CSC411: Assignment #1

Due on Friday, Feb 3, 2016

Gabriel Arpino

February 3, 2017

Part 1

Dataset description
The facescrub dataset contains the name, shape, url, and bounding box coordinates for 265 color images of 6 male actors, and 265 color images of 6 female actors. The actors in the dataset are: Fran Drescher, America Ferrera, Kristin Chenowet, Alec Baldwin, Bill Hader, Steve Carell, Gerard Butler, Daniel Radcliffe, Michael Vartan, Lorraine Bracco, Peri Gilpin, Angie Harmon. The .txt files provided were subsets of the facescrub dataset, each line linking to an image of an actor. The files did not provide the same number of images for each actor/actress, the number of images available for each actor is as follows (calculated using count_dataset.py):

```
Steve Carell 290
Bill Hader 269
Alec Baldwin 271
Gerard Butler 238
Daniel Radcliffe 256
Michael Vartan 225
Fran Drescher 289
America Ferrera 286
Kristin Chenoweth 318
Lorraine Bracco 211
Peri Gilpin 195
Angie Harmon 257
```

The images were downloaded by iterating through each line in the provided facescrub_actors.txt and facescrub_actresses.txt files, and downloading the file from the specified url at that location (`get_data.py` (line 86)). A sample line from the facescrub_actors.txt file is as follows:

```
Alec Baldwin   3209 1862 http://sarcastic-news.com/wp-content/uploads/2013/11/
    ↪ Alec_Baldwin_PETA_Shankbone_2008.jpg 463,450,1785,1772
```

Upon downloading the images from their urls, they appear as follows:

Figure 1: A random image downloaded directly from the dataset. Actor Alec Baldwin. Generated using `get_data.py` (line 76)

Figure 2: A random image downloaded directly from the dataset. Actress Lorraine Bracco. Generated using `get_data.py` (line 76)

Figure 3: A random image downloaded directly from the dataset. Actor Gerard Butler. Generated using `get_data.py` (line 76)

Figure 4: A random image downloaded directly from the dataset. Actress Kristin Chenoweth. Generated using `get_data.py` (line 76)

The images were then cropped, resized to 32x32 pixels, and only the grey channel was analyzed. The grey and cropped images corresponding to the above uncropped images are as follows:

Figure 5: Cropped and grey channel image of actor Alec Baldwin. Generated using `get_data.py` (lines 81-98)

Figure 6: Cropped and grey channel image of actress Lorraine Bracco. Generated using `get_data.py` (line 81-98)

Figure 7: Cropped and grey channel image of actor Gerard Butler. Generated using `get_data.py` (line 81-98)

Figure 8: Cropped and grey channel image of actress Kristin Chenoweth. Generated using `get_data.py` (line 81-98)

Most other cropped and uncropped images are similar. There are, however, certain cropped images that hint at inaccuracies in the bounding boxes. An example is with the image from the 114th url corresponding to actor Bill Hader (counting from the start of the file to the end of the file), where the cropped image does not target the actors face, as shown:

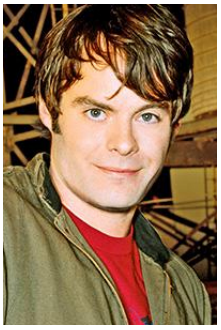


Figure 9: Uncropped 114th image of Bill Hader from the dataset. Generated using `get_data.py` (lines 76)

Figure 10: Cropped and grey channel 114th image of Bill Hader. Generated using `get_data.py` (line 81-98)

The cropped vesion of the image clearly does not capture the entirety of the actors face.
Another remark is that the cropped out faces cannot be aligned with each other or, in other words, they are not always centered or even facing the same direction. Examples shown below are of Steve Carell:

Figure 11: Sample cropped image of Steve Carell, face looking slightly to the left, placed to the right of the image. Generated using `get_data.py` (lines 81-98)

Figure 12: Sample cropped image of Steve Carell, face looking straight, placed closer to the center of the image. Generated using `get_data.py` (lines 81-98)

Figure 13: Sample cropped image of Steve Carell, face looking slightly to the right, placed to the left of the image. Generated using `get_data.py` (lines 81-98)

These small inaccuracies in the bounding boxes introduce uncertainty into the data, causing a lower performance on the training set but slightly hindering possible instances of overfitting.

