Neural Networks for Digit Recognition

Marcus Christiansen, Ernesto Garcia, Konstantine Mushegian

May 2, 2017

Abstract

A neural network (NN) is a information processing paradigm inspired by the biological brain. It is a system of highly interconnected processing units, called artificial neurons, that can be "trained", through the use of training data and updating the connections between the units, to solve a specific problem. A problem that a NN can be used to solve is the digit recognition problem (DRP), where one needs to have a computer recognize discretized images of handwritten digits. In this paper, we further discuss the digit recognition problem, along with the issue of classification, as well as explain and implement a perceptron, a simplified NN, to recognize a series of hand written numbers with two different representations. We "train" this perceptron using a series of training data, before testing our trained perceptron against a set of test data. We vary the input representation, output representation as well as the learning rate of our perceptron to determine the effect of these parameters on the success of our perceptron. From our testing, we conclude that the down-sampled input representation with more output nodes and a higher learning rates leads to a more successful perceptron.

1 Introduction

Artificial Neural Networks, or simply Neural Networks (NNs) as we will refer to them in this paper, are a computational model and a machine learning technique that emerged from computer science and mathematics and is currently used in areas such as finance, medicine, robotics and etc. The first computational model for neural networks was introduced in 1943 by McCulloch and Pitts [2] based on mathematics and something called threshold logic. This model paved way for neural network research for the rest of the century. Neural networks are loosely based on the behavior observed by the human brain - a large collection of simple neural units, analogous to the axons in biological brains. Each neural unit is connected to others, and computes the information received from other neural units using the summation function. Neural units are able to propagate their own computations to neighboring units through these links; however this propagation can be limited in some way.

Neural networks are self-learning systems; instead of being explicitly programmed with rules defined by the human, they are able to learn from data -

a process that is known as training. Since neural networks do not rely on explicit rules specified by a human creator, they excel in areas where the solution is difficult to represent in a traditional computer program. Representation is an important issue in machine learning and we will go into more detail about this in Section 2. Neural networks currently provide amazing solutions to many problems in computer vision, natural language processing, robot locomotion and many others due to their flexibility. Neural networks are able to achieve this level of flexibility by mimicking the human brain and not relying on explicit problem-specific rules.

A problem that a neural network can be used to solve is the digit recognition problem (DRP), where one needs to have a computer recognize discretized images of handwritten digits. This is a very challenging problem for computers, as a computer must be "trained" to classify a certain digit, and as every handwritten digit is unique, it is not feasible to program a computer to solely look for specific characteristics of the digit, however, it is also impossible to train the computer to recognize everyone's unique handwriting. To solve this problem, we implement a perceptron to recognize a series of handwritten digits.

In this paper, we implement a perceptron, a simplified neural network, to classify a series of handwritten digits. We "train" our perceptron using a series of hand written numbers, where the weights between the nodes is updated according to a weight update rule described in Section 3.3. We then test our trained perceptron against a set of testing data to determine the ability of our perceptron to recognize handwritten digits. We explore the impact of the input representation, output representations as well as the learning rate of the perceptron on the performance of the perceptron as a function of the number of training epochs. The input representations we explore are 32x32 bitmaps to represent handwritten digits, as well as 8x8 down-sampled images derived from the bitmaps. The output representations we explore are 1 and 10 output nodes. The learning rates we explore are 0.1, 0.5 and 1. The number of epochs, 50, was kept the same for all runs. We train our perceptron for each combination of the above described parameters, and measure the performance of the trained perceptron against a series of test data by determining the percentage of correctly classified digits. We ran each parameter combination 5 times in order to gain a better estimation of the performance of our perceptron for each combination. For each set of parameters, we recorded the median best solution for each epoch iterations across the 5 runs. We then converted this data into line graphs to be able to determine which parameter combination was best for maximizing the percentage of correctly classified numbers.

From this testing, we discovered that the down-sampled input representation for the digits did very slightly better compared to the 32x32 bitmaps. This may simply be due to chance, as the difference in percentage caused by the two input representations is minimal. We were also able to determine that 10 outputs nodes was significantly better compared to 1 output node. Finally, we also discovered that a larger learning rate was better than a smaller learning rate, for both input representations and output representations.

In this paper, we will be designing and implementing a program that uses a

perceptron to recognize and categorize discretized images of handwritten digits, and then use our perceptron to explore the effects of output representation, input representation and learning rate on the percentage of correctly classified numbers. In Section 2 we will further be describing the issue of digit recognition, why this is a difficult issue, and how a perception can be used to solve this issue. In Section 3 we will describe neural networks, the perceptron, and how we plan to design and implement it. In Section 6, we will describe our experimental methodology, detailing the experiments that we ran, and our reasons behind them. In Section 7, we discuss and analyze the results our experiment. In Section 8, we discuss possible further work for our project, before providing some conclusions in Section 9.

2 Digit Recognition

The human vision system is an exceptionally complicated and impressive mechanism that is able to accurately detect and classify objects even in the most extreme conditions. Our vision system is so powerful because it is supported by a perfect *capturing* system that quickly adapts to changing conditions and delivers high quality *images* to the brain at a high frame-rate. Additionally, our vision system is also powered by a supercomputer that is able to process and interpret all of that information. Things get a little more complicated when we try to make a computer see things.

