CSC411: Project 2

Due on Sunday, March 5, 2017

Zheng Yu Chen, Michael Pham-Hung

October 14, 2017

Part 1

The MNIST data set consists of handwritten digits from 0-9. The raw input image arrays have a single channel and each pixel has an intensity on the range of 0-255 where 0 = black and 255 = white.

In Figure 1, ten images of each digit is represented. Between images of the same digit, there are multiple variations in the way it was written. Most obviously, the digits are written with different **thicknesses** and **orientations**. The digit 1 represents this variation well. Another variation in the data set is the **"font" style** of the written digit and whether or not the **edges** are curved or straight. This can be described as the different sense of style when writing. The digit 2 has some examples of the different ways a 2 can be written (note that one of them looks like a z). In addition, some digits contain extra strokes that add to the style of the writing. For example, the digit 7 can be drawn with and without a horizontal stroke through the center.

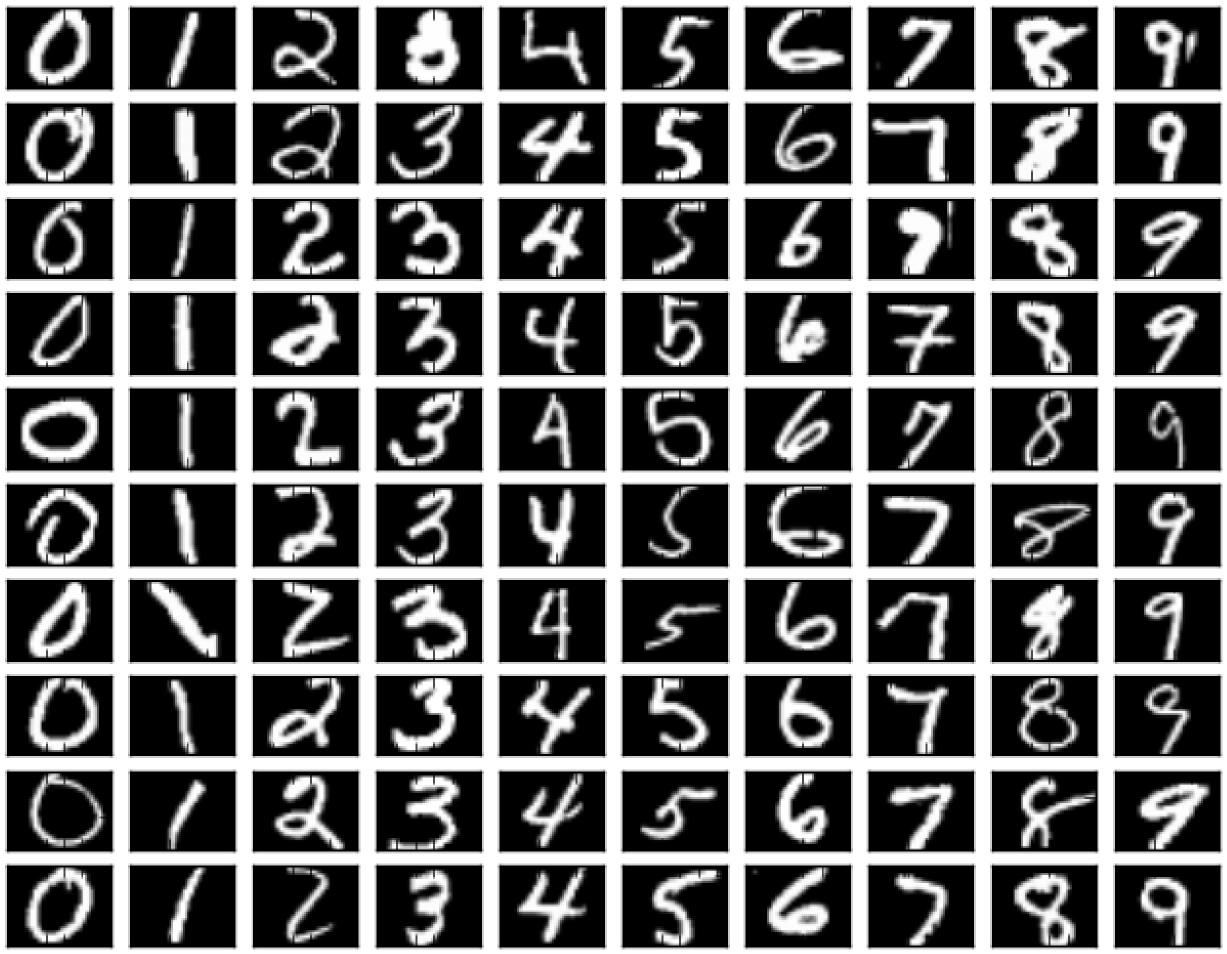

Figure 1: A sample of the digits provided in the MNIST data set

Part 2

To implement the network in Figure 2, which consists of one fully-connected layer with a softmax output, two approaches were possible. First, we could implement as specified using for-loops and assigning each output neuron o_i as a summation $\sum_j w_{ij}x_j + b_j$. However, we can recognize that the output vector o can be simplified into a dot product, which greatly simplifies and optimizes the code.

