1998

Creation of a neural network to assist in deciphering degraded ancient Hebrew texts

Daniel Hentschel

SIMG-503

Senior Research

Creation of a Neural Network to Assist in Deciphering Degraded Ancient Hebrew Texts

Final Report

Daniel B Hentschel
Center for Imaging Science
Rochester Institute of Technology
May 1998

Table of Contents

Creation of a Neural Network to Assist in Deciphering Degraded Ancient Hebrew Texts

Daniel B Hentschel

Table of Contents

Title Page

Creation of a Neural Network to Assist in Deciphering Degraded Ancient Hebrew Texts

Daniel B Hentschel

Abstract

The study of ancient manuscripts is a very important field. These manuscripts hold the keys to our past as a human race. Unfortunately, many ancient manuscripts have been degraded to the point where they are no longer legible. Many techniques have been developed to try to read parts of illegible documents. Neural networks are a fairly new technology with an extremely wide range of applications. The goal of this project is to determine how useful a neural network would be as a tool to help in deciphering ancient manuscripts, and, more specifically, Hebrew manuscripts. An image processing application was created to do preprocessing on the characters, and then two neural networks were created to see how well they could perform when analyzing degraded characters. The results were not completely conclusive, but they seem to indicate that a neural network would not be a very good tool to use in analysis of degraded characters.

Table of Contents

Center for Imaging Science
Rochester Institute of Technology
Rochester, NY 14623-5604

This report is accepted in partial fulfillment of the requirements of the course SIMG-503 Senior Research.

Title: Creation of a Neural Network toAssist in Deciphering Degraded Ancient Hebrew Texts
Author: Daniel B Hentschel
Project Advisor: Dr. Robert Johnston
SIMG 503 Instructor: Joseph P. Hornak

Table of Contents

Creation of a Neural Network to Assist in Deciphering Degraded Ancient Hebrew Texts

Daniel B Hentschel

Introduction

Deciphering ancient manuscripts is an important step to learning about the past of the human race, and to study how we have progressed to the point that we are at today. The study of our past can be applied to build useful predictions for our future, and to decide how best to proceed towards that future. Many ancient Hebrew manuscripts have been found in recent years. Unfortunately, most of them have been degraded to such an extent that only small sections of them are still legible, as shown in **Figure 1**. It is hoped that the results of this research can be applied to help decipher parts of these documents that are currently illegible. The concentration of this project is to decipher ancient Hebrew manuscripts, however the findings need not be limited to this one application. They may also be applicable to any other situation in which degraded images must be analyzed to discover the statistical probability that they match a given pattern.

Figure 1 Degraded Ancient Hebrew text

Not much research has been done in the field of having a computer attempt to recognize characters that are not recognizable by a human. Even computer recognition of characters easily recognizable by humans is problematic at best. [1-4] One method of recognizing handwritten characters is through the use of neural networks. Neural networks are analytical systems designed to solve problems that are not expressly spelled out to the computer. They are especially useful for extremely complex problems in which the interactions between factors are unknown, and are not easily determined. [5,6] This project will try to determine if it is practical to use a neural network as a tool to help decipher degraded ancient Hebrew texts.

Background

Operation of Neural Networks

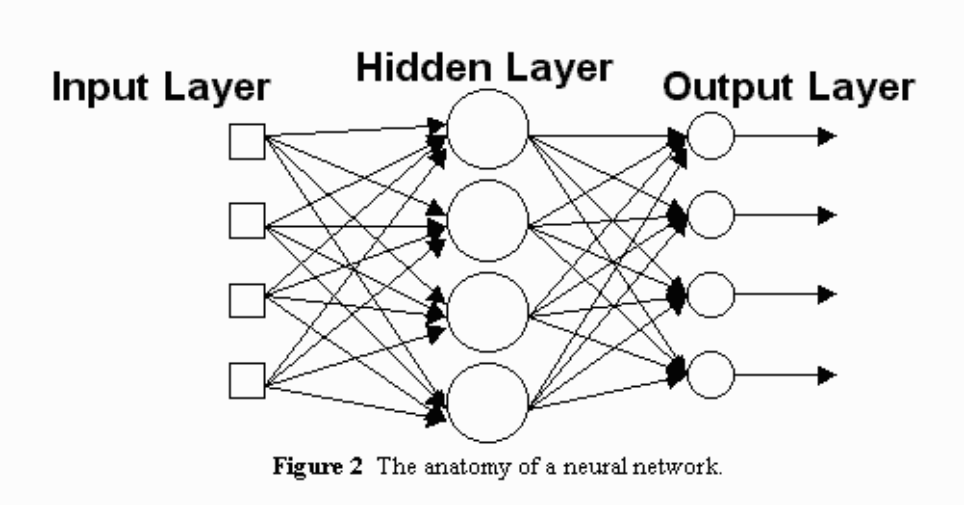

Figure 2 The anatomy of a neural network.

Neural networks can be thought of in three separate sections: 1) Input layer, 2) Hidden layer, and 3) Output layer (**Figure 2**). In the case of OCR (optical character recognition), the input would be specific, quantitative information about the character to be analyzed. The hidden layer would take this information and make weighted comparisons based on previous knowledge. The results of these comparisons would be passed to the output, which would, ideally, indicate what character had been processed at the input stage. [1] The hidden layer is where most of the processing in a neural network takes place. This layer is made up of a number of levels of interconnected nodes called neurons. These nodes are modeled after neurons found in the nervous system of all living creatures. They simply take an input, μ, apply this input to an internal limiting function, $S()$, and return $S(\mu)$ as an output between zero and one. Equation (1) shows the internal function that, as can be seen in **Figure 3**, limits the output of each node to a number between zero and one. The input, μ, is made up of the sum of several inputs taken from the outputs of other nodes. In equation (2), each x_i is one of the several inputs to the node. Each input is weighted by its own scaling factor, w_i, and a bias term, θ_i, which is sometimes called the threshold, is added to it. [5,6]

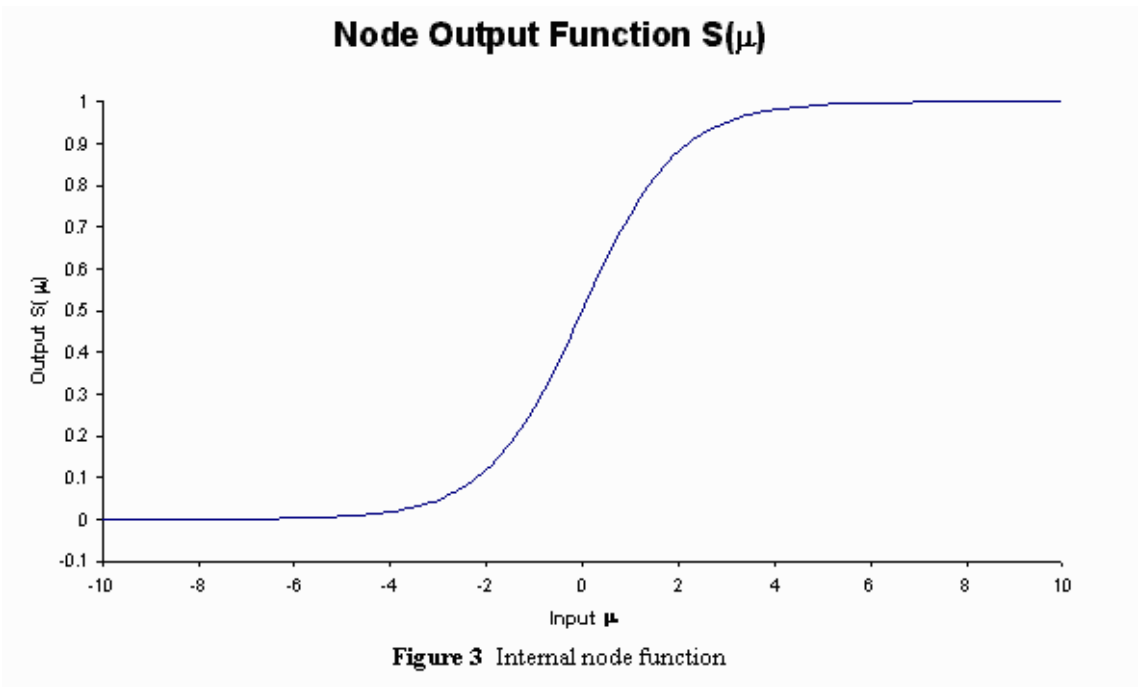

Figure 3 Internal node function

$$(1) \qquad S(\mu) = \frac{1}{1+e^{-\mu}}$$

$$(2) \qquad \mu = \sum_i x_i w_i + \theta_i$$

These nodes can be linked together with the output from any one node going to many other nodes, and the input to any one node coming from many other nodes. This can lead to many varied and complex network designs. Since the exact interaction between input factors that will give the desired output for any given input is not known, this must be "taught" to the neural network after the basic structure is set up. The most common method for neural network "learning" is called backpropagation. This method should be sufficient for this application. [3] In essence, when an input signal is fed into the network, the output from the network is compared to the desired output to produce an error signal. Then this error signal is sent backwards through the network, from the output layer, through the hidden layer, to input layer. As this error term is passed back, the weights and bias terms on each node are adjusted in an attempt to minimize the error. After this learning process is repeated many times with many different inputs, the network becomes better at producing the desired output for any given input. A more detailed description of how backpropagation works can be found in [6] on page 142.

For an example, **Figure 4**
is one of the most simple neural networks possible, but even this very simple neural network can examine some pretty complex problems. The equation that will determine the relationship between the input, *I*, and the output, *O*, is shown in equation (3). The terms w_{A1}, θ_{A1}, refer to the weight on input number 1 of node A, and the bias term for input number 1 on node A. The terms for node B are done similarly. As can be seen, the output, *O*, is actually a function of five variables, *I*, w_{A1}, w_{B1}, θ_{A1}, and θ_{B1}.

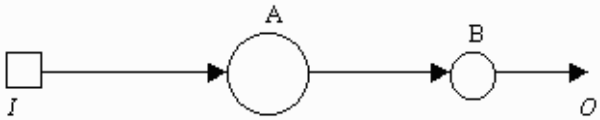

Figure 4 A very simple neural network

$$(3) \qquad O = \frac{1}{1+e^{-\left(w_{B1}\cdot\frac{1}{1+e^{(w_{A1}\cdot I+\theta_{A1})}}+\theta_{B1}\right)}}$$

If the neural network is increased in complexity just a little bit as shown in **Figure 5**, then the equations for the output get a lot more complex. Equations (4) and (5) give the two outputs to the neural network shown in **Figure 5**. Each output is now a function of 14 variables.

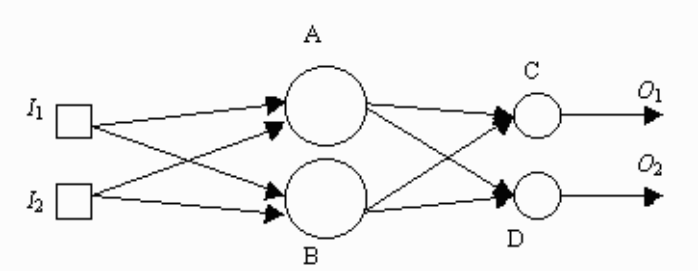

Figure 5 The anatomy of a neural network.

$$(4) \qquad O_1 = \frac{1}{1+e^{-(a+b)}}$$

$$a = \left(w_{C1} \cdot \frac{1}{1+e^{-\left(\left(w_{A1} \cdot I_1 + \theta_{A1}\right)+\left(w_{A2} \cdot I_2 + \theta_{A2}\right)\right)}} + \theta_{C1} \right)$$

$$b = \left(w_{C2} \cdot \frac{1}{1+e^{-\left(\left(w_{B1} \cdot I_1 + \theta_{B1}\right)+\left(w_{B2} \cdot I_2 + \theta_{B2}\right)\right)}} + \theta_{C2} \right)$$

$$(5) \qquad O_1 = \frac{1}{1+e^{-(a+b)}}$$

$$a = \left(w_{D1} \cdot \frac{1}{1+e^{-\left(\left(w_{A1} \cdot I_1 + \theta_{A1}\right)+\left(w_{A2} \cdot I_2 + \theta_{A2}\right)\right)}} + \theta_{D1} \right)$$

$$b = \left(w_{D2} \cdot \frac{1}{1+e^{-\left(\left(w_{B1} \cdot I_1 + \theta_{B1}\right)+\left(w_{B2} \cdot I_2 + \theta_{B2}\right)\right)}} + \theta_{D2} \right)$$

Equation (6) gives a formula for determining how many variables will be in the equation relating the inputs of a neural network to one of the outputs of the network. The variable *i* represents the number of inputs to the network, and the variable *j*
represents the number of nodes in the hidden layer. As can be seen in this equation, increasing the number of nodes in the hidden layer greatly increases the complexity of the output function. As stated before, the neural network is just a tool to solve a complex mathematical model in which the relationship between the inputs and the outputs is not known. The more complex the problem, the more nodes will probably be needed in the hidden layer of the neural network in order to find a suitable solution.

$$(6) \quad \text{Number of Variables} = i + 2ij + 2j$$

Thinning Algorithm

Some of the input features that will be used to describe characters to the neural network will require the character to be thinned to a skeleton before they can be computed. The thinning algorithm used in this project is an adaptation from the algorithm described in [7]. Several modifications to the technique were made since the algorithm didn't seem to work exactly correctly when coded literally as the paper describes it. An image to be thinned is first thresholded. Then each black pixel in the image is analyzed to determine if it is an edge pixel. This is done by comparing neighboring pixels on opposing sides, horizontally and vertically, of the pixel in question. If one neighboring pixel is black and the neighboring pixel on the other side is white, then the pixel in question is an edge pixel. Otherwise it is not. (See **Figure 6**.) If it is an edge pixel, then it is marked to be analyzed later. Once all the edge pixels have been marked, each edge pixel is examined one by one to determine whether or not it should be deleted.