Consider the handwritten digits in Figure 2. You probably recognize them as 5, 7, 8, 3, 4, and 6. However, this seemingly simple and intuitive task becomes much harder when you ask the computer to recognize a sequence of digits in an image. In this paper we focus on recognizing and classifying a single handwritten digit, since identifying a string of them is beyond the scope of this paper, and involves image segmentation techniques.

The difficulty of this task becomes even more apparent when you try to think of how a computer program would recognize digits. The first question we would need to answer in order to get a computer to identify digits is the question of how we would present them to a computer and it is discussed in the section below.

2.1 Input Representation

Representation is an important part of any system that tries to achieve something humans can do. Unlike humans, computers have no concept of digits and their ability of working with data boils down to working with ones and zeros.

As part of this project we worked with two data sets of handwritten images provided by Professor Majercik. The first data set uses 32x32 bitmaps to represent handwritten digits; this format is also known as discretized binary images. This representation is similar to black-and-white images perceived by the human eye, where each pixel is assigned a value of either 0 or 1, where 0 stands

for a white and 1 stands for a black pixel. An example of this representation can be seen in Figure 3.

The second data set consists of 8x8 down-sampled images, created by dividing 32x32 bitmaps into 64 non-overlapping 4x4 blocks and creating a new 8 x 8 matrix whose elements are integers in the range 0-16 and represent the number of 1s in the corresponding 4 x 4 blocks in the original image[3]. An example of this representation can be seen in Figure 4.

We trained our neural network on both data sets in order to see how input representation affects the neural network performance. Our findings are discussed in section 7.

Now that we have a way of giving input to the computer, we can focus on more important tasks, such as making the computer classify the input as one digit or another.

The difficulty of this task becomes apparent when you try to think of ways to explain to a computer what a digit '6' looks like, i.e. when you try to lay out explicit rules for recognizing a '6'. One could try and come up with a set of rules to recognize digits written by a specific person; these rules would get quite complicated as every single detail about each digit's curvature would need to be included. This approach would work assuming that every digit looks exactly the same every single time, which usually isn't the case since people tend to write a little differently every single time. However, even if this approach worked it would be useless - creating a digit classifier for each person's handwriting would be too costly and inefficient, and the explicit rules for generalizing various people's handwriting would get very complicated very quickly.

Thus, a different approach is required to solve the problem of digit recognition. It is discussed in the following sections.

3 Neural Networks

A neural network, as implied by the name, is a set of interconnected neurons that share information across the connections with other neurons in the network. In nature, a neuron is a specialized cell that transmits electrical nerve impulses. The human brain, for example, contains a neural network with hundreds of millions of neurons and tens of billions of connections amongst these neurons. Our experiment creates an artificial neural network that loosely resembles neural networks in the human brain. For our purposes, a neuron is a mathematical function that takes a number of inputs to process and in turn outputs a single output determined by the activation function. The image below shows how a neuron produces an output [1]:

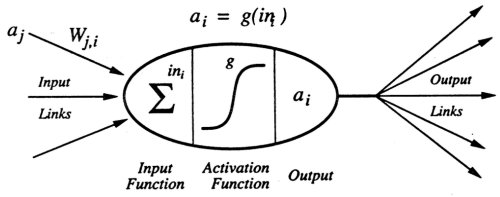


Figure 1

Where:

- a_j: Activation level coming from node j (i.e. value of node j)
- $W_{j,i}$: Weight on link from node j to node i
- in_i: Weighted sum of inputs to node i. $in_i = \sum jW_{j,i}a_j$
- g: Activation function
- a_i: Activation level of node i (i.e. output of node i)

A node takes in inputs $x_1, x_2, ...$ and processes these inputs using an activation function to produce a single output, as described in the figure above. These incoming inputs are coming from other neurons in the neural network. A neural network is typically constructed as layers of nodes, where each layer of nodes is connected to the next, however, the nodes within a certain layer are not interconnected. The edges connecting the neurons are marked with weights. It is the updating of these weights that "trains" the neural network by teaching it to solve a certain problem.

When training the neural network, for every training sample, the input nodes are given information inputs of a certain form from the training sample. This information then travels through the network and is manipulated by the individual nodes in each layer. This propagates through the network until it reaches the output nodes, which then produce a certain output. When an output is produced, the weights are updated according to the error between the output the neural network produced and the expected output. The updating of weights will be further described in section 3.3. The idea is that as the neural network is fed more training data, it will adjust it weights accordingly so as to minimize the error between the two outputs. Over time, it is expected that the neural network will be better to solve the problem is was designed to solve.

Today, there are various types of artificial neurons but for simplicity we will be implementing a neural network using a perceptron as our model of artificial neurons.

3.1 Perceptrons

A Perceptron artificial neuron model was studied and developed by Frank Rosenblatt, however Perceptrons were studied widely in the 1950's and 1960's because

neural networks based on the Perceptron model were the first ones capable of learning. The perceptron nodes act very similar to the description in the section above, however, there are no hidden layers, and thus, the input nodes are connected directly to the output nodes, as seen in figure 5. The pseudocode that we followed when implementing our perceptron can be seen in figure 9.

