were the seven and two digits. Understanding this fact, a researcher could use this sorting
method to find which input patterns have the largest differentiation problem and change the input pattern standards for one of the two patterns. This would probably be difficult to implement in the case of hand written characters, but might help when designing a new product that would need computer classification.

Another interesting result came from one of the sorting methods. Average method’s use of second high sorting demonstrated an interesting ability to find actual characters out of a large field of poor input patterns. 96% of the characters selected by this sorting method could be identified as a specific character by the human expert. This method could not be used as a method for simply detecting whether or not an input pattern was a character for the group of input patterns that cause a high second level is a specialized subset of all characters.

5.3 Auto-Generation Methods

The final research question compares the auto-generation methods themselves. The various methods have demonstrated a variety of strengths and weaknesses. The simplest method is the averaging of the input patterns. These additional input patterns can be created while the initial network is being trained since it is solely derived from the training file itself. Unfortunately, this method produces quite a large number of input patterns and can take quite a while to do so. With a data set of 3823 input patterns, the averaging method created over seven million patterns. I also had to write additional software to process the large number of input patterns through the network and sorting in batches.

Another problem with the averaging method deals with the position of the created input patterns inside the input space of the data set. With the large number of possible input patterns (17^64), the averages between two characters were likely to take the neural network
into an area that the network would not be able to extrapolate properly. The goal of the additional sorting process was to find only those patterns that were on the cusp between two input space classifications that were intersecting. Unfortunately most generated inputs were unable to be used. This method of generation might be more useful when the input space is dominated by proper classifications.

The next method involved the addition of random noise to the network. When using this method, I found no cases where the additional noise made a different classification than the original one from the training file. The addition of patterns with 10% noise did seem to help but became destructive as the noise level increased. Both sorting methods found slightly different areas of difficulty for the neural network and thus showed different digits for classification. Both methods seemed to have the most difficulty with the numeral seven.

The final method of auto-generation was the data extraction from the trained neural network. This method was not successful due to the rather large range of possible input patterns versus the actual amounts of input patterns that the system was trained on. With many areas of uncharted input space, the network was free to find an input that satisfied the output requirements, but had little visual meaning in the context of digits.

5.4 Areas for Future Research

Many possible avenues of research were found during the process. The research could be continued to find other possible sources of auto-generated data or to change a number of the fixed parameters of this study. These parameters include network geometry, type of neural network structure, data type and many others. The first step would be matching an appropriate auto-generation technique with the input data type.

Another possible route for research would involve the another auto-generation technique
that is designed for large input space problems. A new technique would deviate inputs from a
known good classification until the network demonstrates difficulty with classification. This
would minimize the creation of input patterns which have little meaning to the improvement
of the neural net training set.

The data extraction software, though unsuccessful in this research, is an interesting start
in many different fields of research. The software demonstrated an ability to accentuate the
attributes that the neural network found important when fed individual entries from the
training set. When examined in this fashion, one may be able to investigate particular
aspects of a system that cause the smaller set of inputs.
Appendix A

Source Code

A.1 Basic Neural Net

A.1.1 Layer.h

// layer.h V.Rao, H. Rao
// with revisions for specialized training DKW
// header file for the layer class hierarchy and
// the network class

#define MAX_LAYERS 5
#define MAX_VECTORS 4000
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

enum mode{ NO_TRNG_WO_EXP_VAL, NO_TRNG_W_EXP_VAL,
            REG_TRNG, REG_TRNG_W_TST_FILE_OP,
            EXTRACT_INPUTS };

class network;

class layer
{

protected:
int num_inputs;
int num_outputs;
float *outputs; // pointer to array of outputs
float *inputs; // pointer to array of inputs, which
    // are outputs of some other layer

friend network;

public:

    virtual void calc_out()=0;

};

class input_layer: public layer
{

private:

public:

    input_layer(int, int);
    ~input_layer();
    virtual void calc_out();

};
class middle_layer;

class output_layer: public layer
{
    protected:

        float * weights;
        float * output_errors; // array of errors at output
        float * back_errors; // array of errors back-propagated
        float * expected_values; // to inputs
friend network;

public:

output_layer(int, int);
"output_layer();
virtual void calc_out();
virtual void calc_error(float &);
void calc_error_character(float &);
void randomize_weights();
void update_weights(const float);
void update_weights_without_derivative(const float);
void list_weights();
void print_extracted_weights();
void write_extracted_weights(FILE *);
void write_weights(int, FILE *);
void read_weights(int, FILE *);
void list_errors();
void list_outputs();

};

class middle_layer: public output_layer
{

private:

public:

middle_layer(int, int);
"middle_layer();
void calc_error();
void calc_error_without_derivative();
void calc_out();
void calc_out_without_squash();

};
class network
{

private:

    layer *layer_ptr[MAX_LAYERS];
    int number_of_layers;
    int layer_size[MAX_LAYERS];
    float *buffer;
    fpos_t position;
    mode training;

public:
    network();
    ~network();
    network(const network &);
    void set_training(const mode &);
    mode get_training_value();
    void get_layer_info();
    void set_up_network();
    void randomize_weights();
    void update_weights(const float);
    void write_weights(FILE *);
    void read_weights(FILE *);
    void list_weights();
    void write_extracted_weights(FILE *);
    void print_extracted_weights();
    void write_outputs(FILE *);
    void list_outputs();
    void list_errors();
    void forward_prop();
    void backward_prop(float &);
    int fill_I0buffer(FILE *);
    void set_up_pattern(int);
};
A.1.2 Layer.cc

#include "layer.h"

inline float
squash (float input)
// squashing function
// using sigmoid
{
    if (input < -50)
        return 0.0;
    else if (input > 50)
        return 1.0;
    else
        return (float) (1 / (1 + exp (-(double) input)));
}

inline float
unsquash (float input)
// unsquashing function
//
{
    if (input < 0.000000001)
        return -50;
    else if (input > .999999999)
        return 50;
    else
        return (float) -1 * log( (1/(double)input) - 1);
}

inline float
randomweight (unsigned init)
{
}
int num;
// random number generator
// will return a floating point
// value between -1 and 1

if (init == 1) // seed the generator
    srand ((unsigned) time (NULL));

num = rand () % 100;

return 2 * (float (num / 100.00)) -1;
}

// -----------------------------------------
// input layer
// -----------------------------------------
input_layer::input_layer (int i, int o)
{
    num_inputs = i;
    num_outputs = o;

    outputs = new float[num_outputs];
    if (outputs == 0)
    {
        cout << "not enough memory\n";
        cout << "choose a smaller architecture\n";
        exit (1);
    }

    outputs[num_outputs - 1] = 1.00; // Set the Bias node up
}

input_layer::~input_layer ()
{
    delete[]outputs;
}
void
inginput_layer::calc_out ()
{
//nothing to do, yet
//printf("In input layer calc_out\n");
}

// -----------------------------------------
// output layer
//------------------------------------------

output_layer::output_layer (int i, int o)
{
    num_inputs = i;
    num_outputs = o;
    weights = new float[num_inputs * num_outputs];
    output_errors = new float[num_outputs];
    back_errors = new float[num_inputs];
    outputs = new float[num_outputs];
    expected_values = new float[num_outputs];
    if ((weights == 0) || (output_errors == 0) ||
        (back_errors == 0) || (outputs == 0) ||
        (expected_values == 0))
    {
        cout << "not enough memory\n";
        cout << "choose a smaller architecture\n";
        exit (1);
    }
}

output_layer::~output_layer ()
{
    // some compilers may require the array
    // size in the delete statement; those
// conforming to Ansi C++ will not
delete[]weights;
delete[]output_errors;
delete[]back_errors;
delete[]outputs;

}

void
output_layer::calc_out ()
{

    int
    i, j, k;
    float
    accumulator = 0.0;

    //printf("In Output Layer Calc_out\n");
    for (j = 0; j < num_outputs; j++)
    {

        for (i = 0; i < num_inputs; i++)
            // This loop sums the inputs for each output node

        {
            k = i * num_outputs;
            if (weights[k+j] * weights[k+j] > 1000000.0)
            {
                cout << "weights are blowing up\n";
                cout << "try a smaller learning constant\n";
                cout << "e.g. beta=0.02 aborting...\n";
                exit (1);
            }
            outputs[j] = weights[k + j] * (*(inputs + i));
            accumulator += outputs[j];
        }

        // use the sigmoid squash function
        outputs[j] = squash (accumulator);
    }
accumulator = 0;

}

}

// This function will return a 1 or 0
// if it identifies the proper character
void output_layer::calc_error_character (float &error) {
    int i, j, k, max_output_node;
    float accumulator = 0;
    float total_error = 0;
    float max_output;

    // printf("In Output Layer Calc Error\n");

    // Calculate Regular Output Errors for Back Propagation
    for (j = 0; j < num_outputs; j++)
    {
        output_errors[j] = expected_values[j] - outputs[j];
    }

    // Method 1 for determining Total Error
    total_error = 0.0; // start out with no error

    for (j = 0; j < num_outputs; j++)
    {
        if ((outputs[j] < .95 && (expected_values[j] > 0.0)) ||
            (outputs[j] > .05 && (expected_values[j] == 0.0)))
    }
{ 
    total_error=1.0;
    j=num_outputs;
}

// Method 2
// Find Error based on if it picked the character correctly
//max_output = outputs[0];
//max_output_node = 0;

