Implementing an Artificial Neural Network to Recognize Handwritten Digits

New Mexico
Supercomputing Challenge
Final Report
April 4, 2018

Team Number 201
Los Alamos High School

Team Members:
Sam Crooks

Project Mentor:
Jacob Stinnett

Table of Contents

Executive Summary

Computer vision is a rapidly developing field of computer science which aims to autonomously identify objects from digital images and videos. One widely used approach in computer vision is an artificial neural network, which was inspired by the structure of biological neurons in the brain. The objective of this project was to implement an artificial neural network algorithm that could learn to accurately classify handwritten digits.

The algorithm was programmed in Anaconda Python and includes the following stages: data processing, defining the neural network structure, training to optimize the weight and bias matrices, and performance analysis. During data processing, each 28 by 28 pixelated digit is stored as a CSV file and is converted to an input vector. Then, 42,000 labeled handwritten digits from the MNIST dataset are divided into 30,000 training examples and 12,000 test examples. Next, the parameters of the neural network (weights and biases) are initialized. During the training stage, the parameters of the neural network update by gradient descent, where the goal is to minimize a cost function.

After 100 training iterations, the best performing structure correctly classified 97.4% of the test dataset. To evaluate the performance, accuracy and cost were graphed as a function of iterations.

Further work includes varying the number of hidden layers in the neural network, implementing more advanced machine learning techniques, and training the algorithm to recognize other symbols and objects. Applications of this technology include reading addresses on postage, processing images of checks, and autonomous vehicles.

1. Introduction

Computer vision is a rapidly developing field of computer science which aims to autonomously identify objects from digital images and videos. When humans look at symbols and objects, their brains effortlessly recognize them. Similar to the way cone cells in human eyes are used to recognize objects, computer vision systems view objects as pixelated images. One specific task within computer vision is to recognize and classify handwritten digits. In this project, a machine learning technique called an artificial neural network is implemented to recognize handwritten digits.

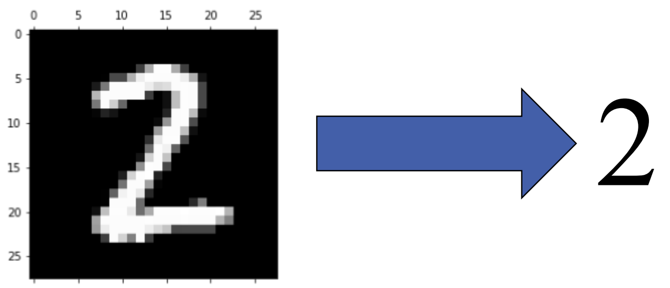


Figure 1: Pixelated image is recognized as the number two

2. Background Research

Simple logical rules cannot accurately classify handwritten digits. Rather than explicitly programming rules, machine learning techniques provide systems with the ability to learn and make accurate predictions on their own. The type of machine learning technique implemented in this project is an artificial neural network. An artificial neural network is a model inspired by the structure of biological neurons in the brain [3]. Neural networks are a highly researched and developed field, and there are a variety of neural network structures.

A feed-forward, fully-connected artificial neural network consists of a number of artificial neurons arranged into layers. The first layer (the input layer) receives input from the real world, and the last layer is the calculated predictions. Each middle (hidden) layer processes the output of the previous layer. Each neuron is connected to each other neuron in neighboring layers by a weight. By adjusting the weights and biases of the network to minimize a cost function, the neural network learns to classify images.

3. Problem Description

One specific type of computer image recognition involves recognizing human handwriting. It can be challenging for a computer algorithm to recognize handwritten digits because not only can the specific pixel values of one digit greatly differ across images of the same digit, but images of different digits may also look very similar. In order to implement a machine learning algorithm for digit recognition, the algorithm must learn from numerous labeled training examples.

4. Objective

The objective of this project is to implement an artificial neural network algorithm that can learn to accurately classify handwritten digits (zero through nine) from the MNIST dataset.