3.2 Activation Function

The activation function of a perception defines the output of a neuron given a set of inputs. The simplest activation function that can be implemented is the step function defined in the following manner:

$$f(x) = \begin{cases} 0 & \text{for x} < \text{t} \\ 1 & \text{x} \geq \text{t} \end{cases}$$

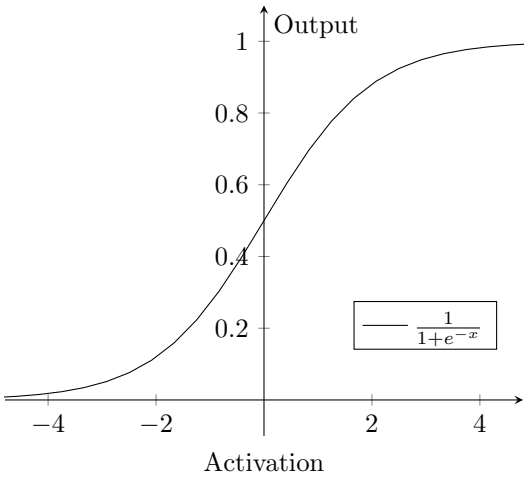
where t is the threshold value for the perceptron and $\text{x} = \sum_i w_i x_i$

Although this activation function may be very efficient, it is not very useful when we only have one perceptron defining the output of our neural network. Since the perceptron only outputs 0 or 1 it is necessary that the choice of output be deliberate. In our case, a perceptron can have its output flipped suddenly with a small change in the input. To fix this we will use an activation function whose output changes smoothly and slowly with the input.

3.2.1 Sigmoid Function

The sigmoid function is defined as follows:

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



The activation level that we are using as our x-axis is the sum of the weighted inputs that our perceptron receives and the output will determine the binary output of our perceptron depending on the threshold value assigned to it. We need the output of the activation function to be smooth but we also need it to be computationally inexpensive, particularly for differentiation.The derivative of the Sigmoid function is

$$f'(x) = f(x) * (1 - f(x))$$

For this reason we utilize the sigmoid function as opposed to any polynomial function that is also differentiable. Differentiation becomes necessary when we begin to adjust the weights on the incoming edges to our perceptron.

3.3 Weight Update

Perhaps the most interesting and useful part of the neural network training routine lies in the weight update procedure that let's the neural network 'learn' a task, which in our case is recognizing handwritten digits.

Since the goal of the training process is to minimize the error between the predictions made by the network and the desired output, we use the gradient descent over the error surface to figure out the direction in which to change the weights on the edges interconnecting the nodes.

3.3.1 Gradient Primer

Gradient is a generalization of the derivative to multiple variables, and is used in place of the derivative for functions of several variables. Gradient of a function $f(x_1, x_2, ..., x_n)$ is a vector that consists of n partial derivatives of f.

Just like the derivative, the gradient of a function also represents the slope of the tangent of the function, albeit in more dimensions. Gradient points in the direction of the greatest rate of increase of the function, and the magnitude of the gradient is the slope of the graph in that direction (pointed to by the gradient).

Following the gradient properties we alter the weights associated with our edges in the direction of the gradient descent, i.e. negative gradient in order to minimize the error.

The weight update function is show below [1]:

$$W_j = W_j + (\alpha * \Delta_i) + M * \Delta_{i-1} \tag{1}$$

Where:

- α: learning rate.
- $\Delta_i = I_j *$ Err $* g'(in)$.
  where:

- I_j: the activation level of node j
- Err: the difference between the output of the perceptron and the expected output
- g': the derivative of the Sigmoid function described in Section 3.2.1.
- in_i: Weighted sum of inputs to node i. $in_i = \sum jW_{j,i}a_j$

- M is a momentum "magic" constant that has been selected by trial and error; set to 0.5 in our implementation.
- Δ_{i-1} is Δ_i from the previous iteration.

We followed the weight update procedure from the slides distributed by Professor Majercik fairly closely, but we added the momentum term[7] that can help the network escape the local minima. Adding the momentum term simply means that we add a fraction (M) of the old weight change to the current one, where M is a constant in the interval [0,1] derived through trial and error. We found $M = 0.5$ to perform better than others. Adding the momentum term lets us go slightly further in the direction of the minimum while keeping M moderately small does not let us skip over the actual minimum.

4 Network Design

Many different artificial neural network structures exist and they wary in complexity based on the needs of individual problems and the resources available for computation. In our implementation we are using the simplest structure that consists of two, input and output, layers. The information in our neural network flows "left to right" from the input towards the output layer. Each layer consists of 1 or more nodes based on the representation of the training data.

The data sets used for training and testing our neural networks consists of 3,823 32x32 discretized binary images and 3,823 8x8 down-sampled images; this results in 1,024 and 64 'pixels' per image respectively. Each pixel is translated into an input node to the neural network, and therefore the size of the input layer is either 1,024 or 64 nodes. Input data representation of training sets used for this project is discussed in detail in Section 2.1.

The output of our neural network can be represented in two ways: using 10 nodes where each node corresponds to a digit from 0 through 9 and has a value between 0 and 1; using 1 node whose value is also between 0 and 1 but is multiplied by 10 and rounded up to compare with the validation data.

All nodes in the neural network are interconnected with weights corresponding to each edge between each node. The values from input nodes are multiplied by these weights as they travel between layers. This process is described in more detail in Section 3.