//for (j = 1; j < num_outputs; j++)
// {
//    if (outputs[j] > max_output)
//    {
//        max_output = outputs[j];
//        max_output_node = j;
//    }
//}
// Now we have found the most likely classification, we need to check to see if it the expected value

// if (expected_values[max_output_node] > 0.0)
//    // it should be one if it is the desired one
//    total_error = 0.0;
// else
//    total_error = 1.0;

// This is the number that is sent back
error = total_error;

// Taking the output errors, we calculate Benefit Term for use when figuring the weight change on the previous layer.
for (i = 0; i < num_inputs; i++)
{
    k = i * num_outputs;
for (j = 0; j < num_outputs; j++)
{
    back_errors[i] = weights[k + j] * output_errors[j]
        * outputs[j] * (1-outputs[j]);
    accumulator += back_errors[i];
}

back_errors[i] = accumulator;
accumulator = 0;
// now multiply by derivative of
// sigmoid squashing function, which is
// just the input*(1-input)
//back_errors[i] *= (*(inputs + i))
// * (1 - (*(inputs + i)));

}

void
output_layer::calc_error (float &error)
{
    int
    i, j, k;
    float
    accumulator = 0;
    float
    total_error = 0;

    // printf("In Output Layer Calc Error\n");

    // Calculate Regular Output Errors for Back Propagation
    for (j = 0; j < num_outputs; j++)
    {
        output_errors[j] = expected_values[j] - outputs[j];
        total_error += fabs( output_errors[j] );
        // cout << output_errors[j] << "\t" <<
        // expected_values[j]<< "\t" << outputs[j]<< endl;
    }

    error = total_error;
/ This is the number that is sent back

// Taking the output errors, we calculate Benefit Term for use when figuring the weight change on the previous layer.
for (i = 0; i < num_inputs; i++)
{
    k = i * num_outputs;

    for (j = 0; j < num_outputs; j++)
    {
        accumulator += back_errors[i];
    }

    back_errors[i] = accumulator;
    accumulator = 0;
    // now multiply by derivative of sigmoid squashing function, which is just the input*(1-input)
    // back_errors[i] *= (*(inputs + i)) * (1 - (*(inputs + i)));
}

void
output_layer::randomize_weights ()
{
    int
    i, j, k;
    const unsigned
    first_time = 1;

    const unsigned
    not_first_time = 0;
    float
discard;
discard = randomweight (first_time);

for (i = 0; i < num_inputs; i++)
{
    k = i * num_outputs;
    for (j = 0; j < num_outputs; j++)
        weights[k + j] = randomweight (not_first_time);
}

void output_layer::update_weights (const float beta)
{
    int i, j, k;
    float temp;

    // learning law: weight_change =
    // beta*output_error*input * output * 1-output

    // The output errors of this level were the
    // backerrors of the level ahead of it

    // i=0;
    // for(j=0;j<num_outputs;j++)
    // {
    //     cout << output_errors[j] << " 
    //     << outputs[j]
    //     " << *(inputs + i) << endl;
    // }

    for (i = 0; i < num_inputs; i++)
    {
        k = i * num_outputs;
        for (j = 0; j < num_outputs; j++)
        {
            temp = beta * output_errors[j] * outputs[j]
            * (1 - outputs[j]) * (*(inputs + i)) ;
            //cout << i << "\t" << j << "\t" << weights[k+j]
        }
    }
}
void output_layer::update_weights_without_derivative (const float beta) {
  int i, j, k;
  float temp;

  // learning law: weight_change =
  // beta*output_error*input *
  // The output errors of this level were the backerrors
  // of the level ahead of it

  for (i = 0; i < num_inputs; i++) {
    k = i * num_outputs;
    
    weights[k + j] =
      (rint((weights[k+j] + temp) *1000000.0)/1000000.0);
    //cout << weights[k+j] << endl;
    //if (unsquash(weights[k+j]) < -8 )
    //  weights[k+j] = squash(-8);
    //if (unsquash(weights[k+j]) > 8)
    //  weights[k+j] = squash(8);

    //temp = beta * output_errors[j] * outputs[j]
    // * (1 - outputs[j]) * (*(inputs + i));
    //cout << j <<"t" << temp<<"t" output_errors[j]
    // << "t" << outputs[j] "t" (*(inputs + i));
    //cout << endl;
  }
}
for (j = 0; j < (num_outputs-1); j++)
{
    //weights[k + j] +=
    // beta * output_errors[j] * (*(inputs + i)) * weights[k + j];
    if (output_errors[j]>0)
        weights[k+j]+=0.1;
    else
        weights[k+j]-=0.1;

    if (weights[k+j]>16)
        weights[k+j]=16;
    if (weights[k+j]<0)
        weights[k+j]=0;
    //temp = beta * output_errors[j] * (*(inputs + i));
    //cout << j <<"\t" << temp<<"\t"<< output_errors[j]
    // << " " << outputs[j] << " "<< (*(inputs + i));
    //cout << endl;
}
}

void
output_layer::list_weights ()
{
    int
    i, j, k;

    for (i = 0; i < num_inputs; i++)
    {
        k = i * num_outputs;
        for (j = 0; j < num_outputs; j++)
            cout << "weight[" << i << "," << j << "] is: "
            << weights[k + j] << " ";
    }
cout << endl;
}

void output_layer::list_errors ()
{
    int i, j;

    for (i = 0; i < num_inputs; i++)
        cout << "backerror[" << i << "] is : " <<
             back_errors[i] << "\n";

    for (j = 0; j < num_outputs; j++)
        cout << "outputerrors[" << j << "] is: " <<
             output_errors[j] << "\n";
}

void output_layer::write_weights
    (int layer_no, FILE * weights_file_ptr)
{
    int i, j, k;

    // assume file is already open and ready for
    // writing

    // prepend the layer_no to all lines of data
    // format:
    // layer_no     weight[0,0] weight[0,1] ...
    // layer_no     weight[1,0] weight[1,1] ...
    // ...

    for (i = 0; i < num_inputs; i++)
    {
        fprintf (weights_file_ptr, "%i ", layer_no);
        "
    }
k = i * num_outputs;

for (j = 0; j < num_outputs; j++)
{
    fprintf
        (weights_file_ptr, "%f ", weights[k + j]);
}
fprintf (weights_file_ptr, "\n");
}

void
output_layer::write_extracted_weights
    ( FILE * weights_file_ptr)
{
    int
        i, j, k;

    // assume file is already open and ready for
    // writing
    // This is getting the weights from layer (should be layer[1])

    for (j = 0; j < num_outputs-1; j++)
    {
        fprintf
            (weights_file_ptr, "%d", (int) rint(weights[j]) );

        if(j!= num_outputs-1)
            fprintf (weights_file_ptr, " ");
    }

    fprintf (weights_file_ptr, "\n");
void output_layer::print_extracted_weights ( )
{
    int j;
    for (j = 0; j < num_outputs; j++)
    {
        printf ("%f",(weights[j]));
        if(j!= num_outputs-1)
            printf (",");
    }
    printf ( "\n");
}

void output_layer::read_weights
    (int layer_no, FILE * weights_file_ptr)
{
    int i, j, k;

    // assume file is already open and ready for
    // reading

    // look for the prepended layer_no
    // format:
    //     layer_no    weight[0,0] weight[0,1] ...
    //     layer_no    weight[1,0] weight[1,1] ...
    //     ...

    while (1)
    {

fscanf (weights_file_ptr, "%i", &j);
if ((j == layer_no) || (feof (weights_file_ptr)))
break;
else
{
    while (fgetc (weights_file_ptr) != '\n')
    {
    }
} // get rest of line

if (!(feof (weights_file_ptr)))
{
    // continue getting first line
    i = 0;
    for (j = 0; j < num_outputs; j++)
    {
        
        fscanf
        (weights_file_ptr, "%f", &weights[j]);
        // i*num_outputs = 0
    }
}
fscanf (weights_file_ptr, "\n");

// now get the other lines
for (i = 1; i < num_inputs; i++)
{
    fscanf (weights_file_ptr, "%i", &layer_no);
    k = i * num_outputs;
    for (j = 0; j < num_outputs; j++)
    {
        fscanf
        (weights_file_ptr, "%f", &weights[k + j]);
    }
} 

fscanf (weights_file_ptr, "\n"); 
} 

else 
    cout << "end of file reached\n"; 

}

void 
output_layer::list_outputs () 
{
    int 
    j;

    for (j = 0; j < num_outputs; j++)
    {
        cout << "outputs[" << j << "] is: " 
             << outputs[j] << " \n";
    }
}

// -----------------------------------------
// middle layer
//------------------------------------------

middle_layer::middle_layer (int i, int o):
output_layer (i, o) 
{
    outputs[o - 1] = 1.00;       //set up the bias
}

middle_layer::~middle_layer () 
{

56
delete[] weights;
delete[] output_errors;
delete[] back_errors;
delete[] outputs;
}

void middle_layer::calc_out () {

    int i, j, k;
    float accumulator = 0.0;

    // printf("In Middle Layer Calc_out\n");
    for (j = 0; j < num_outputs - 1; j++)
        // Last Output is fixed as a bias
        {

            for (i = 0; i < num_inputs; i++)
            {

                k = i * num_outputs;
                if (weights[k+j] * weights[k+j] > 1000000.0)
                {
                    cout << "weights are blowing up\n";
                    cout << "try a smaller learning constant\n";
                    cout << "e.g. beta=0.02 aborting...\n";
                    exit (1);
                }
                outputs[j] = weights[k + j] * (*(inputs + i));
                accumulator += outputs[j];
            }