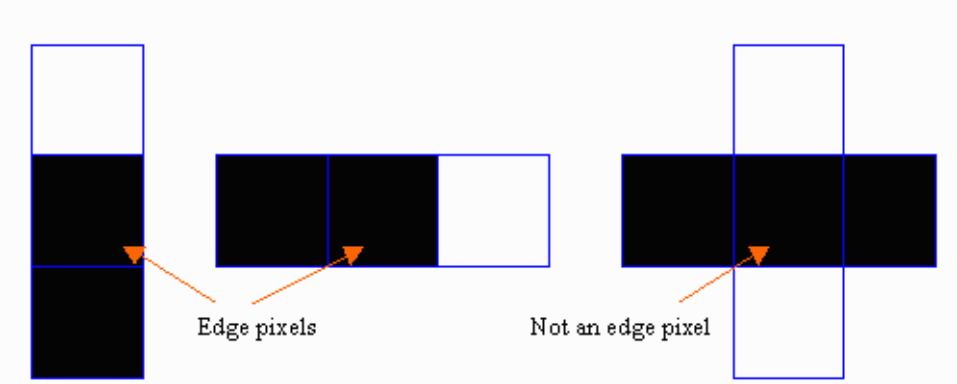

Figure 6 Determining edge pixels for the thinning algorithm

Test 1: The conditions in **Figure 7**
are examined to see which ones apply to the current pixel. If any one of conditions **a** through **l** is true and either condition **m** or **n**
is true, then the pixel cannot be deleted and the algorithm moves on to test the next marked pixel.

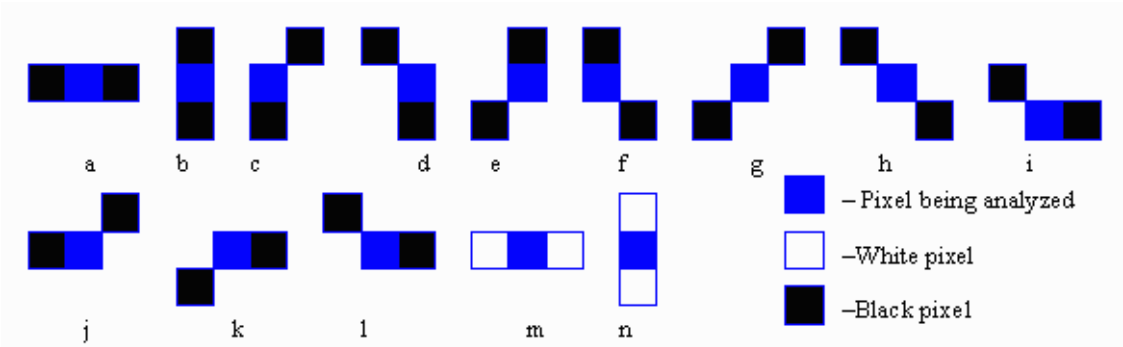

Figure 7 Conditions to determine if a pixel should be deleted

Test 2:
If the pixel has exactly one neighbor and that neighboring pixel has more than two neighbors, then the pixel in question is just an offshoot due to noise. (See **Figure 8**.) The pixel is deleted and the algorithm moves on to test the next marked pixel.

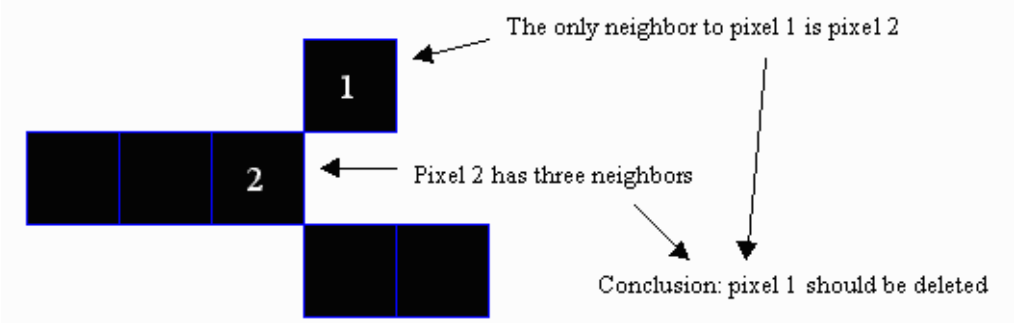

Figure 8 Determining if a pixel with one neighbor is noise

Test 3:
If the pixel has zero neighbors, then it is an isolated pixel and is probably unimportant. The pixel is deleted and the algorithm moves on to test the next marked pixel.

Test 4:
If the pixel has two or more neighbors, then the pixel must be analyzed along with each of its neighbors, one neighbor at a time, comparing their configurations to **Figure 9**. If any of the four conditions in **Figure 9** is true for any one of the neighbors, then deleting this pixel will break connectivity of the line. The pixel cannot be deleted. If none of these conditions are true, then the pixel is deleted. In either case, the algorithm then moves on to test the next marked pixel.

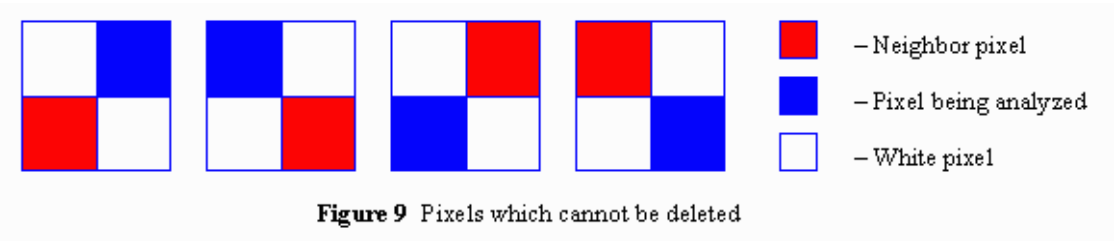

Figure 9 Pixels which cannot be deleted

Once all the marked pixels have been analyzed, there is one more test performed in the thinning algorithm. This test is done to keep lines looking fairly straight. Each black pixel in the image is compared to the four crooked line conditions in **Figure 10**. If one of those conditions exists, then it is changed to the corresponding straight-line condition.

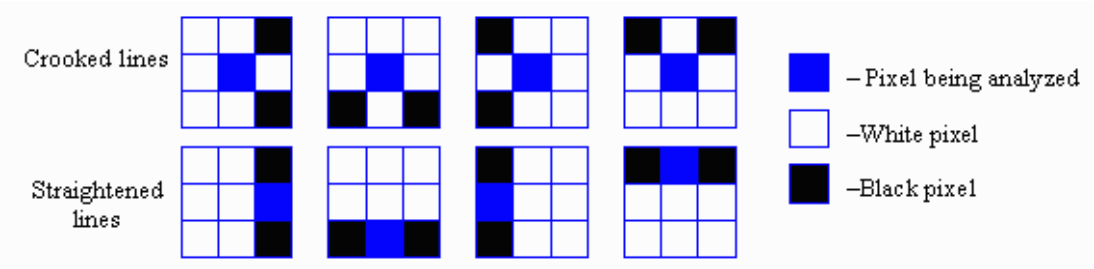

Figure 10 Crooked line test

Now, if any of the edge pixels in the image were deleted when testing the marked pixels in this pass, then the thinning algorithm starts again by marking new edge pixels, analyzing marked pixels to see if they can be deleted, and fixing crooked lines. This process repeats until no more pixels can be deleted.

Application of a Backpropagation Network to OCR

De Bruyne and Korolink [1] used a backpropagation model in their paper on recognizing Hebrew characters. They experienced recognition rates of about 96% after a learning period of 8000 training samples. Here are the criteria

that de Bruyne and Korolink gave to the network as input to produce these results:

1) Number of black regions and their size at three height levels as well as a record of the transition levels.
2) Area of the black part of the character, as a percentage of the circumscribed rectangle.
3) Length of the outline as a percentage of the circumscribed rectangle perimeter.
4) Number of changes of direction encountered in following the outline.
5) Complexity: Square root of the black area divided by the outline perimeter.

The first factor could probably be made more robust for analyzing degraded characters. Currently it calls for the computer to find the number of black regions on a horizontal line at three different heights. This measurement could be badly skewed by small amounts of degradation. For example, in the character shown in **Figure 11**, a small amount of degradation could result in only two black regions being counted near the top of the character, rather than three. If this factor were changed to a calculation of the density of pixels at three height levels, calculated as the sum of all pixel values in a region divided by the number of pixels in the region, it might not be so sensitive. This could be done for vertical as well as horizontal regions. Also, since this will give a good indication of the area of the character compared to the area of the circumscribed rectangle, factor 2 will no longer provide any distinct information, and can be eliminated.

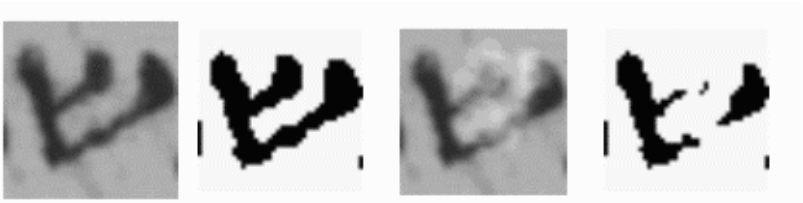

Figure 11 Effect of degradation

In their paper, the authors confess that they had not, at the time of writing, fully explored the possibility of reducing the character to a skeleton. They complain of noise in factor 4, the number of changes in direction, when the outline of the characters is a little rough. It seems that thinning the character using the previously described thinning algorithm should alleviate some of this noise. It should also allow the addition of other useful features including the number of intersections and end points in the skeleton of the character. Skeletonizing the characters should also make the complexity calculation a bit easier. It is difficult to see exactly how the skeleton and the area of a character can be related to its complexity. **Figure 12** illustrates this relation. Each of the three shapes in the image on the left has the exact same area. However, their skeletons all have different lengths. Therefore, the ratio of area to length (complexity factor) will be different for each shape.

Figure 12 Skeletons of three different shapes with the same

It is hoped that after modifying these factors in the way described, the character recognition process will be more resilient when degraded characters are processed. It is understood that with the techniques described here, a computer will never be able to perform character recognition tasks better than (or not even as well as) a human can. The purpose of this research is not to try to replace the human mind, but to create a tool to augment it. For example, suppose that a linguist is studying an ancient manuscript and comes across a character that he can't decipher. If he puts the character into his computer for processing, it would be nice if he could get back from the computer a list of three or four possible characters that it could be. If one of these options is something he hasn't

considered before, then he can try fitting that character into the document, seeing how well it matches with the grammar and sentence structure already present. In this way, the computer doesn't do all the work itself, but is used as a tool by the analyst to suggest possibilities perhaps overlooked.

Methods

Design of a Neural Network

The first step in this project was to design the neural network that would be used to perform character recognition. Design of a neural network is simple once the number of inputs and outputs are known. In this case, there were originally planned to be ten inputs:

1) Density of pixels in the top third of the character.
2) Density of pixels in the middle third of the character.
3) Density of pixels in the bottom third of the character.
4) Density of pixels in the left half of the character.
5) Density of pixels in the right half of the character.
6) Ratio of the length of the character skeleton to half the circumscribed rectangle perimeter.
7) Complexity: Square root of the black area of the character divided by the length of the skeleton.
8) Number of dead ends encountered in the character skeleton.
9) Number of intersections in the character skeleton.
10) Number of changes of direction in the character skeleton.

Due to time constraints, the tenth input, the number of changes of direction in the character skeleton, was left out. This leaves nine inputs to the neural network. To simplify the first experiments, the neural network was initially designed with four outputs. Each output would correspond to a single Hebrew character, as shown in **Figure 13**.

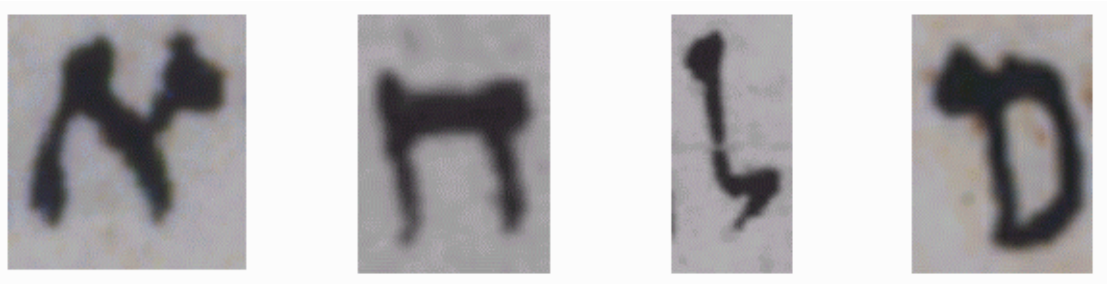

Figure 13 Outputs from the neural network

With the neural network set up this way, nine numbers will be sent to the input of the network. These nine numbers will be the nine features described above, taken from one of the four characters in **Figure 13**. Then, four numbers will be collected at the output. Hopefully the largest of these four numbers will correspond to the character being analyzed. For example, if information about character 2 is sent to the neural network, it is hoped that output 2 will be the highest of the outputs.