The graphical representation of the various cases mentioned above can be seen in Figures 5, 6, 7 and 8.

5 Code Description

This project was fully implemented in Python since the language is fast to prototype is and lets one perform complex operations on arrays in significantly fewer lines of code compared to other languages. However, the major reason for choosing Python was the NumPy library that provides fast and powerful linear algebra toolkit, as well as a native support for N-dimensional arrays. We decided that it was more useful to spend our time understanding the concepts of neural networks instead of spending hours coding up vector and matrix multiplication.

We first wrote our code without NumPy, using simple for-loops and Python primitives, in order to verify the correctness of our logic. Later on we rewrote parts of the program using the data structures and operations provided by NumPy; this resulted in a major speed up of the training process. Without NumPy 1 epoch of training with 3,823 samples took about 5 minutes, with NumPy the same process now takes about 7-8 seconds, which is approximately a 97% decrease in computation time. This let us run a significantly larger number of experiments and thoroughly test the performance of our implementation with various parameters. Our experimental methodology is discussed in Section 6.

We also decided to follow the practical trick suggested by LeCun et at.[5] and shuffle the training data at each epoch in order to present our network with the most 'unexpected' sample at each iteration. This should theoretically prevent us from 'memorizing' the training data.

6 Experimental Methodology

The testing for this project consisted of varying parameters of the neural network and comparing the accuracy of the resulting NNs on the testing data sets provided by Professor Majercik. Since the training process was extremely fast (thanks to NumPy) we were able to vary all parameters and determine which set of parameters produced a neural network with the highest accuracy.

We decided to write a Python script that would let us run a large number of experiments with a single command. We chose Python for this because the language is simple and expressive enough that we could build a comparatively complicated script in a relatively short amount of time. In order to minimize the time taken by experiments we parallelized the testing script according to the number of computing cores available on the machine, which significantly sped up the testing. The results of our automation attempts can be found in auto-test.py file that is included in the project repository.

Once the automated testing framework was completed we set out on our search. Below we present the parameters that we experimented with. We decided to try every possible combination of all parameters in order to find the best one. We trained the perceptron for 50 epochs, and tested it after each training epoch.

- Number of output neurons:
  - 1
  - 10
- Learning rate:
  - 0.1
  - 0.5
  - 1.0
- Input representation:
  - 32x32 discretized binary images (bitmaps)
  - 8x8 downsampled images

Each combination of parameters was run 5 times; this resulted in the grand total of 60 experiments and took about an hour to run (each combination 5 times). The experiments were run on the testing data sets (.tes) provided with the project.

7 Results

7.1 Format

We present our results from our experiments in figure 8 of our paper. The table contains the median best percentage of correctly classified digits and the median final percentage of correctly classified digits for all sets of parameters.

We also transform our data into a series of plots to demonstrate the impact of each variable (input representation, output representation, learning rate) on the performance of our perceptron. We present a total of 4, formatted as follows:

- Learning Rate's Effect on Percentage Correct for the Bit Map Input with One Output Node (Figure 11)
- Learning Rate's Effect on Percentage Correct for the Bit Map Input with Ten Output Nodes (Figure 12)
- Learning Rate's Effect on Percentage Correct for the Down-Sampled Input with One Output Node (Figure 13)
- Learning Rate's Effect on Percentage Correct for the Down-Sampled Input with Ten Output Nodes (Figure 14)

The graphs show median current value for each learning rate for each epoch. The learning rates corresponding to each of the plots on the graph are displayed in the legend on the right.

7.2 Discussion

The first outstanding observation is that when we are working with only one output node, regardless of the input representations, a 1.0 learning rate performed

significantly better in terms of accuracy percentage than a 0.1 or 0.5 learning rate. For the discretized binary input representation the 0.1 and 0.5 learning rate remained between a 0% and 20% accuracy percentage, while with a down-sampled input representation both learning rates stayed between a 0% and 10% accuracy percentage. The 1.0 learning rate, on the other hand, scored upwards of a 90% accuracy rate for both types of input representations. This would suggest that, when we have a single output node, there is some value between 0.5 and 1.0 that would cause a sharp increase in our accuracy percentages.

Having ten output nodes resulted in closer accuracy percentages for all three learning rates in both of our input representation tests. A 1.0 learning rate still returned produced percentages upwards of 90%, but still lower than 95%. In other words we saw a much bigger increase in accuracy for both the 0.1 and 0.5 learning rates, but increasing the number of output nodes did not noticeably improve the accuracy of the 1.0 learning rate.

For our discretized binary testing all the percentages are converging to a common value as the number of epochs increases. They converge at different rates but a surprising observation is that the 0.1 learning rate seems to mitigate its oscillation at a faster rate than that of the 0.5 learning rate. In contrast to our results using one output node, it appears that the learning rate seems to be less important if we have a large enough number of epochs.

This was not the case for our the neural network testing with the down-sampled input representations. Even though the 0.1 and 0.5 learning rates did improve significantly, they still did not perform at the same capacity as a 1.0 learning rate. When trained for less than ten epochs all three learning rates seemed to have a similar level of accuracy but as we trained for more epochs, the 1.0 learning rate consistently outperformed the other two. In addition, the accuracy percentages have much more volatility even as we continue to train the neural network for more epochs. The 1.0 learning rate, however, approaches a steady state after the a certain number of epochs.