            // use the sigmoid squash function
            outputs[j] = squash (accumulator);
            accumulator = 0;
        }

    57
void middle_layer::calc_out_without_squash()
{
    int i, j, k;
    float accumulator = 0.0;

    // printf("In Middle Layer Calc_out\n");
    for (j = 0; j < num_outputs - 1; j++)
    // Last Output is fixed as a bias
    {
        for (i = 0; i < num_inputs; i++)
        {
            k = i * num_outputs;
            if (weights[k+j] * weights[k+j] > 1000000.0)
            {
                cout << "weights are blowing up\n";
                cout << "try a smaller learning constant\n";
                cout << "e.g. beta=0.02 aborting...\n";
                exit (1);
            }
            outputs[j] = weights[k + j] * (*(inputs + i));
            accumulator += outputs[j];
        }
        outputs[j] = accumulator;
        accumulator = 0;
    }

    // The outputs[num_outputs-1] is a fixed output
    // of 1.00 that is set up when initialized
}
void
middle_layer::calc_error ()
{
    int
    i, j, k;
    float
    accumulator = 0;

    //printf("In Middle Layer Calc Error\n");
    for (i = 0; i < num_inputs; i++)
    {
        k = i * num_outputs;
        for (j = 0; j < num_outputs; j++)
        {
            back_errors[i] =
            weights[k+j] * (*(output_errors + j))
            * (*(outputs + j)) * (1 - (*(outputs + j)));
            accumulator += back_errors[i];
        }
        back_errors[i] = accumulator;
        accumulator = 0;

        // now multiply by derivative of
        // sigmoid squashing function, which is
        // just the input*(1-input)
        // back_errors[i] *= (*(inputs + i)) *
        // (1 - (*(inputs + i)));
    }
}

void
middle_layer::calc_error_without_derivative ()
{
    int
    i, j, k;
float
    accumulator = 0;

//printf("In Middle Layer Calc Error\n");
for (i = 0; i < num_inputs; i++)
{
    k = i * num_outputs;
    for (j = 0; j < num_outputs; j++)
    {
        back_errors[i] =
            weights[k + j] * (*(output_errors + j));
        accumulator += back_errors[i];
    }
    back_errors[i] = accumulator;
    accumulator = 0;
}

}

network::network ()
{
    position = 0L;
}

network::~network ()
{
    delete[]buffer;
}

void
network::set_training (const mode &value)
{
    training = value;
}
mode
network::get_training_value ()
{
    return training;
}

void
network::get_layer_info ()
{
    int i;

    //------------------------------------------
    //
    //    Get layer sizes for the network
    //
    //------------------------------------------

    cin >> number_of_layers;
    number_of_layers = 3;

    cout << "\nHard Coded " << number_of_layers
    << " Layer Network\n";
    // Adding the Extra Internal Layer
    if (training == EXTRACT_INPUTS)
        number_of_layers = number_of_layers++;

    cout << " Enter in the layer sizes separated by spaces.\n";
    cout << " For a network with 3 neurons in the input layer,\n";
    cout << " 2 neurons in a hidden layer, and 4 neurons in the\n";
// cout << "output layer, you would enter: 3 2 4.\n";
// cout << "You can have up to 3 hidden layers"
// << ", for five maximum entries: \n\n";

/* for (i = 0; i < number_of_layers; i++)
   {
       cin >> layer_size[i];
   }
*/

// Generating the Layer Size
cout << "Hard Coded Layer Size 64 -> 28 -> 10 \n\n";

layer_size[0] = 64;
layer_size[1] = 28;
layer_size[2] = 10;

if (training == EXTRACT_INPUTS)
// Shifts the Sizes over and creates the first layer
{
    for (i=number_of_layers-1;i>0;i--)
    {
        layer_size[i] = layer_size[i-1];
    }

    layer_size[0] = 1;
}

// size of layers:
// input layer layer_size[0]
// output layer layer_size[number_of_layers-1]
// middle layers layer_size[1]
// optional: layer_size[number_of_layers-3]
// optional: layer_size[number_of_layers-2]

printf ("Beginning Processing\n");
void
network::set_up_network ()
{
    int
    i, j, k;
    //-------------------------------------------------------
    // Construct the layers
    //-------------------------------------------------------

    layer_ptr[0] = new input_layer (0, layer_size[0] + 1);
    // One is added to give a bias node
    //cout << "Making Layer " << 0 << endl;

    for (i = 0; i < (number_of_layers - 2); i++)
    {
        layer_ptr[i + 1] =
            new middle_layer
                (layer_size[i] + 1, layer_size[i + 1] + 1);
        // One is added to give a bias node
        //cout << "Making Layer " << i+1 << endl;
    }

    //cout << "Making Layer " << number_of_layers - 1 << endl;

    layer_ptr[number_of_layers - 1] = new
        output_layer (layer_size[number_of_layers - 2] + 1,
            layer_size[number_of_layers - 1]);
    // One is added to give a bias node,
    // but the output doesn't
    // need a bias node

    for (i = 0; i < (number_of_layers - 1); i++)
    {
        if (layer_ptr[i] == 0)
        {
            cout << "insufficient memory\n";
            cout << "use a smaller architecture\n";
        }
    }
}
exit (1);
}
}

//-------------------------------------------------------
// Connect the layers
//
//-------------------------------------------------------
// set inputs to previous layer outputs for all layers,
// except the input layer

for (i = 1; i < number_of_layers; i++)
    layer_ptr[i]->inputs = layer_ptr[i - 1]->outputs;

// for back_propagation, set output_errors to next layer
// back_errors for all layers except the output
// layer and input layer

for (i = 1; i < number_of_layers - 1; i++)
    ((output_layer *) layer_ptr[i])->output_errors =
        ((output_layer *) layer_ptr[i + 1])->back_errors;

// define the IObuffer that caches data from
// the datafile
i = layer_ptr[0]->num_outputs - 1;  // inputs
j = layer_ptr[number_of_layers - 1]->num_outputs;  // outputs
k = MAX_VECTORS;

if (training == EXTRACT_INPUTS)
    //Handles the pattern extraction
{
    i = layer_ptr[1]->num_outputs - 1;
    // Using the inputs of the second layer
    k = 1;  //one vector at a time
}
buffer = new float[(i + j) * k];
if (buffer == 0)
{
    cout << "insufficient memory for buffer\n";
    exit (1);
}

void
network::randomize_weights ()
{
    int
    i;

    for (i = 1; i < number_of_layers; i++)
        ((output_layer *) layer_ptr[i])->randomize_weights ();
}

void
network::update_weights (const float beta)
{
    int
    i;

    for (i = 1; i < number_of_layers; i++)
    {
        if (training != EXTRACT_INPUTS)
        {
            ((output_layer *) layer_ptr[i])->update_weights (beta);
            // cout << "Updating Layer " << i << endl;
        }
        else
        {
            ((output_layer *) layer_ptr[i])
                ->update_weights_without_derivative (beta);
            i = number_of_layers;
            // This updates the weight in the first layer only
        }
    }
void network::write_weights (FILE * weights_file_ptr)
{
    int i;
    printf ("Writing Weights\n");
    for (i = 1; i < number_of_layers; i++)
        ((output_layer *) layer_ptr[i])
            ->write_weights (i, weights_file_ptr);
}

void network::write_extracted_weights (FILE * weights_file_ptr)
{
    ((output_layer *) layer_ptr[1])
        ->write_extracted_weights (weights_file_ptr);
}

void network::print_extracted_weights ()
{
    ((output_layer *) layer_ptr[1])
        ->print_extracted_weights ();
}

void network::read_weights (FILE * weights_file_ptr)
{
    int i, j, offset;
j = 1;
offset = 0;

if (training == EXTRACT_INPUTS)
{
    // Skips the first layer for extraction
    j = 2;
    offset = -1;
}

for (i = j; i < number_of_layers; i++)
{
    ((output_layer *) layer_ptr[i])
        ->read_weights (i + offset, weights_file_ptr);
    // with training at 4, this will put weights of
    // layer X with numbers of X-1
}

void
network::list_weights ()
{
    int
    i;

    for (i = 1; i < number_of_layers; i++)
    {
        cout << "layer number : " << i << "\n";
        ((output_layer *) layer_ptr[i])->list_weights ();
    }
}

void
network::list_outputs ()
{
    int
    i;

    for (i = 0; i < number_of_layers; i++)
    {
        cout << "layer number : " << i << "\n";
        ((output_layer *) layer_ptr[i])
            ->list_outputs (i);
    }
}
void network::write_outputs (FILE * outfile) {
    int i, ins, outs;
    float temp;
    ins = layer_ptr[0]->num_outputs - 1;
    outs = layer_ptr[number_of_layers - 1]->num_outputs;

    fprintf (outfile, "for input vector:\n");
    for (i = 0; i < ins; i++) {
        temp = layer_ptr[0]->outputs[i];
        fprintf (outfile, "%f ", temp);
    }

    fprintf (outfile, \noutput vector is:\n");
    for (i = 0; i < outs; i++) {
        temp = layer_ptr[number_of_layers - 1]->outputs[i];
        fprintf (outfile, "%f ", temp);
    }

    if (training == NO_TRNG_W_EXP_VAL ||
        training == REG_TRNG ||
        training == REG_TRNG_W_TST_FILE_OP ||
        training == EXTRACT_INPUTS)
    {
        fprintf (outfile, \nexpected output vector is:\n);
for (i = 0; i < outs; i++)
{
    temp =
    ((output_layer *)
        (layer_ptr[number_of_layers - 1]))
        ->expected_values[i];
    fprintf (outfile, "%.6f ", temp);
}
}

