CSC411: Assignment 1

Due on Friday, February 3, 2017

Zi Mo Su (1001575048)

February 3, 2017

Part 1

Dataset description.
The uncropped dataset consists of $1,940$ images of varying sizes. It is composed of six actors and six actresses, each with a varying number of images. For our purposes, we use 120 images of each actor/actress and we have split the actors/actresses into two groups, `act` and `act_test`. Figure 1 shows examples of the raw data.

Figure 1: Examples of the raw images (photos of Alec Baldwin).

The cropped dataset consists of $1,940$ grayscale 32×32-pixel images. Each image is cropped such that the person's face dictates the boundary of the image, as shown in Figure 2.

Figure 2: Examples of the cropped images (photos of Alec Baldwin).

The alignment of the images in Figure 2 is shown in Figure 3, where the transparency of each of the five images has been set to 90% and they have been superimposed. We observe that the overlaid images align fairly accurately at the eyes, nose, mouth, and eyebrows; all of which are desired features of accentuation for facial recognition.

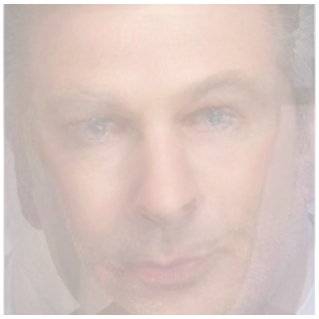


Figure 3: Superimposed images from Figure 2.

The raw data comes in the form of a text file, the quality of which is sub-par; numerous images are corrupted and thus must be disposed of and some images are not of the actor/actress but have been removed and replaced with a filler by the source. The data itself is easily parse-able, however, making the preprocessing fairly simple.

Part 2

Partitioning the dataset.
The dataset, after preprocessing in Part 1, is stored in a dictionary. This dictionary contains key-value pairs of the form:

`{baldwin000:2D array}`

The key is the name of the file without its extension, and the value is the 2D numpy array representing the processed image.

This main dataset is taken and partitioned into a training set with 100 images per actor/actress, and a validation set and test set with 10 images each per actor/actress. The form of these sets is the same as the above dictionary for the master set. The `partition()` function is used to partition the set. It takes in the list of actors (e.g., `act`), the master set, and the sizes for the training, validation and test sets, which are by default 100, 10, and 10 respectively. Since the data is retrieved from a dictionary, the order is uncertain; as such, the data is partitioned according to the filename (e.g., `baldwin000`), specifically the number in the filename (e.g, `000`). The first 10 entries are parted to the validation set, the next 10 to the test set and the next 100 to the training set.

The `partition()` function is shown below:

```
def partition(act, set, train_size = 100, val_size = 10, test_size = 10):
    '''Returns three subsets from the set of all actors/actresses; training,
    validation, and test sets. Each subset's size can be specified and the list
    of actors/actresses to retain in the subset is specified.
    Arguments:
    act -- list of actors/actresses to retain in subsets
    set -- input set of all actors/actresses
    train_size -- number of images per actor/actress to retain in training set
    val_size -- number of images per actor/actress to retain in validation set
    test_size -- number of images per actor/actress to retain in test set
    '''
    val_set = {}
    test_set = {}
    train_set = {}

    # loop through actors/actresses to be retained
    for a in act:
        name = a.split()[1].lower()

        # loop through each element in the dictionary set
        for filename in set:
            if name in filename:

                # save first val_size number of actors to validation set
                if int(filename[-3:]) < val_size:
                    val_set[filename] = set[filename]

                # save next test_size number of actors to test set
                elif int(filename[-3:]) < val_size + test_size:
                    test_set[filename] = set[filename]

                # save next train_size number of actors to training set
                elif int(filename[-3:]) < val_size + test_size + train_size:
                    train_set[filename] = set[filename]

                else:
                    continue

    return train_set, val_set, test_set
```

Part 3

Building a classifier for two actors.
A classifier for images of Bill Hader and Steve Carell is implemented in this part. The cost function minimized is:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}(y^{(i)} - \boldsymbol{x^{(i)}\theta^{T}})^2$$

where m is the number of images that are in the training set, $y^{(i)}$ is -1 if the i^{th} image is of Bill Hader and 1 if the image is of Steve Carell, $\boldsymbol{x^{(i)}}$ is a 1×1025 vector where the first element is 1 and the next 1024 are the pixels of the 32×32 image read left to right and top to bottom, and $\boldsymbol{\theta}$ is a 1×1025 vector with elements $\theta_0 ... \theta_{1024}$, which are the unknown parameters to be optimized over.