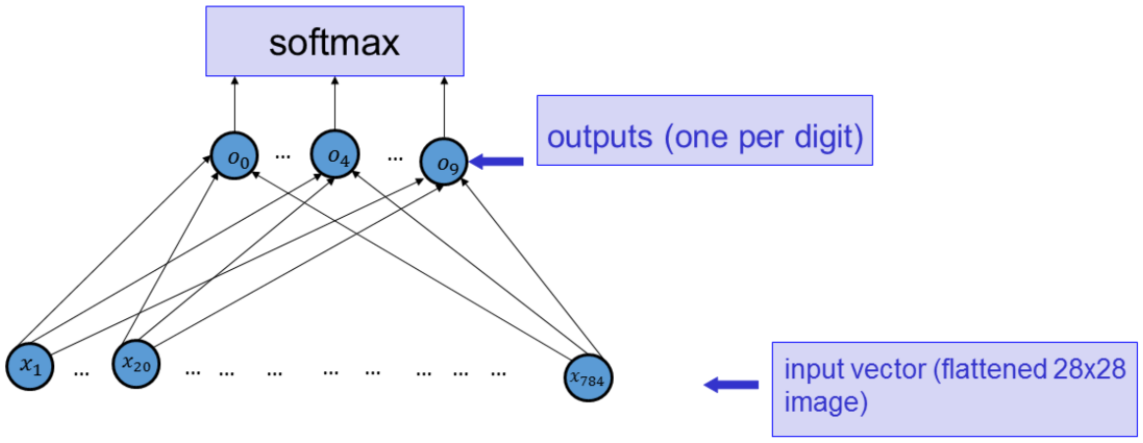

Figure 2: The computational graph of the network

To compute the output, we can compute each layer, starting from the input layer, progressively. The first hidden layer can simply be computed with a dot product of the input and weight matrix. Equation 1 shows how each element of the soft max is calculated. Listing 1 shows the code for implementing the forward pass of the network.

$$p_i = \frac{e^{o_i}}{\sum_j e^{o_j}} \tag{1}$$

Listing 1: Code for Part 2

```
def softmax(y):
    return np.exp(y)/np.tile(np.sum(np.exp(y),0), (y.shape[0],1))

def part2(X, W):
    #Input X is a flattened 28x28 vector representing the image
    #Input W is a 10,785 matrix representing the weight matrix and bias term
    #Ouput the softmax of the nine possible digits
    X = np.vstack( (ones((1, X.shape[1])), X)) #Dummy 1 for bias
    L0 = np.dot(W, X)
    output = softmax(L0)
```

Part 3

We would like to use the sum of the negative log-probabilities of all the training cases as the cost function:

$$C = \sum_k (-\sum_i y_i^{(k)} log(p_i^{(k)})) \tag{2}$$

Where k is the number of training cases. To compute the gradient we can use the linear property of derivatives and the chain rule on each layer. Note that probability p_i is obtained from Equation 1. For multiple training cases, the output vector $o = \underline{\boldsymbol{w}}^T \underline{\boldsymbol{X}}$ then:

$$\begin{aligned} \frac{\delta o}{\delta w_i j} &= \frac{\delta o_i}{\delta w_i j} \\ &= \frac{\delta}{\delta w_{ij}}(\sum_k \sum_j w_{ij} x_j^{(k)}) \\ &= \sum_k x_j^{(k)} \end{aligned} \tag{3}$$

Applying chain rule, we can get the full derivation:

$$\begin{aligned} \frac{\delta C}{\delta w_{ij}} &= \frac{\delta C}{\delta p_i} \frac{\delta p_i}{\delta o_i} \frac{\delta o_i}{\delta w_{ij}} \\ &= \sum_k (p_i - y_i) x_j^{(k)} \end{aligned} \tag{4}$$

Where $\frac{\delta C_k}{\delta p_i} \frac{\delta p_i}{\delta o_i}$ was derived in lecture 7.

Vectorizing this expression, we first start with an expression for the j-th column of the gradient:

$$\frac{\delta C}{\delta \boldsymbol{w_j}} = (\boldsymbol{p} - \boldsymbol{y}) \boldsymbol{x}_j \tag{5}$$

where $\boldsymbol{x_j}$ is the j-th row of the training data input matrix. Finally, we arrive at the result for fully vectorized gradient.

$$\frac{\delta C}{\delta \boldsymbol{w}} = (\boldsymbol{p} - \boldsymbol{y}) \boldsymbol{X}^T \tag{6}$$

Listing 2 is the a snippet of the vectorized cost function and its gradient with respect to the weights as well as a gradient checking function using finite differences.(i.e. $\delta C / \delta \boldsymbol{w} \approx (f(x+h) - f(x-h))/2h$). Note that in the function, a summation of the elements are returned.

Running the gradient check function with a training set size of around 10 and stepsize of 0.001, we get the following outputs:

```
Summation from finite differences:  -24495.3546264
    Summation form direct gradient:  -24507.7
```

Which is a reasonable result given the input data set and h is relatively large.

Listing 2: Code for gradient checking

```
def part3(x, y, init_t):
    grad_check(x,y,init_t,f,df)

def f(X, Y, W):
    # returns cost function
    X = np.vstack( (np.ones((1, X.shape[1])), X))
    #Forward Pass
    L0 = np.dot(W, X)
    L1 = softmax(L0)

    return np.sum(-np.dot(Y.T, np.log(L1)))

def df(X, Y, W): #df
    # Computing the Gradient w.r.t. the Negative Log Cost Function
    X = np.vstack( (np.ones((1, X.shape[1])), X))

    #Forward Pass
    L0 = np.dot(W, X)
    L1 = softmax(L0)

    return np.dot((L1-Y),X.T)

def grad_check(X,Y,W,f,df):
    '''Defining Variables'''
    row = np.linspace(1,W.shape[0],W.shape[0]).astype(int)
    col = np.linspace(1,W.shape[1],W.shape[1]).astype(int)

    ''' Testing Finite Differences for different components of Theta'''
    sum0 =0
    sum1=0
    for i in row[:-1]:
        for j in col[:-1]:
            theta0 = np.zeros(W.shape)
            theta1= np.zeros(W.shape)
            theta2 = np.zeros(W.shape)

            h = 0.0001
            theta1[i][j] = h
            theta2[i][j] = -h

            sum0 += (f(x,y,theta1)-f(x,y,theta2))/(2*h)
            sum1+= df(x, y, theta0)[i][j]

    print "Summation from finite differences:", sum0
    print "Summation form direct gradient:", sum1
```

Part 4

Optimization Procedure
In order to train a neural network with such a large dataset (approx. 60000 images total). A few optimization procedures were required to train the network.