fprintf (outfile, "\n----------------------\n\n");

void
network::list_errors ()
{
    int
    i;

    for (i = 1; i < number_of_layers; i++)
    {
        cout << "layer number : " << i << "\n";
        ((output_layer *) layer_ptr[i])->list_errors () ;
    }
}

int
network::fill_IObuffer (FILE * inputfile)
{
    // this routine fills memory with
    // an array of input, output vectors
    // up to a maximum capacity of
// MAX_INPUT_VECTORS_IN_ARRAY
// the return value is the number of read
// vectors

int
    i, k, count, veclength;

int
    ins, outs;

ins = layer_ptr[0]->num_outputs - 1;
// If we are doing data extraction, we will be looking
// for the number of outputs of the second layer.
if(training == EXTRACT_INPUTS )
    ins = layer_ptr[1]->num_outputs - 1;

outs = layer_ptr[number_of_layers - 1]->num_outputs;

if (training == NO_TRNG_W_EXP_VAL ||
    training == REG_TRNG ||
    training == REG_TRNG_W_TST_FILE_OP ||
    training == EXTRACT_INPUTS)
    veclength = ins + outs;
else
    veclength = ins;
// This is only used when using the network to
// process data without expectations

count = 0;
while((count<1 ||
    (training!=EXTRACT_INPUTS && count<MAX_VECTORS) )
    &&
    (!feof (inputfile)))
{
    k = count * (veclength);
    for (i = 0; i < veclength; i++)
    {
        fscanf (inputfile, "%f", &buffer[k + i]);
        //cout << i << " " << buffer[k + i] << endl;
    }

fscanf (inputfile, "\n");
count++;
}

if (!(ferror (inputfile)))
    return count;
else
    return -1;    // error condition
}

void
network::set_up_pattern (int buffer_index)
{
    // read one vector into the network
    int
        i, k;
    int
        ins, outs;

    ins = layer_ptr[0]->num_outputs - 1;
    outs = layer_ptr[number_of_layers - 1]->num_outputs;

    // We want to put these on the layer 1 weights
    if (training == EXTRACT_INPUTS)
        ins = layer_ptr[1]->num_outputs - 1;

    if (training == NO_TRNG_W_EXP_VAL ||
        training == REG_TRNG ||
        training == REG_TRNG_W_TST_FILE_OP ||
        training == EXTRACT_INPUTS)
        k = buffer_index * (ins + outs);
    else
        k = buffer_index * ins;

    // k is should be 0 for training == 4
    // for we only look at one vector at a time
for (i = 0; i < ins; i++)
{
    if (training != EXTRACT_INPUTS)
    {
        layer_ptr[0]->outputs[i] = buffer[k + i];
    } else {
        ((output_layer *) layer_ptr[1])->weights[i] = buffer[i]; // sets weights to values
    }
}

if (training == EXTRACT_INPUTS)
{
    // sets a constant one output for node 1 and zero for bias
    layer_ptr[0]->outputs[0] = 1.0;
    layer_ptr[0]->outputs[1] = 0.0;

    for (i=ins;i<2*ins;i++)// Turns off bias node
        ((output_layer *) layer_ptr[1])->weights[i] = 0.0;

    ((output_layer *) layer_ptr[1])->outputs[ins] = 1.0; // Turns on bias node for the
}

if (training == NO_TRNG_W_EXP_VAL ||
    training == REG_TRNG ||
    training == REG_TRNG_W_TST_FILE_OP ||
    training == EXTRACT_INPUTS)
{
    for (i = 0; i < outs; i++)
    {
        ((output_layer *) layer_ptr[number_of_layers - 1])
            ->expected_values[i] = buffer[k + i + ins];
        //cout << i << "\t" << buffer[k + i + ins] << endl;
    }
void network::forward_prop()
{
    int i;
    for (i = 0; i < number_of_layers; i++)
    {
        // printf("Calling Forward Prop on layer %d\n",i+1);
        if (training == EXTRACT_INPUTS && i == 1)
            ((middle_layer *)layer_ptr[i])
                ->calc_out_without_squash();
        else
            layer_ptr[i]->calc_out();
        // polymorphic
        // function
    }
}

void network::backward_prop(float &toterror)
{
    int i;
    // error for the output layer
    if (training == EXTRACT_INPUTS)
        ((output_layer *)layer_ptr[number_of_layers - 1])
            ->calc_error(toterror);
    else
        ((output_layer *)layer_ptr[number_of_layers - 1])
            ->calc_error_character(toterror);
// error for the middle layer(s)
if (training == REG_TRNG ||
    training == REG_TRNG_W_TST_FILE_OP )
{
    // prevents calculation of middle errors
    // when just checking output results

    for (i = number_of_layers - 2; i > 0; i--)
    {
        //printf("Calling BackProp Prop on layer %d\n",i+1);
        ((middle_layer *) layer_ptr[i])->calc_error();
    }
}

if (training == EXTRACT_INPUTS )
{
    // prevents calculation of middle errors
    // when just checking output results

    for (i = number_of_layers - 2; i > 0; i--)
    {
        // printf("Calling BackProp Prop on layer %d\n",i+1);
        if (i > 1)
            ((middle_layer *) layer_ptr[i])->calc_error();
        else
            ((middle_layer *) layer_ptr[i])
                ->calc_error_without_derivative();
    }
}
A.1.3 Backpropagation.cc

// backprop.cpp V. Rao, H. Rao
#include "layer.h"
#define TRAINING_FILE "../data/training.dat"
#define WEIGHTS_FILE "../data/weights.dat"
#define OUTPUT_FILE "../data/output.dat"
#define TEST_FILE "../data/test.dat"
#define DESIRED_OUTPUTS_FILE "../data/desired_outputs.dat"

void main (int argc, char* argv[])
{

float error_tolerance = 0.001;
float total_error = 0.0;
float avg_error_per_cycle = 0.0;
float error_last_cycle = 0.0;
float error_last_test_cycle = 0.0;
float avgerr_per_pattern = 0.0; // for the latest cycle
float avgerr_per_test_pattern = 0.0;
float error_last_pattern = 0.0;
float error_last_test_pattern = 0.0;
float learning_parameter = 0.02;
mode mode_temp;
unsigned temp, startup;
long int vectors_in_buffer;
long int max_cycles;
long int patterns_per_cycle = 0;
long int patterns_per_test_cycle = 0;
long int total_cycles, total_patterns;
int i;
int reuse_weights;
int processonly=0;
// create a network object
network backp;

FILE *training_file_ptr, *weights_file_ptr, *output_file_ptr;
FILE *test_file_ptr, *data_file_ptr;

// open output file for writing
if ((output_file_ptr = fopen (OUTPUT_FILE, "w")) == NULL)
{
    cout << "problem opening output file\n";
    exit (1);
}

if (argc > 1 && !(strcmp("processonly",argv[1])))
{
    processonly=1;
    mode_temp = NO_TRNG_WO_EXP_VAL;
}
else
{
    // enter the training mode : 1=training on 0=training off
    cout<< "---------------------------------------------------\n";
    cout<< " C++ Neural Networks and Fuzzy Logic \n";
    cout<< " Backpropagation simulator \n";
    cout<< " version 1 \n";
    cout<< "---------------------------------------------------\n";
    cout<< "Please enter\n";
    cout<< "0 for No training. Data set does not contain expected values\n";
    cout<< "1 for No training. Data set does contain expected values\n";
    cout<< "2 for training. Data set has to contain expected values)\n";
    cout<< "3 for training and display of o/p of test file each iteration\n";
    cout<< "4 for for creating inputs from trained network\n";
    cin >> temp;
    mode_temp = (mode)temp;
}

backp.set_training (mode_temp);

if (!processonly) {
    switch ((int)backp.get_training_value ())
    {
    case 0:
        cout << "--> Training mode is *OFF*. weights will be loaded\n";
        cout << "from the file weights.dat and the current\n";
cout << "(test) data set will be used. For the test\n";
cout << "data set, the test.dat file should contain\n";
cout << "only inputs, and no expected outputs.\n"; break;
case 1:
cout << "--> Training mode is *OFF*. weights will be loaded\n";
cout << "from the file weights.dat and the current\n";
cout << "(test) data set will be used. For the test\n";
cout << "data set, the test.dat file should contain\n";
cout << "inputs and expected outputs.\n"; break;
case 2:
cout << "--> Training mode is *ON*. weights will be saved\n";
cout << "in the file weights.dat at the end of the\n";
cout << "current set of input (training) data\n"; break;
case 3:
cout << "--> Training mode is *ON*. weights will be saved\n";
cout << "in the file weights.dat at the end of the\n";
cout << "current set of input (training) data\n";
cout << "Test File Results will be shown but not used for\n";
cout << "Training (test data does have expected outputs)\n"; break;
case 4:
cout << "--> Training mode is *OFF*. Current Weights will be\n";
cout << "used to generate inputs from the desired outputs\n";
cout << "Generated inputs will save in file\n"; break;
}

if (backp.get_training_value () == REG_TRNG
    || backp.get_training_value () == REG_TRNG_W_TST_FILE_OP
    || backp.get_training_value () == EXTRACT_INPUTS )
{
    // -----------------------------------------
    // Read in values for the error_tolerance,
    // and the learning_parameter
    // -----------------------------------------
    // cout << " Please enter in the error_tolerance\n";
    // cout << " --- between 0.001 to 100.0, try 0.1 to start --\n";