Now that the input and output layers are determined, all that remains is to figure out how large the hidden layer should be. This can be done experimentally by testing how well the neural network performs with different numbers of nodes in the hidden layer. It turns out that two nodes seem to be sufficient, and there doesn't seem to be any performance gain by increasing this number. This will be discussed in more detail later in **Training and Testing of a Neural Network**. **Figure 14**
shows the resulting design for a neural network with nine inputs, two nodes in the hidden layer, and four outputs. According to equation (6), each of the four outputs in this network will be a function of 49 variables.

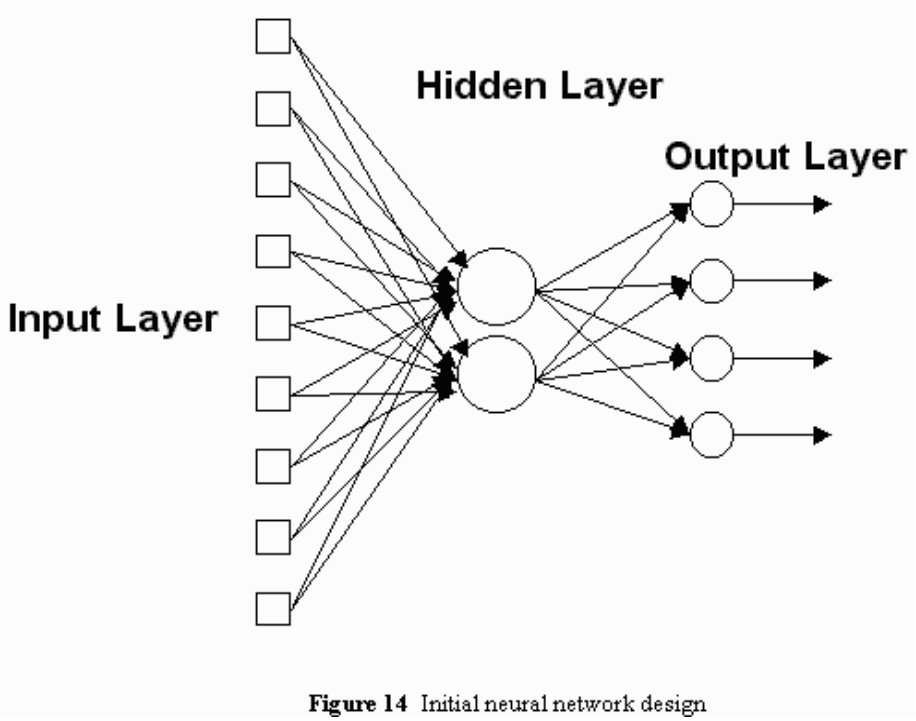

Figure 14 Initial neural network design

Creation of an Image Processing Application

Now that the necessary inputs to the neural network have been determined, an application must be created which can calculate these input values when given an image with characters in it. This application was built using the Microsoft Visual Basic development environment. The application can be found at the following web address:

ocr.zip

The first thing that the program needed to be able to do was to read in images with text in them and segment the individual characters for processing. Since segmentation was not the focus of this research project, a simple, easy to program segmentation algorithm was used. The algorithm first separates lines of text by calculating the average pixel density for each horizontal scan line. It then thresholds this image, and the resulting image clearly shows where the lines of text are, as can be seen in **Figure 15**. Then, the algorithm analyzes each line of text individually, calculating an average pixel value for each vertical scan line and thresholding it. It then uses this image as a mask to separate the individual characters. As can be seen in **Figure 16**, the algorithm is not very robust, and often makes mistakes. It is good enough for the requirements of this application, though.

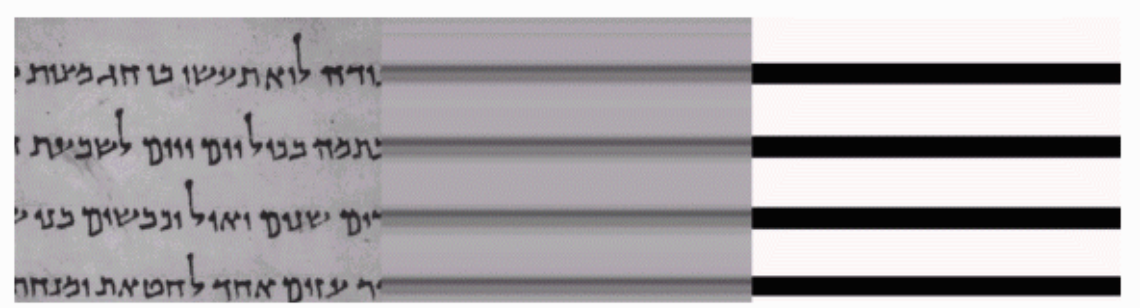

Figure 15 Segmenting lines of text

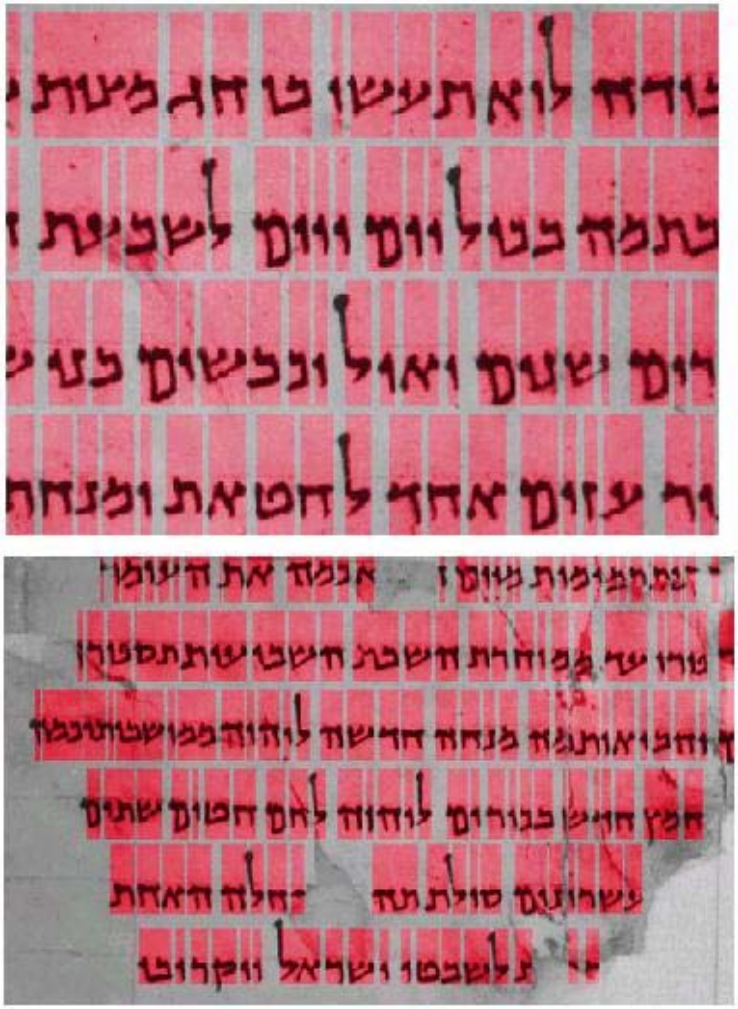

Figure 16 Masks created by the segmentation algorithm

Once the characters have been segmented from a document, the user can select one of them for analysis. Since features 1-6 from the input features to the neural network are very dependent on the size of the character in relation to the size of the circumscribed rectangle, the character is cropped to the smallest rectangle possible when it is selected by the user. This cropping is acheived by thresholding the character at the pixel value corresponding to fifteen percent of the range between the maximum pixel value and minimum pixel value in the character. Then any horizontal scan lines that have no character pixels (black pixels) in them can be cropped from the rectangle.

The application allows the user to perform three image-processing operations to a selected character. It will let the user threshold the character, thin the character to a skeleton (code for the thinning algorithm can be found in Appendix A), or equalize the histogram of the character. The application allows the user to create a new neural network, save the current neural network, and open an old neural network. When creating a new neural network, it allows the user to control the number of nodes in the hidden layer and output layer, the gain of the node function, the learning rate of the network, and the momentum factor in the learning process. The application will allow the user to create a training set of characters to train the neural network with. The application will also allow the user to apply any character to the neural network for processing, and will display the output on the screen. **Figure 17** shows a picture of the application in use.

Figure 17 Imaging application developed for this project

The Visual Basic code for the neural network was adapted from the C code: "Backpropagation Network with Bias Terms and Momentum". [8] The code for the Visual Basic class can be found in Appendix C. When a set of training characters is applied to a neural network, the application will select one of the training characters at random and give it to the neural network. It will then tell the network which character it had been given, and ask the network to compute the total error in its output. The application will do this a number of times, depending on how many characters are in the training set, and then it will average all the errors. If the average of the errors is less than it was last time, then the error is going down, and the network is getting smarter. If the average is more than last time, then the error is no longer going down, and the network is probably as smart as it will get. Once the error goes up, the application will stop training the network since further training will probably have little effect. **Figure 18** shows an example of how the error decreases as the neural network is trained. While it is difficult to see on the large chart, the blowup of the last few points shows that the error does, in fact, go up on the last data point, indicating that the neural network has been trained enough.

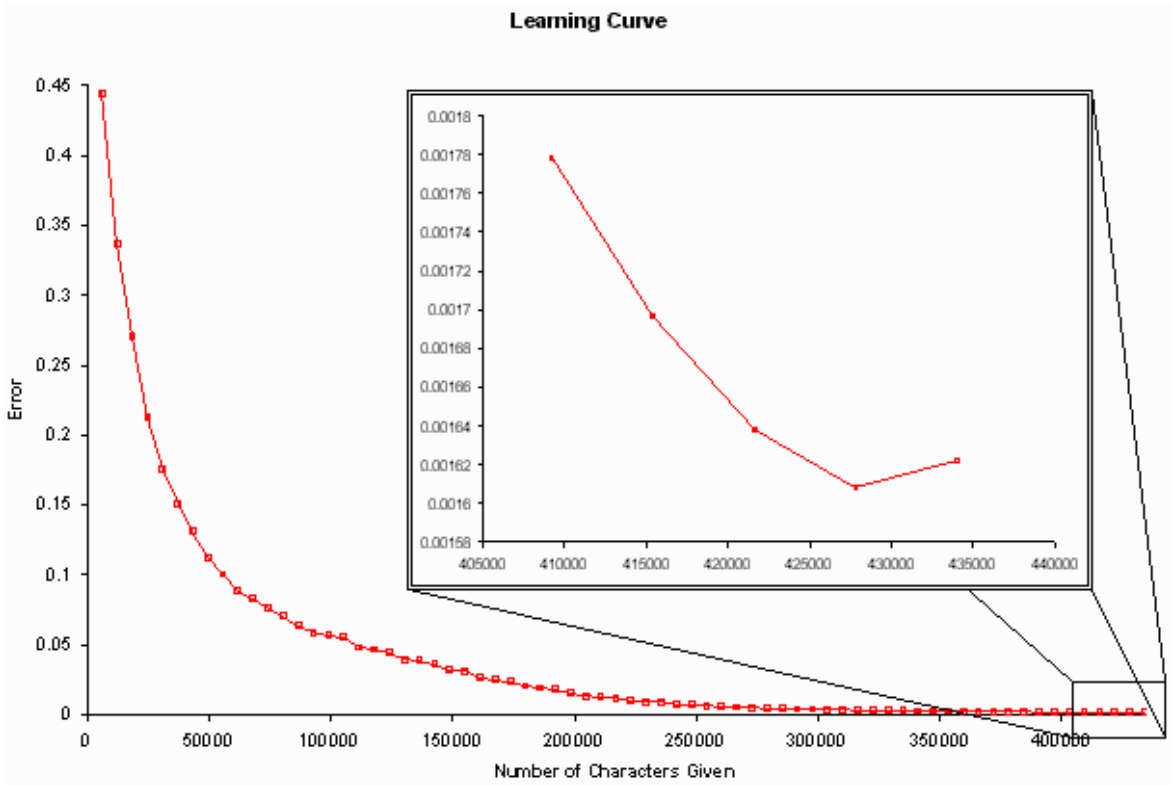

Figure 18 Output error of a neural network as a function of learning cycles

Training and Testing of a Neural Network

The first neural network to be tested was one with four outputs. The training set used is shown in **Figure 19**. Since the neural network will be trained to give a high response in output 1 and a low resonse in all the other outputs whenever a form of the character in the first column is input to it, that character will be referred to from now on as the output 1 character. Similarly, the character in the second column is the output 2 character, the third column is the output 3 character, and the fourth column is the output 4 character. All the neural networks created in this project were trained with a learning rate (Eta) of 0.25, a momentum factor (Alpha) of 0.9, and a gain of 1.0.