Normalization
The raw input image vector had values ranging from 0-255. We normalized the image such that:

$$x_i = (x_i - 127)/255$$

Doing so prevented the neural network from unequally favoring specific weights. Additionally, it allowed us to use a larger step size alpha for each update without diverging to infinity, which helped in tuning the parameter for validation.

Stochastic Gradient Descent/ Batch Training
Instead of running gradient descent over the entire training dataset which is roughly 60000 images, we used mini-batches of the data. Running gradient descent stochastically allowed faster performance and bigger step sizes without diverging. The mini-batches were uniformly sampled from the entire data set each iteration such that overall, we achieved a result equivalent to the expected gradient of the entire set.

Initializing non-zero weight matrix
We initialized the weight matrix to non-zero values on the range of $(-\frac{1}{\sqrt{784}}, \frac{1}{\sqrt{784}})$ as encouraged by Stanford's CS231 notes for neural network optimization (**http://cs231n.github.io/neural-networks-2/**). This ensures that there are no dead neurons at the beginning of the training procedure.

L2 - Regularization
In addition to above mentioned, we tried regularizing the cost function with the L2 regularization such that:

$$\boldsymbol{w} = (1 - \alpha\lambda)\boldsymbol{w} - \alpha\frac{\delta C}{\delta \boldsymbol{w}} \tag{7}$$

where $\boldsymbol{w}$ is the weight matrix. However, we found no considerable effect in performance and left $\lambda = 1$.

Results
After training, the visualization of the weights in Figure 3 show that each weight take the form of their corresponding output. Over such a large data set, the weights show a general shape of each digit, evidence that the model succeeded in not over fit the training set. Figure 4 also supports that the model did not overfit the training set, with a final performance on the test and validation set of 91 percent at the end of training.

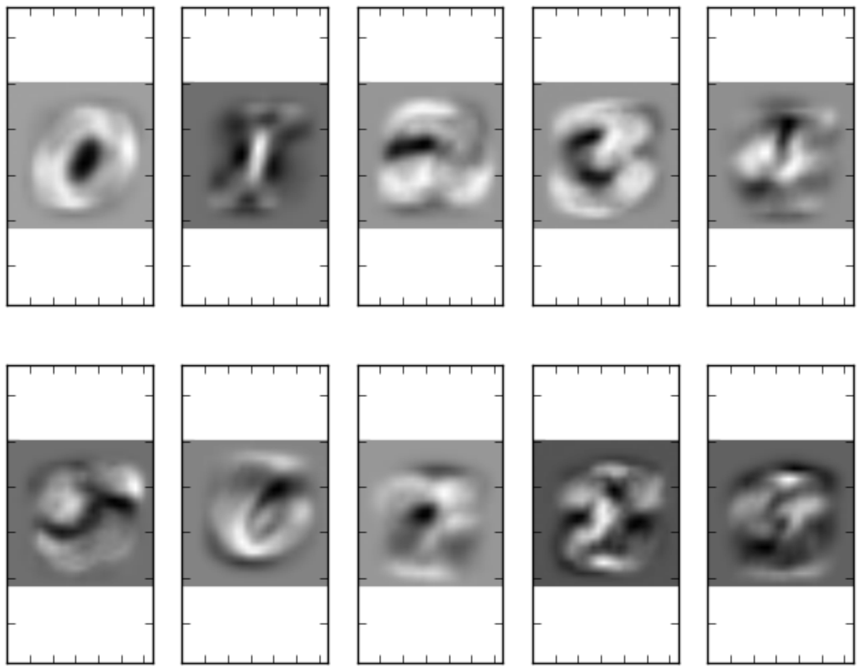

Figure 3: A visualization of the weights after batch-training over the entire dataset.

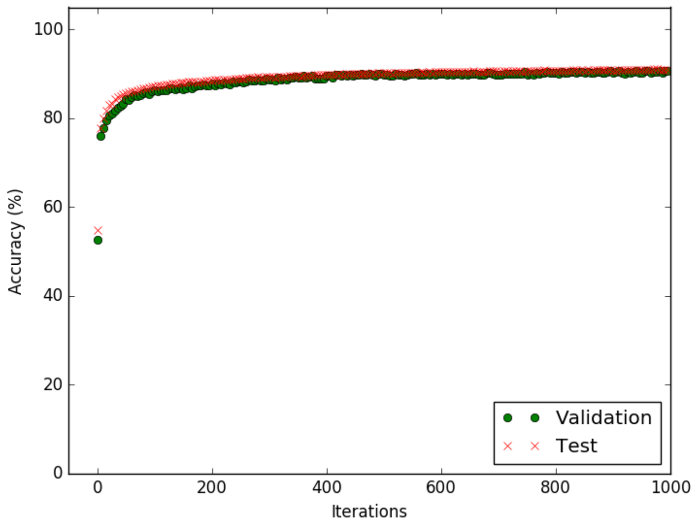

Figure 4: Learning Curve of the model up to 1000 iterations of gradient descent.

Part 5

To demonstrate the advantage of using multinomial logistic regression, we introduced noise in the data set such that the output became very far from the target output. Note that for experimental purposes, we reduced the training set size to 60 images per digit and trained both models with the same training set. To introduce noise into the data set, we simply switched the labels of some images (i.e. an image of 3 would have a target label of 1). Listing 3 shows how the training set was produced within the builddata.py function. Note for this part that 20 percent of the training set included mislabeled images.

For fair comparison, a step size $\alpha = 0.000005$ was chosen. This ensured that the Linear Model actually converged during gradient descent. The same α was used for the Multinomial Log Model in order to compare convergence rate more accurately. Figure 5 shows the performance of each of the models over 1000 iterations. Note that while the linear model rises to the most optimal performance faster than the log model, the final performance of the log model is substantially better.

Looking at the weights of each model, it becomes clear why the log model is substantially better. Figure 6 visualizes the weights that lead to an output of 7 for each model. The log model shows a clear outline of the digit 7 whereas the linear model is mores similar to a noisy image (there is actually a faint 7). The noisy image from the linear model is evidential that the model adjusted the weights heavily from the noisy data.