5. Algorithm Design

5.1 Data Processing

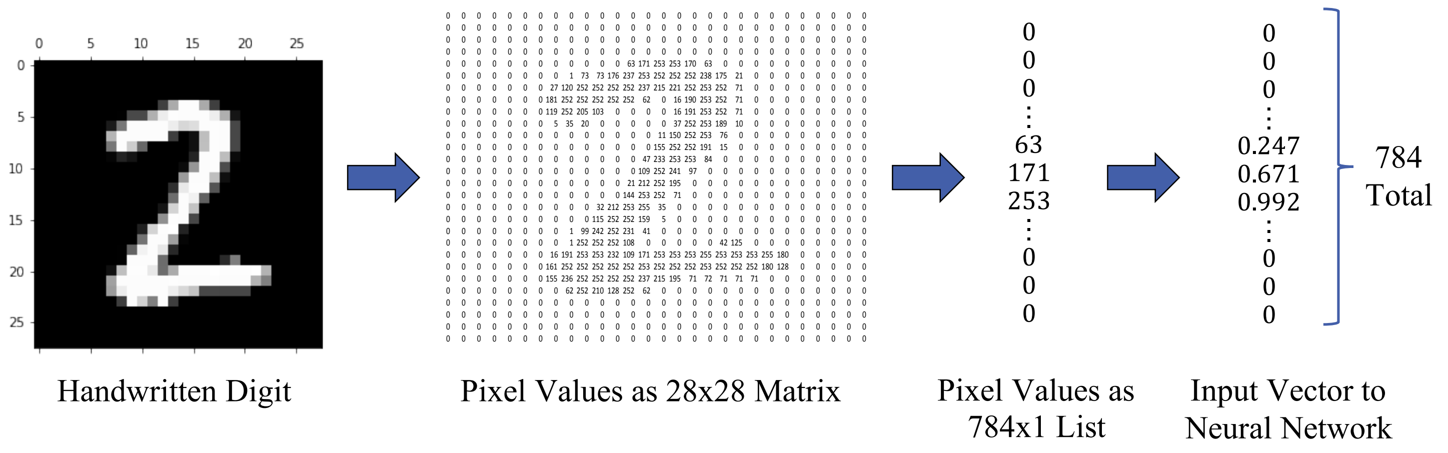


Figure 2: Data Processing Flowchart

A version of the MNIST (Modified National Institute of Standards and Technology) dataset contains 42,000 labeled digits stored as a CSV file [2]. Each digit is 28 by 28 pixels, for a total of 784 pixels. Each row stores each digit’s data. The first column contains a label, and the remaining 784 columns are the pixel values corresponding to the digit. Each pixel value ranges from 0 (black) to 255 (white).

In this stage, images are converted into data that the computer can interpret. First, each digit image is converted to a 28 by 28 matrix. Second, all of the pixel values are put into a list. Third, each pixel value is divided by 255 to normalize the list to values between 0 and 1. Figure 2 demonstrates this process.

The 42,000 labeled digits were divided into 30,000 training examples and 12,000 testing examples. The training examples are used to develop the predictive model, and the testing examples are used to evaluate the performance of the model.

5.2 Neural Network Structure

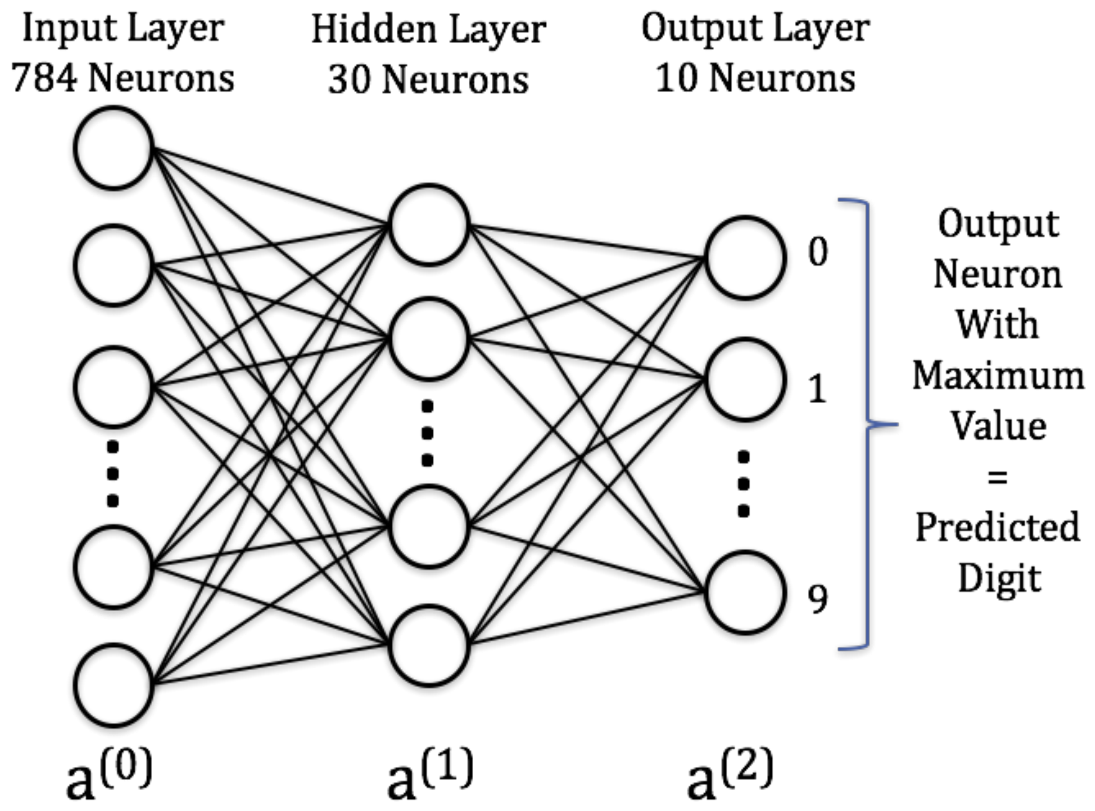


Figure 3: Neural Network Diagram

This neural network structure consists of three layers. Each neuron in the network is a function that outputs a value between 0 and 1 depending on the outputs of all the neurons in the previous layer. The first layer outputs the data of the digits after processing and the last layer's output indicates a confidence that the given image corresponds to each digit 0 through 9. There are a total of 40 biases and 23,820 weights in this network that can be adjusted. Each weight is initialized to a random number between -0.01 and 0.01. Each bias is initialized to 0.