77
// cout << "\n";
// cout << "and the learning_parameter, beta\n";
// cout << " --- between 0.01 to 1.0, try 0.5 to start -- \n\n";
// cout << " separate entries by a space\n";
// cout << " example: 0.1 0.5 sets defaults mentioned :\n\n";

//cin >> error_tolerance >> learning_parameter;

cout << "error tolerance (.005)?";
cin >> error_tolerance;
cout << "\nlearning parameter (.05)";
cin >> learning_parameter;
cout << "\n\n";

if (backp.get_training_value () == REG_TRNG ||
    backp.get_training_value () == REG_TRNG_W_TST_FILE_OP )
{
    if ((training_file_ptr = fopen (TRAINING_FILE, "r")) == NULL)
    {
        cout << "problem opening training file\n";
        exit (1);
    }

    data_file_ptr = training_file_ptr; // training on
}

if (backp.get_training_value () == EXTRACT_INPUTS )
{
    if ((training_file_ptr = fopen (DESIRED_OUTPUTS_FILE, "r")) == NULL)
    {
        cout << "problem opening Desired Output file\n";
        exit (1);
    }

    data_file_ptr = training_file_ptr;
}
// Read in the maximum number of cycles
// each pass through the input data file is a cycle
cout << "Please enter the maximum cycles for the simulation\n";
cout << "A cycle is one pass through the data set.\n";
cout << "Try a value of 10 to start with\n";

cin >> max_cycles;

// Opening Test File also
if (backp.get_training_value () == REG_TRNG_W_TST_FILE_OP)
{
    if ((test_file_ptr = fopen (TEST_FILE, "r")) == NULL)
    {
        cout << "problem opening test file\n";
        exit (1);
    }
}

if (backp.get_training_value () == NO_TRNG_WO_EXP_VAL
    || backp.get_training_value () == NO_TRNG_W_EXP_VAL )
{
    if ((test_file_ptr = fopen (TEST_FILE, "r")) == NULL)
    {
        cout << "problem opening test file\n";
        exit (1);
    }
    
    data_file_ptr = test_file_ptr; // training off
}

// the main loop
//
// training: continue looping until the total error is less than
// the tolerance specified, or the maximum number of cycles is exceeded; use both the forward signal propagation
and the backward error propagation phases. If the error
tolerance criteria is satisfied, save the weights in a file.
no training: just proceed through the input data set once in the
forward signal propagation phase only. Read the starting
weights from a file.
in both cases report the outputs on the screen

// initialize counters
total_cycles = 0; // a cycle is once through all the input data
total_patterns = 0; // a pattern is one entry in the input data

// get layer information
backp.get_layer_info();

// set up the network connections
backp.set_up_network();
// cout << "Network Setup\n";
// initialize the weights
reuse_weights = 0;

if (backp.get_training_value() == REG_TRNG
   || backp.get_training_value() == REG_TRNG_W_TST_FILE_OP)
{
   if (!processonly)
   {
      printf("New weights ??? 1 = Yes 0 = No \n");
      cin >> reuse_weights;
   }
   else
   {
      reuse_weights=0;
   }
}
if (reuse_weights == 1)
{
    // randomize weights for all layers; there is no
    // weight matrix associated with the input layer
    // weight file will be written after processing
    // so open for writing
    if ((weights_file_ptr = fopen (WEIGHTS_FILE, "w")) == NULL)
    {
        cout << "problem opening weights file\n";
        exit (1);
    }
    backp.randomize_weights ();
}
else
{
    // read in the weight matrix defined by a
    // prior run of the backpropagation simulator
    // with training on
    if ((weights_file_ptr = fopen (WEIGHTS_FILE, "r")) == NULL)
    {
        cout << "problem opening weights file\n";
        exit (1);
    }
    backp.read_weights (weights_file_ptr);
    // Reopening file for rewriting the weights file
    if (backp.get_training_value () == REG_TRNG
        || backp.get_training_value () == REG_TRNG_W_TST_FILE_OP)
    {
        fclose (weights_file_ptr);
        if ((weights_file_ptr = fopen (WEIGHTS_FILE, "w")) == NULL)
        {
            cout << "problem opening weights file\n";
            exit (1);
        }
    }
    cout << "Finished Reading Weights" << endl;
}
if (backp.get_training_value () < EXTRACT_INPUTS) {
  // main loop
  // if training is on, keep going through the input data
  // until the error is acceptable or the maximum number of cycles
  // is exceeded.
  // if training is off, go through the input data once. report outputs
  // with inputs to file output.dat

  startup = 1;
  vectors_in_buffer = MAX_VECTORS;  // startup condition

  total_error = 0;
  cout << "Before Main Loop" << endl;
  // This is the main loop for regular training.

  while
    (((backp.get_training_value () == REG_TRNG
        || backp.get_training_value () == REG_TRNG_W_TST_FILE_OP) // training on
        && (avgerr_per_pattern > error_tolerance) // Error greater than desired
        && (total_cycles < max_cycles) // Below desired number of cycles
        && (vectors_in_buffer != 0)) // Vectors to Process
    || ((backp.get_training_value () == NO_TRNG_W_EXP_VAL)
        || (backp.get_training_value () == NO_TRNG_WO_EXP_VAL))
        && (total_cycles < 1)) // This is for non-training runs
    || ((backp.get_training_value () == REG_TRNG
        || backp.get_training_value () == REG_TRNG_W_TST_FILE_OP) //Training on
        && (startup == 1) && max_cycles != 0)) // First Loop
  {
    startup = 0;
    error_last_cycle = 0;  // reset for each cycle
    error_last_test_cycle = 0;  // reset for each cycle
    patterns_per_cycle = 0;
    patterns_per_test_cycle = 0;
    // process all the vectors in the datafile
    // going through one buffer at a time
    // pattern by pattern
  }
while ((vectors_in_buffer == MAX_VECTORS))
{
    // fill buffer
    vectors_in_buffer = backp.fill_IObuffer (data_file_ptr);
    if (vectors_in_buffer < 0)
    {
        cout << "error in reading in vectors, aborting\n";
        cout << "check that there are on extra linefeeds\n";
        cout << "in your data file, and that the number\n";
        cout << "of layers and size of layers match the\n";
        cout << "the parameters provided.\n";
        exit (1);
    }
    // process vectors
    for (i = 0; i < vectors_in_buffer; i++)
    {
        // get next pattern
        backp.set_up_pattern (i);

        total_patterns++;
        patterns_per_cycle++;
        // forward propagate
        backp.forward_prop ();
        //backp.list_outputs ();

        if (backp.get_training_value () == NO_TRNG_WO_EXP_VAL
            || backp.get_training_value () == NO_TRNG_W_EXP_VAL)
            backp.write_outputs (output_file_ptr);

        // back_propagate, if appropriate
        if (backp.get_training_value () == REG_TRNG
            || backp.get_training_value () == REG_TRNG_W_TST_FILE_OP)
        {
        }
backp.backward_prop (error_last_pattern);
error_last_cycle +=
    error_last_pattern * error_last_pattern;
backp.update_weights (learning_parameter);
// backp.list_weights(); // can
// see change in weights by
// using list_weights before and
// after back_propagation
}
if (backp.get_training_value () == NO_TRNG_W_EXP_VAL )
{
    backp.backward_prop (error_last_pattern);
    error_last_cycle +=
        error_last_pattern * error_last_pattern;
}
}

error_last_pattern = 0;
} // End of While loop
if (backp.get_training_value () == REG_TRNG_W_TST_FILE_OP)
    // Parsing the Test File
{
    vectors_in_buffer = MAX_VECTORS;// reset vector;
    while ((vectors_in_buffer == MAX_VECTORS))
    {
        // printf (", ");
        vectors_in_buffer = backp.fill_IObuffer (test_file_ptr);
        if (vectors_in_buffer < 0)
        {
            cout << "error in reading in test vectors, aborting\n";
            exit (1);
        }
    }
// process vectors
for (i = 0; i < vectors_in_buffer; i++)
{
    patterns_per_test_cycle++;
    // get next pattern
    backp.set_up_pattern (i);

    // forward propagate
    backp.forward_prop ();

    // backp.list_outputs ();

    // Calculate the Error
    backp.backward_prop (error_last_test_pattern);
    error_last_test_cycle +=
        error_last_test_pattern * error_last_test_pattern;
}

error_last_test_pattern = 0.0;
}

fseek (test_file_ptr, 0L, SEEK_SET);

avgerr_per_pattern =
    //((float) sqrt ((double) error_last_cycle / patterns_per_cycle));
    error_last_cycle / patterns_per_cycle;
    total_error += error_last_cycle;
total_cycles++;