Listing 3: Snippet of code for building noisy data set

```
for i in range(10):
    for j in range(len(M[cur_train])):
      if j in training_imgs:
        training.append(M[cur_train][j])

        elif j in noise_imgs:
        a = range(10)
        a.remove(i)
        rand_train = "train" + str(np.random.choice(a))
        noisy_img = M[rand_train][j%len(rand_train)]
        training.append(noisy_img)

      elif j in valid_imgs:
        valid.append(M[cur_train][j])

      elif j in test_imgs:
        test.append(M[cur_train][j])
```

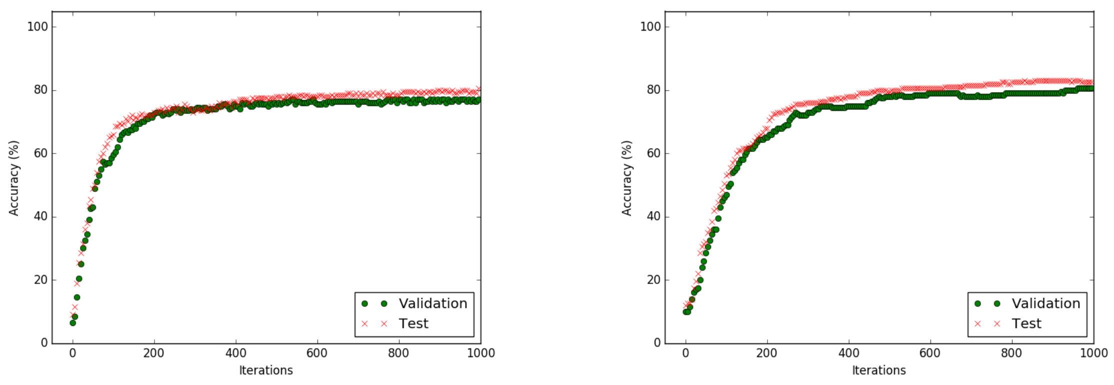

Figure 5: Performance of Linear model is shown on the left and the performance of the Multinomial Log Model is shown on the right

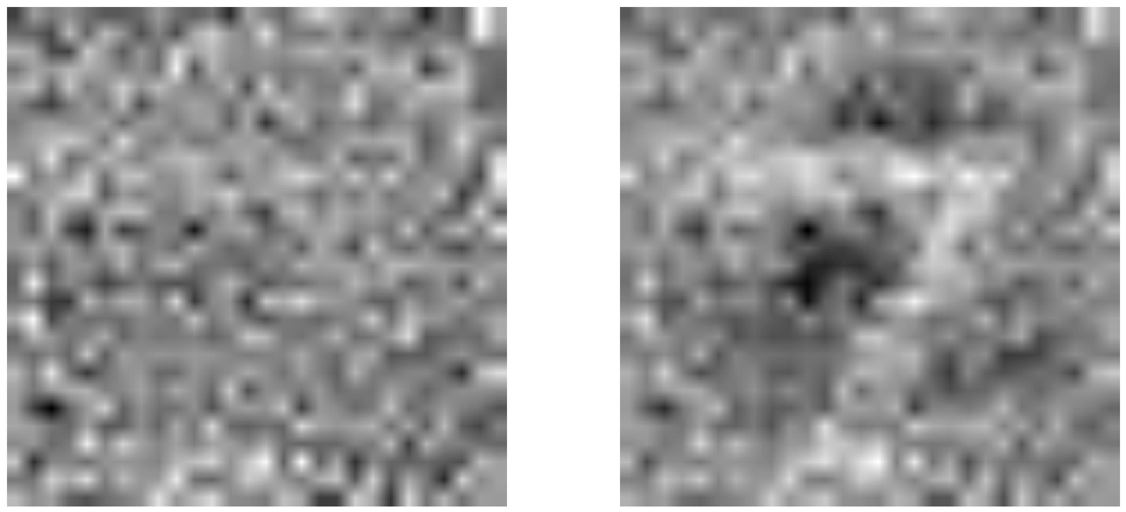

Figure 6: The weights for an output of 7 generated by the Linear model is shown on the left and The weights for an output of 7 generated by the Multinomial Log Model is shown on the right

Part 6

We wish to compare the difference in using fully vectorized back-propagation, as opposed to computing the gradient with respect to each weight individually. Suppose we have a network with N layers and K neurons each and that addition and multiplication operations have a time complexity of $O(1)$. Let us define T_{bad} and T_{back_prop} as the number of operations required for calculating the gradient element-wise and vectorized respectively.
Given N layers and K neurons, calculating the individual derivative with respect to $w_{1,ij}$ we get:

$$\begin{aligned}\frac{\delta C}{\delta w^1_{ij}} &= \frac{\delta C}{\delta L^N}\frac{\delta L^{N-1}}{\delta L_{N-2}}\cdots\frac{\delta L^1}{\delta w^1_{ij}} \\ &= \sum_{l_m=0}^{k}\frac{\delta C}{\delta L^N_{l_m}}(\sum_{l2=0}^{k}\frac{\delta L^N}{\delta L^{N-1}_{lm-1}}\cdots(\frac{\delta L^{(1)}}{\delta w^1_{ij}}))\end{aligned} \tag{8}$$

In otherwords, for every fully connected layer, a nested sum must be computed. For the worst case (computing the first weight gradient), the nested sum is N layers deep and must be computed for every weight $(k \times k)$ resulting in:

$$T_{bad} = K^2 \cdot K(K...(K(1)))) = O(K^{N+2}) \tag{9}$$

Now, assume that a matrix multiplication between two $k \times k$ matrices runs in

$$O(\sum_{i=0}^{k}(\sum_{j'=0}^{k}(\sum_{j=0} A_{ij}B_{j'j})) = k^3$$

where $B_{j'j}$ is the individual element input of the previous layer and j' is the $j'th$ training case. Using backprop we derive the gradient with respect to $\boldsymbol{w}(1)$:

$$\frac{\delta C}{\delta \boldsymbol{w}} = \frac{\delta C}{\delta \boldsymbol{L^n}}\frac{\delta \boldsymbol{L^N}}{\delta \boldsymbol{L^{N-1}}}\cdots\frac{\delta \boldsymbol{L^{(1)}}}{\delta \boldsymbol{w^{(1)}}} \tag{10}$$

