Deep Neural Networks for Handwritten Digit and Face Recognition

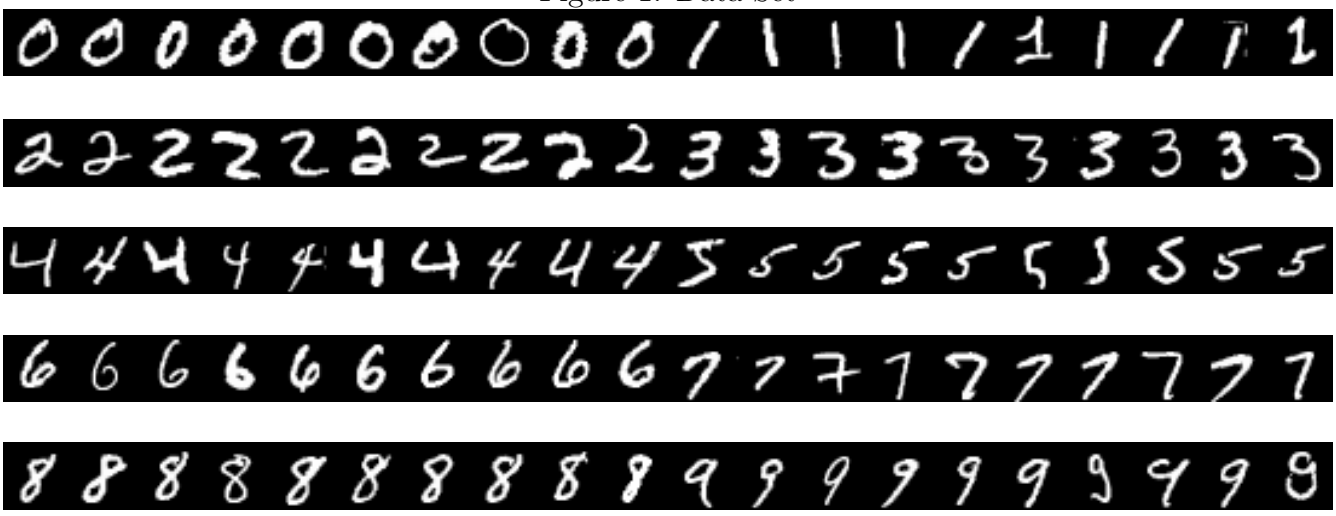
Hao Sheng

March 5^{th} 2017

Part 1

The data set provided for the first part of this assignment was a collection of written digits from zero to nine. In general, a person could identify most of the numbers in the data set. However, the numbers are not always written in the same "style". For example, most of the images of one are written simply as a vertical line, while some others are written with an extra bend and horizontal line. Another example would be some two's written with a loop while others do not, or writing a seven with an extra horizontal dash. This variation in the data set could make it more difficult for the program to correctly identify the digit from the image.

Figure 1: Data Set



Part 2

Below is code used to implement a single layer network with a softmax function output of probabilities. Note that the images are flattened to a column vector of 784 by 1 in the "images" matrix. Moreover, an extra "1" is added to the vector to become a 785 by 1 vector because the bias is included into the weights to allow for calculation of the bias concurrently with the weights.

```
def get_classification(weights, images):
  '''
  Function: generate outputs O_0 to O_9
  Input:
    weights: matrix of weights of size (10 x 785)
    images: matrix of flattened images as column vectors (785 x #images)
  Output:
    classification: classification matrix where the identification for each image
   stored as a column vector (10 x #images)
  '''
  classification = np.dot(weights,images)
  return classification

def softmax(weights, images):
  '''
  Function: generate softmax probabilities
  Input:
    weights: matrix of weights of size (10 x 785)
    images: matrix of flattened images as column vectors (785 x #images)
  Output:
    probability: probability for classifying as each digit for each image is stored
    as a column vector (10 x #images)
  '''
  guess = get_classification(weights, images)
  probability = np.exp(guess)
  probability = probability/probability.sum(axis=0)[None,:]
  return probability
```