The gradient descent algorithm was applied to the training set (size 100) with $\epsilon = 1 \times 10^{-6}$, $\alpha = 1 \times 10^{-7}$. It took $3,369$ iterations and yielded the following results:

i. Cost of the *training set*: $J(\boldsymbol{\theta}) = 0.042$

ii. Cost of the *validation set*: $J(\boldsymbol{\theta}) = 0.42$

iii. Performance of the *training set*: 100.0% accurate

iv. Performance of the *validation set*: 95.0% accurate

To make the gradient descent algorithm work, the parameter α had to be adjusted. When α was too large the system would diverge to infinity. When α was too small the system converged too quickly to a local minimum and thus resulted in poor performance. The α value was tuned by multiples of 10 starting at 1×10^{-4} to 1×10^{-10}, and it was found that 1×10^{-7} yielded effective performance. It is important also to note that a smaller α may yield better performance but will result in longer runtime and overfitting.

Other parameters adjusted for gradient descent included ϵ and the number of max iterations. If ϵ is chosen very small, then it is likely the max iteration will be hit. Alternatively, a larger ϵ may result in completion of gradient descent before the max iteration is hit. ϵ was chosen to be 1×10^{-6}. Larger values were tested but resulted in poor performance, since the number of iterations was not enough. Smaller values resulted in extremely long runtimes between five and 10 minutes. The maximum number of iterations was set to $100,000$, however it is never reached in the reported results.

The following is the function used to compute the output of the classifier. If the computed hypothesis function, $\boldsymbol{x\theta^T}$, is less than zero, the image is classified as Bill Hader, otherwise it is classified as Steve Carell.

```
def performance_hc(set, t):
    '''Evaluates the performance of a given set for hader/carell classification.
    Arguments:
    set -- set to be evaluated
    t -- theta vector generated through training
    '''
    right = 0
    wrong = 0

    for filename in set:
        if "hader" in filename:
            x = make_row(set[filename])
            # x*t < 0 -> -1 (hader)
            if dot(x, t.T) < 0:
                right += 1
            else:
                wrong += 1

        if "carell" in filename:
            x = make_row(set[filename])
             # x*t > 0 -> 1 (carell)
            if dot(x, t.T) >= 0:
                right += 1
            else:
                wrong += 1

    if set == train_set:
        set = "Training set"
    elif set == val_set:
        set = "Validation set"
    else:
        set = "Testing set"

    print set, "is", str(right/((right + wrong)*0.01)) + "% accurate"
```

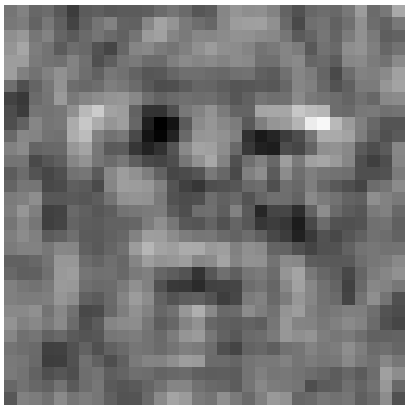
Part 4

Visualizing optimized θ parameters.
The hypothesis function:

$$h_\theta(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + ... + \theta_{1024} x_{1024}$$

is used to compute the approximation of the classification value. The θ parameters that correspond to a minimum for the cost function, generated through gradient descent, can be visualized as a 32×32 image, excluding θ_0.

Two training sets are used to optimize the $\boldsymbol{\theta}$ vector. One with 100 images of each actor and the other with two images of each actor. The results of the visualized $\boldsymbol{\theta}$ vectors are shown in Figure 4



(a) The $\boldsymbol{\theta}$ vector generated from a training set consisting of 100 pictures of each actor (Bill Hader and Steve Carell), visualized as an image.

(b) The $\boldsymbol{\theta}$ vector generated from a training set consisting of two pictures of each actor (Bill Hader and Steve Carell), visualized as an image.

Figure 4

Part 5

Demonstration of overfitting on gender classifiers.
The gender classifiers used to classify actors/actresses as male or female were created by assigning $y = -1$ for actors and $y = 1$ for actresses. Training sets varying in size by multiples of 10 images per actor/actress were used, starting at 10 images per actor/actress and ending at 100. The training set contains six different actors, three male and three female. The parameter $\alpha = 1 \times 10^{-7}$ and the validation set were both held constant throughout the trials. The performance results of the validation set and training set are plotted against the training set size in Figure 5.