In otherwords, this results in N matrix multiplications. Then:

$$T_{backprop} = O(NK^3) \tag{11}$$

As a result, vectorized back-propagation has a better time complexity then computing the gradient element-wise. For further demonstration, we computed the gradient vector from the neural network we trained in Part 4 element-wise and using vectorized back-propagation. Note the substantial difference in terms of computation time for a training set of size 2000.

```
non vectorized gradient time = 0.296000003815
 vectorized gradient time = 0.0469999313354
```

Part 7

In this part, TensorFlow is used to train a fully-connected Neural Network (NN) that has a single hidden layer. This NN was trained to classify the six different actors that were used in Project 1. From Project 1, the same code was used to download the faces of the actors. However, this time we wish to filter out all the bad images using their SHA-256 hashes.

In order to do this, all of the images were downloaded, cropped, and converted to greyscale with their hashes as filenames. Then, they were manually reviewed and the bad images' hashes were recorded in a list. Images were considered bad if they:

- No longer linked to the appropriate image (e.g. replaced with an advertisement or "Image Not Found" placeholder),
- Had a unsatisfactory bounding box which led to a cropped image that did not feature the face, or
- Had a bad contrast (due to lighting conditions) after converting to greyscale. Mostly, these images were very dark and not many features in the face were discernible.

This list of bad hashes can be found in the script that downloads the images, namely `getfaces.py`. After the list of bad hashes were compiled, the images were re-cropped, but this time all the images that were in the bad hashes list were ignored. Again, the code that does this can be found in `getfaces.py`.

The cropped images were then used to build an `M` dictionary similar to the one for the MNIST digits dataset. The code that does this is shown in Listing 4 below. Namely, the function splits the images into a training and testing set, and stores it in a dictionary that is accessible with keys `"testi"` or `"traini"`, where $i = 0, 1, 2, 3, 4, 5$ representing the six actors. This was done so the provided code that builds the training batches and validation/testing arrays can be used, since these functions were implemented for the MNIST dataset. Note that 30 images were used for the test set, as specified. 30 images were used for the validation set as well, and the `get_test()` function was modified to return separate testing and validation sets. Also note that the images selected for the testing and validation sets were randomly selected among all images. Finally, all features (i.e. pixels) were normalized by dividing their values by 255, which is the maximum value a pixel feature can take on.

Listing 4: Code for reading in the images and creating an M dictionary

```python
def get_faces(filenames):
    '''
    Helper function that gets the images with file names from an input list
    'filenames'
    '''
    result = []

    for filename in os.listdir(directory):
        if filename in filenames:
            face = imread(directory+ "/" + filename)
            #flattens the face
            oneD_face = []
            for i in face:
                for j in i:
                    oneD_face.append(j)
            result.append(oneD_face)

    return result

def build_array(names, test_size):
    '''
    Builds a dictionary M that mimics the MNIST digits database. Takes an array
    of actor names and a size for the test+validation sets.
    '''
    print "Building M matrix with testing data of %d random images" % test_size
    M = {}
    for actor in range(len(names)):
        # first, append all the files for the actor to a list
        name = names[actor]
        face_numbers = []
        for filename in os.listdir(directory):
            if filename.startswith(name):
                face_numbers.append(filename)

        # randomly permutes the list of actor's faces
        random_faces = np.random.permutation(face_numbers)

        # splits the permuted list into 2 to use for training and testing
        test_face = random_faces[:test_size]
        training_faces = random_faces[test_size:]

        #mimic the MNIST dataset
        M["train" + str(actor)] = np.array(get_faces(training_faces))
        M["test" + str(actor)] = np.array(get_faces(test_face))

    return M
```

The TensorFlow code provided by the professor was used to train the NN. It was not modified heavily since we elected to build an `M` dictionary that mimics the MNIST dataset, as described above. 300 nodes were used in the hidden layer. The initial weights and biases were generated with a normal distribution of standard deviation 0.01 and mean 0. The learning rate used was 0.0005. A tanh activation function was used. In this part of the project, no regularization was used. The effects of regularization will be studied in Part 8.

These values were determined through trial and error. In other words, the numbers were edited, and the performance on the validation set was tracked. The final values used optimized the performance on the validation set. However, since we are using mini batches for the training set, the performance will vary every time the training takes place.

The learning curve for the training, test, and validation sets are shown below in Figure 7. Notice how the graph jitters due to the effect of training with mini-batches. The final result on the test set, at the last iteration, is:

```
Valid:  80.0000011921
 Test:  83.3333313465
Train:  98.8388955593
   Penalty:  0.0
```

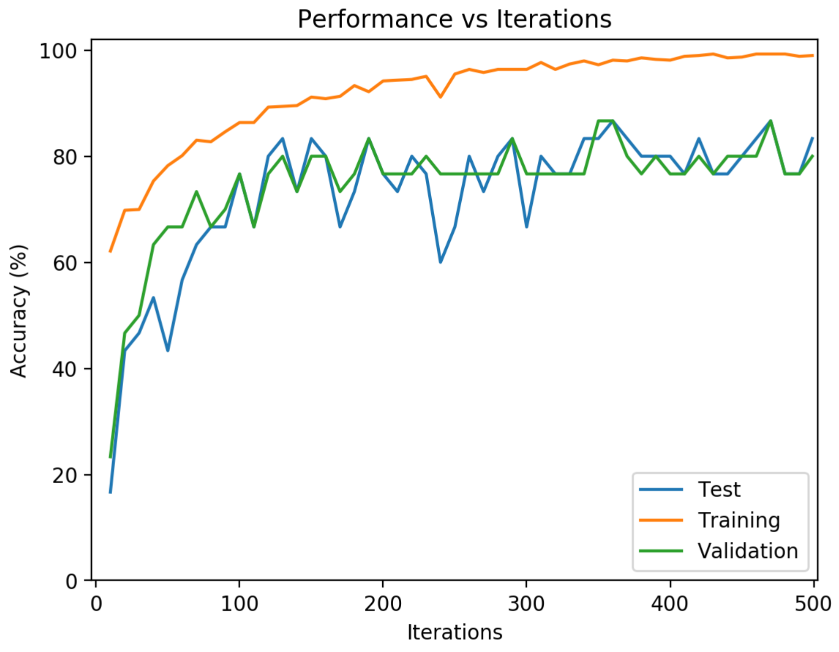

Figure 7: Performance of the NN on the training, validation, and test sets vs the number of iterations or Epochs