Part 2

Separate the dataset into three non-overlapping parts
Initially, all images in the facescrub_actors.txt and facescrub_actresses.txt are downloaded into an "uncropped" folder and labelled to their corresponding actor/actress names using get_data.py. Then, from within this uncropped folder, the code would crop, covert to gray, reshape to 32x32 pixels, and save to a "cropped" folder.
At the begginning of each classification problem, the **faces.py script** would read from this "cropped" folder, append all filenames of the relevant actor into a list, randomly shuffle this list so that there is no bias (using np.random.seed(1) for reproducibility and np.random.shuffle()), and load 100 images into a numpy matrix for training, 10 images into a numpy matrix for validation, and 10 images into a numpy matrix for testing (matrices were created by stacking the image vectors, using np.vstack()). All images were non overlapping since the script would iterate in order through the shuffled list.

Part 3

Build a classifier to distinguish pictures of Bill Hader form pictures of Steve Carell
A linear regression model was used to classify pictures of Bill Hader and Steve Carell. The model was trained through gradient descent, or minimization of the cost function. The gradient was calculated manually and developed as a function in the script. Following are the mathematical expressions for the cost function and gradient.
Cost Function:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(\Theta^{T(i)}X^{(i)} - Y^{(i)})^2$$

Partial Derivative:

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(\Theta^{T(i)}X^{(i)} - Y^{(i)})x_j^{(i)}$$

Gradient:

$$\nabla J = \frac{1}{m}\sum_{i=1}^{m}(\Theta^{T(i)}X^{(i)} - Y^{(i)})X^{(i)}$$

Where X is the input vector, Θ is the vector of parameters which we train, Y is a vector of targets, m is the number of input vectors.
A classifier was run for 2000 iterations of gradient descent on 100 training images of Bill Hader, and 100 training images of Steve Carell. These were then tested against 10 validation and 10 test images for each actor. The values of the cost function for the training and validation sets after 2000 iterations are:

```
Training Cost: 0.15061987679
Validation Cost: 0.173595062825
```

There were some fluctuations, however, and they can be summarized with this plot of cost against number of iterations:

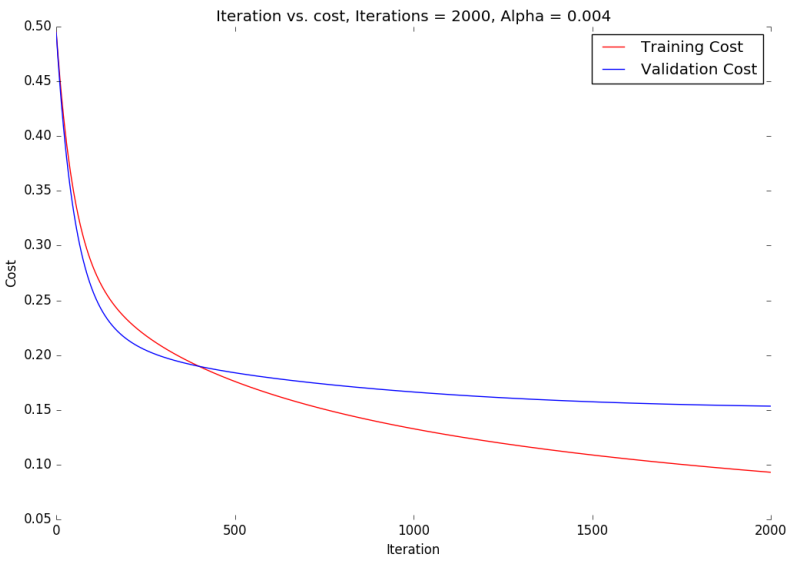


Figure 14: Plot of training and validation costs versus number of gradient descent iterations (for 2000 iterations), using a learning rate of 0.004. Obtained from `faces.py`

It is observed that the validation cost surpasses the training cost at around iteration 400, hinting that the model is training features that allow it to fit to the training set better, features that may not exactly correspond to the validation set. It can be seen that the validation cost plateaus and even seems to increase a little, while the training cost always decreases. This explains the larger validation cost over training cost mentioned previously. Again, this is likely because the model is showing signs of overfitting its training data. In terms of the percent accuracy of the model, the results are similar. Following are the percent accuracies (percentage of correctly classified images) on both the training and validation sets after running 2000 iterations of gradient descent:

```
Percent Validation Accuracy: 95.0%
Percent Train Accuracy: 96.0%
```

These percent accuracies also follow a trend, and for 2000 iterations the trend appears in Figure 16.