Part 3a

To calculate the gradient of the cost function with respect to weight W_{ij}, we know:

$$Cost = \sum_{k=0}^{num\_images-1} (Cost^{(k)})$$

$$Cost^{(k)} = (-\sum_{j=0}^{9} y_j^{(k)} logp_j^{(k)})$$

where,

$$p_i = \frac{e^{o_i}}{\sum_{j=0}^{9} e^{o_j}}$$

o_j is the jth output of the neural net

$y_j^{(k)}$ is the the jth element of the one-hot classification of the kth image

and,

$$\frac{\partial p_i}{\partial o_l} = \frac{\frac{\partial}{\partial o_l}(o_i)(\sum_{j=0}^{9}(e^{o_j})) - (o_i)\frac{\partial}{\partial o_l}(\sum_{j=0}^{9}(e^{o_j}))}{(\sum_{j=0}^{9} e^{o_j})^2}$$

when $i \neq l$

$$\frac{\partial p_i}{\partial o_l} = \frac{-(o_i)(o_l)}{(\sum_{j=0}^{9} e^{o_j})^2} = -p_i p_l$$

when $i = l$

$$\frac{\partial p_i}{\partial o_l} = \frac{(o_i)(\sum_{j=0}^{9}(e^{o_j})) - (o_i)(o_i)}{(\sum_{j=0}^{9} e^{o_j})^2} = (1 - p_i)p_i$$

Moreover,

$$\frac{\partial Cost}{\partial W_{ij}} = \sum_{k=0}^{num\_images-1} (\frac{\partial Cost^{(k)}}{\partial p_m}\frac{\partial p_m}{\partial o_i}\frac{\partial o_i}{\partial W_{ij}})$$

Where,

$$\frac{\partial Cost^{(k)}}{\partial p_m} = \sum_{m=0}^{9}(-\frac{y_m^{(k)}}{p_m^{(k)}})$$

$$\frac{\partial p_m^{(k)}}{\partial o_i^{(k)}} = p_m^{(k)}(1 - p_m^{(k)}) \text{ or } \frac{\partial p_m^{(k)}}{\partial o_i^{(k)}} = \frac{-(o_m^{(k)})(o_i^{(k)})}{(\sum_{j=0}^{9} e^{o_j})^2} = -p_m^{(k)} p_i^{(k)}$$

$$\frac{\partial o_i^{(k)}}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}}(W_i X^{(k)} + b_i) = X_j^{(k)}$$

Thus,

$$\frac{\partial Cost}{\partial W_{ij}} = \sum_{k=0}^{num\_images-1} ((\sum_{m=0, m\neq i}^{9} ((-\frac{y_m^{(k)}}{p_m^{(k)}})(-p_m^{(k)} p_i^{(k)})) - (\frac{y_i^{(k)}}{p_i^{(k)}})(p_i^{(k)}(1 - p_i^{(k)})))X_j^{(k)})$$

$$\frac{\partial Cost}{\partial W_{ij}} = \sum_{k=0}^{num\_images-1} ((\sum_{m=0, m\neq i}^{9} (y_m^{(k)} p_i^{(k)}) - y_i^{(k)}(1 - p_i^{(k)}))X_j^{(k)})$$

$$\frac{\partial Cost}{\partial W_{ij}} = \sum_{k=0}^{num\_images-1} ((\sum_{m=0, m\neq i}^{9} (y_m^{(k)} p_i^{(k)}) - y_i^{(k)} + y_i^{(k)} p_i^{(k)}))X_j^{(k)})$$

$$\frac{\partial Cost}{\partial W_{ij}} = \sum_{k=0}^{num\_images-1} ((\sum_{m=0}^{9} (y_m^{(k)} p_i^{(k)}) - y_i^{(k)})X_j^{(k)})$$

$$\frac{\partial Cost}{\partial W_{ij}} = \sum_{k=0}^{num\_images-1} ((p_i^{(k)} \sum_{m=0}^{9} (y_m^{(k)}) - y_i^{(k)}) X_j^{(k)})$$

