Application of Neural Networks for Digit Recognition

Bo Bleckel[1], Jasper Houston[1], Dylan Parsons[1]
[1]*Bowdoin College, Brunswick, Maine, USA*
{*lbleckel, jhouston, dparsons*}*@bowdoin.edu*

Keywords: NN, neural nets, neural networks, digit recognition, perceptron

Abstract: A Neural Network (NN) is an artificial system built to mimic the connectivity and functionality of the human brain. In this paper, we explore the application of Neural Networks in solving digit recognition problems. We create a perceptron, a form of classifier that categorizes an input based on a set of learned rules. In doing so, we will explore the difficulties associated with input and output representation: how can digits best be represented in order to produce an effective perceptron? We find that the most effective combination was a 32x32 input map, and 10 output nodes. While the nets constructed with one output node ran significantly faster, they vastly underperformed those with 10 output nodes.

1 INTRODUCTION

In a time of increasing need for computers to function like humans, the problem of reproducing cognitive processes is an important task. From classification to supervised learning, there is no limit to the ambitions and abilities of machine learning. Amongst these tasks, recognition is a powerful tool for machines that allows them to interpret data inputs as a part of a larger pattern. Speech, facial, and handwriting recognition are three examples that have a range of applications across different disciplines. In this paper, we explore a subset of handwriting recognition: digit recognition.

Using a form of Neural Network called a perceptron, we create a system capable of recognizing a digit based on a certain kind of input. A perceptron is a classifier that functions with supervised learning, a form of network training. The perceptron determines which, if any, category of digit (integers from 0 to 9) the input belongs in. Since the perceptron's performance is based largely on the input, the matter of representation becomes a serious issue. We deal with several issues regarding representation, including representing the input and output.

In Section 2, we describe the problem of digit recognition in more depth. In Section 3, we explain the functionality and details of Neural Networks, followed by a close description of our experimental methods in Section 4. Section 5 outlines our results, Section 6 offers further thoughts and considerations, and Section 7 concludes our report.

2 DIGIT RECOGNITION

Our project will focus on training our neural network to recognize handwritten digits. Digit recognition is generally considered a relatively easy problem that is straightforward to solve using neural networks. The dataset that we use is called the MNIST dataset, and consists of 60,000 training images, and 10,000 testing images. These images represent the handwritten digits as a matrix of values between 0 and 1. Figure 1 shows a visual representation of the matrix, which resembles a handwritten 5. Each of these

Figure 1: Sample image from the MNIST dataset.

images has an associated number that represents the value that of the digit in the image. Using both the image data and the associated number, we are able to train a neural network to learn to recognize digits. The overall goal is to allow the network to associate some representation of an image of a digit with a value. Generally this is done by setting up the neural network with an output vector size of 10. In Section 3 we talk more in depth on this topic, but using an output vector of size 10 allows us to infer with some level of certainty what digit it is, or at least, what digit the neural network thinks it is. For example, if the output vector looked like this:

0.33
0.27
0.97
0.18
0.02
0.57
0.11
0.76
0.49
0.001

we could infer that the most likely value for this image would be 2. This is because that is the number with the highest associated classification value from the neural network. The next most likely value for the image would be 8.

The MNIST dataset is generally used as the standard for learning how to create a neural network and for running small tests. The fact that it has so much training data, as well as an impressive amount of testing data, means that it is perfect for learning how to work with neural networks. On top of that, each image is the same size and the digits are centered in the window. In fact Google's Tensorflow project offers the MNIST dataset as the first exercise in learning how to use their neural network library.

3 NEURAL NETWORKS

Neural Networks (NN) are a model loosely based on the function of the brain. The main use for Neural Networks is in function approximation through supervised learning. These networks are expressed as a network of nodes with weights on each edge. Each node represents an artificial neuron which is connected with other neurons just as they are in the brain. Each node functions as seen in Figure 2 where

- a_j is the activation level coming from node j
- $W_{j,i}$ is the weight on the link from node j to node i
- in_i is the weighted sum of inputs to node i, $in_i = \sum_j W_{j,i} a_j$
- g is the activation function
- a_i is the activation level of node i

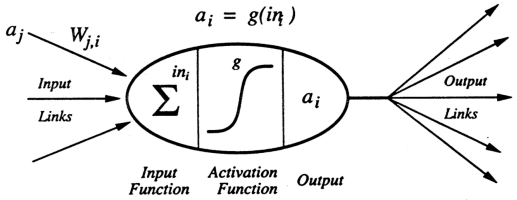

Figure 2: A Node in a Neural Network.

3.1 Activation Functions

There are three activation functions that are most commonly used. They are the step function, the sign function, and the sigmoid function.

3.1.1 Step Function

The step function is as seen in Figure 3.