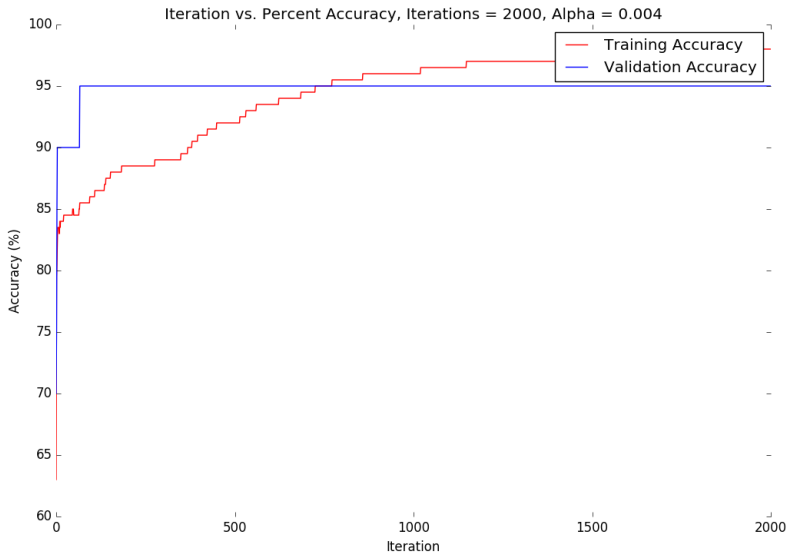


Figure 15: Plot of training and validation percent accuracies versus number of gradient descent iterations (for 2000 total iterations), using a learning rate of 0.004. Obtained from `faces.py` (lines 81-98)

The accuracies are discrete percentage values because the number of training images is discrete, and so is the number of correctly classified images. We can see from the plot that the training accuracy increases with the validation accuracy, but remains a little higher.
The code to compute the output of the classifier (classifying between Steve Carell or Bill Hader) involved a simple matrix multiplication emulating the $\Theta^T X = Y$ formula:

```
def predictions(weights, inputs):
    return np.dot(inputs, weights)
```

This predictions function was used to compute the output of the classifier. The predictions were then summed up and devided by the total number of targets to get the percent accuracy:

```
def percent_accuracy(weights, inputs, test_targets):
    accuracy = 0
    pred = predictions(weights, inputs)
    for i in range(len(pred)):
        if ((abs(pred[i] - 1) < abs(pred[i] + 1))):                # If predicted
            ↪ value is closer to 1 than -1
```

```
            accuracy = accuracy + (test_targets[i] == 1)
        else:
            accuracy = accuracy + (test_targets[i] == -1)

    return accuracy/float(len(test_targets))
```

In order to make the system work, many approaches were taken.
The initial theta was chosen empirically to optimize accuracy on validation data. It was noticed that an initial theta of zero for all elements yielded faster convergence and higher validation accuracy, likely because all thetas are between the midpoint of the targets, -1 and 1. Random initial thetas were tested, but these did not yield results that were as good.
Additionally, the system would converge very slowly if the images were not normalized. Using a maximum learning rate of 10^{-7}, the code would take over 2 minutes to converge and the accuracy would cap at 75%. This was fixed by performing an approximate normalization on the data, where the image vector was divided by 255.0 before being added to the input matrix as follows:

```
for i in range(0,10):                                   # Load the 10 overfit test
    ↪ images
    img = imread(test_files[i])                         # Load image into numpy matrix
    flat_img = img.flatten()            # Flatten Image into single vector
    flat_img = flat_img/255.0                       # Approximately normalize image
    flat_img = np.append(flat_img, 1)                   # Append Bias Unit
    overfit_test_inputs.append(flat_img)                    # Append image to vector
        ↪ containing all images
```

After this normalization, my alpha (learning rate) was changed from 10^{-7} to 10^{-3}, and the classifier converged within seconds to a percent accuracy above 90%.
Another requirement for a working system was the selection of an appropriate learning rate alpha. If the learning rate was too large (> 0.007), the cost function would increase at every iteration, and the percent accuracy would decrease at every iteration, as shown in Figures 16 and 17.