    // most character displays are 25 lines
    // user will see a corner display of the cycle count
    // as it changes
    cout << total_cycles << "\t" << avgerr_per_pattern
    << "\t" << error_last_cycle ;
if(backp.get_training_value () == REG_TRNG_W_TST_FILE_OP)
{
    avgerr_per_test_pattern =
        error_last_test_cycle / patterns_per_test_cycle;
    cout << "\t" << avgerr_per_test_pattern << "\t"
        << error_last_test_cycle;
}
cout << "\n";

fseek (data_file_ptr, 0L, SEEK_SET); // reset the file pointer
// to the beginning of
// the file

vectors_in_buffer = MAX_VECTORS; // reset
}
    // end main loop

cout << "\n\n\n\n\n\n\n"
cout << "-----------------------------------\n";
cout << " done: results in file output.dat\n";
cout << " training: last vector only\n";
cout << " not training: full cycle\n";
if (backp.get_training_value () == REG_TRNG ||
    backp.get_training_value () == REG_TRNG_W_TST_FILE_OP)
{
    backp.write_weights (weights_file_ptr);
    backp.write_outputs (output_file_ptr);
    //avg_error_per_cycle =
    // (float) sqrt ((double) total_error / total_cycles);
    //error_last_cycle = (float) sqrt ((double) error_last_cycle);
    avg_error_per_cycle = total_error / total_cycles;
    error_last_cycle = error_last_cycle;

cout << " weights saved in file weights.dat\n";
cout << "\n";
cout << "---->average error per cycle = " << avg_error_per_cycle << " <---\n";
cout << "---->error last cycle = " << error_last_cycle << " <---\n";
cout << "->error last cycle per pattern= " << avgerr_per_pattern << " \n";
}
if (backp.get_training_value () == NO_TRNG_W_EXP_VAL )
{
    //avg_error_per_cycle =
    // (float) sqrt ((double) total_error / total_cycles);
    //error_last_cycle = (float) sqrt ((double) error_last_cycle);
    avg_error_per_cycle = total_error / total_cycles;
    error_last_cycle = error_last_cycle;

    cout << "---->average error per cycle = " << avg_error_per_cycle << " <---\n";
    cout << "---->error last cycle = " << error_last_cycle << " <---\n";
    cout << "->error last cycle per pattern= " << avgerr_per_pattern << " <---\n";
}

cout << "---------------->total cycles = " << total_cycles << " <---\n";
cout << "---------------->total patterns = " << total_patterns << " <---\n";
cout << "---------------------------------------------\n";
}

// Loop for extracting inputs from a trained loop
if (backp.get_training_value () == EXTRACT_INPUTS) 
{
    vectors_in_buffer = backp.fill_IObuffer (data_file_ptr);

while (vectors_in_buffer != 0)
{

    if (vectors_in_buffer < 0)
    {
        cout<< "error in reading in vectors, aborting\n";
        cout<< "check that there are on extra linefeeds\n";
        cout<< "in your data file, and that the number\n";
cout<< "of layers and size of layers match the\n";
cout<< "the parameters provided.\n";
exit (1);
}

backp.set_up_pattern (0);
total_cycles = 0;
error_last_pattern = 1;
error_last_cycle = 1;
int temp;

while (total_cycles < max_cycles &&
    error_last_cycle > error_tolerance)
{
    backp.forward_prop ();
    //backp.list_outputs ();
    backp.backward_prop (error_last_pattern);
    //backp.list_errors ();
    backp.update_weights (learning_parameter);
total_cycles++;
    //cout << total_cycles << " " <<
    //    error_last_pattern<< endl;
    error_last_cycle =
        error_last_pattern * error_last_pattern;
    //backp.print_extracted_weights();
    // cin >> temp;
}
cout << total_cycles << " " <<
    error_last_pattern<< endl;

    //backp.print_extracted_weights();
backp.write_extracted_weights(output_file_ptr);

    // fill buffer
vectors_in_buffer = backp.fill_IObuffer (data_file_ptr);
// backp.list_weights();
{

// close all files
fclose (weights_file_ptr);
fclose (data_file_ptr);
fclose (output_file_ptr);

}
A.2 File Adaptation

A.2.1 Adaptfile.c

/********************
Adaptfile.c

This program changes the data files from the postal characters into a format that the neural net software can interpret for training data.

The only difference in to the two software formats is that the character classification (which is a number 0 -> 9) is represented as a series of true and false bits for each classification, thus any number would be represent by a series of 0 except for the particular number which it represents which would be 1.

a seven character would be shown as
0 0 0 0 0 0 1 0 0

The output goes to stout
********************/

#include <stdio.h>

int main (){
    FILE *filename;
    int charac[64];
    int number_rep;
    int i;
    char line[200];

    filename = fopen("../data/optdigits.tra", "r");

    while (!feof(filename)) {
        
}
/ * Reads in the Line */
for (i = 0; i < 64; i++)
{
    fscanf (filename, "%d," , &charac[i]);
}
scanf (filename, "%d", &number_rep);
fgets (line, 10, filename);

/* Prints the line out */
for (i = 0; i < 64; i++)
{
    if (charac[i] < 10)
        printf (" %d ", charac[i]);
    else
        printf ("%d ", charac[i]);
}

/* Prints out 1s and 0s to represent the desired output for net training */
for (i = 0; i < 9; i++)
{
    if (i == number_rep)
        printf ("1 ");
    else
        printf ("0 ");
}
if (number_rep == 9)
    printf ("i\n");
else
    printf ("0\n");
}
A.2.2 Create_avg_inputs.c

#include <stdio.h>

#define MAX_LINES 4000

int
main ()
{
    FILE *filename;
    int charac[64];
    int charac_2[64];
    char garbage[200];
    int number_of_lines = 0;
    int tempsum;
    int i, j, k;

    fpos_t position_array[MAX_LINES];

    filename = fopen ("../data/optdigits.tra", "r");

    /* Count the number of lines */
    while (!feof (filename))
        {
            fgetpos (filename, &position_array[number_of_lines]);
            fscanf (filename, "%s\n", garbage);
            number_of_lines++;
        }
    /* printf ("Total Lines is %d\n", number_of_lines); */

    /* number_of_lines = 5; */

    for (j = 1; j < number_of_lines; j++)
        {
            fsetpos (filename, &position_array[j - 1]);

            /* Reads in the Line */
for (i = 0; i < 64; i++)
{
    fscanf (filename, "%d", &charac[i]);
}

fscanf (filename, "%s\n", garbage);

for (k = j + 1; k <= number_of_lines; k++)
{
    fsetpos (filename, &position_array[k - 1]);
    for (i = 0; i < 64; i++)
    {
        fscanf (filename, "%d", &charac_2[i]);
    }

    fscanf (filename, "%s\n", garbage);

    /* Now Create New Line */

    /* Prints the line out */

    for (i = 0; i < 64; i++)
    {
        tempsum = charac[i] + charac_2[i];
        if (tempsum % 2 == 1)
        {
            tempsum++;
        }
        tempsum /= 2;
        if (tempsum < 10)
        {
            printf (" %d ", tempsum);  
        }
        else
        {
            printf ("%d ", tempsum);
        }
    }
    printf("\n");
}
}
A.2.3 Addnoise.c

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

/* This program takes a data file with expected values and adds
noise to it using the following algorithm.

A noise percentage is entered by the user and the program will
compute this as a number of bits to flip.

Since there are 64 squares with 0-16 bits, there are 1024 possible changes
that can occur. A 10% noise would be 102 bits would change state.

The information is not so specific that you can simply change the state
of a particular bit so the software will narrow down the changes to the
64 squares with each square having no more than 16 changes

Once the number of changes are determined per block, the type of
change is determined (darker or lighter) the change will be +/-
16. When added to the original number, the amount will range
from -16 to +32.

To determine the newvalue, the function will be the following
x < 0 x = -x
x >= 0 && x<= 16 x = x
x > 16 x = 32 – x

The new values are printed without the expectation for they will be sent
to the user for classification.

*/

void addnoise (float noisepercentage, int * numberarray)
{
    int bits[1024];
    int bits_to_change;
}
/*This is based on the character geometry */
int rand_number;
int i,j;
int direction;

bits_to_change
=( (int)ceil ( (double)( ((double)noisepercentage/100.0) * 64.0 * 16.0 )));

if (bits_to_change > 1024) /* Prevents noisepercentages over 100 */
   bits_to_change=1024;
if (bits_to_change < 0) /* Prevents negative noisepercentages*/
   bits_to_change=0;

/*clear memory */
memset ((void *) bits, 0, sizeof(bits));

/*seed the generator*/
srand ((unsigned) time (NULL));

while ( bits_to_change > 0)
{
   rand_number=(int) (1024.0*rand()/(RAND_MAX+1.0));
   while ( bits[rand_number] == 1 )/*Finds a bit that has not been set */
   {
      rand_number=(int) (1024.0*rand()/(RAND_MAX+1.0));
   }
   bits[rand_number] = 1; /*Sets the bit*/
   bits_to_change--;
}

for (i=0;i<64;i++)
{
   if(rand()>=.5) /* sets if area will darken or lighten*/
      direction = -1;
   else
direction = 1;

for(j=0;j<16;j++)
{
    numberarray[i] += direction * bits[i*16+j];
    /*adds noise to the number array if bit is set*/
}

if (numberarray[i] < 0)
    numberarray[i] = -numberarray[i];
if (numberarray[i] > 16)
    numberarray[i] = 32 - numberarray[i];
    /*These lines handle if the amount subtraction or addition are greater than the bits available*/

}

int main (int argc, char* argv[])
{
    FILE *filename;
    int numbers[64];
    int number_rep;
    int i;
    char line[200];
    float noise = 1;

    if (argc != 3)
    {
        printf( "usage: %s %s %s\n", argv[0] );
        exit(1);
    }
    else
    {
        noise = (float)atof(argv[2]);
        filename = fopen (argv[1], "r");
    }
    if (!filename)
{  
    printf( "Error opening file: %s \n", argv[1] );
    exit(i);
}

while (!(feof(filename))  )
{
    for (i = 0; i < 64; i++)
    {
        fscanf (filename, "%d,", &numbers[i]);
    }
    fscanf (filename, "%d", &number_rep);
    fgets (line, 10, filename);
    addnoise(noise, numbers);

    for (i = 0; i < 64; i++)
    {
        if (numbers[i] < 10)
            printf (" %d ", numbers[i]);
        else
            printf ("%d ", numbers[i]);
    }
    printf("\n");
}
}
A.3 File Sorting and Classification

