**MNIST Handwriting Recognition**

Daniel Nguyen

March 4th, 2018

**Abstract**

In this project, we will use neural networks in order to construct an algorithm that identifies MNIST Handwriting digits from 0-9. We will start from a simple one-layer network, then on to a two-layer network, and we will generalize the logistics of our algorithm into neural networks with more layers.

**Contents**

**1 Introduction**

**2 Theory**

**3 MATLAB Procedure**

**4 Results**

**5 Summary and Conclusions**

**Appendix A MATLAB Functions Used and Extra Graphs**

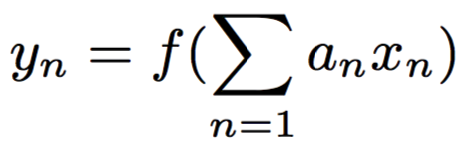
**Appendix B MATLAB CODE**

**1 Introduction**

Handwriting recognition is important to modern innovations such touch-based/pen-based software, which can be found in electronic devices such as tablets. For this assignment, we will be working with the MNIST database of handwritten digits. The MNIST data set was developed by Yann LeCun, Corinna Cores and Christopher Burges in order to evaluate the differences between machine learning models on the handwritten digit classification problem. Extremely strong state-of-the-art prediction models typically have an error of less than 0.1%, but we will be less ambitious in our work and analyze what we get from building a basic one-layer neural network, and we will work our way up to two-layers to see the difference.

**2 Theory**

A deep neural network (DNN) is composed of a layer of input nodes called “perceptrons”, layers of output nodes, and one or more “hidden layers” that connects the inputs to the outputs. We can use a DNN to model non-linear relationships in data, and we can use DNN as a learning algorithm by modifying the weights of the connections between nodes in each layer of the network. If we have to be our input data, to be our output data, and to be the weight of each input’s connection, then we can model the output with the following equation:



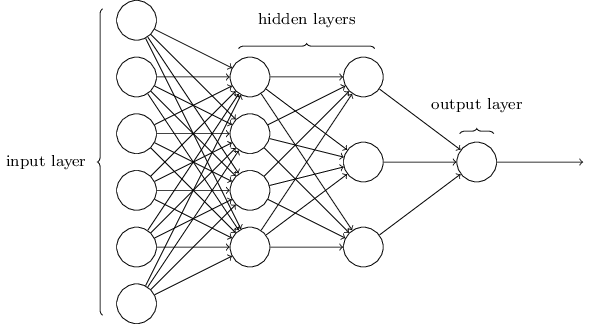


Figure 1: A visualization of a neural network.

Where is a transfer function. We can represent this linearly in the matrix form , where is a matrix of weights . These weights can be found using Stochastic Gradient Descent (SGD). Gradient descent is a method for finding a point in a set of data where the derivative in all directions are zero (in a 3D example, this could be thought of as rolling down a hill to a minimum point in the graph). However, we run into problems as it is possible to fall into a local minimum rather than the global minimum. That is why we need a stochastic implementation of this concept, where every once in a while, the algorithm kicks our point randomly by some distance step so as to avoid getting stuck inside a local minimum.

The reason why we want to find a point where the derivative in all directions are zero is because we want to minimize the error of our function. One can think back to the concept of least-squares fitting, In order to find the solution, or the correct constants of our line or curve, we need to set the gradient equal to zero, where the sum of the squares are the least. For a line, this is all fine and good, but for a data set with significantly higher dimensions, it becomes infeasible to calculate thousands of partial derivatives for instance. This is why gradient descent is such a useful tool – it is an efficient method of finding a zero gradient while cutting down computation.

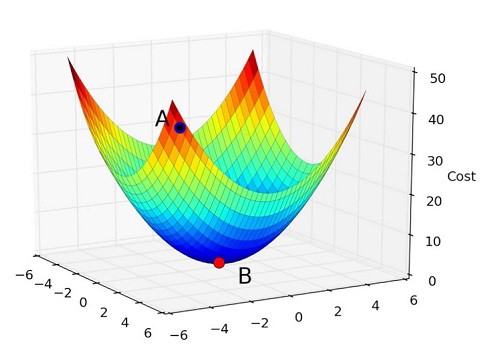


Figure 2: A visualization of gradient descent, where point A travels in steps to find point B.

**3 MATLAB Procedure**

We are given a data set of 70,000 handwritten digits from 0-9. Each digit is represented by a 28x28 pixel grayscale image, yielding 784 pixels in total. The digits are given in two sets: 42,000 digits for training and 28,000 digits for testing.