5.3 Training

A single training iteration exposes the neural network to the entire training dataset, and this consists of forward propagation and backward propagation. Training is done through a process called gradient descent. During forward propagation, each digit input vector in the training dataset is sent through the neural network and predictions are made. During backward propagation, the weights and biases of the neural network adjust to minimize error.

Forward Propagation:

During forward propagation, each of the digit input vectors is sent through the neural network. First, each neuron receives a weighted sum of the output from the previous layer. Then, each neuron activates depending on its input; the sigmoid function outputs a value between 0 and 1. These operations are performed with matrices.

$$\begin{aligned}
a^{(0)} &= \text{Input Vector to Neural Network} \\
z^{(1)} &= a^{(0)} \bullet w^{(1)} + b^{(1)} \\
a^{(1)} &= \text{sigmoid}(z^{(1)}) \\
z^{(2)} &= a^{(1)} \bullet w^{(2)} + b^{(2)} \\
a^{(2)} &= \text{sigmoid}(z^{(2)})
\end{aligned}$$

$$\begin{bmatrix} a_0{}^{(1)} \\ a_1{}^{(1)} \\ \dots \\ a_{783}{}^{(1)} \end{bmatrix} = \begin{matrix} s \\ i \\ g \\ m \\ o \\ i \\ d \end{matrix} \left(\begin{bmatrix} w_{(0,0)} & w_{(0,1)} & \dots & w_{(0,783)} \\ w_{(1,0)} & w_{(0,1)} & \dots & w_{(1,783)} \\ \dots & \dots & \dots & \dots \\ w_{(269,0)} & w_{(269,1)} & \dots & w_{(269,783)} \end{bmatrix} \begin{bmatrix} a_0{}^{(0)} \\ a_1{}^{(0)} \\ \dots \\ a_{783}{}^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \dots \\ b_{30} \end{bmatrix} \right)$$

Backward Propagation:

The goal of backward propagation is to minimize the error function. The error function used is cross-entropy cost [4]:

$$\text{Cost} = -\frac{1}{M}\sum_{i=1}^{M}\left(y_i^T \cdot \log(a_i^{(2)}) + (1-y_i)^T \cdot \log(1-a_i^{(2)})\right)$$

M is the number of training examples
y is a 10 by 1 target matrix that corresponds to the label of a digit (the expectation as to what the neural network should output)
$a^{(2)}$ is a 10 by 1 matrix that contains the activations of the output layer (confidences of a digit)

To minimize the cost function, the algorithm calculates the derivatives of the cost with respect to each adjustable parameter in the model: w_2, b_2, w_1, b_1 using chain rule.

Ex:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

Then, those parameters are adjusted in the direction to minimize the cost.

Ex:

$$w_2 = w_2 - \alpha \cdot \frac{\partial L}{\partial w_2}$$

w_2 contains adjustable parameters in the model.
a is the learning rate and determines the leap size during gradient descent.
$\frac{\partial L}{\partial w_2}$ is the derivative of the cost function with respect to w_2

6. Performance Analysis and Optimization

6.1 Accuracy and Error Graphs

To evaluate the performance, the accuracy on the training set, the accuracy on the testing dataset, and the cost were graphed as a function of training iterations.