Knowing that $y^{(k)}$ is a vector of zeros with only one index having a one,

$$\sum_{m=0}^{9} (y_m^{(k)}) = 1$$

Therefore,

$$\frac{\partial Cost}{\partial W_{ij}} = \sum_{k=0}^{num\_images-1} ((p_i^{(k)} - y_i^{(k)}) X_j^{(k)})$$

Part 3b

Below is the code used to compute the gradient using finite differences:

```
def get_finite_difference(weights, images, classifier,step_size):
  '''
  Function: Returns a gradient vector or matrix of the direction to minimize cost
   function
  Input:
    weights: classifier function (matrix)
    images: image data used to calculate output of classifier function (matrix)
    classifier: desired result from classifier function (matrix)
    step_size: step sized used in approximation of gradient (float)
  Output:
    gradient: a gradient vector or matrix of the direction to minimize cost
   function
  '''
  gradient = np.ones(weights.shape)
  probability = logreg_softmax(weights, images) #calculate softmax
  bwd = logreg_get_cost(probability, classifier) #f(x)
  for i in range(weights.shape[0]):
    for j in range(weights.shape[1]):
      weights[i][j]+= step_size
      probability_stepped = logreg_softmax(weights, images)
      fwd = logreg_get_cost(probability_stepped, classifier) #f(x+h)
      gradient[i][j] = (fwd-bwd)/(step_size) #update slope
      weights[i][j]-= step_size
  return gradient

def unit_vector(vector):
  '''
    Function: Returns the unit vector of the vector
    '''
  return vector / np.linalg.norm(vector)

def angle_between(v1, v2):
  '''
  Function: compresses v1 and v2 into vectors and returns the angle between the
   vectors
  Input:
    v1: first vector or matrix
    v2: second vector or matrix
  Output:
    angle between v1 and v2 in radians
  '''
  v1 = v1.reshape(1, np.size(v1)) #reshape into vector
  v1 = v1[0]
  v2 = v2.reshape(1, np.size(v2)) #reshape into vector
  v2 = v2[0]
  v1_u = unit_vector(v1) #calculate unit vector direction
  v2_u = unit_vector(v2) #calculate unit vector direction
  angle = np.arccos(np.clip(np.dot(v1_u.T, v2_u), -1.0, 1.0))
  if angle > math.pi: angle = angle-math.pi
  return angle
```

Figure 2 below plots the angle difference between the finite difference gradient and the actual gradient used in the program with respect to $J(\theta)$, the point where the gradient is evaluated. The same perturbation of 0.001 was used for all finite difference gradient calculations for consistency. The angle was calculated by resizing each gradient matrix to a vector. Thus the angle between the two compressed matrices was used as a method of comparison of the two methods for acquiring a gradient. Figure 2 in particular demonstrates that the angle between the gradients grows at lower cost values. Thus if the finite difference gradient was used to perform gradient descent, this demonstrates that this gradient may no longer be as accurate.