A.3.1 Sorting

Sort.h

#include <iostream.h>
#include <list>
#include <fstream.h>

#define NUMBER_INPUTS 64
#define NUMBER_OUTPUTS 10
#define MAX_ENTRIES 4000
#define TOP_ENTRIES_TO_OUTPUT 100

class ENTRY
{
private:
    int * inputs;
    float * outputs;
    float max_value;
    float second_max_value;

public:

    ENTRY();
    ENTRY(const ENTRY &);
    ENTRY & operator= (const ENTRY &);
    bool operator== (const ENTRY &);

    int get_input_value (int) const;
    float get_output_value (int) const;
    float get_max_value () const;
    float get_2nd_max_value () const;

    void set_input_values (const int *);
    void set_output_values (const float *);
};
void set_max_value();

};

struct COMPARE
{
    bool operator()(ENTRY & E1, ENTRY & E2) const
    {
        return (E1.get_max_value()) < (E2.get_max_value());
    }
};

struct COMPARE_SECONDS
{
    bool operator()(ENTRY & E1, ENTRY & E2) const
    {
        return (E1.get_2nd_max_value()) < (E2.get_2nd_max_value());
    }
};
Sort.cc

#include "sort.h"

ENTRY::ENTRY()
{
    inputs = new int[NUMBER_INPUTS];
    outputs = new float[NUMBER_OUTPUTS];
    max_value = 0;
    second_max_value = 0;
}

ENTRY::~ENTRY()
{
    delete [] inputs;
    delete [] outputs;
    inputs=0;
    outputs=0;
}

ENTRY::ENTRY(const ENTRY & rhs)
{
    int i;
    inputs = new int[NUMBER_INPUTS];
    outputs = new float[NUMBER_INPUTS];

    for(i=0;i<NUMBER_INPUTS;i++)
    {
        inputs[i]=rhs.get_input_value(i);
    }
    for(i=0;i<NUMBER_OUTPUTS;i++)
    {
        outputs[i]=rhs.get_output_value(i);
    }
}
max_value = rhs.get_max_value();
second_max_value = rhs.get_2nd_max_value();

ENTRY & ENTRY::operator=(const ENTRY & rhs)
{
    int i;
    if (this == &rhs)
        return *this;

    for(i=0;i<NUMBER_INPUTS;i++)
    {
        inputs[i]=rhs.get_input_value(i);
    }
    for(i=0;i<NUMBER_OUTPUTS;i++)
    {
        outputs[i]=rhs.get_output_value(i);
    }
    max_value = rhs.get_max_value();
    second_max_value = rhs.get_2nd_max_value();
}

bool ENTRY::operator==(const ENTRY & rhs)
{
    bool returnvalue = true;
    int i;

    for(i=0;i<NUMBER_INPUTS;i++)
    {
        if(inputs[i]!=rhs.get_input_value(i))
        {
            returnvalue=false;
        }
    }
    return (returnvalue);
inline int ENTRY::get_input_value(int item) const
{
    return (inputs[item]);
}

inline float ENTRY::get_output_value(int item) const
{
    return (outputs[item]);
}

inline float ENTRY::get_max_value() const
{
    return (max_value);
}

inline float ENTRY::get_2nd_max_value() const
{
    return (second_max_value);
}

void ENTRY::set_input_values(const int * location)
{
    int i;

    for(i=0;i<NUMBER_INPUTS;i++)
    {
        inputs[i]=location[i];
    }
}

void ENTRY::set_output_values(const float * location)
{
int i;

for(i=0;i<NUMBER_OUTPUTS;i++)
{
    outputs[i]=location[i];
}

void ENTRY::set_max_value()
{
    int i;
    float tempmax;
    float temp2ndmax;

    if (outputs[0]>=outputs[1])
    {
        tempmax = outputs[0];
        temp2ndmax = outputs[1];
    }
    else
    {
        tempmax = outputs[1];
        temp2ndmax = outputs[0];
    }

    for(i=2;i<NUMBER_OUTPUTS;i++)
    {
        if (outputs[i]>temp2ndmax && outputs[i] <= tempmax)
            temp2ndmax=outputs[i];
        if (outputs[i]>tempmax)
        {
            temp2ndmax=tempmax;
            tempmax=outputs[i];
        }
    }

    max_value=tempmax;
    second_max_value=temp2ndmax;
Sortmain.cc

#include "sort.h"

int main()
{
  int intarray[NUMBER_INPUTS] ;
  float floatarray[NUMBER_OUTPUTS];

  int i, j, counter;
  ENTRY entryarray[MAX_ENTRIES];
  char line[1000];
  char *next_pointer;
  list <ENTRY> entrylist;
  list <ENTRY>::iterator iter;

  //Reads File
  ifstream fin("../data/output.dat");
  i = 0;
  fin.getline(line, 1000);// Discards Input Line Text

  do
  {
    fin.getline(line, 1000);// Grabs Weights
    intarray[0]= (int) strtod(line, &next_pointer);
    counter=1;
    while(counter<NUMBER_INPUTS)
    {
      intarray[counter]=(int)strtod(next_pointer, &next_pointer);
      counter ++;
    }

    fin.getline(line, 1000);// Discards Output Line Text
  }
  fin.getline(line, 1000);// Grabs Outputs
  floatarray[0]= strtod(line, &next_pointer);
  counter=1;
while(counter<NUMBER_OUTPUTS) {
    floatarray[counter] = strtod(next_pointer, &next_pointer);
    counter ++;
}

fin.getline(line, 1000); //Discards dashed line

entryarray[i].set_input_values ( intarray );
entryarray[i].set_output_values ( floatarray );
entryarray[i].set_max_value ();
entrylist.push_back(entryarray[i]);
i++;

fin.getline(line, 1000); // Discards Input Line Text

}while (!fin.eof() ) ;

// All of the lines have been put into the list of Entries

//Listing the max and 2nd max values as entered
/* for (iter = entrylist.begin();iter != entrylist.end(); iter++)
{
    cout << iter->get_max_value()
        << " " <<iter->get_2nd_max_value() << endl;
}

cout <<endl <<endl;
* /

// Sorting by the First Max Values

/* entrylist.sort(COMpare());
for (iter = entrylist.begin();iter != entrylist.end(); iter++)
{
    cout << iter->get_max_value()
        << " " <<iter->get_2nd_max_value() << endl;
}
cout <<endl <<endl; /*

// Sorting by the Second Max Values
entrylist.sort(COMPARE_SECONDS());
counter=0;
for (iter = entrylist.begin(); counter < TOP_ENTRIES_TO_OUTPUT ; iter++)
{
    for (int i=0;i<NUMBER_INPUTS;i++)
        cout << iter->get_input_value(i) << " ";
    cout << endl;
    //cout << iter->get_max_value() <<" " <<iter->get_2nd_max_value() << endl;
    counter++;
}

return 1;
}
Batchsort.cc

/* batchsort.cc

This program will take an extremely large input file of values
and batch process them through the neural net. After each
iteration, the program will read the output and keep a list of
top values from the sorting method.

The output is the top input values from the processing */

#include <fstream.h>
#include "sort.h"

const int batchsize = 3000; // amount processed each iteration
const int finalsize = 100;  // amount listed at end of process

int
main ()
{
    bool endoffile = false;
    int batchcounter = 0, counter = 0;
    char filename[] = "../data/averagedata";
    char tempfilename[] = "../data/tempbatchdata";
    char processcommand[] = "./main processonly > /dev/null";
    char buffer[255];
    char commandstring[255];

    int intarray[NUMBER_INPUTS];
    float floatarray[NUMBER_OUTPUTS];
    int i, j, internalcounter;
    ENTRY entryarray[MAX_ENTRIES];
    char line[1000];
    char *next_pointer;
    list < ENTRY > entrylist;
    list < ENTRY >::iterator iter;
//initialize the list to size of batchsize + finalsize

//open the input file and process up to the final size
ifstream fin (filename);

// open the temp file
ofstream fout (tempfilename);

while (!endoffile && counter < finalsize)
{
  // read a line
  if (!fin.getline (buffer, 255))
  {
    endoffile = true; // end of file

  }
  else
  {
    fout << buffer << "\n";
    // print the line to the temp file
  }

  counter++;  
}

// close temp file
fout.close();
// Allow Neural Net to process the tempfile

strcpy (commandstring, "rm ../data/test.dat;ln -s ");
strcat (commandstring, tempfilename);
strcat (commandstring, " ../data/test.dat");

system ("./main processonly");

// Pull all of the results into the list

ifstream fin_output ("../data/output.dat");
i = 0;
fin_output.getline (line, 1000); // Discards Input Line Text

do
{
    fin_output.getline (line, 1000); // Grabs Weights
    intarray[0] = (int) strtod (line, &next_pointer);
    internalcounter = 1;
    while (internalcounter < NUMBER_INPUTS)
    {
        intarray[internalcounter] = (int) strtod (next_pointer, &next_pointer);
        internalcounter++;
    }

    fin_output.getline (line, 1000); // Discards Output Line Text
    fin_output.getline (line, 1000); // Grabs Outputs
    floatarray[0] = strtod (line, &next_pointer);
    internalcounter = 1;

    while (internalcounter < NUMBER_OUTPUTS)
    {
        floatarray[internalcounter] = strtod (next_pointer, &next_pointer);
        internalcounter++;
    }
}
fin_output.getline (line, 1000); //Discards dashed line

entryarray[i].set_input_values (intarray);
entryarray[i].set_output_values (floatarray);
entryarray[i].set_max_value ();
entrylist.push_back (entryarray[i]);
i++;
fin_output.getline (line, 1000); // Discards Input Line Text