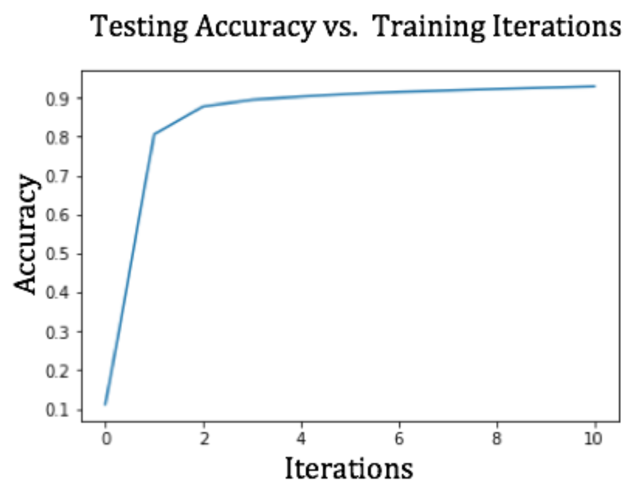


Figure 4: Plot of testing accuracy after 10 training iterations

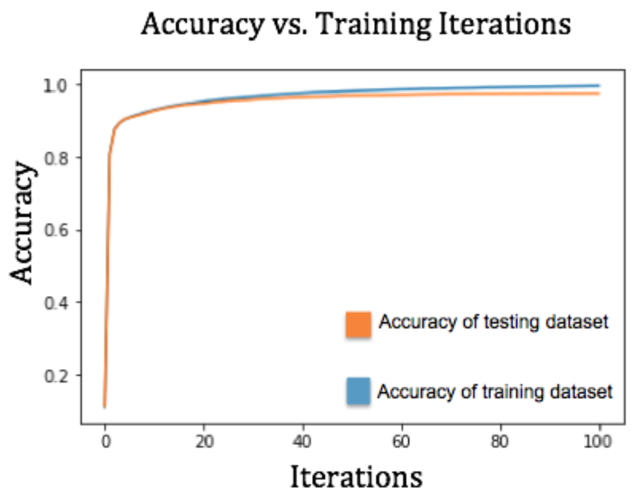


Figure 5: Plot of testing and training accuracy after 100 training iterations

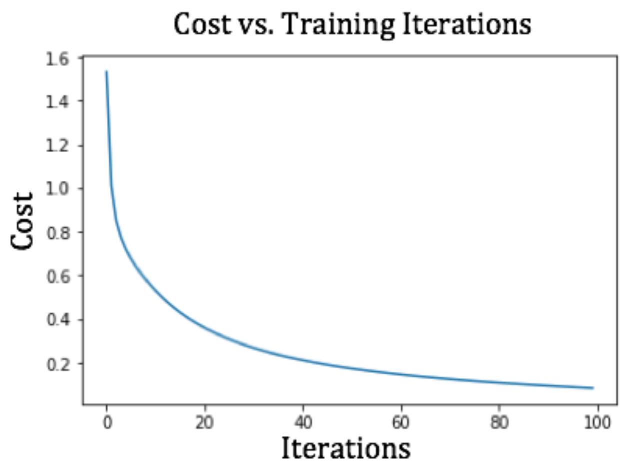


Figure 6: Plot of cost (error) per digit after 100 training iterations

The shape of the graph in Figure 6 decreases quickly during the initial iterations and gradually flattens out. The shape of the graphs in Figures 4 and 5 increase quickly during the initial iterations and gradually flatten out. As the parameters update to reach a minimum of error, the derivative of the cost function with respect to each parameter was expected to be, on average, flatter. Thus, the neural network was expected to learn at a gradually decreasing rate. These results demonstrate the neural network is effectively being trained on the dataset.

6.2 Optimization

To optimize the neural network, the number of hidden neurons was varied in the neural network.

Hidden Neurons	1	10	30	90	270	810
Accuracy	11.3%	90.5%	95.6%	97.1%	97.4%	97.3%

Figure 7: Accuracy on test dataset after 100 training iterations for varying number of hidden neurons.

Method	Test Accuracy
Mine: 2 layers, 270 hidden neurons	97.4%
Linear classifier (1-layer NN)	88.0%
K-nearest-neighbors, Euclidean (L2)	95.0%
3-layer NN, 300+100 hidden neurons	97.0%
Committee of 35 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	99.8%

Figure 8: Comparison to other published methods [1]

The best performing structure was the neural network with 270 hidden neurons. After 100 training iterations, it correctly classified 97.4% of the test dataset. A possible explanation for why the accuracy of the neural network structure with 810 hidden neurons was lower than the that of 270 hidden neurons could be due to overfitting. Overfitting occurs when a model learns the patterns and noise in training data to the extent that it negatively impacts accuracy on data it has never seen.

6.3 Applying the Model to the Real World

To test the model on my own handwritten digits, the following steps were conducted. First, a digit was handwritten on a piece of paper using a pen. Second, a PNG image file of the digit was scanned by creating a signature in the desktop application Preview. Third, code was written to load the digit into the program and process from an image file to an input vector. Fourth, the input vector was sent through the neural network (forward propagation) and it produced a prediction. The figures below display the handwritten digit after scanning and the neural network's confidences given that digit.