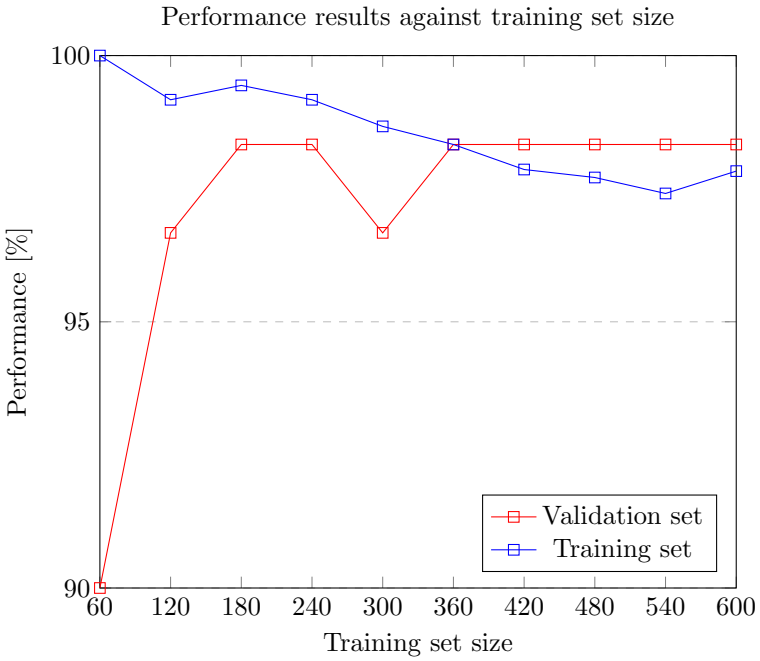


Figure 5

We observe the consistent decrease in performance of the training set. At a small training set size, it is easy to fit the training data accurately, hence the 100.0% performance on the training set when the size is 60. As the size increases, the data becomes harder to fit, and thus poorer performance results arise. On the other hand, the validation set fits the data the worst at the beginning because of overfitting. The data is overfitting the training set; it captures patterns that are unique to the training set. As such, since these patterns are not exhibited in the validation set at times, the performance is lowered. In general, we observe the validation set performance remains stable after the first couple training set sizes. A larger training set results in poorer performance on the training data because the fit becomes more generalized.

Testing the classifier on the set of actors/actresses that are not in the training data (i.e., `act_test`), yields an accuracy of 88.0%. This performance accuracy is quite good; as such it is fair to say that there is a significant enough difference between males and females for classification purposes. It would be interesting to observe the effects of added images of females with short hair and males with long hair.

Part 6

Classification with multiple labels.

a) The cost function is:

$$J(\boldsymbol{\Theta}) = \sum_{i=1}^{m}\sum_{j=1}^{k}(\boldsymbol{\Theta}^T\boldsymbol{x^{(i)}} - \boldsymbol{y^{(i)}})_j^2$$

For a single image, $\boldsymbol{\Theta}^T\boldsymbol{x^{(i)}} - \boldsymbol{y^{(i)}}$ is:

$$\begin{bmatrix}\theta_{01} & \theta_{11} & \dots & \theta_{(n-1)1}\\ \theta_{02} & \theta_{12} & \dots & \theta_{(n-1)2}\\ \vdots & \vdots & \ddots & \vdots\\ \theta_{0k} & \theta_{1k} & \dots & \theta_{(n-1)k}\end{bmatrix}\begin{bmatrix}1\\ x_1^{(i)}\\ \vdots\\ x_{n-1}^{(i)}\end{bmatrix} - \begin{bmatrix}y_1^{(i)}\\ y_2^{(i)}\\ \vdots\\ y_k^{(i)}\end{bmatrix}$$

The expansion of which is:

$$(\boldsymbol{\Theta}^T\boldsymbol{x^{(i)}} - \boldsymbol{y^{(i)}})_j = \theta_{0j} + \theta_{1j}x_1^{(i)} + ... + \theta_{(n-1)j}x_{n-1}^{(i)} - y_j^{(i)}$$

The partial derivative of the cost function with respect to θ_{pq} is thus:

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \theta_{pq}} = 2\sum_{i=1}^{m}\sum_{j=1}^{k}(\boldsymbol{\Theta}^T\boldsymbol{x^{(i)}} - \boldsymbol{y^{(i)}})_j x_p^{(i)}$$

b) The following are definitions for $\boldsymbol{\Theta}$, $\boldsymbol{X}$, and $\boldsymbol{Y}$:

$$\boldsymbol{\Theta}^T = \begin{bmatrix}\theta_{01} & \theta_{11} & \dots & \theta_{(n-1)1}\\ \theta_{02} & \theta_{12} & \dots & \theta_{(n-1)2}\\ \vdots & \vdots & \ddots & \vdots\\ \theta_{0k} & \theta_{1k} & \dots & \theta_{(n-1)k}\end{bmatrix} \in \Re^{k\times n}$$

$$\boldsymbol{X} = \begin{bmatrix}1 & 1 & \dots & 1\\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)}\\ \vdots & \vdots & \ddots & \vdots\\ x_{n-1}^{(1)} & x_{n-1}^{(2)} & \dots & x_{n-1}^{(m)}\end{bmatrix} \in \Re^{n\times m}$$

$$\boldsymbol{Y} = \begin{bmatrix}y_0^{(1)} & y_0^{(2)} & \dots & y_0^{(m)}\\ y_1^{(1)} & y_1^{(2)} & \dots & y_1^{(m)}\\ \vdots & \vdots & \ddots & \vdots\\ y_k^{(1)} & y_k^{(2)} & \dots & y_k^{(m)}\end{bmatrix} \in \Re^{k\times m}$$

where k is the number of labels (6), m is the number of training images, and n is one more than the number of pixels per image (1025).

We can compute $(\boldsymbol{\Theta}^T\boldsymbol{X} - \boldsymbol{Y})^T$:

$$(\boldsymbol{\Theta}^T\boldsymbol{X} - \boldsymbol{Y})^T = \begin{bmatrix} \boldsymbol{\Theta}^T\boldsymbol{x^{(1)}} - \boldsymbol{y^{(1)}} \\ \boldsymbol{\Theta}^T\boldsymbol{x^{(2)}} - \boldsymbol{y^{(2)}} \\ \vdots \\ \boldsymbol{\Theta}^T\boldsymbol{x^{(m)}} - \boldsymbol{y^{(m)}} \end{bmatrix}$$

where $\boldsymbol{x^{(i)}}$ and $\boldsymbol{y^{(i)}}$ are the i^{th} columns of $\boldsymbol{X}$ and $\boldsymbol{Y}$, respectively.

Multiplying $\boldsymbol{X}$ by this result yields:

$$\boldsymbol{X}(\boldsymbol{\Theta}^T\boldsymbol{X} - \boldsymbol{Y})^T = \begin{bmatrix} \boldsymbol{x^{(1)}} & \boldsymbol{x^{(2)}} & \dots & \boldsymbol{x^{(m)}} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Theta}^T\boldsymbol{x^{(1)}} - \boldsymbol{y^{(1)}} \\ \boldsymbol{\Theta}^T\boldsymbol{x^{(2)}} - \boldsymbol{y^{(2)}} \\ \vdots \\ \boldsymbol{\Theta}^T\boldsymbol{x^{(m)}} - \boldsymbol{y^{(m)}} \end{bmatrix}$$

This expands to:

$$\boldsymbol{X}(\boldsymbol{\Theta}^T\boldsymbol{X} - \boldsymbol{Y})^T = \boldsymbol{x^{(1)}}(\boldsymbol{\Theta}^T\boldsymbol{x^{(1)}} - \boldsymbol{y^{(1)}}) + \boldsymbol{x^{(2)}}(\boldsymbol{\Theta}^T\boldsymbol{x^{(2)}} - \boldsymbol{y^{(2)}}) + ... + \boldsymbol{x^{(m)}}(\boldsymbol{\Theta}^T\boldsymbol{x^{(m)}} - \boldsymbol{y^{(m)}})$$

Each element in this matrix is equal to the partial derivative found in part a) (without the coefficient 2). As such,

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\Theta}} = 2\boldsymbol{X}(\boldsymbol{\Theta}^T\boldsymbol{X} - \boldsymbol{Y})^T$$

c) The cost function is implemented as follows in `f_v()`:

```
def f_v(X, Y, THETA, m):
    '''Returns the result of the cost function. The cost function is
    (THETA.T*X-Y)^2 (element wise square) summed vertically and then
    horizontally.
    Arguments:
    X -- matrix of size n x m containing image data
    Y -- array of size k x m containing classification data
    THETA -- array of size n x k containing theta parameters to be minimized
    where:
    n -- number of pixels per image + 1
    m -- number of training examples
    k -- number of possible labels
    '''
    return (1./(2*m))*sum(sum((dot(THETA.T, X) - Y)**2, 0))
```