Figure 16: Plot of cost vs. number of iterations when alpha is too large. Generated using `faces.py`

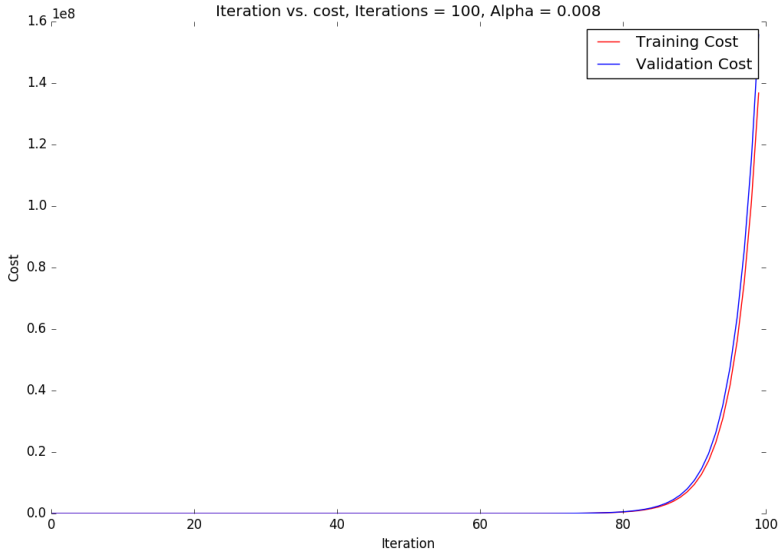


Figure 17: Plot of percent accuracy vs. number of iterations when alpha is too large. Generated using `faces.py`

It can be seen that the error in the classifier spirals quickly, as it can be viewed already at 100 iterations. The learning rate was then adjusted by trial and error, the best learning rate being that which classifies the problem in seconds, and achieves a peak validation accuracy. Figure 18 is a plot of performance vs. learning rate for different iterations. This plot was used to decide on the alpha to be used.

Figure 18: Plot of cost vs. learning rate for 1000 iterations. Generated using `faces.py`

It can be observed that the cost blows up at a learning rate of 0 and a learning rate above 0.007. In between these values, the learning rates become a tradeoff between overfitting with slow convergence when small and

underfitting with fast convergence when large. A learning rate of 0.004 was chosen because it is close to the midpoint of the boundaries, trading off between speed of convergence and overfitting rate.

Part 4

Visualizing Theta
The theta matrix was visualized for different alphas with a full training set, and with a training set containing only two images. The theta matrices were downloaded directly through their grey channel, hence their grey rather than heat map appearance. Following is the theta matrix visualized with the full training set:

Figure 19: Theta for 2000 iterations, learning rate 0.004, 100 training files. Generated using `faces.py`

Figure 20: Theta for 4000 iterations, learning rate 0.004, 100 training files. Generated using `faces.py`

Now, two thetas visualized for only two images present in the training set:

Figure 21: Theta for 2000 iterations, learning rate 0.004, 2 training files. Generated using `faces.py`

Figure 22: Theta for 4000 iterations, learning rate 0.004, 2 training files. Generated using `faces.py`

We can observe that the thetas trained from only 2 training images are largely more overfit, as they appear more as faces, and they likely appear very similar to the two training image faces used. The thetas using 100 training images still appear like faces, but less so due to the need to generalize the theta to all 100 images.It can be also observed that, as the number of iterations increases, the faces become less visible likely due to overfitting.

Part 5

Classifying actors as male or female and overfitting
Now, we attempt to classify actors based as male or female from the set of six actors:

```
act =['Fran Drescher', 'America Ferrera', 'Kristin Chenoweth', 'Alec Baldwin', 'Bill
    ↪ Hader', 'Steve Carell']
```

After this classification, we perform a classification on 10 images of each of the following non-trained actors to analyze overfitting:

```
act_test = ['Gerard Butler', 'Daniel Radcliffe', 'Michael Vartan', 'Lorraine Bracco',
    ↪ 'Peri Gilpin', 'Angie Harmon']
```

Following are the final results from the classification procedure for 1000 iterations using a learning rate of 0.004, training on 100 images:

```
Final Percent Test Accuracy: 90.0 %
Final Percent Validation Accuracy: 93.3333333333 %
FInal Percent Train Accuracy: 93.0 %
Final Percent Overfit Test Accuracy: 91.6666666667 %
```

Following are the final results from the classification procedure for 3000 iterations using a learning rate of 0.004, training on 100 images:

```
Final Percent Test Accuracy: 88.3333333333 %
Final Percent Validation Accuracy: 96.6666666667 %
FInal Percent Train Accuracy: 97.3333333333 %
Final Percent Overfit Test Accuracy: 90.0 %
```

It can be observed that the final test, validation, and train accuracy are very high (above 90%). The overfit test accuracy, however, is a little lower than the validation and training accuracy because there was likely overfitting on the training test. This can be seen more obviously in the second set of results where the model was run for 3000 iterations, leading to more overfitting and a smaller overfit test accuracy than the 1000 iterations case. Figure 24 and 25 are plots of the performance of the model versus number of training examples for two different numbers of iterations:
It can be observed that, for both 1000 and 2000 iterations, the training accuracy is higher than the validation accuracy, and that is because the model is optimizing using the gradient from the training set, meaning it will only learn distinguishing features from the training set. These features will likely also be present in the validation set, but probably not all. The validation set accuracy, however, increases a little bit as the number of training images increases because, the more different training images the model has, the harder it is to overfit.