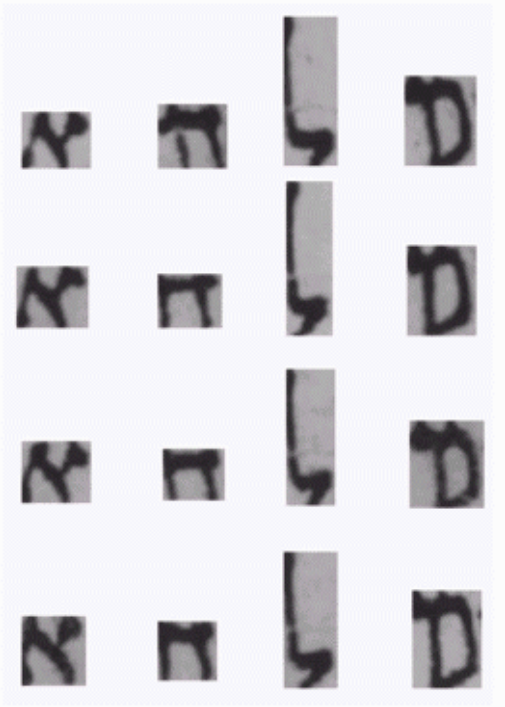

Figure 19 A sixteen character training set

In order to determine the optimum number of nodes in the hidden layer, several different configurations were trained with the same training set until a good configuration was found. There were two criterion for a good configuration. The neural network had to be able to identify each character in the training set with 100% accuracy when fully trained. The neural network also had to be able to identify new characters with better than 90% accuracy. First, a neural network was tried with one node in the hidden layer, then another was tried with two nodes, then three nodes, etc. After the optimum configuration of the neural network was found, this network was trained using the training set in **Figure 19**
and then saved for later use. Several images with characters in them were selected to use as test images for this network to see how well it performs. **Figures 20-23** show the images that were selected.

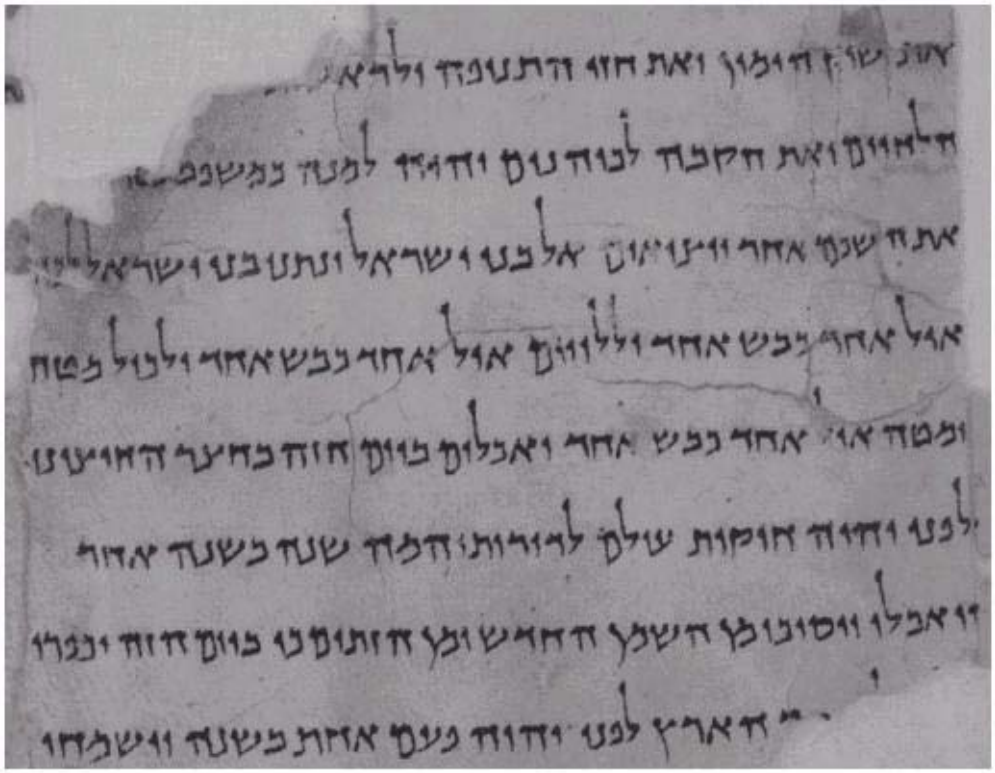

Figure 20 Test characters with minimal degradation

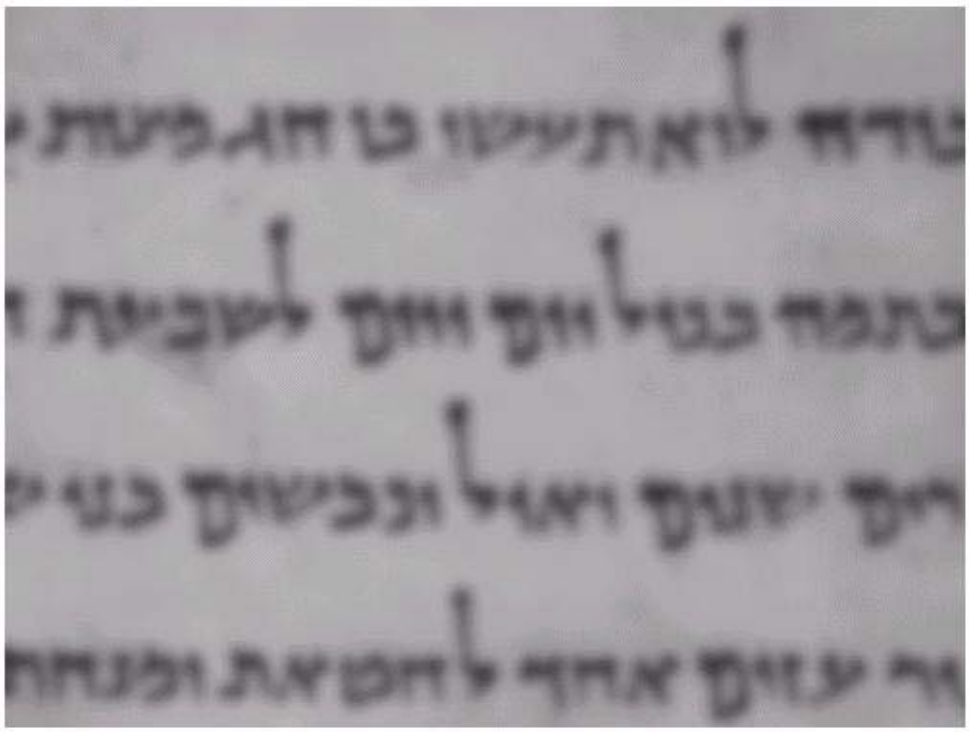

Figure 21 Characters blurred with a Gaussian filter

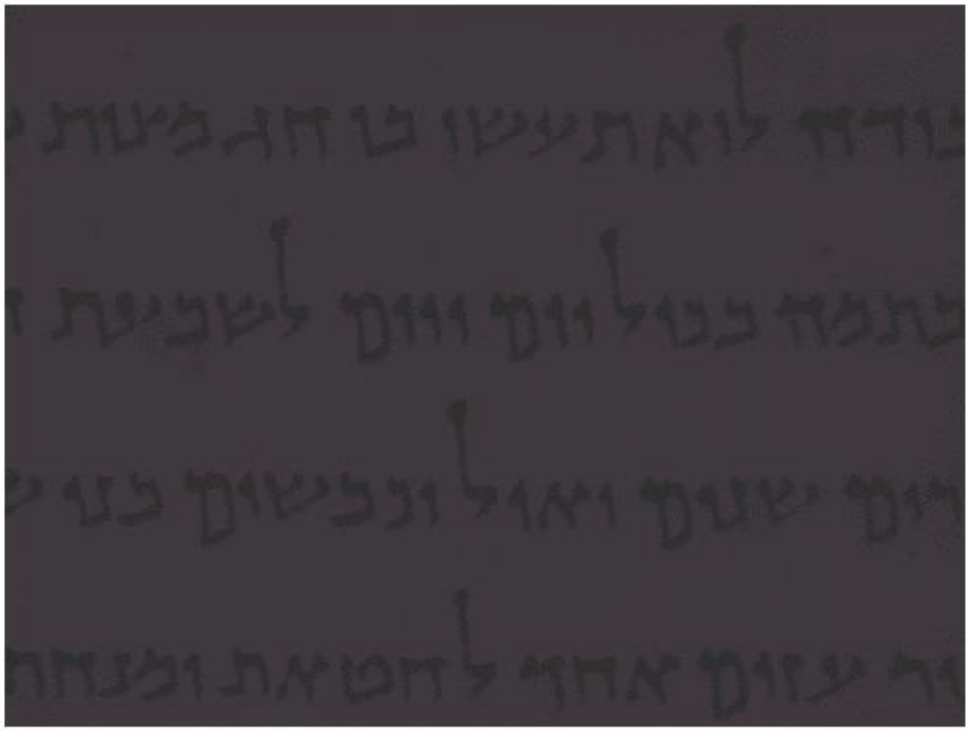

Figure 22 Characters with decreased contrast and brightness levels

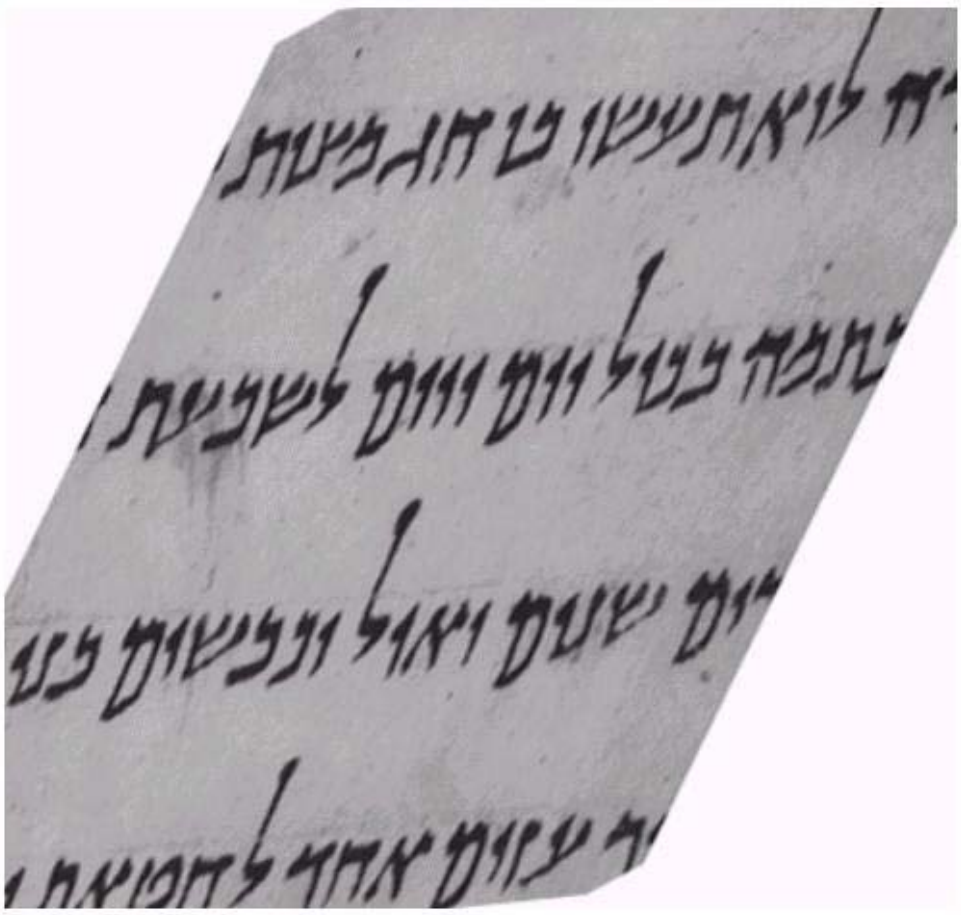

Figure 23 Badly skewed characters

After analyzing the performance of this network with four outputs, a new network was created with ten outputs. Again, several models were tested to determine how many nodes would be needed in the hidden layer. **Figure 24**

shows the training set used for this network.

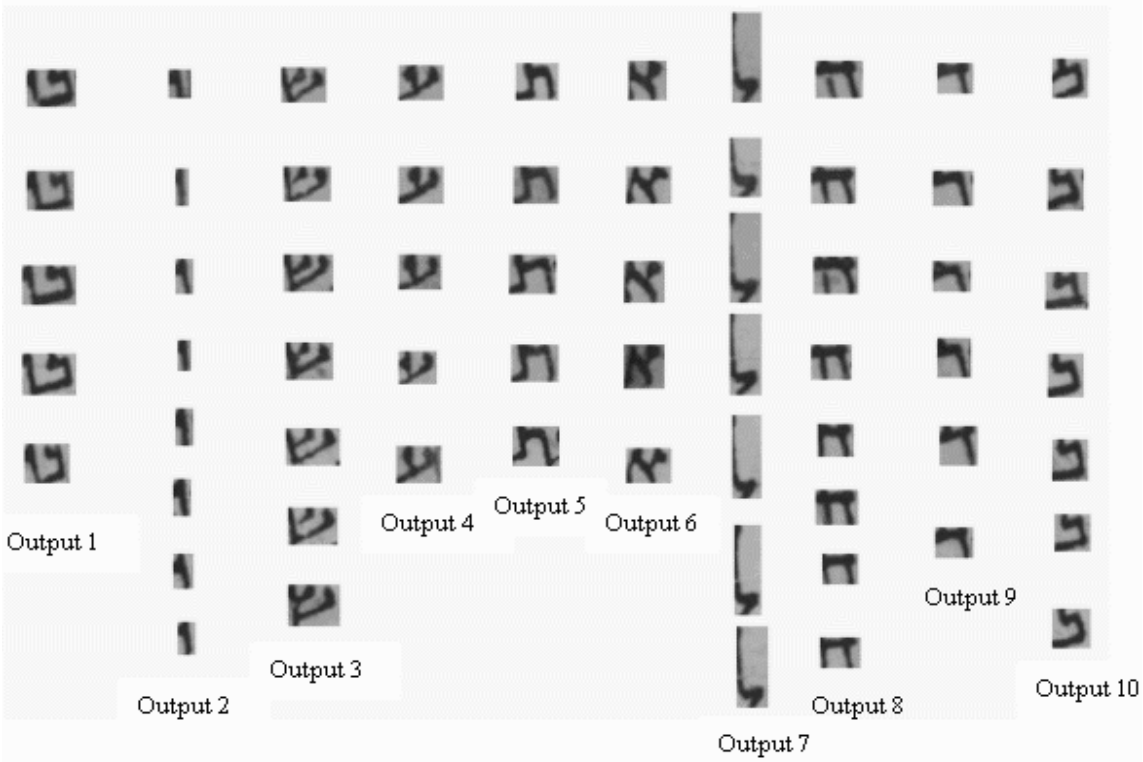

Figure 24 A sixty-three character training set

Once a good configuration for this network was found and trained, its performance on **Figures 20-23** was also analyzed. The results from this analysis and from the performance of the four output network were studied to determine if a neural network would be helpful in analysis of degraded characters.