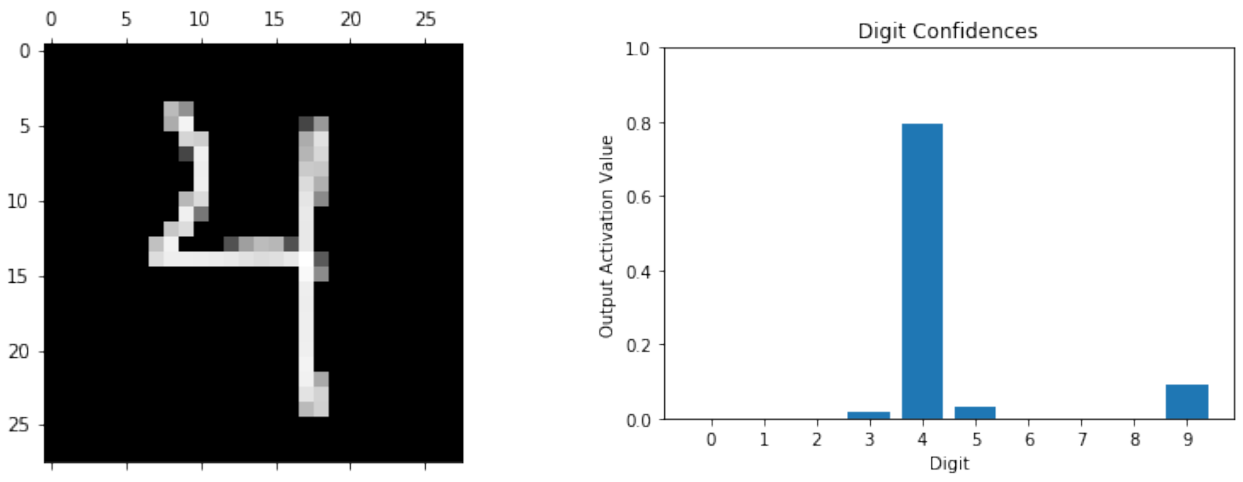


Figure 9: Scanned handwritten digit and output layer activation (digit confidences) bar plot from model

7. Conclusion

During this project, a neural network was implemented that could accurately classify digits from the MNIST dataset. This algorithm was programmed in Anaconda Python without any neural network libraries. Future projects could possibly improve the accuracy of this neural network by varying the number of hidden layers or by implementing more advanced techniques (regularization, deep learning). In addition, future projects could train the model using different or more complex datasets containing letters, symbols, shapes, etc. Even further, applications of this technique include reading addresses on postage, processing images of checks, and autonomous vehicles.

8. Acknowledgements

I appreciate the Los Alamos Mesa Public Library for providing quiet space with fast internet. I thank my mentor, Jacob Stinnett, for offering great guidance, helping me debug code (even late at night), and teaching me calculus concepts that I needed to understand for my project.

9. References

[1] Y. LeCun, C. Cortes, and C. Burges, “MNIST Handwritten Digit Database,” [Online]. Available at: http://yann.lecun.com/exdb/mnist/. [Accessed 14 Jan. 2018].

[2] “Digit Recognizer,” Kaggle, 2013. [Online]. Available at: https://www.kaggle.com/c/digit-recognizer. [Accessed 14 Jan. 2018].

[3] XeronStack, “Overview of Artifical Neural Networks and its Applications,” Medium, July 16, 2017. [Online]. Available at: https://medium.com/@xenonstack/overview-of-artificial-neural-networks-and-its-applications-2525c1addff7. [Accessed 14 Jan. 2018].

[4] Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015

10. Source Code

Code was written in Anaconda Python using Jupyter Notebook.

In [1]:

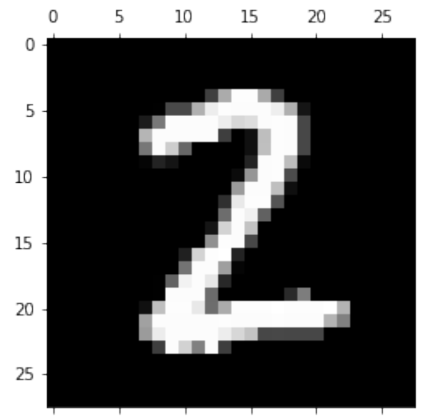
```
#Load the csv file into a numpy data matrix
import numpy as np
import matplotlib.pyplot as plt
import csv

data = [] # Create empty data matrix

#Load csv file into data ---
with open('digitData.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    for row in readCSV:
        if len(row) != 0:
            data = data + [row]
#---

data =  np.asarray(data) #Convert the regular array to numpy array
print(data)
print("done")
```

```
[['label' 'pixel0' 'pixel1' ... 'pixel781' 'pixel782' 'pixel783']
 ['1' '0' '0' ... '0' '0' '0']
 ['0' '0' '0' ... '0' '0' '0']
 ...
 ['7' '0' '0' ... '0' '0' '0']
 ['6' '0' '0' ... '0' '0' '0']
 ['9' '0' '0' ... '0' '0' '0']]
done
```

In [2]:

```
#Visualize original digits
displayPixelValues = data[23][1:]  #Gather all pixel values from first dat
a point

displayPixelValues = np.reshape(displayPixelValues, (-1, 28)) #Reshape dat
a to 28 by 28 array instead of 784 by 1 array
displayPixelValues = displayPixelValues.astype(np.int) #Convert the data i
nto integers from strings
#print(displayPixelValues) #Print out array of single digit
```

```
plt.matshow(displayPixelValues, fignum=10,cmap=plt.cm.gray) #Make grayscal
e representation of digit
plt.show() #Show grayscale representation of digit
```



In [3]:

```
#Sigmoid function
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

In [4]:

```
np.random.seed(2)

layer0_neurons = 784 #Number of pixels = 784
layer1_neurons = 270 #Number of middle layer neurons. I picked 30 for now
layer2_neurons = 10 #Output neurons here.
                    #The activation of the output neurons are the "guesses
"
#Initialize random weights and biases

layer1_b = np.zeros((layer1_neurons,1))#Initialize layer_1 biases to 0. Th
is is a 270 by 1 matrix
layer1_w = (2*np.random.random((layer1_neurons,layer0_neurons)) - 1) / 100
#Initialize layer_1 weights to a value between -.01 and .01 This is a 270
by 784 matrix
layer2_b = np.zeros((layer2_neurons,1)) #Initialize layer_2 biases to 0. T
his is a 10 by 1 matrix
layer2_w = (2*np.random.random((layer2_neurons,layer1_neurons)) - 1) / 100
#Initialize layer_2 weights to to a value between -.01 and .01 This is a 1
0 by 270 matrix
```

In [5]:

```
#Function evaluates accuracy given predictions, targets, weights, and bias
es
def evaluateNetworkAccuracy(x,y,w1,b1,w2,b2):
    z_1 = np.dot(w1,x)+b1
    a_1 = sigmoid(z_1)
    z_2 = np.dot(w2,a_1)+b2
    a_2 = sigmoid(z_2)

    predictions = np.argmax(a_2, axis = 0)
    #print(predictions)

    return(np.sum(np.equal(y,predictions))/len(y))
```

In [6]:

```
#Separation of training (Xdata) and testing (Xtest) digits

trainingSize = 30000 #Number of training sample
Xdata = data[1:trainingSize+1,1:] #Get pixel values of data samples
Xdata = Xdata.T #Transpose - Make columns rows and rows columns
Xdata = Xdata.astype(np.int) #Convert values from string to integer
Xdata = Xdata/255 #Remap grayscale values (normalize)

Xtest = data[trainingSize+1:,1:]
Xtest = Xtest.T #Transpose - Make columns rows and rows columns
Xtest = Xtest.astype(np.int) #Convert values from string to integer
Xtest = Xtest/255 #Remap grayscale values (normalize)

labels = data[1:,0].astype(np.int)  #Get labels of each digit an numpy lis
t

Xdata.shape
```

Out[6]:

```
(784, 30000)
```

In [7]:

```
#Cost function; If labels highly differ from targets, cost is high
def cross_entropy_cost(y, a):
    error = np.multiply(y,np.log(a)) + np.multiply((1-y),np.log((1-a)))
    errorSum = np.sum(error)
    meanErrorSum = -errorSum/(y.shape[1])
    return meanErrorSum
```

In [7]:

```
#Get target matricies from labels
nData = len(data) - 1
target = np.zeros([10,nData])
for i in range(nData):
    target[labels[i],i] = 1
```

```
trainingTarget = target[:,:trainingSize]
testTarget = target[:,trainingSize:]
```

In [8]:

```
accList=[] #List keeps track of training accuracy
testAccList = [] #List keeps track of test accuracy
def trackAccuracyAndCost():
    cost = cross_entropy_cost(batchTarget,a_2)
    costList.append(cost)
    #print(cost)
    acc = evaluateNetworkAccuracy(Xdata,labels[0:trainingSize],layer1_w,la
yer1_b,layer2_w,layer2_b)
    accList.append(acc)


    testAcc = evaluateNetworkAccuracy(Xtest,labels[trainingSize:],layer1_w
,layer1_b,layer2_w,layer2_b)
    testAccList.append(testAcc)
```

In [11]:

```
accList=[] #List keeps track of training accuracy
testAccList = [] #List keeps track of test accuracy
costList = [] #List keeps track of cost


#Initialize random weights and biases
layer1_b = np.zeros((layer1_neurons,1))#Initialize layer_1 biases to 0. Th
is is a 784 by 1 matrix
layer1_w = (2*np.random.random((layer1_neurons,layer0_neurons)) - 1) /100
#Initialize layer_1 weights to a value between -.01 and .01 This is a 30 b
y 784 matrix
layer2_b = np.zeros((layer2_neurons,1)) #Initialize layer_2 biases to 0. T
his is a 30 by 1 matrix
layer2_w = (2*np.random.random((layer2_neurons,layer1_neurons)) - 1) /100
#Initialize layer_2 weights to to a value between -.01 and .01 This is a 1
0 by 30 matrix


iterations = 9 #Number of times NN is exposed to entire dataset
chunkSize = 256 #Number in digits in a batch
learningRate = 0.00075


#Calculate Accuracy before training (Iteration 0)
acc = evaluateNetworkAccuracy(Xdata,labels[0:trainingSize],layer1_w,layer1
_b,layer2_w,layer2_b)
accList.append(acc)
testAcc = evaluateNetworkAccuracy(Xtest,labels[trainingSize:],layer1_w,lay
er1_b,layer2_w,layer2_b)
testAccList.append(testAcc)
```

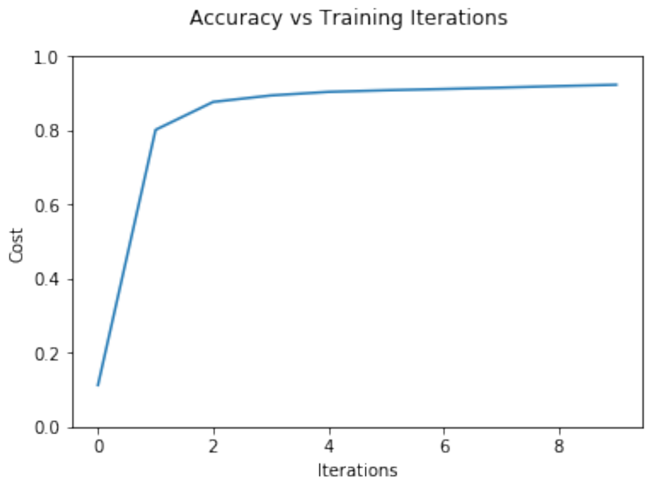
```
print('Iter 0','Training Accuracy: ',acc,'Testing Accuracy: ', testAcc)