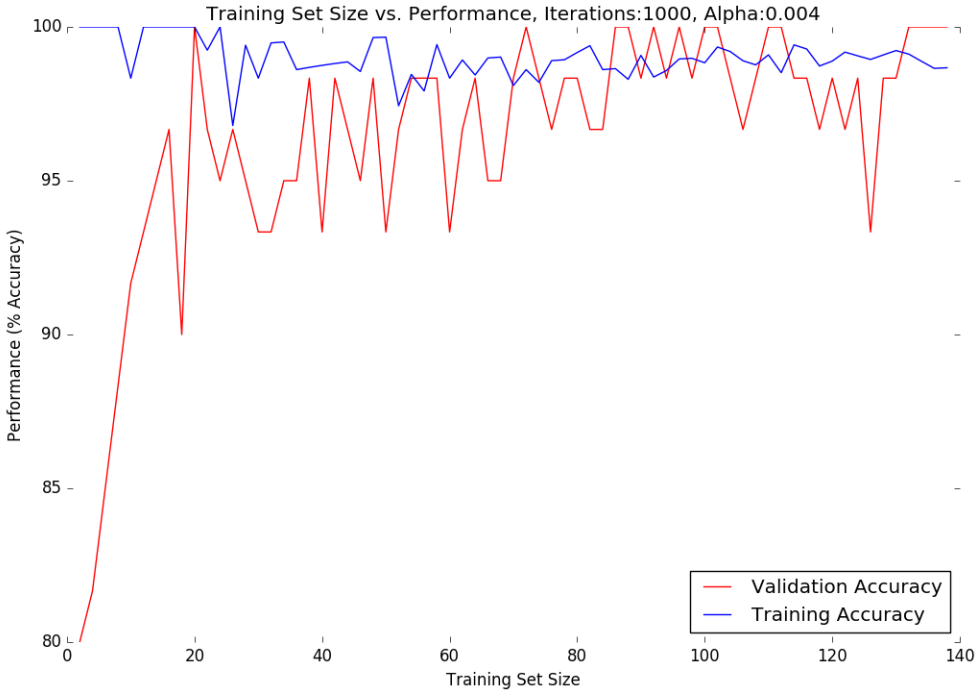


Figure 23: Model performance vs. training set size for 1000 iterations. Initial parameters all zero. Generated using `faces.py`

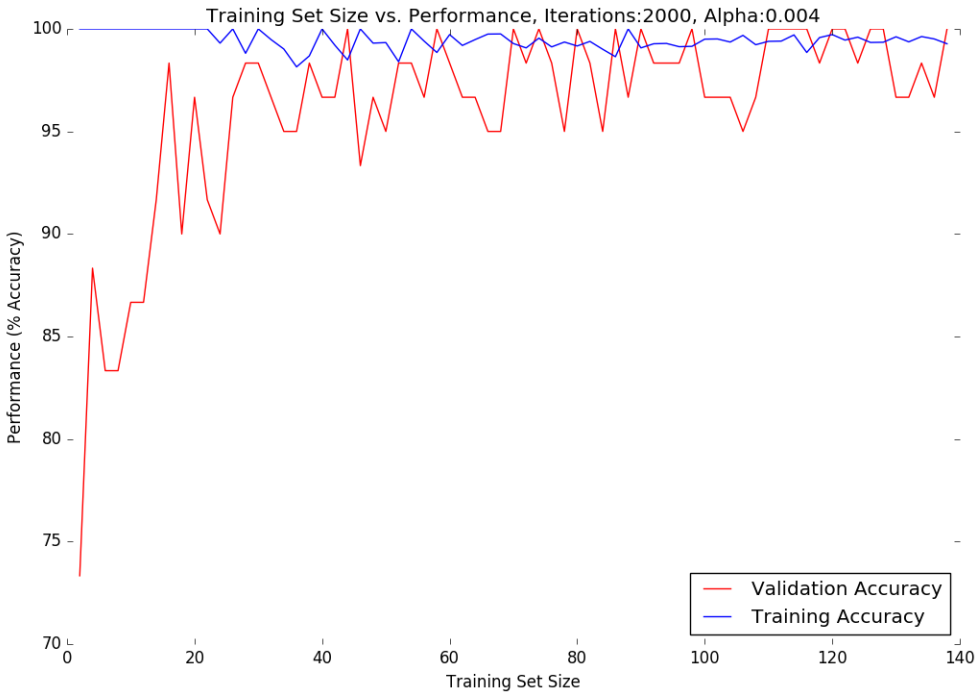


Figure 24: Model performance vs. training set size for 2000 iterations. Initial parameters all zero. Generated using `faces.py`