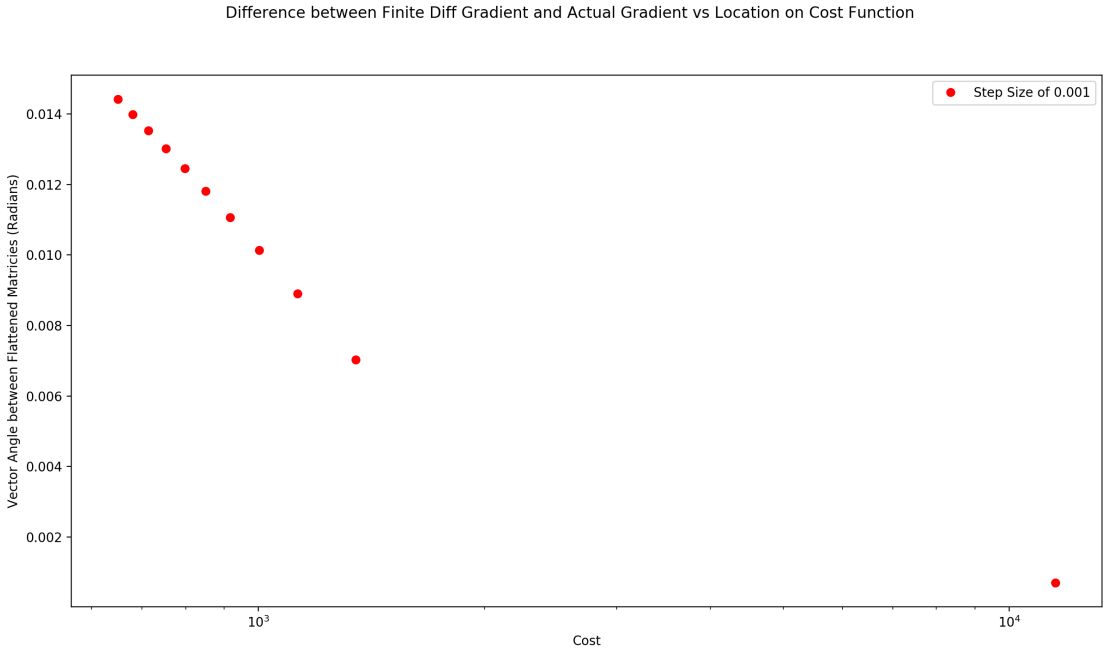


Figure 2: Angle Difference vs Cost

Part 4

Below are two plots of the performance of the neural network and the observed weights below the plots. On the left, 3 demonstrates performance for when training used the entire batch while on the right, 3 demonstrates performance performance for when training used for a mini batch of 5000 images. Although in terms of computation speed, the mini-batch was much faster, we can see the performance of the test set plateaus early due to the limited ability of generalization of a mini batch versus a full batch.

The optimization was performed with a gradient descent. Using the gradient calculated in Part 3a, the cost function was minimized iteratively by stepping in the opposite direction of the calculated gradient, to reduce cost. For the weight visualization, the weights displayed are actually very early on in the optimization process. This is to reduce over fitting of the weights to the noise of the training set and more towards the general features of each number.