Part 8

While regularization was not utilized in Part 7, it will be analyzed here to investigate its uses. Furthermore, it will be utilized in a similar way in Part 9 in order to produce weights that look nicer when visualized. A few reasons why regularization may be useful are discussed below.

Firstly, regularization is useful when the ratio of the training data size to the number of features is small, i.e. when the size of the training set is very small. The reason is because it will become "easy" for the algorithm to fit weights that perfectly describe the training set, since there are so few images. This is particularly true when there are a lot of learning iterations on the same small dataset. An example of this is shown in Figure 8. With 8 data points, it is possible to fit a polynomial of order $8 - 1 = 7$ to perfectly describe the training set. When this happens, **notice how the weights, represented the coefficients of the polynomial, tend to be large**. Regularization will force the weights to be smaller; the more regularization, the more the final fit will resemble the 1st order polynomial fit (i.e. a flat line).

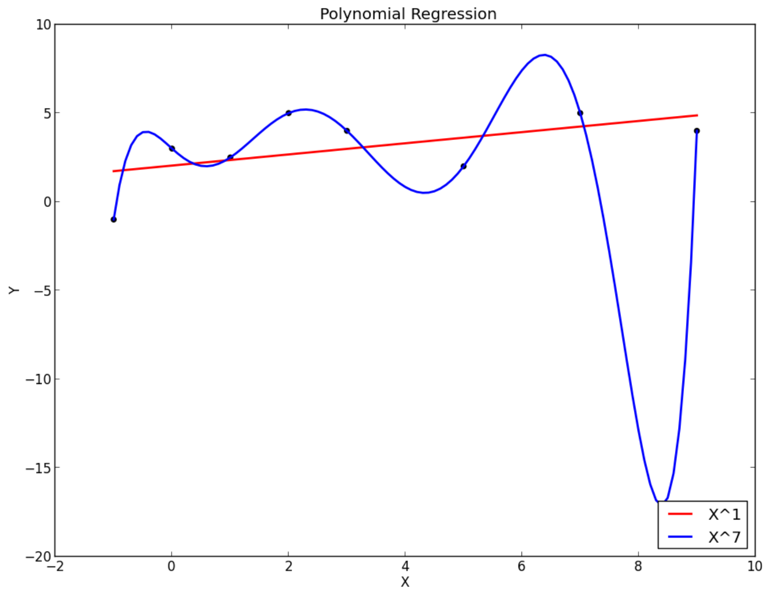

Figure 8: Example of overfitting with 8 points in the training set. It is possible to fit a 7th order polynomial with large weights to perfectly describe the training set.

These two reasons (small training set and large number of iterations) were combined and a specific training set was selected to portray these situations. More specifically, *only two images* of each actor was used per iteration. Then, the number of iterations/epochs was increased to 750.

Finally, a regularization $\lambda = 0.05$ was used. This value was determined through trial-and-error. Specifically, λ was increased and decreased and the performance on the validation set was monitored. The λ selected corresponded with an increase in performance when regularization was utilized. Unfortunately, although in theory regularization should improve the performance when using a small training set, it did not have too much of a measurable impact during our experiments. Regardless, the results will be reported. The code that was run is shown below:

```
part7(act, 2, "part8_noreg.png", 700, 0, train_rate = 0.00005, plt_title =
"Without Regularization")
part7(act, 2, "part8_reg.png", 700, 0.005, train_rate = 0.00005, plt_title =
"With Regularization")
```

The learning curves produced are shown below in Figure 9 below. As previously mentioned, the improvement is not extremely evident. However, working with the last iteration (699), the performance on the test set is 46.67% without regularization, and 50% with regularization. This represents a **5.3%** decrease in error. Additionally, the performance on the training set is slightly worse when using regularization. This is expected because penalizing the weights will make the fit on the training data worse.

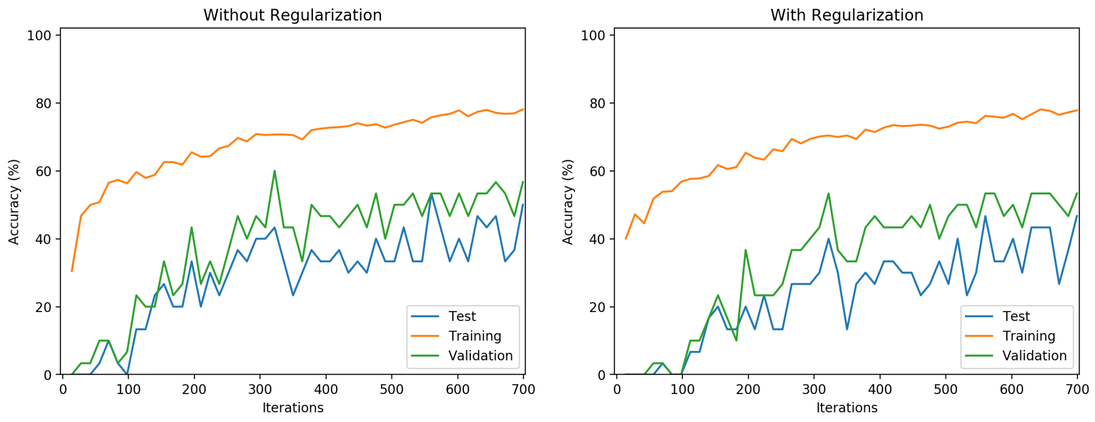

Figure 9: The learning curves for 700 iterations TensorFlow on two images per actor with and without regularization. With regularization is shown on the left, and without is shown on the right.