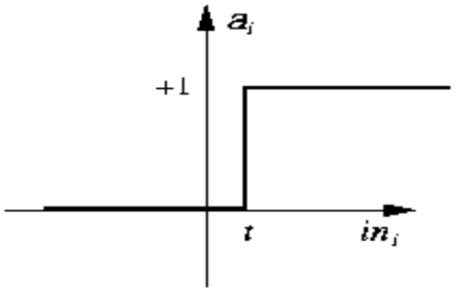

Figure 3: The Step Function.

Where $Step_t(x) = 1 \; if \; x \geq t, \; else \; 0 \quad threshold = t$

3.1.2 Sign Function

The sign function is as seen in Figure 4.
Where $Sign(x) = +1 \; if \; x \geq 0, \; else \; -1$

3.1.3 Sigmoid Function

The sigmoid function is as seen in Figure 5.
Where $Sigmoid(x) = \frac{1}{1+e^{-x}}$

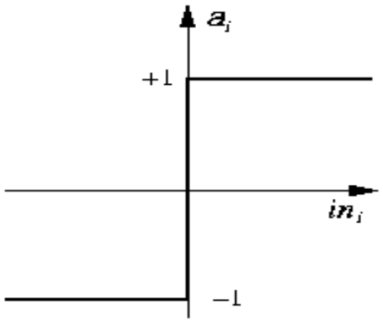

Figure 4: The Sign Function.

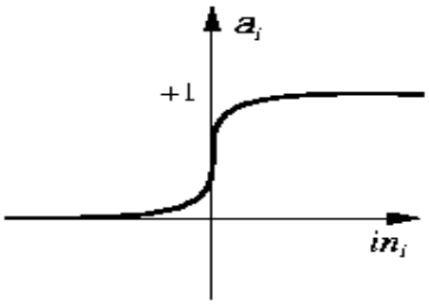

Figure 5: The Sigmoid Function.

3.2 Perceptrons

In our work, we will be focusing on perceptrons, also known as single-layer, feed-forward networks. A perceptron network is shown in Figure 6.

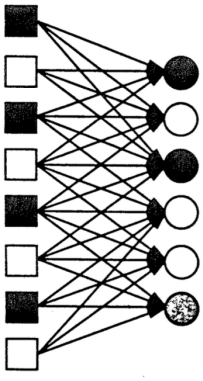

Figure 6: Perceptron Network.

In the perceptron networks we are looking at, there will be a given number of input nodes and a given number of output nodes as defined by the user. In the case of Figure 6, there are 8 input nodes and 6 output nodes. Typically, as discussed in Section 2, we will be using 10 output nodes in our networks. In out networks, all input nodes will be connected to all output nodes. We can think of this more closely by looking at a single perceptron such as in Figure 7. This simplifies things because each edge weight only affects a single output.

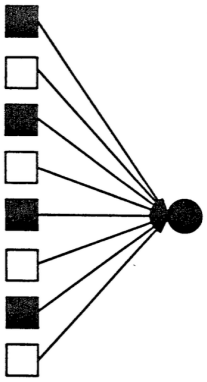

Figure 7: Single Perceptron.

3.3 Perceptron Learning

```
BEGIN
  given a set of training examples:
  for each epoch (iteration) of training:
    for each training example:
      present the input
      calculate error in output
      adjust the weights
END
```

Figure 8: Pseudocode for Perceptron Learning.

The weights are adjusted as follows:

$$W_j = W_j + (\alpha * I_j * Err * g'(in))$$

where

- α is the learning rate
- I_j is the input value at input node j
- $Err = T - O$ where T is the target value and O is the output
- $in = \sum_j W_j I_j$

4 EXPERIMENTAL METHODOLOGY

As mentioned in Section 2, we trained and tested our neural network on the MNIST hand-written digit dataset. We trained the network on 3823 images and tested it on 1797 images. We tested multiple output and input representations, experimenting with the intent of finding the optimal input and output representations. We also ran our experiments with the goal of finding the optimal value for the learning rate α given each possible input and output representation. We evaluated the effectiveness of the network based on the percentage of images correctly classified. For each experiment we recorded the percentage correctly classified at each epoch of training, as well as the percentage correctly classified on the test set after training was complete. The results from these experiments can be found in Section 5. We trained each network with 50 epochs on the entire training image set.

4.1 Network Implementation

We implemented the neural network using a C++ class for both the input and output nodes, as well as a separate class for the network itself. The network class held a vector for the input nodes, as well as a vector for the output nodes. It also held a two-dimensional vector for the weights of each perceptron. The weights of the network govern the learning of the entire network. We implemented the sigmoid activation function, as described in Section 3.1.3, however, we used a shifted version:

$$g(x) = \frac{1.0}{1 + e^{0.5-x}}.$$

Each input node was fully connected to each output node, thus creating a fully-connected neural network. The number of input and output nodes depended on the input and output representation, which is described in Section 4.2. We initialized the weights to values between -0.15 and 0.15. We found that this worked much better than the suggested values of -1 and 1. We also added a bias node to the input layer of our network which is anode that has a constant activation level of 1. This node has a connection and weight to every output node. Those weights are trained just the same as they are for any other node.