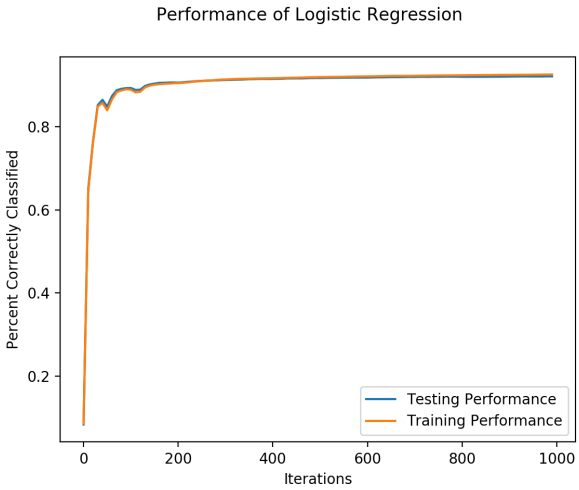


Figure 3: Batch Training Set

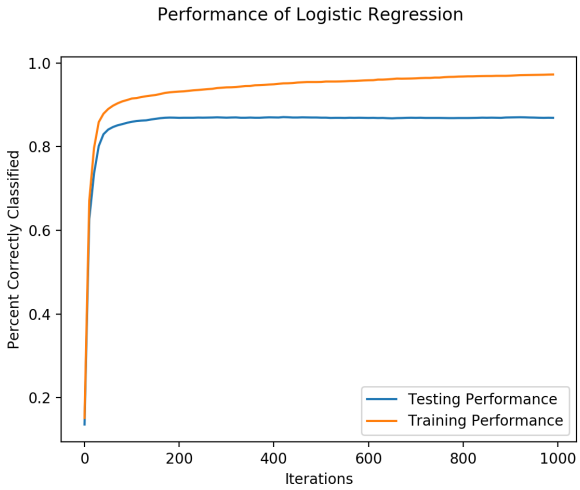
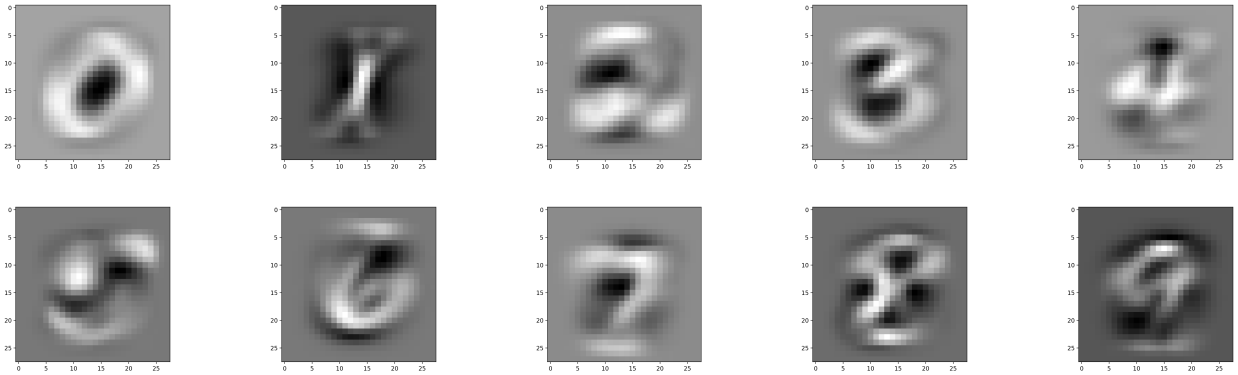


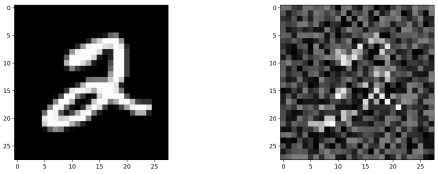
Figure 4: Mini Batch Training Set

Figure 5: Weight Visualization



Part 5

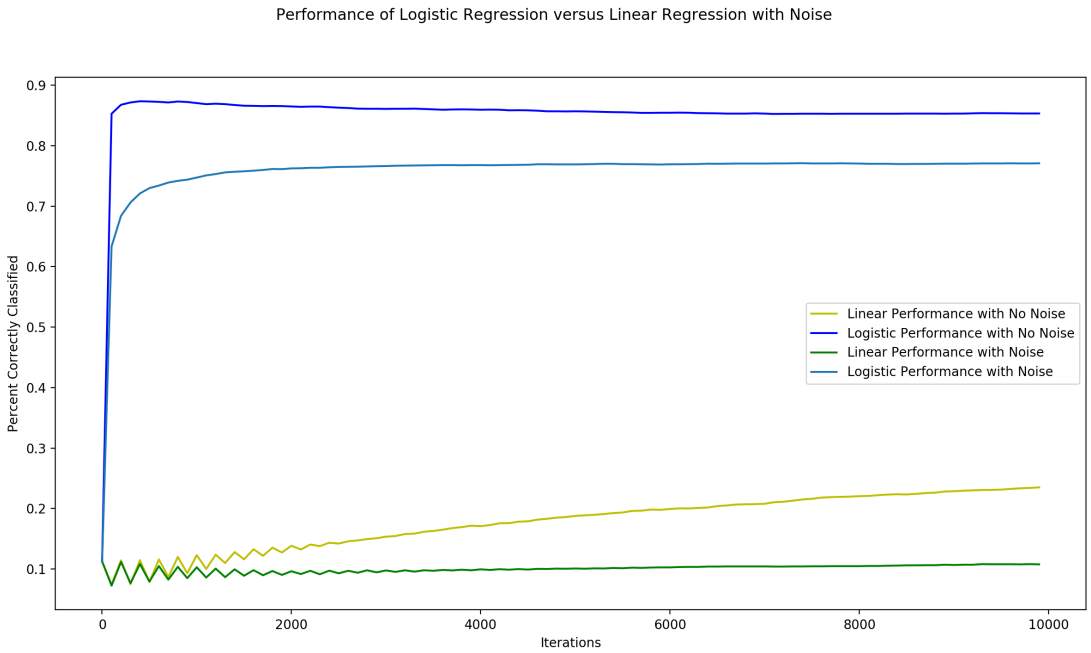
Below is the visualization of the same image, however the one of the left is the image without any noise and the one on the right is the image after adding noise.