Results

Neural Network with Four Outputs

For the neural network with four outputs, five configurations of the hidden layer were tested. **Table 1** shows the results of this testing.

Table 1 Effectiveness of Five Different Four Output Networks

Nodes in Hidden Layer	*Training Cycles*	*Final Output Error*	*Incorrectly Classified Training Characters*
1	11,200	0.194	4
2	108,800	0.00030	0
3	110,400	0.00022	0

4	49,600	0.00040	0
8	41,600	0.00037	0

With more nodes in the hidden layer, the network learned more quickly, but its overall performance didn't change noticeably. **Figure 25** shows the learning curves for each of the configurations tested.

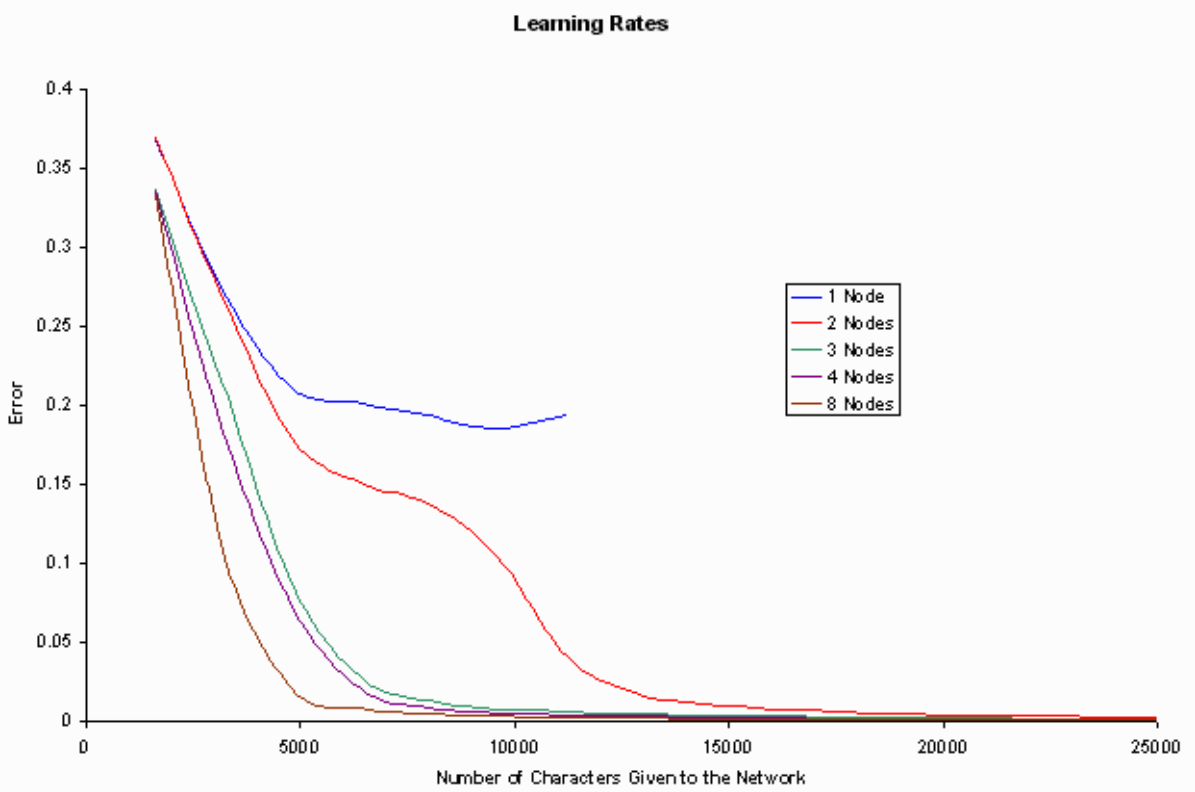

Figure 25 Output error curves for five different configurations of a four output neural network

The neural network configuration with two nodes in the hidden layer was chosen for the rest of the analysis. The cropped characters in **Figure 26** are the characters from **Figure 20** that were tested with this network. There were twenty-three in all, and of those, four were incorrectly classified by the neural network. Those four are circled in red on **Figure 26**. They are all output 3 characters, and each one of them appears to be poorly cropped, cutting off most of the stem on top. This probably accounts for why they were incorrectly classified. They don't look much like an output 3 character should.

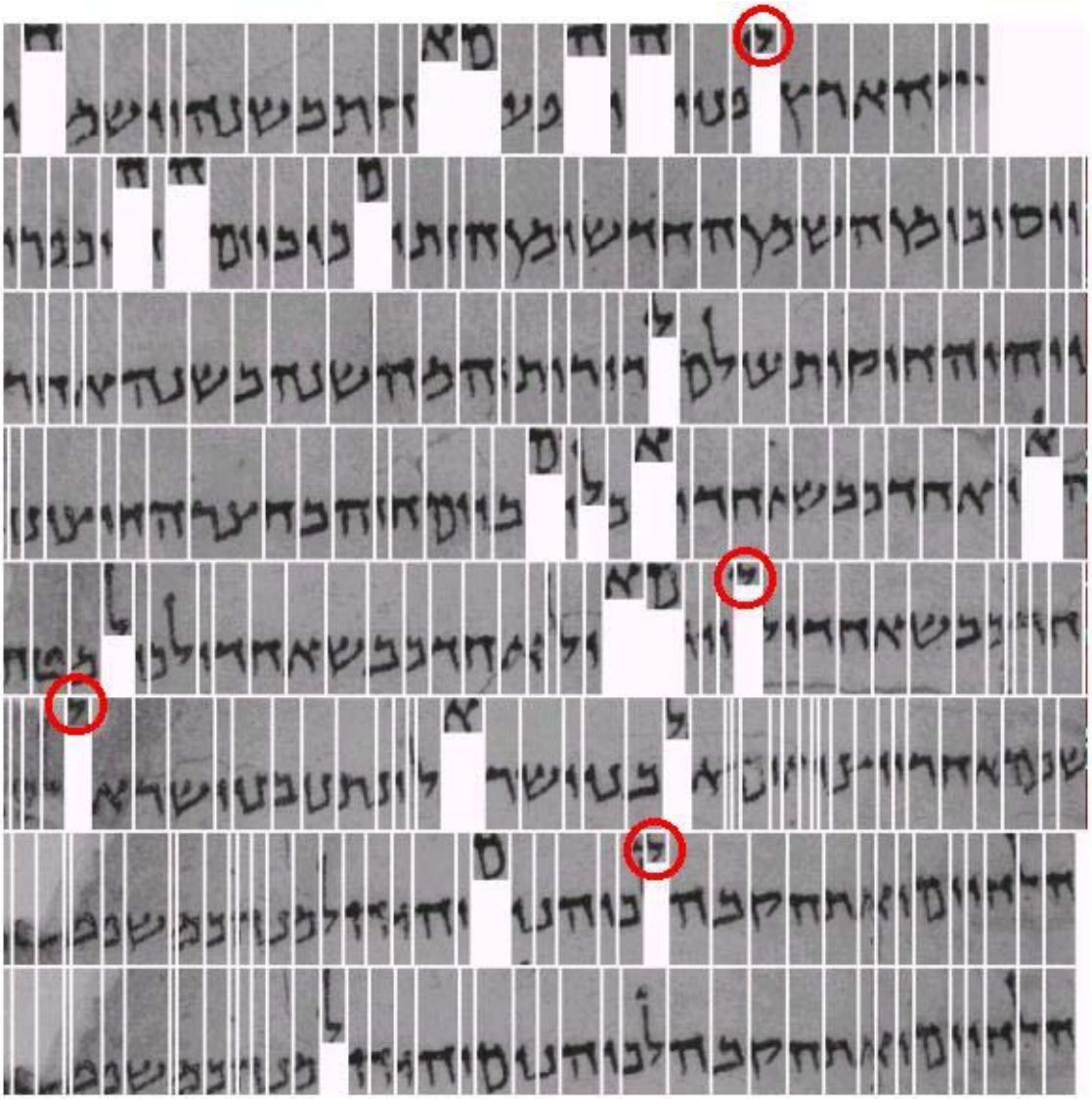

Figure 26 Characters segmented from **Figure 20**. Characters analyzed by the neural network are cropped. Circled characters were incorrectly classified.

There was a problem with testing the network's performance on the degraded images, **Figures 21-23**. The segmentation algorithm in the application was not able to segment the blurred, darkened, or skewed characters. Instead, characters were selected by drawing a box around them by hand in order to input them into the neural network. Twenty selections were made for the blurred image, **Figure 21**. Occasionally more than one selection was made on a single character. These selections were fed into the neural network to be analyzed. The selected characters are shown in **Figure 27**. **Table 2** shows the output of the neural network for each selection.

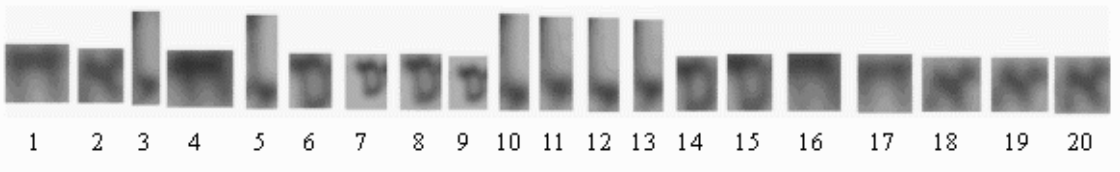

Figure 27 Blurred characters selected from **Figure 21**

Table 2 Neural Network Output for Blurred Characters

Selection number	*Output 1*	*Output 2*	*Output 3*	*Output 4*	*Correct*
1	.01	.99	.00	.02	yes
2	.02	.99	.00	.01	no
3	.82	.00	.09	.00	no
4	.02	.99	.00	.01	yes
5	.39	.00	.41	.00	yes
6	.00	.97	.00	.07	no
7	.00	.00	.02	.07	yes
8	.00	.28	.00	.33	yes
9	.03	.00	.84	.00	no
10	.06	.00	.99	.00	yes
11	.00	.00	.98	.01	yes
12	.07	.00	.99	.00	yes
13	.01	.00	.99	.00	yes
14	.00	.00	.00	.94	yes
15	.00	.11	.00	.89	yes
16	.00	.99	.00	.03	yes
17	.00	.99	.00	.02	yes
18	.00	.99	.00	.02	no
19	.69	.66	.00	.00	yes
20	.02	.99	.00	.00	no

Selection numbers 6, 7, 8, and 9 were all the same character. Also selection numbers 5 and 10 were the same character and selection numbers 3 and 12 were the same character. The selection makes a big difference in the output from the neural network.

Eight selections were made for the darkened image, **Figure 22**. These selections were fed into the neural network to be analyzed. The selected characters are shown in **Figure 28**. **Table 3** shows the output of the neural network for each selection.

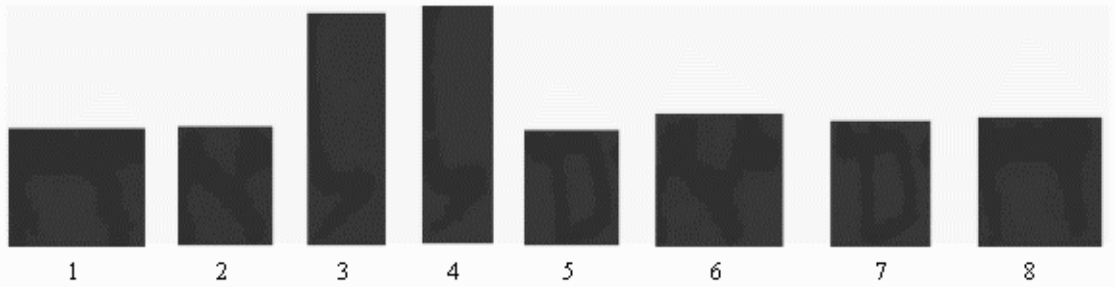

Figure 28 Darkened characters selected from **Figure 22**

Table 3 Neural Network Output for Darkened Characters

Selection number	*Output 1*	*Output 2*	*Output 3*	*Output 4*	*Correct*
1	.01	.99	.00	.01	yes
2	.89	.00	.05	.00	yes
3	.00	.00	.99	.00	yes
4	.03	.00	.99	.00	yes
5	.00	.00	.00	.90	yes
6	.96	.00	.02	.00	yes
7	.00	.00	.03	.99	yes
8	.00	.96	.00	.03	yes

Eight selections were made for the skewed image, **Figure 23**. These selections were fed into the neural network to be analyzed. The selected characters are shown in **Figure 29**. **Table 4** shows the output of the neural network for each selection.