The vectorized gradient is implemented as follows in `df_v()`:

```
def df_v(X, Y, THETA, m):
    '''Returns the vector gradient of the cost function.
    Arguments:
    X -- matrix of size n x m containing image data
    Y -- array of size k x m containing classification data
    THETA -- array of size n x k containing theta parameters to be minimized
    where:
    n -- number of pixels per image + 1
    m -- number of training examples
    k -- number of possible labels
    '''
    return (1./m)*dot(X, (dot(THETA.T, X) - Y).T)
```

In this implementation, the coefficient of the cost function, $\frac{1}{2m}$, where m is the number of images in the training set, has been included. This allows for efficient computation times and increased performance.

d) The finite difference gradient is computed using the following code. At each θ_{ij}, the partial derivative is calculated and placed in the gradient. The finite difference computation is shown below:

```
def finite_diff(X, Y, THETA, m, h):
    '''Returns the finite difference calculation of the gradient of the cost
    function defined by f_v.
    X -- matrix of size n x m containing image data
    Y -- array of size k x m containing classification data
    THETA -- array of size n x k containing theta parameters to be minimized
    h -- step size
    where:
    n -- number of pixels per image + 1
    m -- number of training examples
    k -- number of possible labels
    '''
    G = np.zeros((1025, 6))
    for i in range(1025):
        for j in range(6):
            H = np.zeros((1025, 6))
            H[i,j] = h
            # compute the partial derivative of f with respect to theta_ij and
            # save it in the gradient G
            G[i,j] = (f_v(X, Y, THETA + H, m) - f_v(X, Y, THETA, m))/(h*1.)
    return G
```

To identify the similarity between the analytically computed gradient and the numerically computed one through the finite difference method, we take the norm of the difference between the two. This is plotted against varying step sizes used in the finite difference computation, as shown in Figure 6.

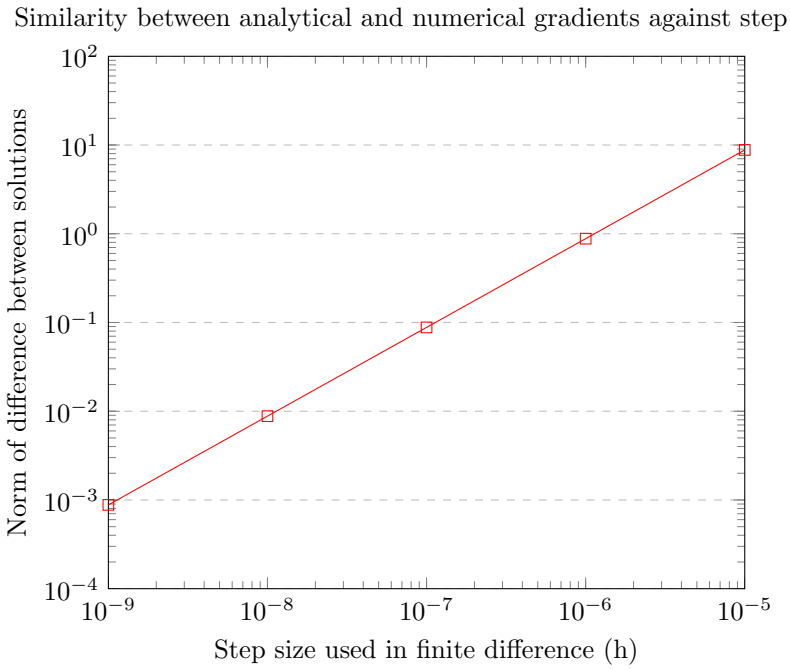


Figure 6

We see that as the step size used for the numerical computation of the gradient decreases, the difference between the analytical and numerical solutions decreases as well.

Part 7

Performance of facial recognition classifier.
Using the set of actors/actresses in `act`, with 100 images of each actor/actress, training was executed with gradient descent. The performance of the classifier is shown in Figure 7.