Part 9

Now, we wish to visualize the weights on the hidden units (AKA neurons) to see which features of the actors' faces are the most important to the NN when it is processing an image. To do this, we first have to figure out the units of the hidden layer that the NN is using. For this part, we will use only 50 units in the hidden layer.

The NN was then trained again on all six actors. More specifically, the following lines were run to produce the results in Part 9:

```
act = ['drescher', 'ferrera', 'chenoweth', 'baldwin', 'hader', 'carell']
     part7(act, 20, None, 100, 0.00005, visualize = True, nhid = 50)
```

50 units were used in the hidden layer as mentioned. 20 images of each actor was used per batch. $\lambda = 0.0005$ was used, as mentioned in Part 8, to regularize in order to make the visualizations nicer. The reason the visualizations are nicer with regularization is because regularization keeps the weights smaller, leading to a smaller variance between the weights.

With the parameter `visualize` set to `True`, the code shown in Listing 5 below was run. First, the NN was trained with the six actors. Baldwin and Ferrera were the two actors whose faces were used to visualize the weights. They are actors 3 and 1 respectively in the `M` dictionary. One image of both actors was selected randomly and stored in `vis_imgs`. The images were then fed into `layer1` of the NN, i.e. the hidden layer with 50 units. The output of this represents the weights on the hidden units. Any unit with a positive weight was useful for the classification. **However, since there were many positive weights, the unit with the highest weight was visualized** (i.e. unit $i = argmax(output)$). This was done by running the weights after training into the first layer, and displaying the weights corresponding with unit i.

Listing 5: Code for visualizing the weights

```
vis_imgs, vis_imgs_y = get_train_batch(M, 1, actors = actors)

if visualize:
    for actor in vis_actors:
        feed_img = vis_imgs[actor].T
        output = sess.run(layer1, feed_dict={x: [feed_img]}).T
        feed_img = feed_img.reshape((32, 32))
        feed_img = imresize(feed_img, (320,320))
        imsave('part9/feed_img_%d.png' % actor, feed_img, cmap=cm.Greys)

        #visualize weight
        i = argmax(output)
        theta = sess.run(W0)
        print 'Saving image %d' % i
        theta_show = np.reshape(theta.T[i], (32,32))

        theta_resized = imresize(theta_show, (320,320))
        imsave('part9/%d.png' % i, theta_resized, cmap=cm.coolwarm)
```

The visualization of the weights and the image fed into the hidden layer were saved in a directory named `part9/`. The results are shown below in Figure 10.

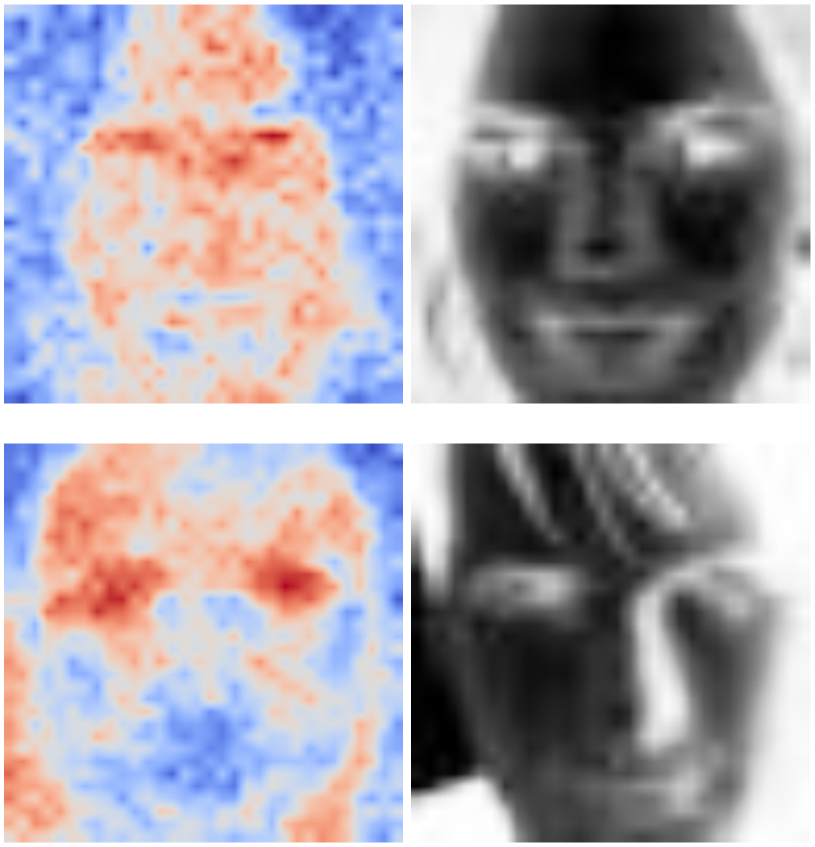

Figure 10: A visualization of the weights that corresponded with the most important unit in the hidden layer for each actor. The visualization is shown on the left, and the image of the actor that was fed into the hidden layer is shown on the right. Ferrera is shown on top, and Baldwin is shown on the bottom.

Part 10

In this part, we will investigate the effects of using features that are not simply the grayscale pixels. To obtain some interesting new features from the images, we will use a few layers of a Convolutional Neural Network (CNN) in AlexNet. We can apply the pictures to this CNN and extract the activation values at a layer. Then, we can feed these activation values into the NN in Part 7 and see if the performance improves.

To do this, the images were first converted to RGB 277x277 images (i.e. (277, 277, 3) matrices). Recall that in Part 7, the images were cropped to 32x32 (using the given bounding boxes), converted to grayscale, and stored in the `cropped` folder. This time, the images are cropped to 277x277, not converted to grayscale, and saved in the `cropped_p10` folder. This was done through a separate script, submitted as `get_data_10.py`. This script is imported in the main Part 10 file.