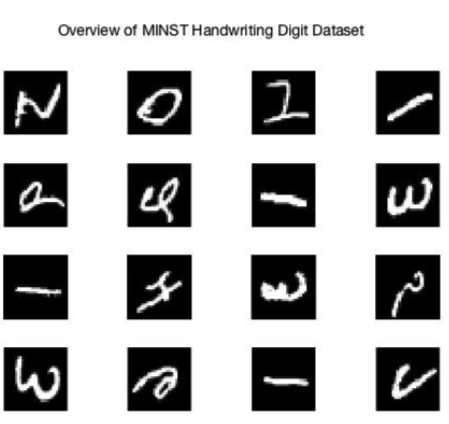


Figure 3: A small overview of the MNIST data set.

With our digits loaded into the script, we need to come up with a labeling system for our digits. We should add 1 to all the digits to use as our label, since we can’t use 0 as an index when working with matrix structures in MATLAB.

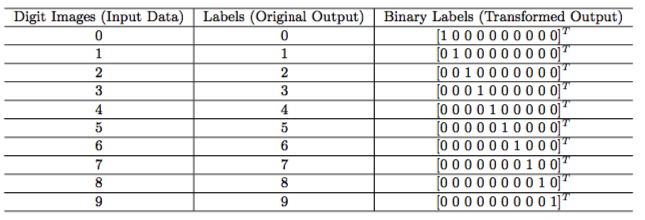
****

Figure 4: Labeling scheme for each digits.

And most importantly, we need a way to cross-validate our data. I have done this by using the randperm() command in order to shuffle the indexes of the digits that will be used for each iteration. I will choose to cross-validate 5 times for each case that I will be exploring.

We then feed our training data into the neural net, which will use stochastic gradient descent in order to find the weights of the connections in the neural network. I will be using the Neural Network Toolbox offered by MATLAB, and I will use the patternnet() command and net.layers.transferFcn{1} to do this. We then use those computed weights in order to label our digits according to the values of our output later. Once we have done that, we can find the error by determining the number of digits that have been misclassified, and we find the accuracy by subtracting the error from 100%.

I will want to explore 6 cases. Cases 1-3 will explore the effect of the training size on the accuracy for a one-layer network and cases 4-6 will vary with the same training sizes but on a two-layer network. The number of digits that I will be using for my training set will be 30,000, 35,000 and 40,000.

**4 Results**

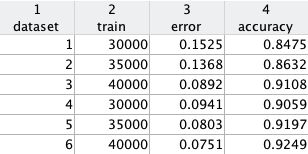
****

Figure 5: The errors for each case that I have explored. Data sets 1-3 deal with a 1-layer network while the rest deal with a 2-layer network.

It is quite easy to see from figure 5 that the error tends to go down as we either increase the size of our data set or increase the number of layers. On the lowest size data set, we go down to 84.75%. But when we increase to two layers, we get 90.59% on the first run.

**5 Summary and Conclusions**

We found that the accuracy of our network always increases as we increase our number of training examples. This makes sense since as the size of the training set increases, the matrix A would have more training which would lead to better weights. We have also found that as we add more layers into our dataset, the accuracy also changes.

All in all, in order to get an extremely high accuracy rate like the big companies do nowadays, a lot of manipulation in the construction of a neural network is necessary.

**Appendix A MATLAB Functions Used**

csvread();

randperm();

patternnet();

transferFcn();

view(net);

perform();

**Appendix B MATLAB Code**

clear all;

close all;

clc;

%%

digit\_train = csvread('train.csv',1,0);

digit\_test = csvread('test.csv',1,0);

labels\_train = digit\_train(:,1);

digit\_train = digit\_train(:,2:785);

%%

trainnum = 35000;

testnum = 28000;

randind = randperm(42000);

train = randind(1:trainnum);

testingIndex = randind(trainnum:end);

trainingData = digit\_train(train,:);

LTrain = labels\_train(train);

testingData = digit\_train(testingIndex,:);

LTest = labels\_train(testingIndex);

labels = zeros(2,trainnum);

for i = 1:35000

j = LTrain(i);

labels(j+1,i) = 1;

end

x = trainingData';

x2 = testingData';

net = patternnet(10,'trainscg');

net.layers{1}.transferFcn = 'tansig';

net = train(net,trainingData',labels);

view(net)

y = net(x);

y2= net(x2);

performance = perform(net,labels,y);

classes2 = vec2ind(y);

classes3 = vec2ind(y2);

subplot(4,1,1), bar(y(1,:),'FaceColor',[.6 .6 .6],'EdgeColor','k')

subplot(4,1,2), bar(y(2,:),'FaceColor',[.6 .6 .6],'EdgeColor','k')

subplot(4,1,3), bar(y2(1,:),'FaceColor',[.6 .6 .6],'EdgeColor','k')

subplot(4,1,4), bar(y2(2,:),'FaceColor',[.6 .6 .6],'EdgeColor','k')

wrong = LTest' - classes3;

for i = 1:length(wrong)

if wrong(i) == -1

wrong(i) = 0;

else

wrong(i) = 1;

end

end

accuracy = 1 - mean(wrong);