This was done by adding random values between -1 and 1 to each pixel of the image. Moreover after adding the "noise", the absolute value of the pixel was used to ensure that there would be no negative values (pixel values shouldn't be negative). Here is the code used to add noise, note that only indicies 1:785 were affected because index 0 represented the bias term and remained as 1:

```
train_images_noise =  train_images.copy()
train_images_noise[1:785,:] += (np.random.rand(train_images.shape[0]-1,
  train_images.shape[1]) -0.5)*2
train_images_noise[1:785,:] = abs(train_images_noise[1:785,:])
```

Below is the performance of logistic and linear regression on the testing set. It is clear that with noise, logistic regression still performs quite well while linear regression is unable to classify many images correctly when noise is introduced. In essence, linear regression is not able to identify generalized features as well as logistic regression and can get more caught up in the noise of the data.



Part 6

Assumptions:

- The neural network consist N layers and K neurons in each layer.
- All layers are fully connected.
- Output is a single node.
- Each derivative operation cost is constant C.

Method 1: Calculate each gradient

For this method, the program needs to for gradient for each layer separately. For i th layer from the output in the neural network.

Consider,

$$x = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_k \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_k \end{bmatrix}$$

where x and y are the first and final layer respectively.

Moreover,

$$W^i = \begin{bmatrix} W_{i,1,1} & W_{i,1,2} & W_{i,1,3} & \dots & W_{i,1,k} \\ W_{i,2,1} & W_{i,2,2} & W_{i,2,3} & \dots & W_{i,2,k} \\ \dots & \dots & \dots & \dots & \dots \\ W_{i,k,1} & W_{i,k,2} & W_{i,k,3} & \dots & W_{i,k,k} \end{bmatrix}$$

where W^i is the weight matrix for the i^{th} layer and $W^i_{i,j,l}$ is the j,l element of W^i.

Thus,

$$y = W^{N-1} \dots W^2 W^1 x$$

For the i^{th} layer of the neural net, x is matrix multiplied by a weight matrix W^i. Thus each layer takes $O(k^2c)$. With N-1 weight matrices, the total cost is $O(k^{2(N-1)}c)$

Moreover, the derivative needs to be calculated with respect to each weight. There are $(N-1)k^2$ individual weights. Thus it takes $O((N-1)k^2c)$ to compute

Overall, the cost becomes $O(k^{2(N-1)}c) + O((N-1)k^2c) = O((k^{2(N-1)}c)$

Method 2: Backpropagation

The backpropagation calculate the gradient of a layer with respect to its upper layer. This means the individual gradient computation will only cost k^2c. There are N layer to be computed. Thus, the total cost of all the layer is O(Nk^2c)

When K and N are large, O(Nk^2c) is much less than $O((k^{2(N-1)}c)$. This will lead to the backpropagation to be much faster than calculate each gradient.