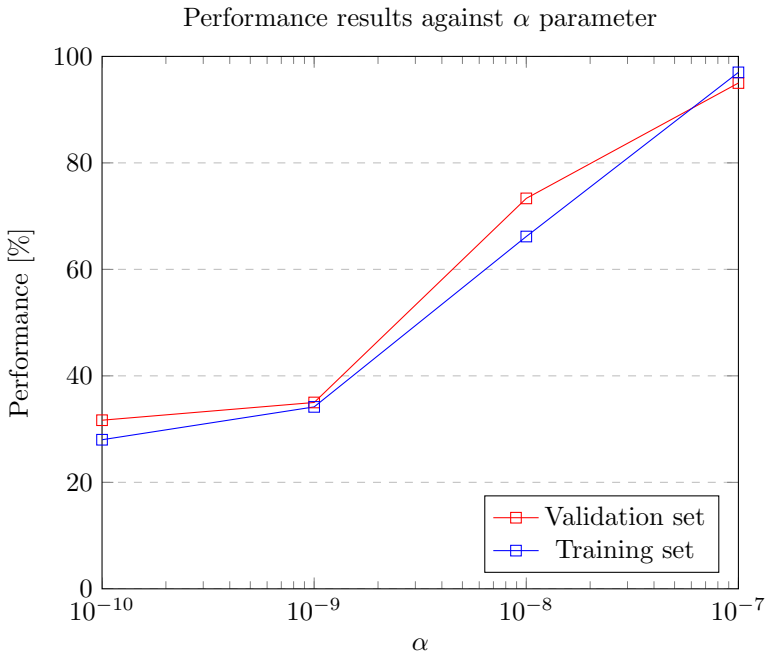
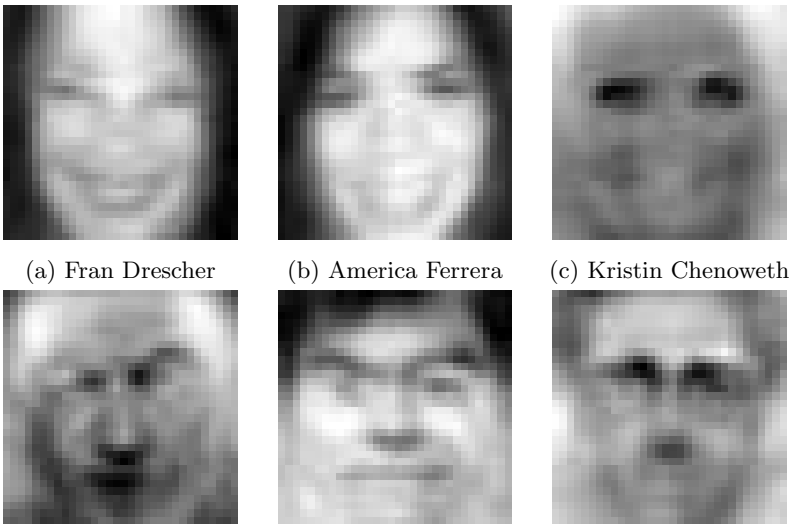


Figure 7

We observe the classifier performs better as α increases. This is expected as the number of iterations also increases. A small α gets gradient descent to converge to a local minimum, resulting in a large cost. As the α value is increased, these local minima are surmounted and the convergence to a lower cost results in better performance. We note when $\alpha = 1 \times 10^{-7}$, the performance of the training set is better than that of the validation set for the first time. This may be indicative of overfitting.

Part 8

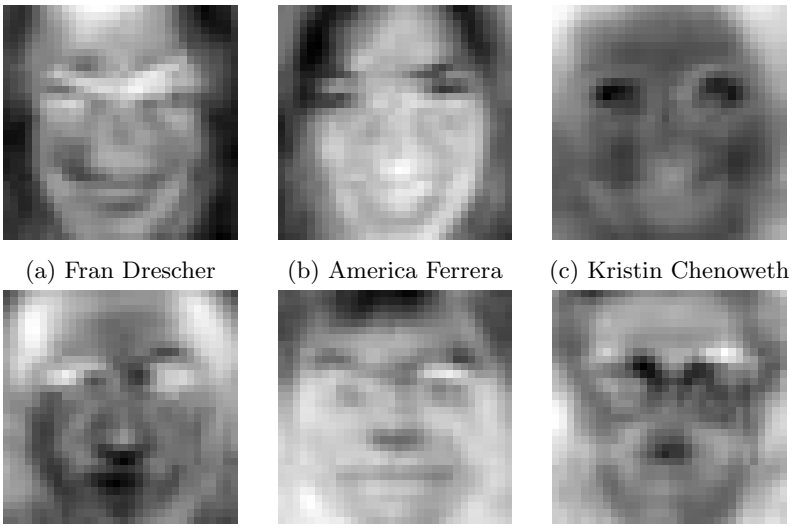
Visualizing optimized θ parameters with multiple classifications.
Figure 8 shows the images of the theta parameters (extracted as in part 4). The actor/actress associated with the image is labeled. This result is observed when running with $\alpha = 4 \times 10^{-9}$, which yields a performance of 50.0% on the training set and 56.7% on the validation set. Figure 9 and Figure 10 show the results for larger α values. Notice the decrease in fluency of the actors'/actresses' faces as a result of a better fitting and overfitting. The performance increases as expected since there are more iterations, reaching over 95.0% performance on both sets at $\alpha = 1 \times 10^{-7}$.