Something else we noticed was that for the smaller learning rates, there seemed to be more fluctuation in the percentage of correctly classified digits. This was not as much the case with the 1.0 learning rate, as this percentage slowly converged over the course of the epochs.

Additionally, we also saw that for the down-sampled input representation, there was significantly more fluctuation in the percentage of correctly classified digits compared to the bit map, where the bit map tended to converge to a stable percentage.

8 Further Work

We were able to successfully implement the program that uses a Perceptron artificial neuron model to recognize discretized images of handwritten digits. We were quite satisfied with the correctness level that our program was able to achieve, thus further work would focus on improving the efficiency of the system.

An obvious improvement would be to speed up the training process; this could be achieved by parallelizing the training procedure and potentially running it on the Bowdoin High Performance Cluster (HPC) and utilizing Graphics Processing Units (GPUs) which are optimized for high performance numerical computations, as opposed to Central Processing Units (CPUs) that are contained in every regular computer.

Another improvement would be to introduce learning rate adaptation [6] to the training process of our Perceptron artificial neural model implementation, which has been shown to result in higher rates of convergence in comparison to approaches where the learning rate is fixed.

Finally, an additional improvement for this project would be to implement a visualizer that would display the network structure along with the learning progress, similar to the MatLab interface. An intuitive GUI would let users get a better understanding of what the program was doing, as well as keep track of the progress made during the learning stage.

9 Conclusions

Over the course of this project we were able to successfully train our neural network without hidden layers to accurately classify handwritten digits. We trained our neural network on varying parameters which included the type of input representation,the number of output nodes, and different learning rates. What we found was that the increasing the learning rates for our neural network did not have a direct and linear correlation with accuracy as one would expect. Instead, the accuracy of our neural network depended largely on the number of output nodes that we implemented.

We only tested on two different numbers of output nodes: one and ten. What we found was that, when we have a single output node, the learning rate does not change the level of accuracy of our neural network across different input representations i.e the 1.0 learning rate performed similarly for both input representations etc. This does not mean, however, that a higher learning rate does not improve performance in our network. This is highlighted by the fact that changing our learning rate from 0.5 to 1.0 increased our percentage of accuracy by greater than 70% for both input representations. This indicates that there is a sharp increase in accuracy for some value of our learning rate between 0.5 and 1.0 that we would have attempted to isolate had we had more time.

Additionally, we found that when we have have a discretized binary input representation and ten output nodes the learning rate becomes less significant as the number of epochs that we train our neural network for increases. The percentage of correct classifications converge to the same value for all three learning rates that we tested. This was not the case for the downsized input representation and ten output nodes test. Even though the lower learning rates performed better than when we there was only one output node, they still did not reach the same percentage of correct classifications as the 1.0 learning rate.

10 Figures

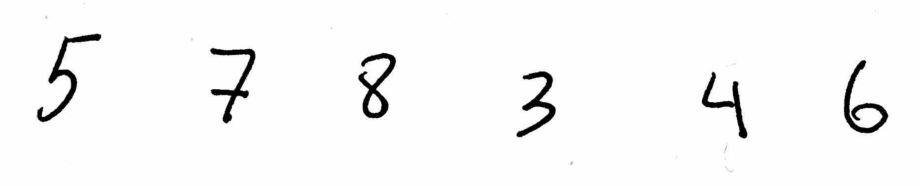


Figure 2: Handwritten Digits

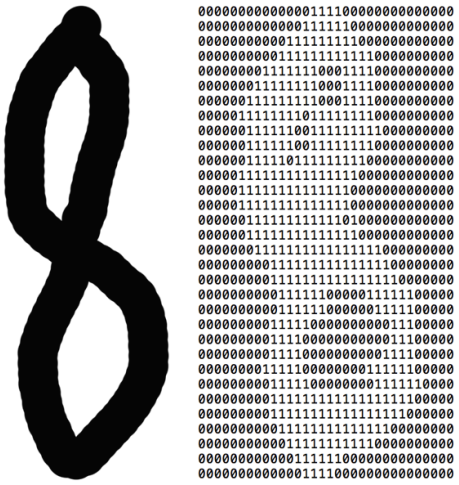
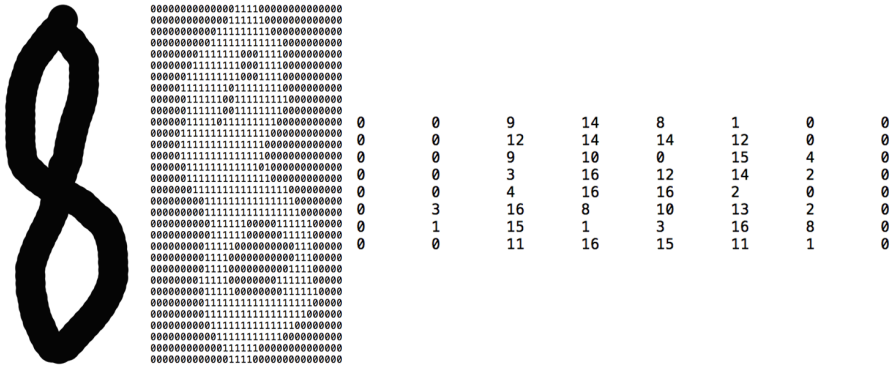