Part 6

Multi-class classification

6a)

The cost function to be minimized for multi-classification has a derivative that can be derived as follows (taking the case of classifying 4 different actors):

$$\begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} & \dots & \theta_{1n} \\ \theta_{21} & \theta_{22} & \theta_{23} & \dots & \theta_{2n} \\ \theta_{31} & \theta_{32} & \theta_{33} & \dots & \theta_{3n} \\ \theta_{41} & \theta_{42} & \theta_{43} & \dots & \theta_{4n} \end{bmatrix} * \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1m} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & \dots & x_{nm} \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} & y_{13} & \dots & y_{1m} \\ y_{21} & y_{22} & y_{23} & \dots & y_{2m} \\ y_{31} & y_{32} & y_{33} & \dots & y_{3m} \\ y_{41} & y_{42} & y_{43} & \dots & y_{4m} \end{bmatrix}$$

Where each column of the x matrix is an image and the rows are the pixels. Theta is an n x k matrix, and each x vector in the x matrix is nx1. So, for each image $x^(i)$, we have:

$$\theta_{11} * x_{11} + \theta_{12} * x_{12} + \theta_{13} * x_{13} + \cdots + \theta_{1n} * x_{1n} = y_{11}$$
$$\theta_{21} * x_{21} + \theta_{22} * x_{22} + \theta_{23} * x_{23} + \cdots + \theta_{2n} * x_{2n} = y_{21}$$
$$\theta_{31} * x_{31} + \theta_{32} * x_{32} + \theta_{33} * x_{33} + \cdots + \theta_{3n} * x_{3n} = y_{31}$$
$$\theta_{41} * x_{41} + \theta_{42} * x_{42} + \theta_{43} * x_{43} + \cdots + \theta_{4n} * x_{4n} = y_{41}$$

And, when we differentiate, we get:

$$\frac{\partial \Theta^T X}{\partial \theta_{pq}} = x_{pq}$$

And when we add this to the chain rule of the derivative of the cost function and sum over all images, we get:

$$\frac{d}{d\theta_{pq}}(\Theta^T X^{(i)} - Y^{(i)})^2 = 2(\Theta^T X^{(i)} - Y^{(i)})(\frac{d}{d\theta_{pq}}(\Theta^T X^i))$$
$$\frac{\partial J}{\partial \theta_{pq}} = 2\sum_i (\Theta^T x^{(i)} - y^{(i)}) x_{pq}^{(i)}$$

6b)

As seen in Part 6a), the derivative of the cost function involves the difference between the prediction and targets, multiplied by an x:

$$(\Theta^T x^{(i)} - y^{(i)}) x_{pq}^{(i)}$$

This, however, when summed over for both dimensions of X (since we also saw in section 6a that X is a two dimensional matrix), we can simplify the $\sum_j$ and the $\sum_i$ (sum over images and sum over target indexes) into a matrix multiplication involving just X. As mentioned in part 6a, the X matrix is an nxm matrix, where n is the number of pixel in the images and m is the number of training examples.
If we analyze the partial derivative of J:

$$\frac{\partial J}{\partial \Theta} = \begin{bmatrix} \frac{\partial J}{\partial \Theta_1} \\ \dots \\ \frac{\partial J}{\partial \Theta_j} \\ \dots \\ \frac{\partial J}{\partial \Theta_N} \end{bmatrix}$$

$$\frac{\partial J}{\partial \Theta} = \begin{bmatrix} 2(x^{(i)}\Theta - y^{(i)})(x_1) \\ \dots \\ 2(x^{(i)}\Theta - y^{(i)})(x_j) \\ \dots \\ 2(x^{(i)}\Theta - y^{(i)})(x_N) \end{bmatrix}$$

Now, recall that matrix multiplication is a dot product between row i and column j, so the equation from part 6a can be rearranged to:

$$\frac{\partial J}{\partial \theta_{pq}} = 2 \sum_{alldimensions} ((\Theta^T x^{(i)} - y^{(I)}) x_{pq}^{(i)})$$

which then simplifies to yield:

$$2X(\Theta^T X - Y)^T$$

as the derivative of the cost function for all thetas.