4.2 Input and Output Representation

We read in the training and test images in two different forms. One form was the original 32×32 image, consisting of 1s and 0s. Figure 9 shows the representation of one of the images. We called this the 32×32 representation.

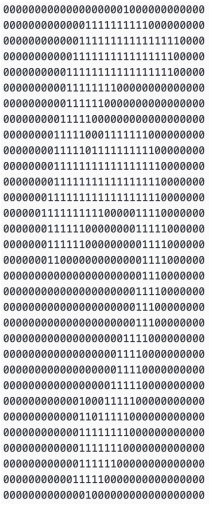

Figure 9: Bit representation of a handwritten digit (the digit 5 in this case) from the MNIST dataset.

This image could also be represented by breaking it up into 64 non overlapping 4×4 blocks that are given by the sum of the bits in that region. This gives an 8×8 representation that does not lend well to a visual representation, but provides a more compact form to represent each input. We ran our experiments with the goal of comparing these two input representations.

We also designed our network to support two different output types. The first being ten output nodes, each corresponding to a digit. The node with the highest activation signifies what the network thinks the digit is. The second option, is a single output node whose output value is contained in the activation value, multiplied by 10. We tested both of these output representations, along with the two input representations mentioned above in our experiments.

4.3 Learning Rate

In our experiments, we tested four values for the learning rate α. For each of the four output-input representations, we tested these four values for α, thus giving 16 total experiments. The results from these experiments are described in Section 5.

5 RESULTS

Our goal while investigating Neural Networks was to explore different input representations, different

output representations, and how varying the learning rate affects the accuracy of the network. It is worth noting that for the graphs referenced in this section, we are looking at the percent correct in the learning epochs rather than the testing results. The table in Figure 17 shows the results of the testing after the training is completed.

5.1 Input Representations

The two input representations we explored were a 32x32 input and an 8x8 input. Due to the loss of information in the 8x8 representation, we expected the 32x32 representation to outperform the 8x8 version.

A comparison of the input representations and the effect on performance can be seen in Figure 10. As expected, performance is better with the 32x32 representation than with the 8x8 representation. However, it is interesting to note that the 8x8 had a much longer initial learning period before plateauing than the 32x32 did. In the 32x32, the learning plateaued almost immediately after just 2 or 3 epochs, but with the 8x8 representation, it took nearly 15 epochs before the learning plateaued. Overall, the 32x32 was more accurate by approximately 10%.

5.2 Output Representations

The two output representations we explored were 1 and 10 output nodes. For the single output node, a value was calculated between 0 and 1. That value was then multiplied by 10 and rounded down to the closest integer to get the output value. For example, if the activation was calculated to be 0.3256, that would be multiplied by ten, 3.256, then rounded down to 3. The 10 output node method worked a bit differently. Each node represented each one of the digits, 0-9, and each had its own set of weights. A value was calculated for each node, and the node with the highest value was selected as the calculated output.

A comparison of the output representations and the effect on performance can be seen in Figure 11 for the 32x32 input representation and in Figure 12 for the 8x8 input representation. In both cases, the 10 node output representation far outperforms the 1 node output. However, it is very interesting to observe the behavior of the two. For the 32x32 representation, the two output representations behave very similarly in that they increase for the first three or four epochs then plateau and do not improve significantly for the rest of the learning period. For the 8x8 representation, however, the 10 node output learns for 10-15 epochs then plateaus while the 1 node output doesn't seem to improve at all through the learning. However, it also does not truly plateau, but instead seems to oscillate and not make any significant improvements or retrogressions.

5.3 Learning Rate

We originally had planned to explore three learning rates, 0.1, 0.5, and 1.0. However, after some testing we noticed that the performance was not quite what we had expected. With the prior knowledge that a learning rate of 0.01 is a good rule of thumb, we decided to add that to our tests. The results were drastic across the board confirming that 0.01 was the optimal learning rate of the forms that we tested.

A comparison of the learning rates tested can be seen in Figure 13, Figure 14, Figure 15, and Figure 16. We begin our analysis looking at Figure 13, which varies the learning rate for the 8x8 input and the 1 output node. It is quite clear that the learning rate of 0.01 is far better than the other tested learning rates of 0.1, 0.5, and 1.0. In fact, with the higher learning rates, there was an immediate plateau. When we investigated this plateau in the learning, we discovered that when using the higher learning rates, the NN was correctly identifying all the zeros but nothing else. This being said, there was not any improvement in performance over the epochs with the learning rate of 0.01, and it oscillated, getting better then worse without any sense of a trend from epoch to epoch.

Figure 14 shows varying the learning rate for the 8x8 input and 10 output nodes. Similar to the 8x8 with 1 output node, the learning rate of 0.01 significantly outperforms the other learning rates we tested. However, in this case it is clear that there is actual learning happening for both 0.01 and 0.1. For 0.5 and 1.0, however, it appears that there is still no learning happening, and once again it is only identifying the zeros.