Figure 3: Handwritten digit 8 and its discretized 32x32 binary counterpart



0	0	9	14	8	1	0	0
0	0	12	14	14	12	0	0
0	0	9	10	0	15	4	0
0	0	3	16	12	14	2	0
0	0	4	16	16	2	0	0
0	3	16	8	10	13	2	0
0	1	15	1	3	16	8	0
0	0	11	16	15	11	1	0

Figure 4: Handwritten digit 8 along with its discretized 32x32 binary and 8x8 down-sampled counterparts

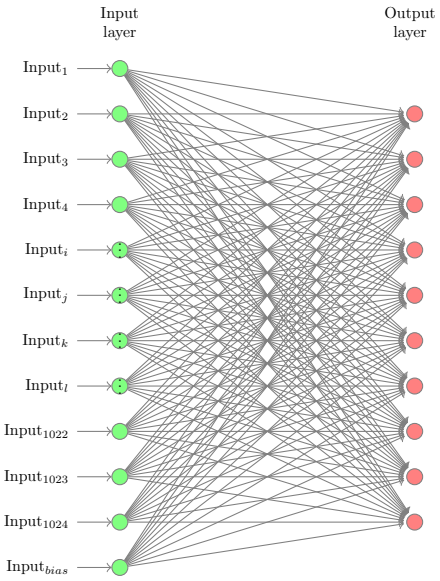


Figure 5: 1024 input nodes, 10 output nodes, and a bias node

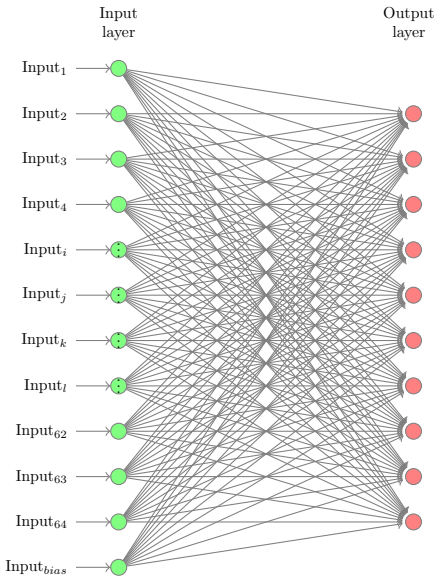


Figure 6: 64 input nodes, 10 output nodes, and a bias node

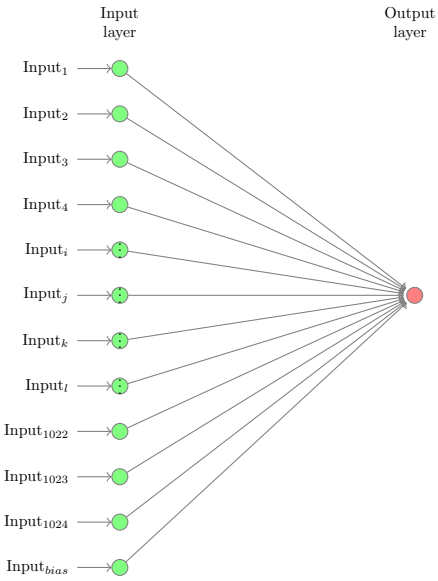


Figure 7: 1024 input nodes, 1 output node, and a bias node

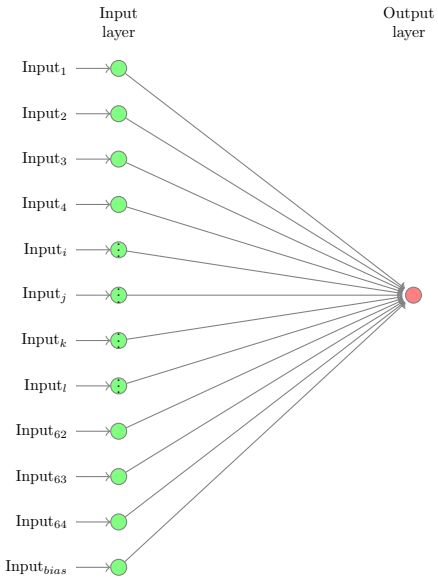


Figure 8: 64 input nodes, 1 output node, and a bias node

Algorithm 1 Perceptron

```
Input: numInputs, numOutputs, numEpochs, learnRate, trainingExamples,
inActivations, outActivations
weights ← initializeRandWeightArray[numInputs , numOutputs]
for numEpochs do
    for i ∈ trainingExamples do
        x ← expected output
        initializeInputActivations(inActivations)
        for j ∈ outActivations do
            inputSum = 0
            for k ∈ inActivations do
                inputSum += inActivations[k] · allWeights[k]
            outActivations[j] = sigmoid(inputSum)
        y ← calculateOutput(outActivations)
        Err ← x − y
        for j ∈ allWeights do
            UpdateWeight(j, Err, learnRate)
```