Afterwards, the images were fed into the CCN in AlexNet. Specifically, the `conv4` layer was used (and therefore the subsequent layers were ignored). An excerpt from the AlexNet code (`deepfaces.py`) is shown below in Listing 6. The images are read in one by one, separated by actors. The images are read in as (277, 277, 3) matrices and normalized. Then, to extract the activation for the images, they are passed into the CNN layer `conv4`.

The result from the `conv4` layer is then flattened to match the format of the features in previous parts of this assignment. Then, the flattened features are stored into a `M` array just in Part 7 (i.e. it mimics the MNIST digits dataset). This is so that we can pass this back to the code in Part 7 easily and train the same NN.

Listing 6: AlexNet code

```
################################################################################
#IMPORTS
################################################################################

# sets directory for cropped images
directory = "cropped_p10"

def get_activations(filenames):

    print "Running AlexNet to get activation values (conv4)"

    train_x = zeros((1, 227,227,3)).astype(float32)
    train_y = zeros((1, 1000))
    xdim = train_x.shape[1:]
    ydim = train_y.shape[1]

    ############################################################################
    #Read Image, and change to BGR

    imgs = []
    for filename in os.listdir(directory):

        if filename in filenames:
            face = (imread(directory+ "/" + filename, mode = "RGB")[:,:,:3])
            face = face.astype(float32)
            face = face - mean(face)
            face[:, :, 0], face[:, :, 2] = face[:, :, 2], face[:, :, 0]
            imgs.append(face)

    ############################################################################
    #ALEXNET INIT AND CONV CODE
    ############################################################################

    #Output:
    activations = np.array(sess.run(conv4, feed_dict = {x:imgs}))
    features = [ i.flatten() for i in activations ]

    return features

def build_array(names, test_size):
    print "Building M matrix with testing data of 30 random images"
    M = {}
    for actor in range(len(names)):
        # first, count the number of faces for the actor
        name = names[actor]
        total_faces = 0
        face_numbers = []
        for filename in os.listdir(directory):
            if filename.startswith(name):
                face_numbers.append(filename)
                total_faces += 1

        # array of random integers to use for training, validating, and testing
        random_faces = np.random.permutation(face_numbers)
        test_face = random_faces[:test_size]
        training_faces = random_faces[test_size:]

        M["train" + str(actor)] = np.array(get_activations(training_faces))
        M["test" + str(actor)] = np.array(get_activations(test_face))

    return M
```

In Listing 6, the imports and the code that initializes the layers for AlexNet is not shown, but can be of course found in the submitted file. This code that is not shown is largely untouched from the provided AlexNet file. Finally, the code to reproduce the results of Part 10 can be found in the TensorFlow file that was used for Parts 7, 8, and 9. The file imports the AlexNet code in Listing 6 and the results of Part 10 can be reproduced by running the `part10()` function.

The NN had a single hidden layer with 300 units. The weights and biases are randomly generated with a normal distribution $N(0, 0.01^2)$. A tanh activation function was utilized. No regularization was used, and the learning rate was 0.0005. These parameters were selected on a trial-and-error basis. In other words, the parameters were edited and the performance on the validation set was tracked. The final values produced the best results on the validation set.

Finally, the learning curve produced by TensorFlow is shown below in Figure 11. Notice how the performance on all data sets is drastically improved over Part 7. In fact, the performance on the all three sets are nearly 100%. Additionally, only 35 iterations/epochs were necessary before the performance plateaued near 100%.

Finally, notice that the lines for validation/training sets seem "choppier". This is because there are only 30 images in the validation/testing sets, so the performance can only take on a few finite values near 100%. The results for the final iteration are shown below as well. The performance on the test set for the last iteration is 96.67% **The performance, compared to Part 7, is a 80% decrease in error rate**.

```
Valid:  96.6666638851
 Test:  93.3333337307
Train:  99.2795407772
   Penalty:  0.0
```

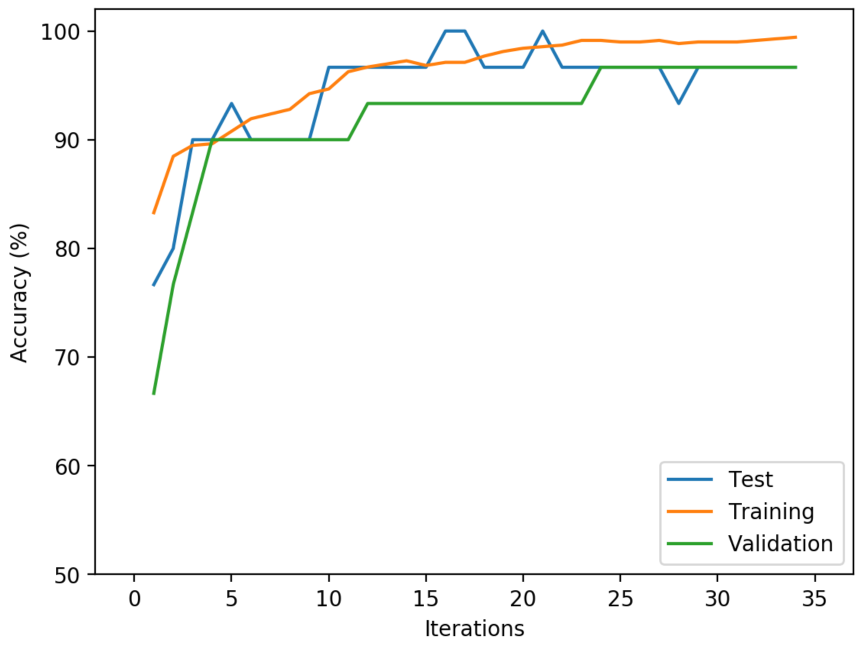

Figure 11: Performance of the NN on the training, validation, and test sets (with AlexNet activation values as features) vs the number of iterations or epochs