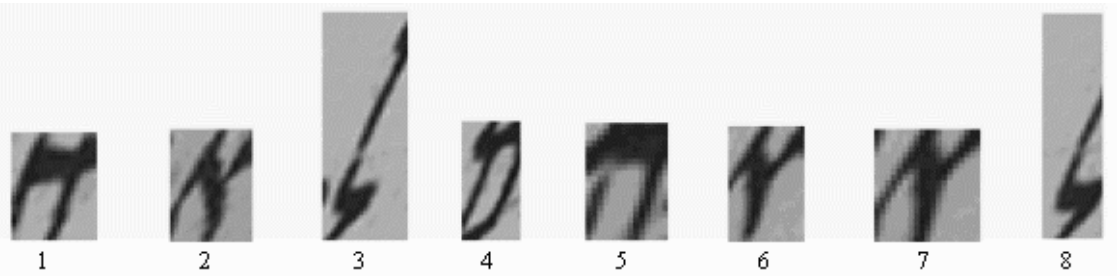

Figure 29 Skewed characters selected from **Figure 23**

Table 4 Neural Network Output for Skewed Characters

Selection number	*Output 1*	*Output 2*	*Output 3*	*Output 4*	*Correct*
1	.93	.00	.00	.00	no
2	.10	.00	.99	.00	no
3	.16	.00	.98	.00	yes
4	.00	.00	.80	.10	no
5	.11	.95	.00	.00	yes
6	.00	.00	.00	.07	no
7	.94	.00	.08	.00	yes
8	.04	.00	.99	.00	yes

Neural Network with Ten Outputs

For the neural network with ten outputs, the configurations tried were 2, 4, 5, 6, 7, and 8 nodes in the hidden layer. **Table 5** shows the results of each configuration.

Table 5 Effectiveness of Six Different Ten Output Network Configurations

Nodes in Hidden Layer	*Training Cycles*	*Final Output Error*	*Incorrectly Classified Training Characters*
2	56,700	0.235	16
4	88,200	0.057	4
5	107,100	0.037	2
6	113,400	0.021	2
7	119,700	0.020	1
8	119,700	0.017	1

In the case of this neural network, it did not seem possible to teach it to correctly classify all sixty-three of the training characters. One character continually stumped it. That character is shown in **Figure 30**. The learning curves for the six configurations tried are shown in **Figure 31**.

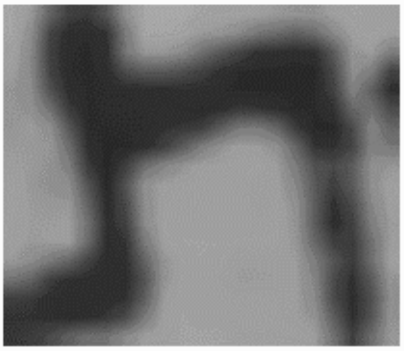

Figure 30 The one training character that the ten output neural network could not be taught

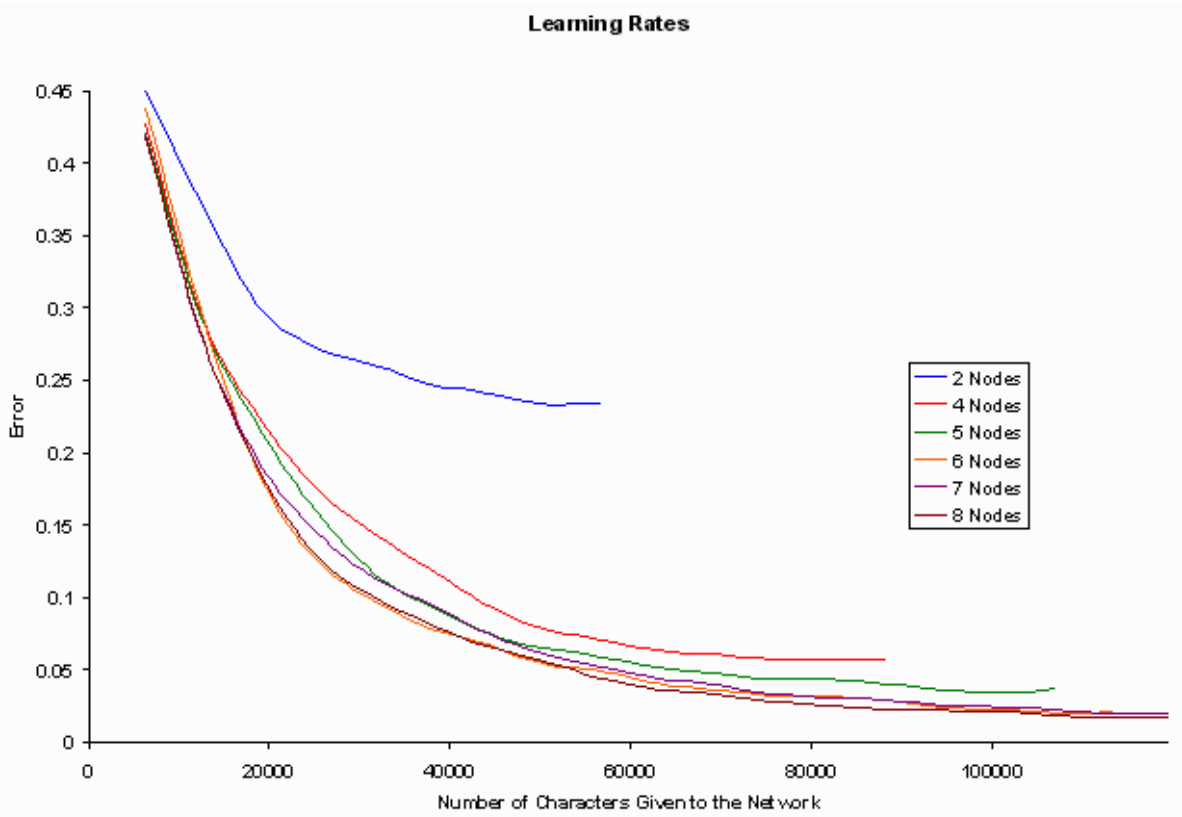

Figure 31 Output error curves for six different configurations of a ten output neural network

The neural network configuration with seven nodes in the hidden layer was used for the rest of the analysis. The cropped characters in **Figure 32** are the characters from **Figure 20** that were tested with this network. There were thirty-six in all, and of those, eleven were incorrectly classified by the neural network. Those eleven are circled in red on **Figure 32**.

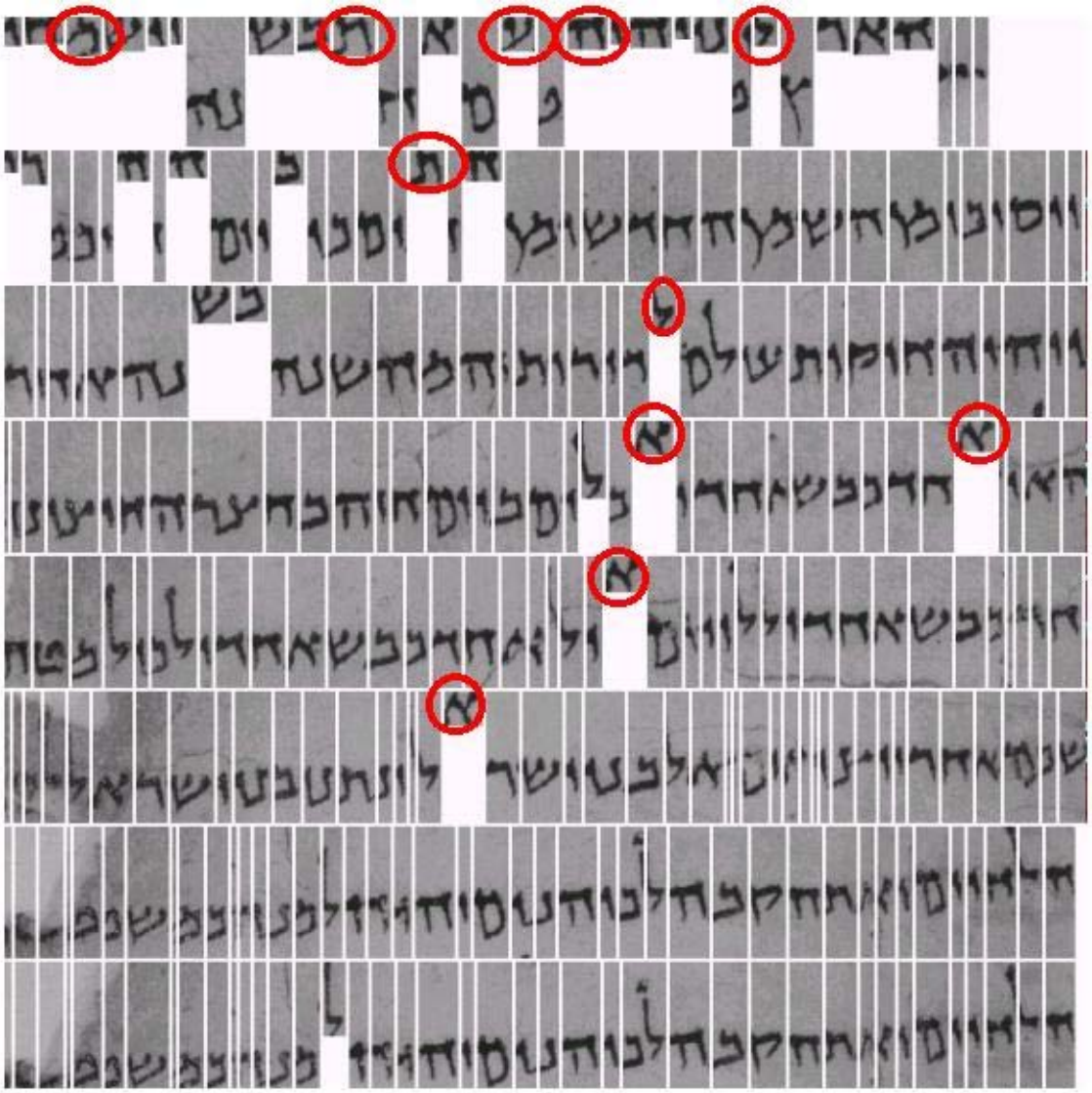

Figure 32 Characters segmented from **Figure 20**. Characters analyzed by the neural network are cropped. Circled characters were incorrectly classified.

Since the ten output neural network did not perform very well when classifying characters which have not been degraded, no analysis will be done on the degraded images with this network.

Discussion

The decision of how many nodes to use in the hidden layer for the four output neural network depends on what is going to be done with the network. As shown in **Table 1**, the network with two nodes in the hidden layer performs at least as well as the other configurations once it has been fully trained. It has the second lowest error of all the networks tried. The problem with it is that it needs more input than some of the other networks in order to reach that level of "intelligence". This means that the training time will be significantly longer for the two-node network than for the four or eight node network. The reason the two-node network was chosen for this project is because once it was trained, it performed slightly faster than the networks with more nodes in the hidden layer. This is because fewer nodes translates to less calculations needing to be performed when analyzing an input. Since all the networks had already been fully trained in order to get the data in **Table 1**, training time was no longer a

consideration, and the two node network was chosen only because it would make getting the rest of the results go more quickly.

The fact that the error on the two-node network was lower than the four and eight node networks was not a factor. The difference between 0.0003, 0.0002, and 0.0004 are not large enough differences to translate into a very large performance difference for the four output network. As can be seen by studying **Table 1** and **Table 5**, the output error very roughly corresponds to the percentage of the training set that the network will get wrong. The four output network with one node in the hidden layer had an output error of 0.194. Since there are sixteen characters in the training set shown in **Figure 19**, it is expected that this network will get 3.1 of them wrong since 3.1 is nineteen percent of sixteen. It, in fact, got four characters wrong, a fairly good approximation. The neural network with ten outputs and two nodes in the hidden layer had an error of about twenty-three and a half percent, and a sixty-three character training set, so it would be expected to get 14.8 characters wrong in this training set. It actually got sixteen characters wrong, again a pretty good approximation. When thinking about the error in these terms, the difference between 0.03%, 0.02%, and 0.04% are negligible.

The performance of the four output layer on characters that had not been very degraded looks very promising. As shown in **Figure 26**, the only input characters that it incorrectly classified were some that had been very poorly segmented. This happened often with the output 3 characters because the tall stem was often slanted and overlapped the neighboring character a bit. Since the very simple segmentation algorithm I wrote assumed that all characters are in their own rectangle, separate from other characters, it often chopped off the stem of the output 3 characters when cropping them to a rectangle. It is understandable that the network could not correctly classify these cropped characters since they really do not look much at all like the output 3 characters that the network was trained with from **Figure 19**. It seems likely that with a better segmentation algorithm this network could have classified any of the four output characters correctly close to one hundred percent of the time.

When classifying degraded characters, segmentation again seemed to be a big problem for this network. The size and positioning of the selection that was made around the character to be analyzed had a large impact on the results. From **Figure 27**, of the blurred characters, selections 6, 7, 8, and 9 are all the same output 4 character. Selection 6 was classified as an output 2 character. Selection 9 was classified as an output 3 character. Selections 7 and 8 were correctly classified as an output 4 character, but the network wasn't very sure of either decision. Neither got higher than 0.5 in output 4, and each one had another output that was only 0.05 less than output 4. **Figure 29**, showing the skewed characters, also had a character in it that was selected twice. Selections 6 and 7 are the same character with only slightly different selections around it. However, this very small, barely noticeable difference had a very large impact on the output of the neural network. The character is an output 1 character, but selection 6 had a 0.00 value in output 1. When the selection was slightly changed, though, this jumped to a 0.94 for the output 1 of selection 7. Again, it is very possible that the network may have benefited from a more robust segmentation algorithm.