Figure 15 shows varying the learning rate for the 32x32 input and 1 output node. Once again, a learning rate of 0.01 is the most accurate over time, and we see it mostly plateau after 15-20 epochs and stay relatively steady. With a learning rate of 0.1, it also plateaus, though at a lower value, but it is not as steady and there is some oscillation up and down. This is emphasized at a learning rate of 0.5 in that it plateaus at an even lower value and has more significant oscillation. With the learning rate of 1, we once again see the NN not learning effectively and it is again only identifying the zeros.

Figure 16 shows varying the learning rate for the 32x32 input and 10 output nodes. In this case, it is not as clear which learning rate is most optimal, as the percent correct is very close for 0.01, 0.1, and 0.5.

However, upon closer inspection of the results found in Figure 17, we found that using 0.01 and 0.1 for the learning rate yielded the same accuracy and were slightly more accurate than using a learning rate of 0.5. These were all significantly better than a learning rate of 1.0, but it is worth noting that in this case, using a learning rate of 1.0 actually did see the NN improving over the epochs rather than only identifying the zeros.

5.4 Testing Results

After training was complete for each set of parameters, we tested the NN using the test files mentioned previously in this paper. The results from this portion of the testing can be seen in the table in Figure 17. It was clear from these results that a learning rate of 0.01 was optimal, using 10 output nodes was better than using just 1 output node, and the 32x32 input representation was more accurate than the 8x8.

6 FURTHER WORK

6.1 Stagnation

Clearly, our networks converged after a small number of epochs. To improve overall performance, we could consider implementing a cutoff, stopping training when little to no progress is being made by the network.

6.2 Timing

Though we know, in general, the running times of each network configuration, it might prove insightful to formally explore the effects of various configurations on the running time of the network's testing and training process, and, specifically, whether there are significant changes in how quickly the network converges to its 'best' performance.

6.3 Rounding

During the process of testing for networks of one output node, the node's value is multiplied by 10 and floored in order to determine the digit value. However, this method means that a value of 6.999 would be mapped to 6, while it is far closer to 7. Instead of flooring the result, we might see improved performance by rounding normally: that is, round up for $x.5$ and greater, and round down otherwise. There are two disadvantages to this method, however. First, it introduces the potential to output 10, which is not a digit. This could be remedied by only rounding to a digit. Second, it unevenly distributes the results, by only outputting 0 for 0.5 and less, and, depending on how the first issue is resolved, outputting 9 for a larger interval.

7 CONCLUSIONS

In our work, we tested the usefulness of perceptrons in the relevant problem of digit recognition. We explored the benefits and downfalls of each combination of input (32x32 or 8x8 maps) and output (1 or 10 nodes) representations. Unsurprisingly, we observed that our networks constructed with 10 output nodes (and hence requiring many more operations per epoch) performed significantly slower than those with just 1; surprisingly, however, when it came to recognition, those same networks vastly outperformed their speedier kin. In some cases, sacrificing performance for runtime is worthwhile; in the case of a perceptron, however, only so much performance can be sacrificed. In this case, the networks with 1 output node would be useless in practice, classifying approximately 80% incorrectly. In reflection, the perceptron was both relatively easy to implement and very effective (given optimal setup), showing the usefulness of such a network for real world problems. It could potentially be extended to all written letters and, paired with a program to scan and convert handwritten documents, used to digitize a wide range of source materials.