(a) Fran Drescher (b) America Ferrera (c) Kristin Chenoweth

(d) Alec Baldwin (e) Bill Hader (f) Steve Carell

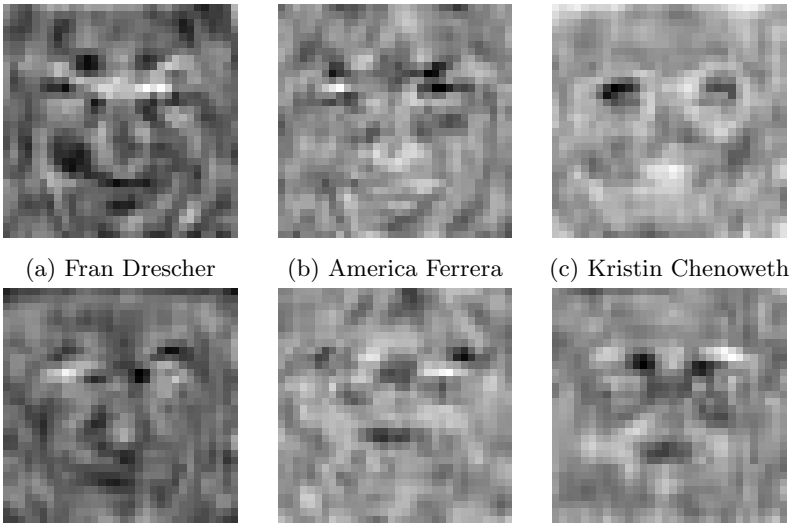
Figure 8: Visualizations of the θs using $\alpha = 4 \times 10^{-9}$ with performance of 50.0% (test) and 56.7% (validation). The cost is 0.41 ending on iteration 43.



(a) Fran Drescher (b) America Ferrera (c) Kristin Chenoweth

(d) Alec Baldwin (e) Bill Hader (f) Steve Carell

Figure 9: Visualizations of the θs using $\alpha = 1 \times 10^{-8}$ with performance of 66.2% (test) and 73.3% (validation). The cost is 0.32 ending on iteration 391.



(a) Fran Drescher (b) America Ferrera (c) Kristin Chenoweth

(d) Alec Baldwin (e) Bill Hader (f) Steve Carell

Figure 10: Visualizations of the θs using $\alpha = 1 \times 10^{-7}$ with performance of 97.0% (test) and 95.0% (validation). The cost is 0.12 ending on iteration 3550.