Figure 9: Perceptron pseudocode

Representation	Output Nodes	Learning Rate	Best Result	Final Result
Bit Map	1	0.1	10.68%	10.51%
		0.5	10.74%	9.62%
		1.0	94.21%	93.54%
	10	0.1	93.43%	84.91%
		0.5	93.43%	92.43%
		1.0	93.98%	93.65%
Down-sampled	1	0.1	16.86%	11.35%
		0.5	15.63%	14.80%
		1.0	94.60%	93.82%
	10	0.1	93.98%	93.43%
		0.5	94.21%	93.48%
		1.0	94.26%	93.65%

Figure 10: Table summarizing the experiment results

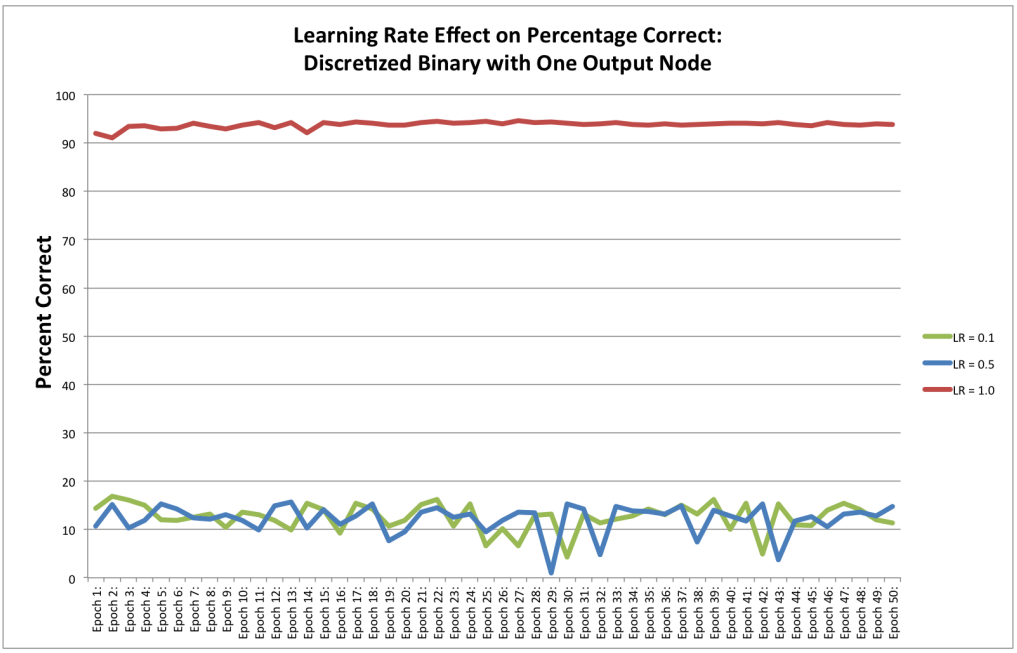


Figure 11

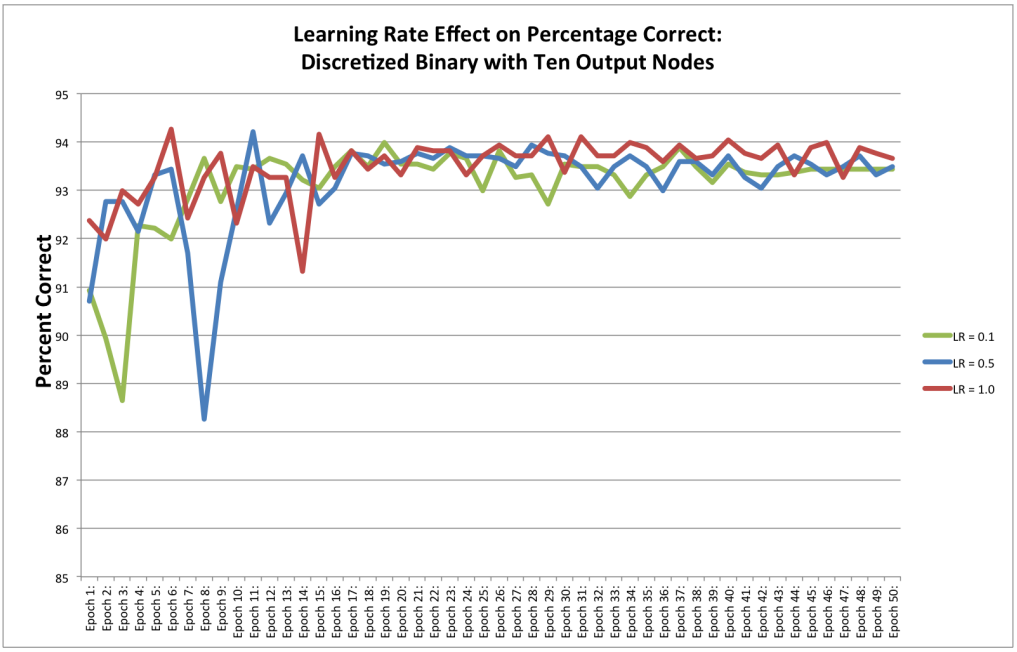


Figure 12

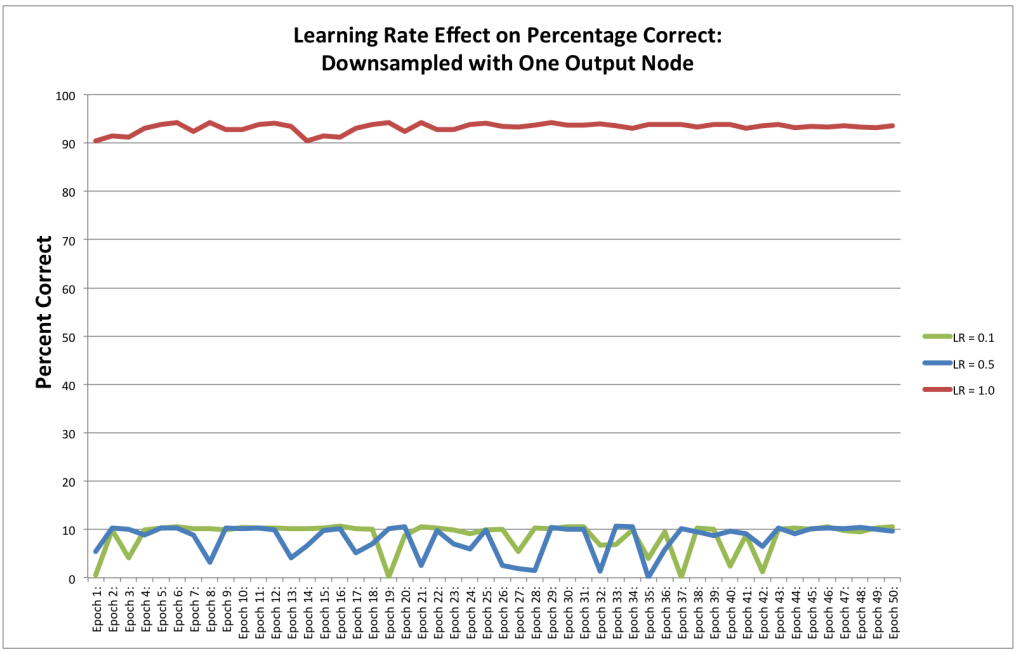


Figure 13

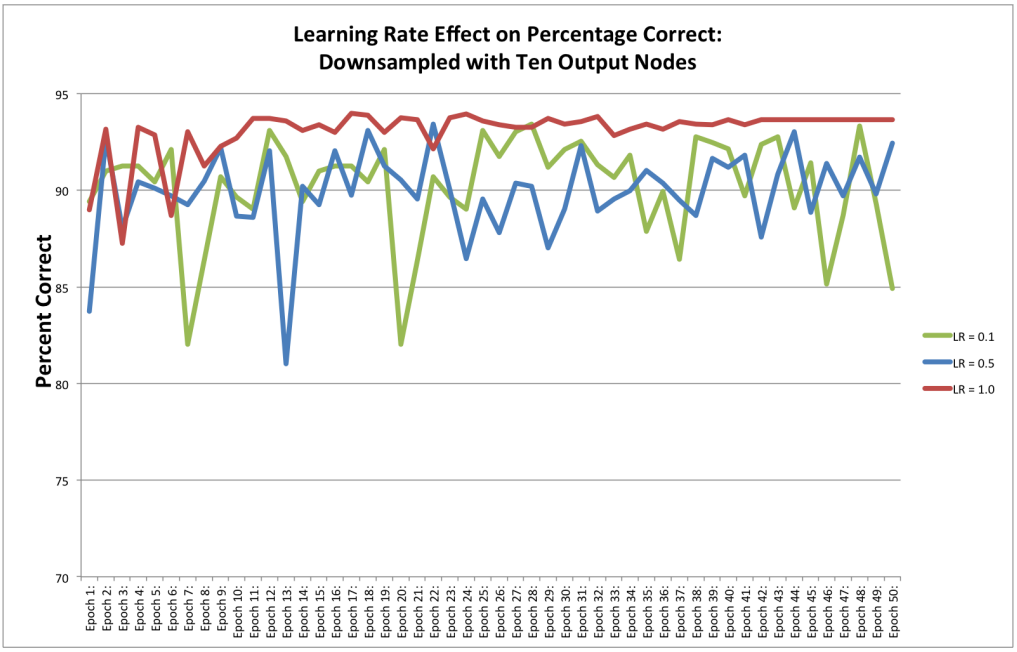


Figure 14

References

[1] Majercik, Stephen. "Neural Networks". Lecture, Bowdoin College, April 10, 12, 17 and 19, 2017

[2] McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." The bulletin of mathematical biophysics 5.4 (1943): 115-133.

[3] Majercik, Stephen. "Neural Networks for Digit Recognition". Project assignment. 2017. 1.

[4] Wikipedia contributors. "Gradient." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 22 Mar. 2017. Web. 2 May. 2017

[5] LeCun, Yann A., et al. "Efficient backprop." Neural networks: Tricks of the trade. Springer Berlin Heidelberg, 2012. 9-48.

[6] Jacobs, Robert A. "Increased rates of convergence through learning rate adaptation." Neural networks 1.4 (1988): 295-307.

[7] Orr, Genevieve (Jenny) B. "Momentum and Learning Rate Adaptation." Momentum and Learning Rate Adaptation. Willamette University, n.d. Web. 1 May 2017. https://www.willamette.edu/ gorr/classes/cs449/momrate.html.