}
while (!fin_output.eof ());

fin_output.close ();

// Begin Batch Processing
while (!endoffile)
{
    counter = 0;
    ofstream fout (tempfilename);
    cout << "Processing Batch " << batchcounter << endl;
    while (!endoffile && counter < batchsize) // individual batch process
    {
        if (!fin.getline (buffer, 255))
        {
            endoffile = true;
            fout.close ();
            //close the tempfile
        }
        else
        {
            fout << buffer << "\n"; // print the line to the temp file
        }
        counter++;
    }
    fout.close ();
// close tempfile

// Allow Neural Net to process the tempfile

strcpy(commandstring, "rm ../data/test.dat\n ln -s ");
strcat(commandstring, tempfilename);
strcat(commandstring, " ../data/test.dat");
system("./main processonly");

// Pull all of the results into the last batchsize list items
ifstream fin_output("../data/output.dat");
i = finalsize;
fin_output.getline(line, 1000); // Discards Input Line Text
do {
    fin_output.getline(line, 1000); // Grabs Weights
    intarray[0] = (int) strtod(line, &next_pointer);
    internalcounter = 1;
    while (internalcounter < NUMBER_INPUTS)
    {
        intarray[internalcounter] = (int) strtod(next_pointer, &next_pointer);
        internalcounter++;
    }

    fin_output.getline(line, 1000); // Discards Output Line Text
    fin_output.getline(line, 1000); // Grabs Outputs
    floatarray[0] = strtod(line, &next_pointer);
    internalcounter = 1;

    while (internalcounter < NUMBER_OUTPUTS)
    {
        floatarray[internalcounter] = strtod(next_pointer, &next_pointer);
        internalcounter++;
    }
}
fin_output.getline (line, 1000);  //Discards dashed line
entryarray[i].set_input_values (intarray);
entryarray[i].set_output_values (floatarray);
entryarray[i].set_max_value ();
entrylist.push_back (entryarray[i]);
i++;
fin_output.getline (line, 1000);  // Discards Input Line Text
}
while (!fin_output.eof ());

// sort the list
entrylist.sort(COMPARE_SECONDS());

batchcounter++;
//get rid of the lessor values
counter=0;
for (iter = entrylist.begin(); counter < finalsize ; iter++)
{
    counter++;
}
entrylist.erase( iter ,entrylist.end());

}

// output the final size of the list
counter=0;
for (iter = entrylist.begin(); counter < finalsize ; iter++)
{
    for (int i=0;i<NUMBER_INPUTS;i++)
        cout << iter->get_input_value(i) << " ";
    cout << endl;
    //cout << iter->get_max_value() <<" " <<iter->get_2nd_max_value() << endl;
}
counter++;  
}  
fin.close();  
}
A.3.2 Classifydata.tcl

#!/usr/bin/wish

proc drawgray {num_list} {
  set counter 0
  for {set y 0} {$y < 8} {incr y} {
    for {set x 0} {$x < 8} {incr x} {
      set tempnumber [lindex $num_list $counter]
      while {$tempnumber == ""} {
        incr counter
        set tempnumber [lindex $num_list $counter]
      }
      set number $tempnumber
      if {$number == 16} {
        set number 15
      }
      set number [expr 15 - $number]
      set colorstring [format "#%X%X%X" $number $number $number]
      .main create rectangle [expr $x*24] \ 
        [expr $y*24] \ 
        [expr $x*24 + 24] \ 
        [expr $y*24 + 24] \ 
        -outline {} -fill $colorstring -width 0
      .smallmain create rectangle [expr $x*8] \ 
        [expr $y*8] \ 
        [expr $x*8 + 8] \ 
        [expr $y*8 + 8] \ 
        -outline {} -fill $colorstring -width 0
      incr counter
    }
  }
}

proc drawscatter {num_list} {

set counter 0
for {set y 0} {$y < 8} {incr y} {
    for {set x 0} {$x < 8} {incr x} {
        set tempnumber [lindex $num_list $counter]
        while {$tempnumber == ""} {
            incr counter
            set tempnumber [lindex $num_list $counter]
        }
        set number $tempnumber
        for {set i 0} {$i < 17} {incr i} {
            array set setarray [list $i 0]
        }
        for {set i $number} {$i != 0} {incr i -1} {
            set flag 0
            while {$flag == 0} {
                set rand_num [expr int((rand()*16) + 1)]
                if {[[lindex [array get setarray $rand_num] 1] == 0]} {
                    .main create rectangle \
                    [expr $x*24 + (($rand_num-1) % 4) *6] \ 
                    [expr $y*24 + (($rand_num-1) / 4)*6] \ 
                    [expr $x*24 + 6 + (($rand_num-1) % 4)*6] \ 
                    [expr $y*24 + 6 + (($rand_num-1) / 4)*6] \ 
                    -outline {} -fill #000 -width 0
                    .smallmain create rectangle \
                    [expr $x*8 + (($rand_num-1) % 4) * 2] \ 
                    [expr $y*8 + (($rand_num-1) / 4)*2] \ 
                    [expr $x*8 + 2 + (($rand_num-1) % 4)*2] \ 
                    [expr $y*8 + 2 + (($rand_num-1) / 4)*2] \ 
                    -outline {} -fill #000 -width 0
                    set flag 1
                    array set setarray [list $rand_num 1]
                }
            }
        }
    }
}

117
incr counter
}
}

set filename "../data/preclassify.dat"
set outputfilename "../postclassify.dat"
set filehandle [open $filename r]

set textline ""

button .switchview -text "Switch Viewing Method" \
    -command {set selection "switch"}
label .title -text "Grey Scale Mode"
canvas .main -height 192 -width 192 -background white
canvas .smallmain -height 64 -width 64 -background white
frame .topbuttons
frame .bottombuttons
button .notknown -text "Unknown" -command {set selection "unknown"}
label .filler -text ""
button .quit -text "Quit" -command {set selection "quit"}

button .topbuttons.b0 -text "0" -width 2 \
    -command {set selection "1 0 0 0 0 0 0 0 0 0"}
button .topbuttons.b1 -text "1" -width 2 \
    -command {set selection "0 1 0 0 0 0 0 0 0 0"}
button .topbuttons.b2 -text "2" -width 2 \
    -command {set selection "0 0 1 0 0 0 0 0 0 0"}
button .topbuttons.b3 -text "3" -width 2 \
    -command {set selection "0 0 0 1 0 0 0 0 0 0"}
button .topbuttons.b4 -text "4" -width 2 \
    -command {set selection "0 0 0 0 1 0 0 0 0 0"}
button .bottombuttons.b5 -text "5" -width 2 \
    -command {set selection "0 0 0 0 0 1 0 0 0 0"}
button .bottombuttons.b6 -text "6" -width 2 \
    -command {set selection "0 0 0 0 0 0 1 0 0 0"}
button .bottombuttons.b7 -text "7" -width 2 \
    -command {set selection "0 0 0 0 0 0 0 1 0 0"}
button .bottombuttons.b8 -text "8" -width 2 \
    -command {set selection "0 0 0 0 0 0 0 0 1 0"}
button .bottombuttons.b9 -text "9" -width 2 \
    -command {set selection "0 0 0 0 0 0 0 0 0 0"}
pack .title -fill x
pack .main
pack .smallmain
pack .topbuttons .bottombuttons
pack .notknown -fill x
pack .filler -fill x
pack .switchview -fill x
pack .quit -fill x

pack .topbuttons.b0 .topbuttons.b1 .topbuttons.b2 \
    .topbuttons.b3 .topbuttons.b4 -side left -fill x
pack .bottombuttons.b5 .bottombuttons.b6 .bottombuttons.b7 \
    .bottombuttons.b8 .bottombuttons.b9 -side left -fill x

gets $filehandle textline
while {! [eof $filehandle]} {
    set numberlist [split [string trim $textline]]
    drawgray $numberlist
    set mode "gray"
    vwait selection
    while {$selection == "switch"} {
        if {$mode == "gray"} {
            .main delete all
            .smallmain delete all
            drawscatter $numberlist
            set mode "scatter"
            .title configure -text "Pixel Scatter Mode"
        } else {
            .main delete all
            .smallmain delete all
            drawgray $numberlist
            set mode "gray"
        }
    }
}
.title configure -text "Grey Scale Mode"

} 
vwait selection
}

if {$selection == "quit"} {
    exit
}
if {$selection != "unknown"} {
    puts -nonewline $textline
    puts "$selection"
}
gets $filehandle textline
exit
A.3.3 Gnuplot Script

```plaintext
set size .5625, .5625
set term png
set output "figure.png"
set yrange [0:100]
set ytics 10
set xlabel "Iterations"
set ylabel "Incorrect Classification Percentage"
plot 'graphoutput.dat' every 4 using 1:($2*100) t "Training File" w lp, \\
    'graphoutput.dat' every 4 using 1:($4*100) t "Test File" w lp
```
Bibliography