6c)

The code for multi-class cost function and gradient was done with the transpose of all matrices for my ease of reading, so I based my code off of the equation:

$$X^T \Theta = Y^T$$

Which yields the same results. Following is the code:

```
from scipy.misc import *
import glob
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from scipy.misc import imsave

def predictions(weights, inputs):
    return np.dot(inputs, weights)

def cost_function(weights, inputs, test_targets):
    return (1.0/(2.0*m))*np.sum((predictions(weights,inputs) - test_targets)**2)

def gradient(weights, inputs, test_targets):
    return (1.0/m)*np.dot(inputs.T, (predictions(weights,inputs) - test_targets))
```

Where weights corresponds to the Θ matrix, inputs corresponds to the X^T matrix, and test_targets corresponds to the Y^T matrix.

6d)

All components of the gradient were approximated using finite differences, where the derivative of the cost is approximated:

$$\frac{\partial J}{\partial \theta_{pq}} = \frac{f(\Theta_{pq} + h, x) - f(\theta_{pq}, x)}{h}$$

The process involved disturbing the theta matrix by a small perturbation (a small step size h) and comparing the change in the cost function to the gradient. The following code was used:

```
def finite_differences(weights, inputs, test_targets, perturbation = 0.005):
    ''' Approximates the gradient through finite differences and compares it to the
        ↪ gradient function'''
    print "Calculating Finite Difference and Gradient Error"
    total_error = 0
    for row in range(theta.shape[0]):
        for col in range(theta.shape[1]):
            prev_cost = cost_function(theta, input_mat, test_targets)
            deriv = gradient(theta, input_mat, test_targets)
            theta[row][col] += perturbation
            after_cost = cost_function(theta, input_mat, test_targets)
            cost_diff = (after_cost - prev_cost)/float(perturbation)
            total_error += cost_diff - deriv[row][col]

    avg_error = total_error/float(theta.shape[0]*theta.shape[1])
    return avg_error
```

The function not only calculates the error between the finite difference calculation and gradient for each theta, but it also averages these errors over the entire shape of theta. Following are the raw outputs of the code:

```
Perturbation of 0.005: Calculating Finite Difference and Gradient Error
0.000807192275581 average error
Perturbation of 0.00005 Calculating Finite Difference and Gradient Error
8.07190109219e-06 average error
```

A perturbation size of 0.005 results in an average error of 0.000807192275581, and an even smaller perturbation size of 0.00005 results in an average error of 8.07190109219e-06, even smaller than the first. The finite differences method therefore suggests that our gradient function is correct, because it is approximated with high accuracy through the method, and the average error of the approximations for all thetas will eventually reach zero as the perturbation used gets smaller.

Part 7

Multi Classification gradient descent
Gradient descent was run on the following set of six actors:

```
actors_list = ['Fran Drescher', 'America Ferrera', 'Kristin Chenoweth', 'Alec Baldwin'
    ↪ , 'Bill Hader', 'Steve Carell']
```

The parameters used for optimization were a learning rate of 0.004 (chosen by the same criteria mentioned in part 3). The initial theta was all zeros because this resulted in the fastest convergence and highest validation accuracy, and the number of iterations is 4000. Following are the final validation, train, and test accuracies:

```
Beggining Classification
Learning rate: 0.004
Final Percent Test Accuracy: 86.6666666667 %
Final Percent Validation Accuracy: 85.0 %
Final Percent Train Accuracy: 87.5 %
```

The learning rate was chosen through trial and error and based off of the fact that the Bill Hader and Steve Carell classification done previously demonstrated a best learning rate of 0.004. It was noticed that the model cost blows up with a learning rate of 0.007, so the learning rate would have to be smaller than 0.007 to converge, but not too much smaller so that we overfit the training data (similar learning rate restrictions as described in Part 3).
To confirm this, a plot of validation accuracy versus learning rate was created:

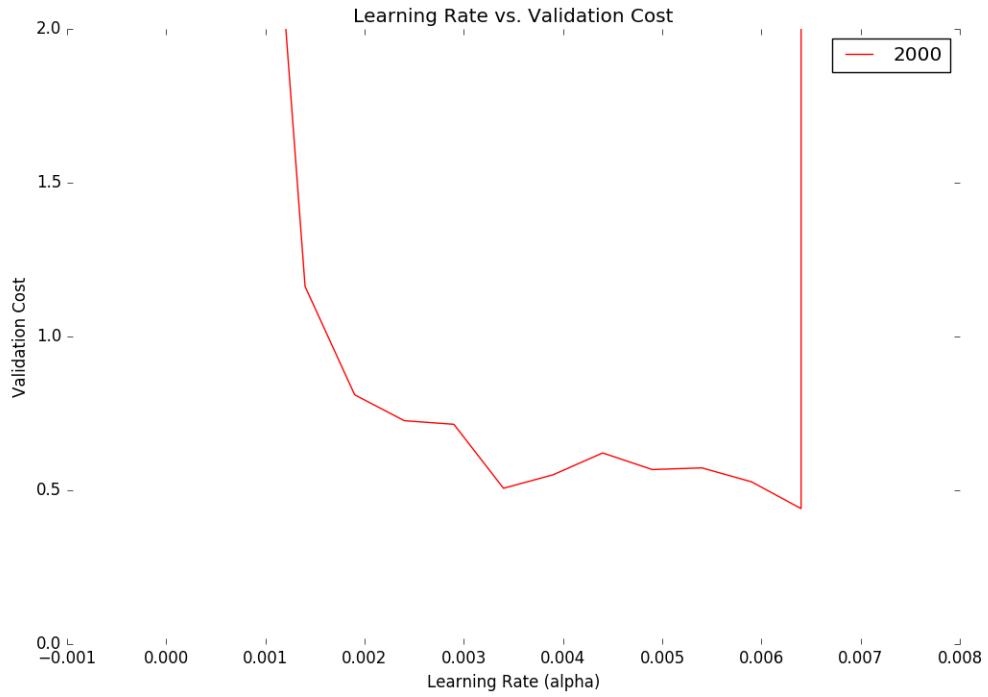


Figure 25: Model validation performance accuracy vs. learning rate for 200 iterations. Initial parameters all zero. Generated using `faces.py`

It can therefore be seen that the lowest cost is achieved at an unclear value between 0 and 0.007. A learning rate of 0.004 was therefore chosen because it is in between the two boundaries and is a midpoint tradeoff

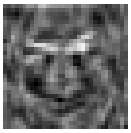
between a chance of overfitting with slow conversion and a chance of underfitting with faster conversion. The number of iterations was chosen as 4000 by testing 1000,2000,3000, and 4000 iteration performances, and the raw output followed:

```
Beggining Classification
Learning rate: 0.004 Num Iterations: 1000
Final Percent Test Accuracy: 86.6666666667 %
Final Percent Validation Accuracy: 85.0 %
Final Percent Train Accuracy: 87.5 %
Beggining Classification
Learning rate: 0.004 Num Iterations: 2000
Final Percent Test Accuracy: 76.6666666667 %
Final Percent Validation Accuracy: 78.3333333333 %
Final Percent Train Accuracy: 91.8333333333 %
Beggining Classification
Learning rate: 0.004 Num Iterations: 3000
Final Percent Test Accuracy: 83.3333333333 %
Final Percent Validation Accuracy: 81.6666666667 %
Final Percent Train Accuracy: 93.8333333333 %
Beggining Classification
Learning rate: 0.004 Num Iterations: 4000
Final Percent Test Accuracy: 91.6666666667 %
Final Percent Validation Accuracy: 88.3333333333 %
Final Percent Train Accuracy: 96.3333333333 %
```

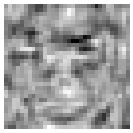
It can be seen from the raw output that 4000 iterations results in the highest value for the validation and training set results. Any more than this number of iterations would overfit the model to the training set.

Part 8

Visualizing Thetas
The theta matrices were downloaded directly through their grey channel, hence their grey rather than heat map appearance, and they appear as follows (there are 6 visual thetas per classification, each one corresponding to each actor):



(a) Theta for Fran Drescher for alpha 0.004 in 4000 iterations



(b) Theta for America Ferrera for alpha 0.004 in 4000 iterations

(c) Theta for Kristin Chenoweth for alpha 0.004 in 4000 iterations

(d) Theta for Alec Baldwin for alpha 0.004 in 4000 iterations

(e) Theta for Bill Hader for alpha 0.004 in 4000 iterations

(f) Theta for Steve Carell for alpha 0.004 in 4000 iterations

Figure 26: Another example of a completely missed crop

The algorithm works in that it produces thetas that resemble faces of their respective actors. This makes sense because the matrix multiplication essentially invokes a dot product between the thetas and the inputs, and the dot product is maximized when the theta looks as close as possible to its input images.