for i in range(1, iterations+1):
    chunk = chunkSize
    while(chunk <= trainingSize):
        batch       = Xdata[:,chunk - chunkSize: chunk] #Get batch of trai
ning data
        batchTarget = target[:,chunk - chunkSize: chunk] #Get targets of b
atch




        #Forward Propagation
        z_1 = np.dot(layer1_w,batch)+layer1_b
        a_1 = sigmoid(z_1)
        z_2 = np.dot(layer2_w,a_1)+layer2_b
        a_2 = sigmoid(z_2)

        #Back Propagation: Derivative Calculations
        der_a2 = -np.divide(batchTarget,a_2) + np.divide(1-batchTarget, 1-
a_2)
        der_z2 = der_a2*a_2*(1-a_2)
        der_w2 = np.dot(der_z2,a_1.T)
        der_b2 = np.sum(der_z2,axis = 1,keepdims=True)/trainingSize
        der_a1 = np.dot(layer2_w.T,der_z2)
        der_z1 = der_a1*a_1*(1-a_1)
        der_w1 = np.dot(der_z1,batch.T)
        der_b1 = np.sum(der_z1,axis = 1,keepdims=True)/trainingSize

        #Back Propagation: Parameter Update
        layer2_w = layer2_w - learningRate*der_w2
        layer2_b = layer2_b - learningRate*der_b2
        layer1_w = layer1_w - learningRate*der_w1
        layer1_b = layer1_b - learningRate*der_b1


        chunk+=100
    if(i % 1 == 0):
        cost = cross_entropy_cost(batchTarget,a_2)
        acc = evaluateNetworkAccuracy(Xdata,labels[0:trainingSize],layer1_
w,layer1_b,layer2_w,layer2_b)
```

```
        testAcc = evaluateNetworkAccuracy(Xtest,labels[trainingSize:],laye
r1_w,layer1_b,layer2_w,layer2_b)
        trackAccuracyAndCost()
        print('Iter',i,'Training Accuracy: ', acc,'Testing Accuracy: ', te
stAcc)
```

```
Iter 0 Training Accuracy:  0.11106666666666666 Testing Accuracy:  0.112666
66666666666
Iter 1 Training Accuracy:  0.8087333333333333 Testing Accuracy:  0.8061666
666666667
Iter 2 Training Accuracy:  0.8776333333333334 Testing Accuracy:  0.8761666
666666666
Iter 3 Training Accuracy:  0.8946666666666667 Testing Accuracy:  0.8946666
666666667
Iter 4 Training Accuracy:  0.9033666666666667 Testing Accuracy:  0.90425
Iter 5 Training Accuracy:  0.9096666666666666 Testing Accuracy:  0.90875
Iter 6 Training Accuracy:  0.9145 Testing Accuracy:  0.9120833333333334
Iter 7 Training Accuracy:  0.9185666666666666 Testing Accuracy:  0.9163333
333333333
Iter 8 Training Accuracy:  0.9227666666666666 Testing Accuracy:  0.9196666
666666666
Iter 9 Training Accuracy:  0.9261 Testing Accuracy:  0.9245
```

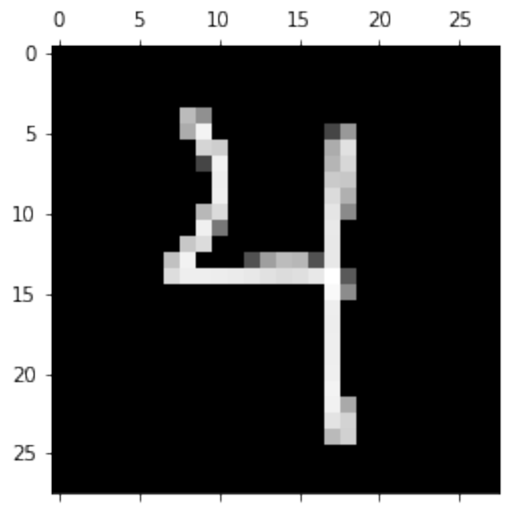
In [11]:

```
#Display Accuracy Plot
plt.plot(np.arange(0,10,1), testAccList)
plt.suptitle('Accuracy vs Training Iterations')
plt.ylabel('Cost')
plt.xlabel('Iterations')
plt.ylim((0,1))
plt.show()
```



In [68]:

```
from PIL import Image
im = Image.open("userImage11.png")