Part 7

A single hidden layer neural network with 200 hidden units is used in part7. The input takes a 227 by 227 gray scale picture from six actors. Each pixel's value is divide by 255 the feed as the input into the input layer. All the inputs are full connected with all 200 hidden units. Then, 200 hidden units are fully connected to six output units. Finally, we used softmax method to regularize the output so that each output represents the probably of the input is a certain act.
The program removes the invalid face pictures by implementing SHA-256 hash function. All the weights in the neural network are initialize to some random small value around 0. We used tanh function as our activation function for the hidden units. Then we use negative log-loss function to compute our cost. The training is done through mini-batch and each batch contains 10 image inputs.
In order to get the best performance, we tuned our parameter with different number of hidden units. It turn out that ,with very small number of hidden units(less than 50), the performance is poor. However, with the a bigger and bigger number of hidden unit, the performance was not getting any better on the validation set and the running gets slower. At the end, we choose the number of hidden units to be 200. Then we tried to tune the batch size against the validation set. It turns out the batch size(10 to 100) do not change the overall performance of the neural network significantly. We picked the batch size to be 10 because it runs the fastest.
The learning curve fluctuates frequently, because we used a relatively small batch size. The best performance of the test set is around 73%.

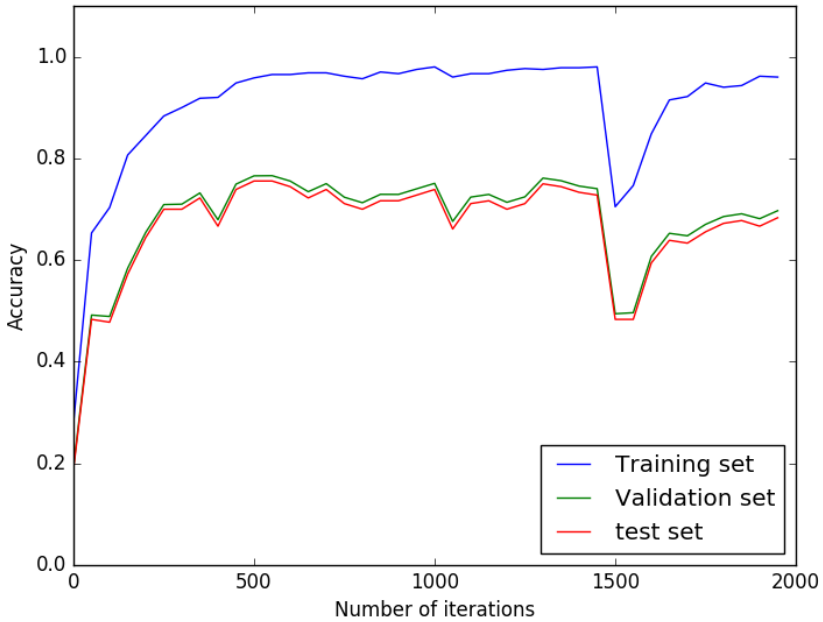


Figure 6: The learning curve of single layer neural network

Part 8

```
lam = 0.1
decay_penalty =lam*tf.reduce_sum(tf.square(W0))+lam*tf.reduce_sum(tf.square(W1))
```

The program uses both L1 and L2 regularization to prevent overfitting. The effect of regularization is not evident when the number of hidden layer is small. However, the regularization will noticeably increase the accuracy if the number of hidden layer is set to 1,000. With no regularization, after 3,000 iteration, the training accuracy is 100% and the test accuracy is at 78.88%. With the regularization coefficient "lam" set to 0.1, after 3,000 iteration, the training accuracy is 100% and the test accuracy is at 83.33%.
In this case, the regularization term successfully minimizes the overfitting and achieves greater accuracy on the test set.

Part 9

The weights visualization is shown below. For one particular output of an actor, we pick the biggest weight that go into this output node from the hidden layer. For example, the output, associated with Steve Carell, receives 300(number of hidden units) connections from all hidden units. From all those connection, we pick the hidden unit which has the largest weight on the connection that going into the output. Then, all the weights associated with that hidden unit and the input layer units are visualized, first weight becomes the first pixel and so on (just like how we visualize the image array to form a image).