The results of the ten output neural network were much less encouraging than those of the four output network. None of the configurations tried could reach the goal for a "good configuration" established in the **Training and Testing of a Neural Network**
section. None of them could classify all sixty-three of the training characters with 100% accuracy. Each configuration always incorrectly classified the character in **Figure 30**. This may also be because of the way that this character was segmented. **Figure 33**
shows this character after being thresholded and thinned by the application. The small part of another character in the top right part of the image makes a significant difference in the appearance of the skeleton for this character, adding an extra intersection and dead end to it.

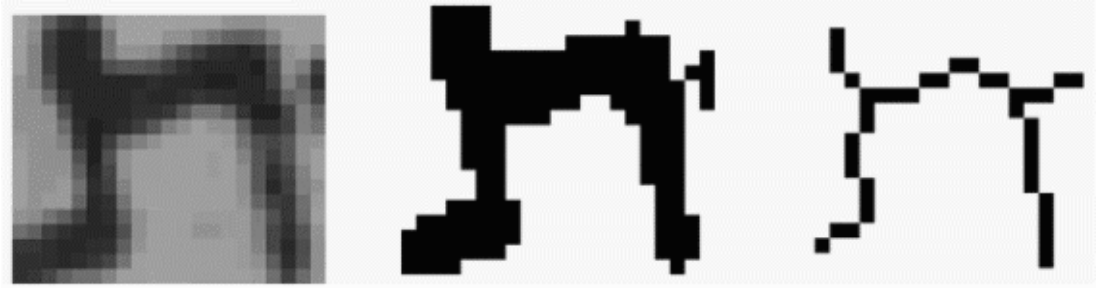

Figure 33 Processing of the character shown in **Figure 30**

This was apparently not the only problem with this character, though. The output 5 character was consistently incorrectly classified in **Figure 32**. When looking at the direct output values from the neural network, it is obvious that it really had problems with this character. In on case, output 3 had the highest value, followed by output 9, and then output 5. This was the only time that output 5 was even one of the top three results, though. Another output 5 character gave the highest value on output 2, then output 4, and then output 10 with output 5 showing up as the fifth highest value. Another one had output 3 as highest, then output 6, then output 10, and again, output 5 was the fifth highest. Another case was even worse with output 6 highest, then output 10, then output 8, and output 5 was the eighth highest value. The output 5 character was not the only character that the network had difficulty with. the only output 4 character in **Figure 32**
was incorrectly classified. Also, several of the output 6 characters were incorrectly classified. It is not surprising that these three characters were giving such problems. Looking at the network with two nodes in the hidden layer from **Table 5**, of the sixteen characters in the training set that this network got wrong, fifteen of them were output 4, 5, and 6 characters. Since there are only five of each in the training set, this means that the two node network could learn almost every other character, but it couldn't recognize a single one of these three characters. Of the four characters that the four node neural network had difficulty with, three were output 5 characters, and one was an output 4 character. It would possibly be helpful to study why these three characters were such a problem and try to determine a way to improve on the current process so that these characters can be more effectively classified.

Conclusions

It was thought that this project would be able to determine if a neural network can be an effective tool for analyzing degraded patterns and returning a measure of how closely the degraded pattern matches another pattern. More specifically, this was applied to recognizing characters on ancient Hebrew documents. Unfortunately, not enough work was completed on this project to know for certain whether or not the neural network could be made to be effective. Neither of the two networks created in this project would be a helpful tool to anyone analyzing degraded ancient Hebrew documents. The four output network was very adept at analyzing the four characters it was designed for as long as those characters weren't degraded. If they were degraded, then the segmentation algorithm ran into difficulties before it was determined whether or not the neural network could be effective in analyzing the degraded characters. Since input features 1-6 from **Design of a Neural Network** each depend on the size of the cropping rectangle around the character, the fact that the degraded characters were not cropped consistently strongly affected the network's output in these tests. It would probably be informative to see how well the network performs with degraded characters if a better, more consistent segmentation algorithm is employed.

The ten output neural network did not even perform satisfactorily on characters that weren't degraded. Three of the ten output characters were consistently a problem. It is possible that the input features being used were not descriptive enough to distinguish these characters from each other or from other characters. The number of turns

in the skeleton of characters was not used as an input feature due to time constraints. It is possible that this may add enough information to make classification of these characters easier for the neural network. There is no reason to think it will help a lot, though. What is really needed is a better way to describe, with one or two numbers, the overall shape/appearance of the characters. The number of curves in the skeleton will help, and that coupled with the number of intersections and the number of dead ends can give a pretty good description of the appearance of the character, but it doesn't give any placement for each of these features. Perhaps a count of the number of dead ends, intersections and curves appearing in each quarter of the character would be more helpful. For example, output 4 for the ten output network has one dead end in the top left quarter, one in the top right quarter, and one in the bottom left quarter. It also has one intersection in the bottom left quarter. Another possibility would be an addition of moment calculations on the characters as described in [9] and [10]. This could be an effective way of describing overall character properties in numbers. With the addition of some of these input features, it may be possible to remove some of the current input features that may no longer be necessary. A study would need to be done to determine how significant each of the current inputs is.

Looking at the function being used inside the nodes in this network shown in **Figure 3**, it is plain to see that there is a very sharp threshold between a one and a zero output. This may have something to do with how sensitive the network is as a whole. If an input to a node changes just a little bit, then the slope of this function can cause a large change in the output of the node. It might be interesting to try a network with a lower gain on the internal node function. This type of network may be less sensitive to deviations from the training set than the current model.

Based only on the results of the studies on these pages, it is necessary to conclude that a neural network would not be an effective tool for study of degraded characters. While the neural network is very good at matching patterns that fairly closely resemble each other, it is very sensitive to small deviations in the pattern, such as stray lines or marks, breaks in the character, blurring, rotation, etc. How sensitive a neural network is to each of these factors depends entirely on what features from the character are input into the network. It would certainly be possible to create a neural network which is less sensitive to rotation by inputting features of the character that do not depend on the orientation of the character. This network, however, would very possibly be more susceptible to errors due to stray marks or blurring since information that does not depend on orientation would necessarily be the shape of the character, and the shape can be changed by marks or by blurring. A network that was designed to be resistant to blurring would probably employ density of pixels in regions in order to decipher a character. This network, however, would probably be very sensitive to changes in the orientation of the character. It does not seem likely that a neural network can be found that will give meaningful output for characters affected by various types of degredation.

Table of Contents

References

1. de Bruyne, P., R. Korolnik, "Segmentation and Recognition of Calligraphic Text," *Second International Conference on Artificial Neural Networks*, No. 349, pp. 214-218, November 1991.
2. Nellis, J., T. J. Stonham, "A Fully Integrated Hand-Printed Character Recognition System Using Artificial Neural Networks," *Second International Conference on Artificial Neural Networks*, No. 349, pp. 219-223, November 1991.
3. Elliman, D. G., R. N. Banks, "A Comparison of Two Neural Networks for Hand-Printed Character Recognition," *Second International Conference on Artificial Neural Networks*, No. 349, pp. 224-228, November 1991.
4. Lee, Seong-Whan, Eung-Jae Lee, "Integrated Segmentation and Recognition of Connected Handwritten Characters with Recurrent Neural Network," Document Recognition III, *Proceedings SPIE - The International Society for Optical Engineering*, Vol. 2660, pp. 251-261, January 1996.
5. Chester, Michael, *Neural Networks: A Tutorial*, Prentice-Hall, New Jersey, 1993.
6. Haykin, Simon, *Neural Networks: A Comprehensive Foundation*, Macmillan College Publishing Company, Inc., New York, 1994.
7. Flores, Edna, Eder N. Rezende, Gilberto A. Carrijo, Joao B. T. Yabu-tti, "A Fast Thinning Algorithm for Characters," 1995 *IEEE Workshop On Nonlinear Signal And Image Processing*, Web page: http://poseidon.csd.auth.gr/Workshop/papers/, June 1995.
8. Kutza, Karsten, "Backpropagation Network with Bias Terms and Momentum," Web page: http://www.geocities.com/CapeCanaveral/1624/bpn.html, April 1996.
9. Hanson, Adam, *Character Recognition of Optically Blurred Textual Images Using Moment Invariants*, Rochester Institute of Technology Center for Imaging Science,1993.
10. Sanders, Lee C., *Recognition of Optically Blurred Characters Using Invariant Moments*, Rochester Institute of Technology Center for Imaging Science,1994.

Creation of a Neural Network to Assist in Deciphering Degraded Ancient Hebrew Texts

Daniel B Hentschel

Appendix A: Thinning Algorithm

The ImagePixel class is listed in Appendix B.

```
Public Sub Thin(Optional Pic As Object, Optional displayScale As Integer)
    Dim current As ImagePixel
    Dim test As ImagePixel
    Dim marked As New Collection

    Dim cnwidth As Integer
    Dim cnheight As Integer
    Dim deleted As Long


    Do



        deleted = 0

        ' Mark edge pixels

        For cnheight = 0 To height - 1

            For cnwidth = 0 To bmi.bmiHeader.biWidth - 1

                If tempdata(cnwidth, cnheight) = 0 Then

                    Set current = New ImagePixel

                    current.InitPos cnwidth, cnheight

                    current.CountNeighbors tempdata

                    If current.VerifyDiversion Then

                        If Not current.FunctionVerifyDiversion(tempdata) Then

                            If current.isContour Or current.neighbors = 0 Then

                                marked.Add current, (cnwidth & "," & cnheight)

                            End If

                        End If

                    Else

                        If current.isContour Or current.neighbors = 0 Then

                            marked.Add current, (cnwidth & "," & cnheight)
```

```
                        End If

                    End If

                End If

            Next

        Next


        ' Check edge pixels to see if they should be deleted

        For Each test In marked

            test.CountNeighbors tempdata

            If (Not test.Figure2) Then
                If test.neighbors = 1 Then
                    If test.CheckNeighbors(tempdata) Then
                        tempdata(test.x, test.y) = 255
                        deleted = deleted + 1
                    End If
                Else

                    If test.neighbors >= 2 Then

                            If Not test.CheckConnectivity(tempdata) Then

                                tempdata(test.x, test.y) = 255
                                deleted = deleted + 1

                            End If

                    End If

                    If test.neighbors = 0 Then

                        tempdata(test.x, test.y) = 255
                        deleted = deleted + 1

                    End If
                End If

            End If

            marked.Remove (test.x & "," & test.y)

        Next

        If Not (Pic Is Nothing) Then DisplayTemp Pic, 0, 0, CSng(displayScale)


    Loop While deleted > 0
End Sub
```