#im.show()
digitArray = np.asarray(list(im.getdata()))
userDigit = []
for i in range(len(digitArray)):
    pixel = digitArray[i]
    #print(pixel)
    pixelBrightness = pixel[0]*0.299 + pixel[1]*0.587 + pixel[2]*0.114
    pixelBrightness = (-(pixelBrightness) + 255)
    userDigit.append(pixelBrightness)
maxVal = np.max(userDigit)
userDigit = userDigit/maxVal
userDigit = np.reshape(userDigit, (-1, 1))
userDigit = np.power(userDigit, .25)
displayUserDigit = np.reshape(userDigit, (-1, 28)) #Reshape data to 28 by
28 array instead of 784 by 1 array


plt.matshow(displayUserDigit, fignum=10,cmap=plt.cm.gray) #Make grayscale
representation of digit
plt.show() #Show grayscale representation of digit
```

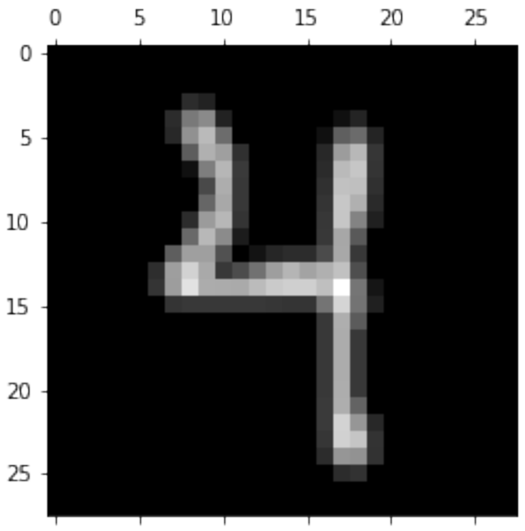


In [69]:

```
def blur(input_image):
    input_shape = input_image.shape
    output_image = np.zeros(input_shape) + input_image
    n_row = input_image.shape[0]
    n_col = input_image.shape[1]
    for i in range(n_row):
        for j in range(n_col):
            pixelVal = input_image[i,j]
            pixel_add = pixelVal * 1

            if i >0:
                pixel_top = (i-1, j)
                output_image[i-1,j] += pixel_add
            if i < n_row-1:
                pixel_bot = (i+1, j)
                output_image[i+1,j] += pixel_add
            if j > 0 :
                pixel_left = (i, j-1)
                output_image[i,j-1] += pixel_add
            if j < n_col -1:
                pixel_right = (i,j+1)
                output_image[i,j+1] += pixel_add
    return output_image / np.max(output_image)
```

In [70]:

```
processedImage = blur(displayUserDigit)
#processedImage = blur(processedImage) #blur again if wanted
plt.matshow(processedImage, cmap=plt.cm.gray) #Make grayscale representati
on of digit
plt.show() #Show grayscale representation of digit
testImage = np.reshape(processedImage, ( 784,1))
testImage = np.power(testImage, .75)
```

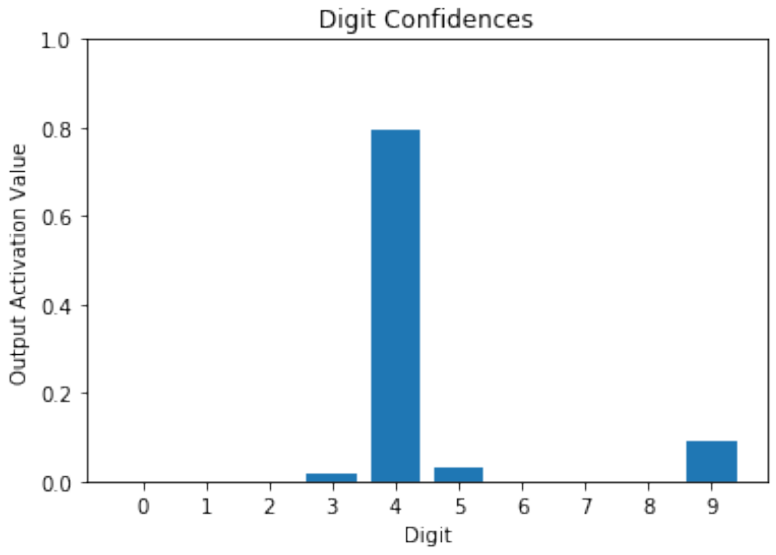


In [71]:

```
#Forward propagation of user's digit
z_1 = np.dot(layer1_w,testImage)+layer1_b
a_1 = sigmoid(z_1)
z_2 = np.dot(layer2_w,a_1)+layer2_b
a_2 = sigmoid(z_2)
```

In [72]:

```
#Display confidences (aka guesses) for the given digit by the user
x = ['0','1','2','3','4','5','6','7','8','9']
plt.bar(x,a_2[:,0])
plt.title('Digit Confidences')
plt.ylabel('Output Activation Value')
plt.xlabel('Digit')
plt.ylim([0,1])
plt.show()
```


Digit Confidences
Output Activation Value
Digit