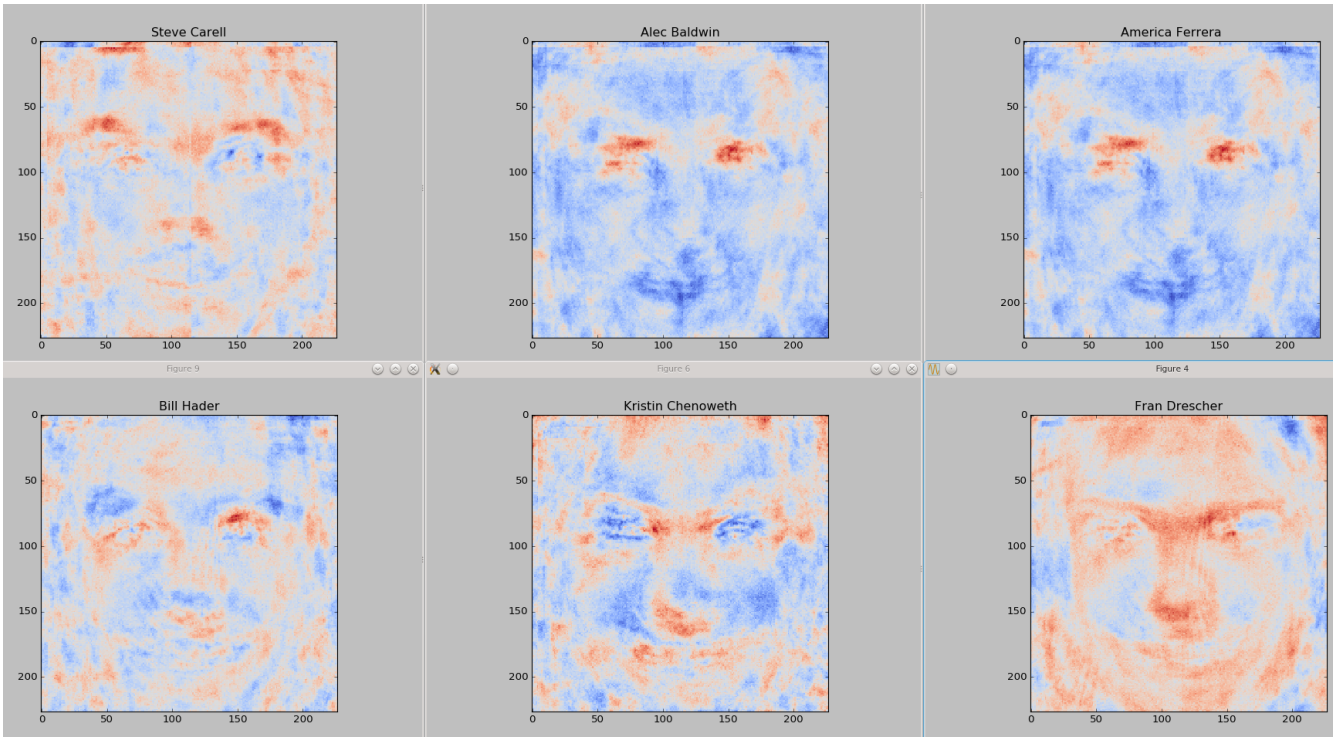


Figure 7:

Part 10

Values were extracted from the activations of AlexNet on the face images by extracting a convolutional layer of Alexnet.This layer would represent the feature map that Alexnet creates during its classification process of the image it is given. We then implemented a single layer neural network in tensorflow with these feature maps as the input. Thus our neural network took in a matrix (nx64896) of flattened convolution layers, where each layer used to be of dimensions (13x13x384) and n is the number of images. The neural network was then trained in the same way as was done in Part 7, to identify the correct actor/actress. In essence, we have only replaced images as our input with the feature map of AlexNet.

A minibatch of 50 images per actor/actress was used for training and 30 images of each actor/actress was used for testing. The performance of the test set plateaued around 92%, considerably better than the Part 7 results. This is likely due to the better feature "recognition" of AlexNet which would then be "learned" by our neural network. To some degree our single layer would represent the last layer of a deep neural network, which would identify complex features given the identification of lower level features from the activations of AlexNet and the elimination of noise from each individual image. On the otherhand, our single layer network in Part 7 would be more susceptible to noise because without multiple layers, it would not be able to filter it out and focus on general features of each actor/actress very well.