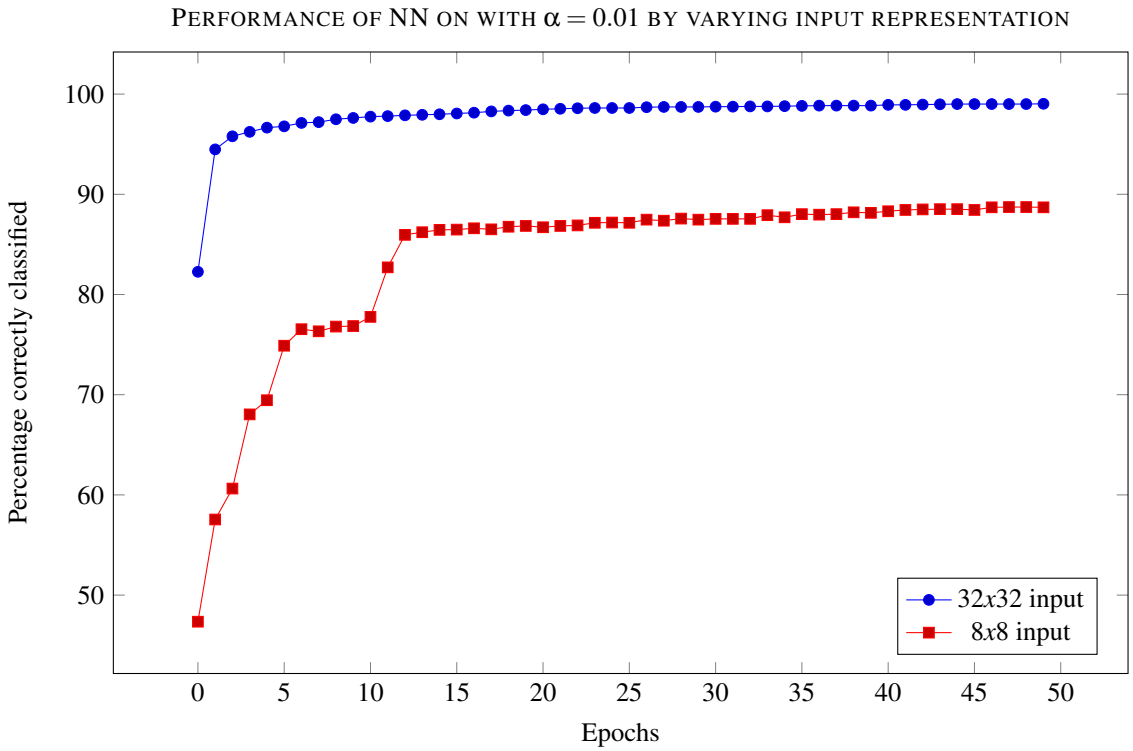

Figure 10: Neural Network trained using 10 output nodes and a learning rate of 0.01 for each input representation.

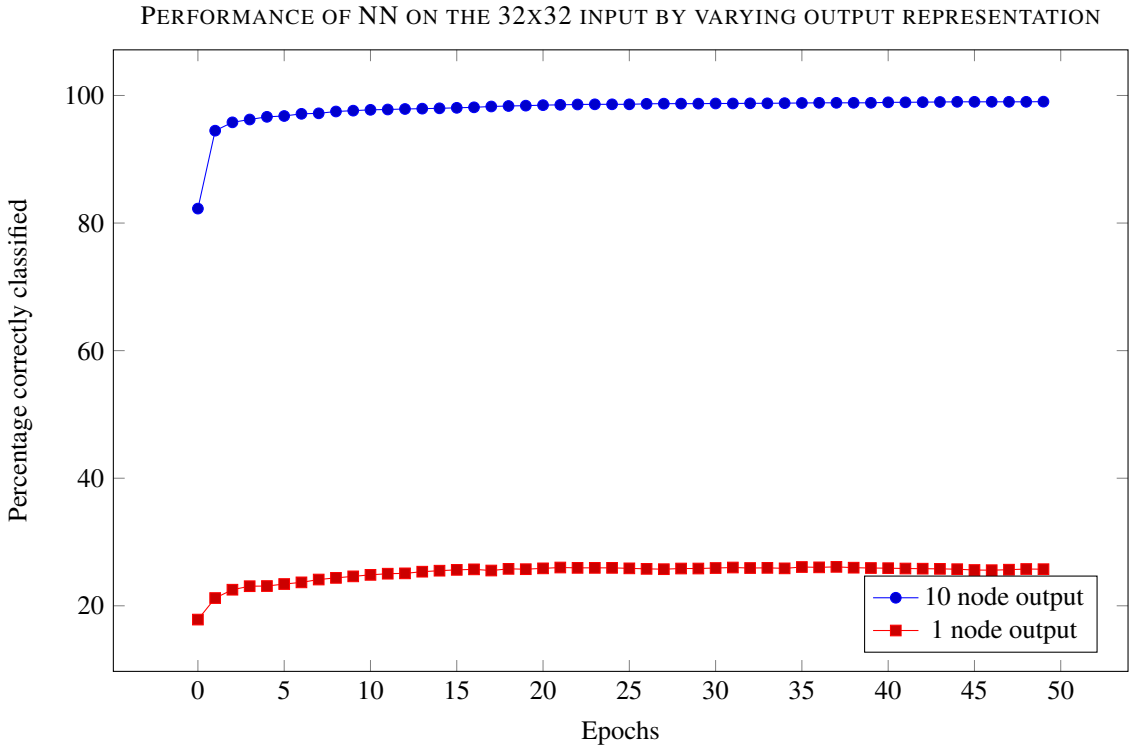

Figure 11: Neural Network trained using the 32x32 input representation and a learning rate of 0.01.