Appendix B: ImagePixel Class

```
Option Explicit

Private xx As Integer
```

```
Private yy As Integer
Private code As Byte
Private nPixels As Collection
Private nCount As Byte
Private direction As Byte

Public Sub InitPos(x As Integer, y As Integer)
    xx = x
    yy = y

    code = 0
    nCount = 0
    Set nPixels = New Collection

    ' Put a meaningless value in direction

    direction = 8
End Sub

Public Sub SetDirection(bDirection As Byte)
    direction = bDirection
End Sub

Public Function GetDirection() As Byte
    GetDirection = direction
End Function

Public Sub CountNeighbors(data() As Byte)
    Dim errorY As Boolean
    Dim errorX As Boolean
    Dim errorXX As Boolean
    Dim nPixel As ImagePixel

    nCount = 0

    code = 0
    Set nPixels = Nothing
    Set nPixels = New Collection


    On Error GoTo ErrorHandler
    If data(xx + 1, yy) = 0 Then
        code = code Or &H1
        Set nPixel = New ImagePixel
        nPixel.InitPos xx + 1, yy
        nPixel.SetDirection 0
        nPixels.Add nPixel
        nCount = nCount + 1
    End If
    If data(xx + 1, yy - 1) = 0 Then
        code = code Or &H2
        Set nPixel = New ImagePixel
        nPixel.InitPos xx + 1, yy - 1
        nPixel.SetDirection 1
        nPixels.Add nPixel
        nCount = nCount + 1
    End If

EndXPlus:

    If data(xx, yy - 1) = 0 Then
        code = code Or &H4
        Set nPixel = New ImagePixel
        nPixel.InitPos xx, yy - 1
        nPixel.SetDirection 2
        nPixels.Add nPixel
        nCount = nCount + 1
    End If

    If data(xx - 1, yy - 1) = 0 Then
```

```
        code = code Or &H8
        Set nPixel = New ImagePixel
        nPixel.InitPos xx - 1, yy - 1
        nPixel.SetDirection 3
        nPixels.Add nPixel
        nCount = nCount + 1
    End If
EndYMinus:
    If data(xx - 1, yy) = 0 Then
        code = code Or &H10
        Set nPixel = New ImagePixel
        nPixel.InitPos xx - 1, yy
        nPixel.SetDirection 4
        nPixels.Add nPixel
        nCount = nCount + 1
    End If

    If data(xx - 1, yy + 1) = 0 Then
        code = code Or &H20
        Set nPixel = New ImagePixel
        nPixel.InitPos xx - 1, yy + 1
        nPixel.SetDirection 5
        nPixels.Add nPixel
        nCount = nCount + 1
    End If

EndXMinus:
    If data(xx, yy + 1) = 0 Then
        code = code Or &H40
        Set nPixel = New ImagePixel
        nPixel.InitPos xx, yy + 1
        nPixel.SetDirection 6
        nPixels.Add nPixel
        nCount = nCount + 1
    End If

    If data(xx + 1, yy + 1) = 0 Then
        code = code Or &H80
        Set nPixel = New ImagePixel
        nPixel.InitPos xx + 1, yy + 1
        nPixel.SetDirection 7
        nPixels.Add nPixel
        nCount = nCount + 1
    End If


    Exit Sub

ErrorHandler:

    If xx + 1 > UBound(data, 1) And Not errorXX Then
        errorXX = True
        Resume EndXPlus
    End If

    If yy - 1 < 0 And Not errorY Then
        errorY = True
        Resume EndYMinus
    End If

    If xx - 1 < 0 And Not errorX Then
        errorX = True
        Resume EndXMinus
    End If

End Sub


Public Property Get neighbors() As Byte
    neighbors = nCount
```

```
End Property

Public Property Get x() As Integer
    x = xx
End Property

Public Property Get y() As Integer
    y = yy

End Property

' returns the number of neighbors with 2 or fewer neighbors to it
Public Property Get CheckNeighbors(data() As Byte) As Boolean
    Dim nPixel As ImagePixel

    For Each nPixel In nPixels

        nPixel.CountNeighbors data
        If nPixel.neighbors > 2 Then

                CheckNeighbors = True
                Exit Property

        End If

    Next
End Property


Public Property Get isContour() As Boolean
     If (code And &H44) = &H40 Or _
        (code And &H11) = &H10 Or _
        (code And &H44) = &H4 Or _
        (code And &H11) = &H1 Then isContour = True
End Property




Public Property Get Figure2() As Boolean
    If (((code And &H11) = &H11 Or _
         (code And &H44) = &H44 Or _
         (code And &H42) = &H42 Or _
         (code And &H48) = &H48 Or _
         (code And &H24) = &H24 Or _
         (code And &H84) = &H84 Or _
         (code And &H22) = &H22 Or _
         (code And &H88) = &H88 Or _
         (code And &H9) = &H9 Or _
         (code And &H12) = &H12 Or _
         (code And &H21) = &H21 Or _
         (code And &H90) = &H90) And _
        ((code And &HEE) = code Or _
         (code And &HBB) = code)) _
        Then Figure2 = True

End Property


Public Property Get VerifyDiversion() As Boolean
    If (code And 2) / 2 + (code And 8) / 8 + (code And 32) / 32 + (code And 128) _
        / 128 = 2 And nCount = 2 Then VerifyDiversion = True
End Property

Public Function FunctionVerifyDiversion(data() As Byte) As Boolean

    If (code And 2) = 2 And (code And 128) = 128 Then
        data(xx + 1, yy) = 0
        data(xx, yy) = 255
    End If
```

```
    If (code And 8) = 8 And (code And 32) = 32 Then
        data(xx - 1, yy) = 0
        data(xx, yy) = 255
    End If
    If (code And 2) = 2 And (code And 8) = 8 Then
        data(xx, yy - 1) = 0
        data(xx, yy) = 255
    End If

    If (code And 32) = 32 And (code And 128) = 128 Then
        data(xx, yy + 1) = 0
        data(xx, yy) = 255
    End If
    If data(xx, yy) = 255 Then FunctionVerifyDiversion = True
End Function

Public Property Get CheckConnectivity(data() As Byte) As Boolean
    Dim nPixel As ImagePixel

    For Each nPixel In nPixels

        nPixel.CountNeighbors data

        If nPixel.NeighborSkeletonConnectivity Then
            CheckConnectivity = True
            Exit Property
        End If

    Next
End Property

Public Property Get NeighborSkeletonConnectivity() As Boolean
    If direction = 5 Then
        If (code And 1) = 0 And (code And 4) = 0 Then
            NeighborSkeletonConnectivity = True
            Exit Property
        End If
    End If


    If direction = 7 Then
        If (code And 4) = 0 And (code And 16) = 0 Then
            NeighborSkeletonConnectivity = True
            Exit Property
        End If
    End If
    If direction = 1 Then
        If (code And 16) = 0 And (code And 64) = 0 Then
            NeighborSkeletonConnectivity = True
            Exit Property
        End If
    End If
    If direction = 3 Then
        If (code And 1) = 0 And (code And 64) = 0 Then
            NeighborSkeletonConnectivity = True
            Exit Property
        End If
    End If

End Property




Public Property Get IsNeighbor(otherPixel As ImagePixel) As Boolean
    Dim nPixel As ImagePixel
    For Each nPixel In nPixels
```

```
        If nPixel.x = otherPixel.x And nPixel.y = otherPixel.y Then
            IsNeighbor = True
            Exit Property
        End If

    Next
End Property
```

Appendix C: BPNetwork Class

```
Option Explicit

Private Type LAYER
    nodes As Integer
    NodeOutput() As Double
    NodeError() As Double
    NodeWeight() As Double
    NodeWeightSave() As Double
    NodeDeltaWeight() As Double

End Type

Private TotalLayers As Integer
Private Layers() As LAYER
Private inputMax() As Double
Private inputMin() As Double
Private alpha As Double
Private eta As Double
Private gain As Double
Private error As Double
Private Const LO = 0.2
Private Const HI = 0.8
Private Const BIAS = 1




Public Sub SaveNet(filename As String)
    Open filename For Binary Access Write As #1

    Put #1, , TotalLayers
    Put #1, , alpha
    Put #1, , eta
    Put #1, , gain
    Put #1, , Layers
    Put #1, , inputMin
    Put #1, , inputMax

    Close #1
End Sub


Public Sub OpenNet(filename As String)
    Open filename For Binary Access Read As #1

    Get #1, , TotalLayers
    Get #1, , alpha
    Get #1, , eta
    Get #1, , gain
    ReDim Layers(0 To TotalLayers - 1)
    Get #1, , Layers
    ReDim inputMin(0 To Layers(0).nodes)
    ReDim inputMax(0 To Layers(0).nodes)
    Get #1, , inputMin
    Get #1, , inputMax
```

```
    Close #1

End Sub



Public Function GetError() As Double
    GetError = error
End Function

' Random number generators
Private Function GetiRAND(Low As Integer, High As Integer) As Integer
    Randomize
    GetiRAND = Rnd() Mod (High - Low + 1) + Low

End Function

Private Function GetdRAND(Low As Double, High As Double) As Double
    Randomize

    GetdRAND = Rnd * (High - Low) + Low
End Function







' Inputs should be normalized between HI and LO
Public Property Get High() As Double
    High = HI
End Property
Public Property Get Low() As Double
    Low = LO
End Property




' Set up the network: initialize values and allocate space
Public Sub InitNetwork(ByVal dAlpha As Double, ByVal dEta As Double, ByVal dGain As Double, _
                       ByVal nLayers As Integer, ParamArray nNodes())

    Dim l As Integer

    TotalLayers = nLayers

    ReDim Layers(0 To TotalLayers - 1)

    For l = 0 To TotalLayers - 1


        Layers(l).nodes = nNodes(l)

        ReDim Layers(l).NodeOutput(0 To nNodes(l))
        ReDim Layers(l).NodeError(0 To nNodes(l))

        ' The 0th output element of each node is its bias.

        Layers(l).NodeOutput(0) = BIAS

        If l <> 0 Then
            ReDim Layers(l).NodeWeight(0 To nNodes(l), 0 To nNodes(l - 1))
            ReDim Layers(l).NodeWeightSave(0 To nNodes(l), 0 To nNodes(l - 1))
            ReDim Layers(l).NodeDeltaWeight(0 To nNodes(l), 0 To nNodes(l - 1))
        End If

    Next
```

```
    ReDim inputMin(0 To nNodes(0))
    ReDim inputMax(0 To nNodes(0))

    alpha = dAlpha
    eta = dEta
    gain = dGain


    ' Set initial weights to random values

    RandomizeWeights

End Sub

Private Sub RandomizeWeights()
    Dim l As Integer
    Dim i As Integer
    Dim j As Integer

    For l = 1 To TotalLayers - 1

        For i = 1 To Layers(l).nodes

            For j = 0 To Layers(l - 1).nodes

                Layers(l).NodeWeight(i, j) = GetdRAND(-0.5, 0.5)

            Next

        Next

    Next
End Sub


Public Sub SetInputLimits(nNode As Integer, max As Double, min As Double)
    inputMin(nNode) = min
    inputMax(nNode) = max
End Sub

' Set the input values and calculate the output values by propagating through the net.
Public Sub SetInput(ParamArray inputVals())
    Dim cn As Integer


    For cn = 1 To Layers(0).nodes

        Layers(0).NodeOutput(cn) = ((inputVals(cn - 1) - inputMin(cn - 1)) / _
            (inputMax(cn - 1) - inputMin(cn - 1))) * (HI - LO) + LO
    Next
    PropagateNet
End Sub



' Get the output values
Public Sub GetOutput(outputVals() As Double)
    Dim cn As Integer

    ReDim outputVals(0 To Layers(TotalLayers - 1).nodes - 1)
    For cn = 1 To Layers(TotalLayers - 1).nodes
        outputVals(cn - 1) = Layers(TotalLayers - 1).NodeOutput(cn)

    Next
End Sub


' Calculate the value of each node on the upper layer based on the
' output of the lower layer
Private Sub PropagateLayer(Lower As LAYER, Upper As LAYER)
```

```
    Dim i As Integer
    Dim j As Integer
    Dim Sum As Double

    On Error GoTo OverflowHandler

    For i = 1 To Upper.nodes

        Sum = 0

        For j = 0 To Lower.nodes

            Sum = Sum + Upper.NodeWeight(i, j) * Lower.NodeOutput(j)

        Next

        Upper.NodeOutput(i) = CDbl(1# / (1# + CDbl(Exp(-gain * Sum))))
    Next

    Exit Sub

    ' An overflow occurs if the exponential in the above
    ' calculation becomes too small or too high.

    ' If it's too small, then this error handler sets the output to
    ' an extremely small number. Otherwise it sets it to an
    ' extremely high number.
OverflowHandler:
    If -gain * Sum > 700 Then
        Upper.NodeOutput(i) = 1E-300
    Else
        Upper.NodeOutput(i) = 1E+300
    End If

    Resume Next
End Sub

' Calculate each layer
Private Sub PropagateNet()
    Dim l As Integer

    For l = 0 To TotalLayers - 2

        PropagateLayer Layers(l), Layers(l + 1)

    Next
End Sub


' Based on the target values, how far away was the result?
Private Sub ComputeOutputError(Target() As Double)
    Dim i As Integer
    Dim Out As Double
    Dim Err As Double

    error = 0

    For i = 1 To Layers(TotalLayers - 1).nodes

        Out = Layers(TotalLayers - 1).NodeOutput(i)
        Err = Target(i) - Out
        Layers(TotalLayers - 1).NodeError(i) = gain * Out * (1# - Out) * Err
        error = error + 0.5 * Err ^ 2

    Next

End Sub


' Calculate the lower node error based on the error on the upper nodes
```

```
' and the weighting between the upper and lower node
Private Sub BackpropagateLayer(Upper As LAYER, Lower As LAYER)
    Dim i As Integer
    Dim j As Integer
    Dim Out As Double
    Dim Err As Double

    For i = 1 To Lower.nodes

        Out = Lower.NodeOutput(i)
        Err = 0

        For j = 1 To Upper.nodes
            Err = Err + Upper.NodeWeight(j, i) * Upper.NodeError(j)
        Next

        Lower.NodeError(i) = gain * Out * (1# - Out) * Err

    Next
End Sub

' Calculate the errors for each layer
Private Sub BackpropagateNet()
    Dim l As Integer

    For l = TotalLayers - 1 To 2 Step -1

        BackpropagateLayer Layers(l), Layers(l - 1)

    Next

End Sub




' Adjust the weights (and the bias) based on the calculated errors.
Private Sub AdjustWeights()
    Dim l As Integer
    Dim i As Integer
    Dim j As Integer
    Dim Out As Double
    Dim Err As Double
    Dim dWeight As Double

    For l = 1 To TotalLayers - 1

        For i = 1 To Layers(l).nodes

            For j = 0 To Layers(l - 1).nodes

                Out = Layers(l - 1).NodeOutput(j)
                Err = Layers(l).NodeError(i)
                dWeight = Layers(l).NodeDeltaWeight(i, j)
                Layers(l).NodeWeight(i, j) = Layers(l).NodeWeight(i, j) + eta * Err * _
                                             Out + alpha * dWeight
                Layers(l).NodeDeltaWeight(i, j) = eta * Err * Out

            Next

        Next

    Next

End Sub

' This subroutine is the publicly accessable routine that calls all
' the other routines needed to train the network
Public Sub TrainNetValues(ParamArray targetVals())
    Dim Target() As Double
```

```
    Dim cn As Integer

    ReDim Target(LBound(targetVals) To UBound(targetVals))

    For cn = LBound(targetVals) To UBound(targetVals)

        Target(cn) = targetVals(cn)

    Next
    ComputeOutputError Target
    BackpropagateNet
    AdjustWeights

End Sub


' This subroutine is the publicly accessable routine that calls all
' the other routines needed to train the network
Public Sub TrainNetArray(targetVals() As Double)
    Dim Target() As Double
    Dim cn As Integer

    ReDim Target(1 To Layers(TotalLayers - 1).nodes)

    For cn = 1 To Layers(TotalLayers - 1).nodes

        Target(cn) = targetVals(cn)

    Next
    ComputeOutputError Target
    BackpropagateNet
    AdjustWeights

End Sub
```

Table of Contents | Thesis