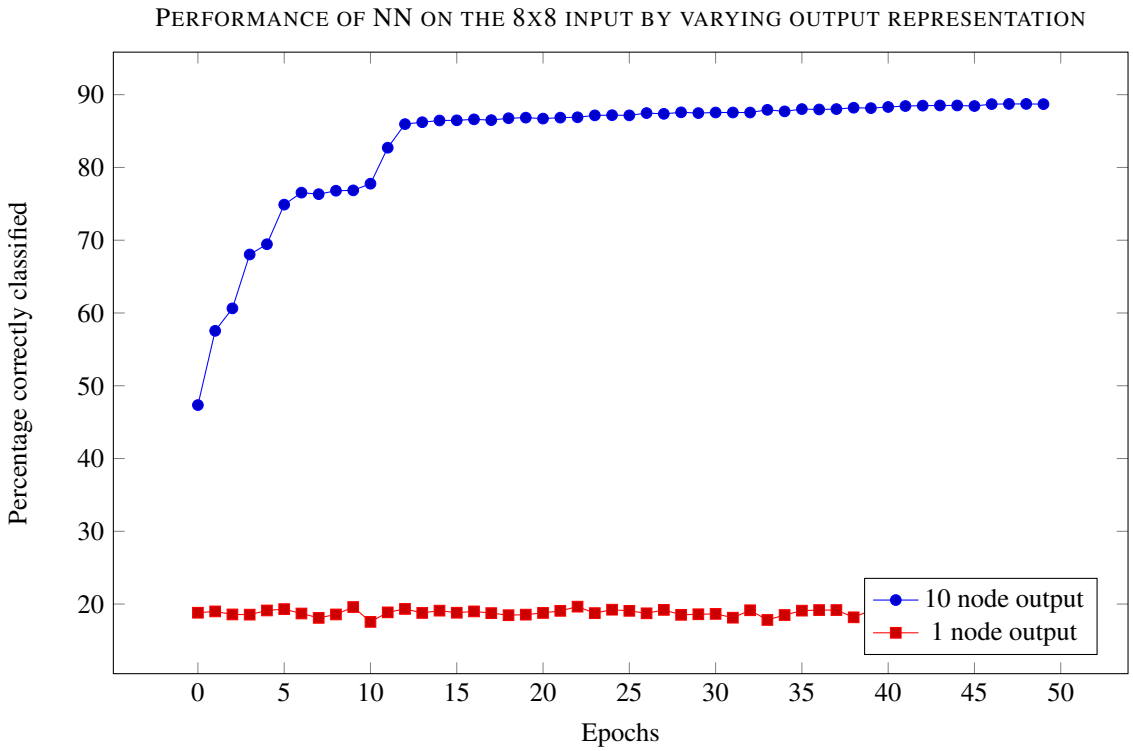

Figure 12: Neural Network trained using the 8x8 input representation and a learning rate of 0.01.

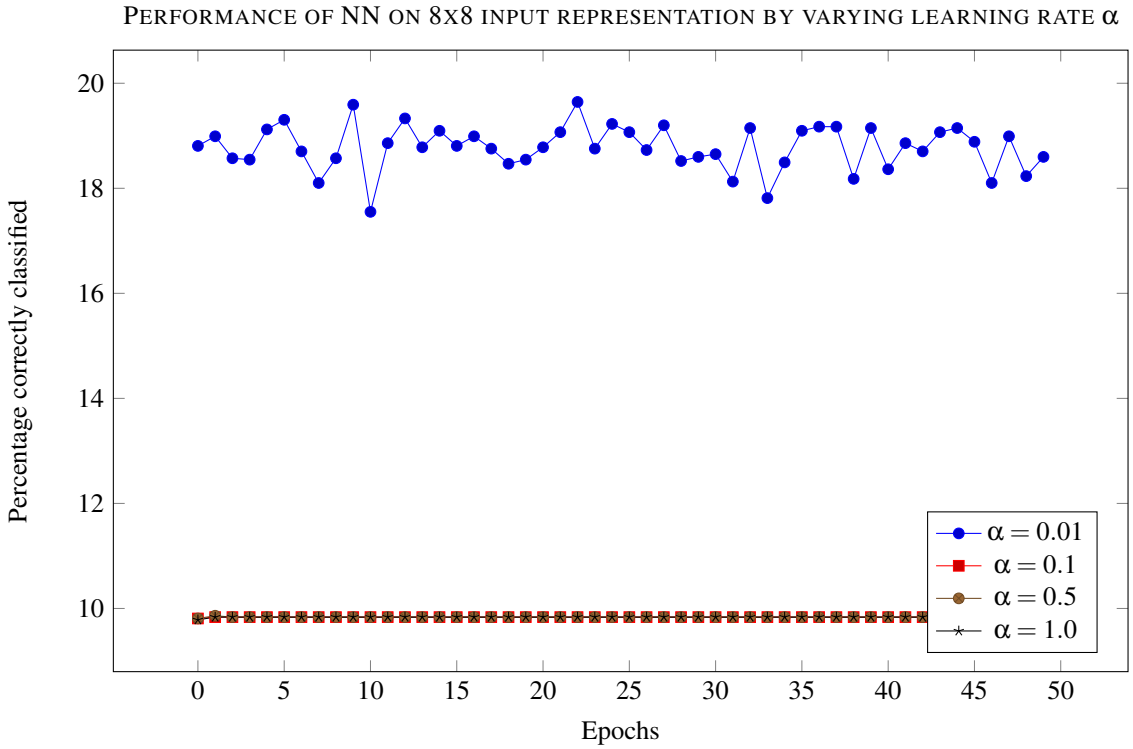

Figure 13: Neural Network trained on the 8x8 representation of the files with varying learning rates, and 1 output node.

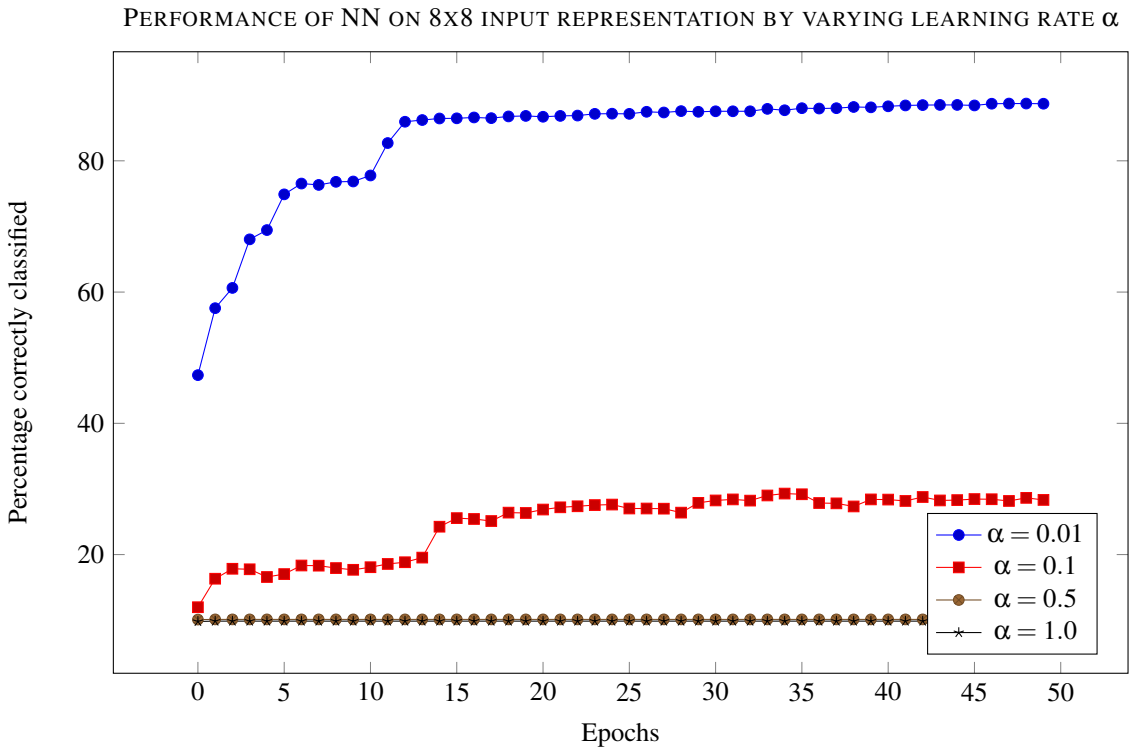

Figure 14: Neural Network trained on the 8x8 input representation of the files with varying learning rates, and 10 output nodes.

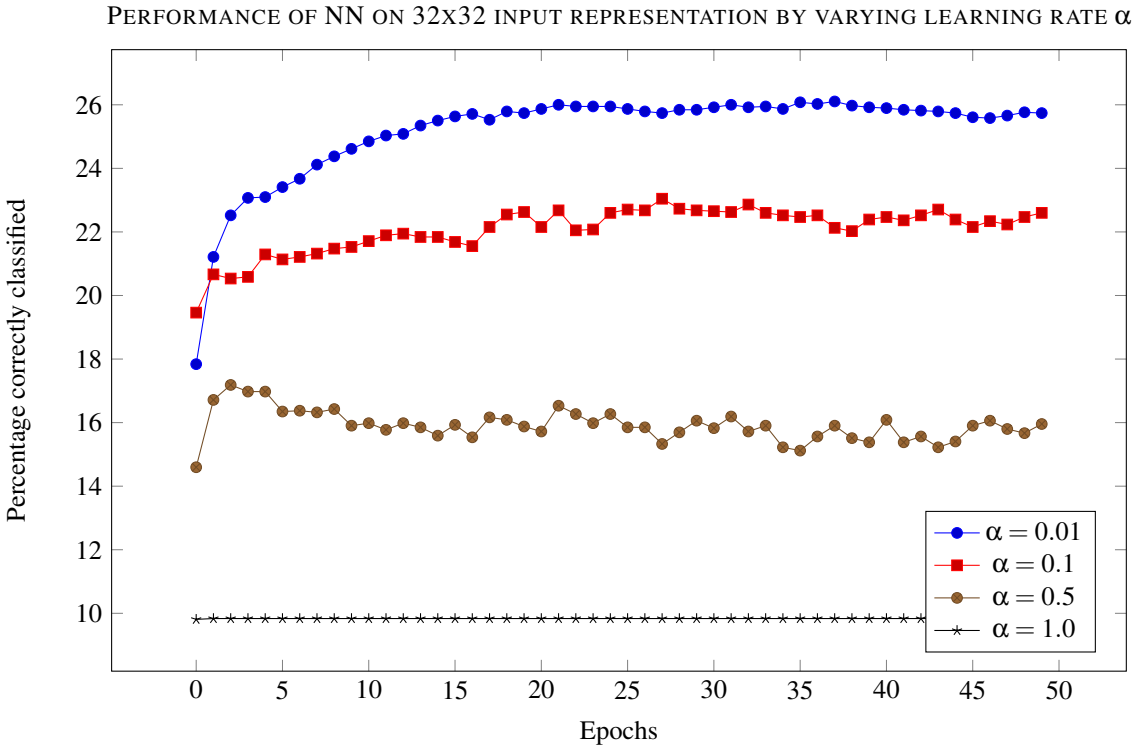

Figure 15: Neural Network trained on the 32x32 input representation of the files with varying learning rates, and 1 output node.

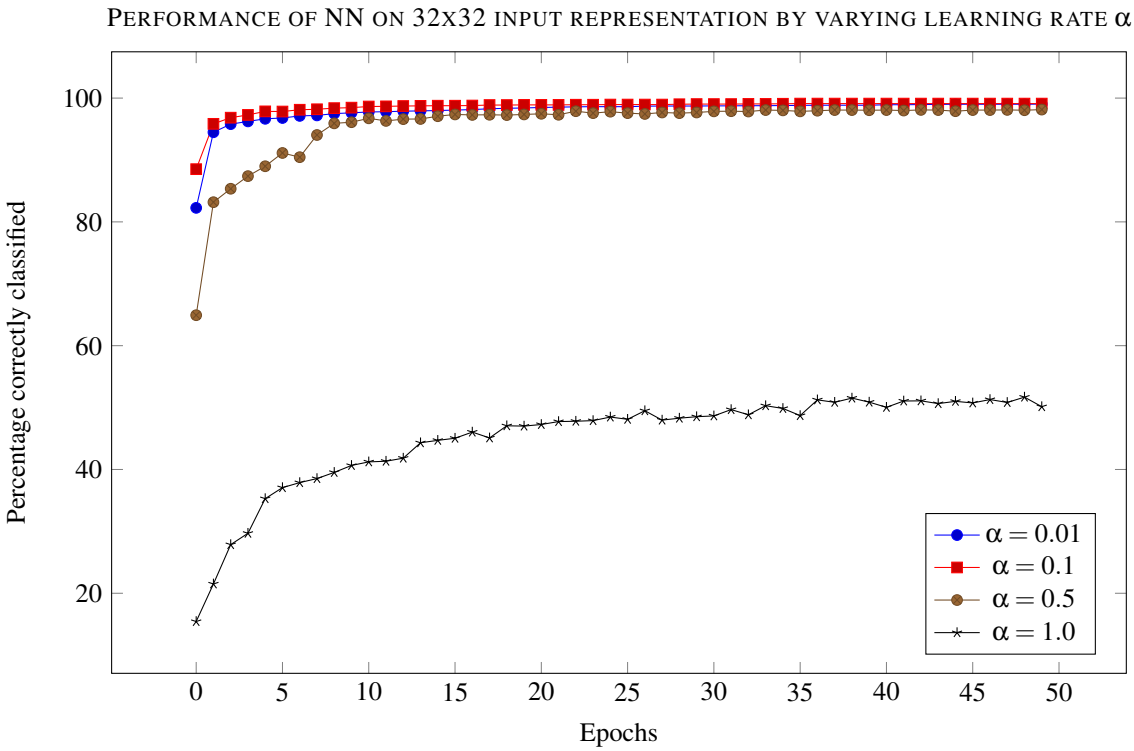

Figure 16: Neural Network trained on the 32x32 input representation of the files with varying learning rates, and 10 output nodes.

Input	Output	α	% Correctly Classified
8x8	1	0.01	20.9238
8x8	1	0.1	9.9054
8x8	1	0.5	9.9054
8x8	1	1.0	9.9054
8x8	10	0.01	85.4201
8x8	10	0.1	28.7145
8x8	10	0.5	10.0723
8x8	10	1.0	10.0723
32x32	1	0.01	22.8158
32x32	1	0.1	19.1987
32x32	1	0.5	15.9154
32x32	1	1.0	9.9054
32x32	10	0.01	94.7691
32x32	10	0.1	94.7691
32x32	10	0.5	94.2126
32x32	10	1.0	50.9738

Figure 17: Percentage of test images correctly classified in network testing given different input and output representations and their optimal learning rates.