Appendix A: Code

```
'''CSC411: Assignment 1
Due: Friday, February 03, 2017
Author: Zi Mo Su (1001575048)
'''


#-----------------------------------IMPORTS-------------------------------------
from pylab import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import random
import time
from scipy.misc import imread
from scipy.misc import imresize
from scipy.misc import imsave
import matplotlib.image as mpimg
import os
from scipy.ndimage import filters
import urllib


#-----------------------------------FUNCTIONS-----------------------------------
# PART 1 ---------------------------------------------------------------------
def timeout(func, args=(), kwargs={}, timeout_duration=1, default=None):
    '''From:
    http://code.activestate.com/recipes/473878-timeout-function-using-threading/
    '''
    import threading
    class InterruptableThread(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.result = None

        def run(self):
            try:
                self.result = func(*args, **kwargs)
            except:
                self.result = default

    it = InterruptableThread()
    it.start()
    it.join(timeout_duration)
    if it.isAlive():
        return False
    else:
        return it.result

def crop(img, coords):
    '''Return cropped RGB image, represented as a numpy array of size n x m x 3.
    Image is cropped according to coordinates given in data file.
```

```
    Arguments:
    img -- raw uncropped image file.
    coords -- coordinates of the crop in the form [x1, y1, x2, y2], where
    (x1, y1) is the top-left pixel and (x2, y2) is the bottom right pixel.
    '''
    # create image array
    img_array = imread(img, mode="RGB")
    return img_array[int(coords[1]):int(coords[3]), int(coords[0]):int(coords[2])]

def rgb2gray(rgb):
    '''Return the grayscale version of the RGB image rgb as a 2D numpy array
    whose range is 0..1.
    Arguments:
    rgb -- an RGB image, represented as a numpy array of size n x m x 3. The
    range of the values is 0..255
    '''
    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b
    return gray/255.

def create_set(act):
    '''Return set containing all pictures of each actor/actress in the list act
    in the form of a dictionary with key-value pairs {(name###):(2D array)}.
    Arguments:
    act -- list of actors/actresses
    '''
    testfile = urllib.URLopener()
    set = {}

    # loop through each actor/actress in act
    for a in act:
        name = a.split()[1].lower()
        i = 0

        # loop through each line in the raw data (every image of every
        # actor/actress is in 'faces_subset.txt'
        for line in open("faces_subset.txt"):
            if a in line:
                # filename is of the form: '[name][###].[ext]'
                filename = name+str(i).zfill(3)+'.'+line.split()[4].split('.')[-1]
                # A version without timeout (uncomment in case you need to
                # unsupress exceptions, which timeout() does)
                # testfile.retrieve(line.split()[4], "uncropped/"+filename)
                # timeout is used to stop downloading images which take too long to
                    download
                timeout(testfile.retrieve, (line.split()[4], "uncropped/"+filename),
                    {}, 30)

                # remove images that are unreadable
                try:
                    imread("uncropped/"+filename)
                except IOError:
                    if os.path.isfile("uncropped/"+filename):
```

```
                        os.remove("uncropped/"+filename)
                    print "IOError"
                    continue

                if not os.path.isfile("uncropped/"+filename):
                    continue

                print filename

                # crop image
                coords = line.split("\t")[4].split(",")
                img = crop("uncropped/"+filename, coords)
                # imsave("cropped/crop_"+filename, img)

                # convert image to grayscale
                img = rgb2gray(img)
                # imsave("gray/gray_"+filename, img)

                # resize image to 32 x 32
                try:
                    img = imresize(img, (32,32))
                except ValueError:
                    os.remove("uncropped/"+filename)
                    print "ValueError"
                    continue

                # imsave("resized/rs_"+filename, img)

                # store image in set
                set[name+str(i).zfill(3)] = img

                i += 1

    return set

# PART 2 ---------------------------------------------------------------------
def partition(act, set, train_size = 100, val_size = 10, test_size = 10):
    '''Returns three subsets from the set of all actors/actresses; training,
    validation, and test sets. Each subset's size can be specified and the list
    of actors/actresses to retain in the subset is specified.
    Arguments:
    act -- list of actors/actresses to retain in subsets
    set -- input set of all actors/actresses
    train_size -- number of images per actor/actress to retain in training set
    val_size -- number of images per actor/actress to retain in validation set
    test_size -- number of images per actor/actress to retain in test set
    '''
    val_set = {}
    test_set = {}
    train_set = {}

    # loop through actors/actresses to be retained
    for a in act:
```

```
        name = a.split()[1].lower()

        # loop through each element in the dictionary set
        for filename in set:
            if name in filename:

                # save first val_size number of actors to validation set
                if int(filename[-3:]) < val_size:
                    val_set[filename] = set[filename]

                # save next test_size number of actors to test set
                elif int(filename[-3:]) < val_size + test_size:
                    test_set[filename] = set[filename]

                # save next train_size number of actors to training set
                elif int(filename[-3:]) < val_size + test_size + train_size:
                    train_set[filename] = set[filename]

                else:
                    continue

    return train_set, val_set, test_set

# PART 3 -----------------------------------------------------------------------
def f(X, y, t, m):
    '''Returns the result of the cost function.
    Arguments:
    X -- matrix of size m x 1025 containing image data
    y -- array of size 1 x m containing -1s and 1s depending on classification
    t -- array of size 1 x 1025 containing theta parameters
    m -- the number of images
    '''
    return 1/(2.*m)*sum((y.T - dot(X, t.T))**2)

def df(X, y, t, m):
    '''Returns the gradient of the cost function.
    Arguments:
    X -- matrix of size m x 1025 containing image data
    y -- array of size 1 x m containing -1s and 1s depending on classification
    t -- array of size 1 x 1025 containing theta parameters
    m -- the number of images
    '''
    return -1./m*sum((y.T - dot(X, t.T))*X, 0)

def make_row(arr):
    '''Returns a 1 x 1025 array representing the image. The first element is 1
    and the next 1024 are the pixels of the image read from left to right and
    top to bottom.
    Arguments:
    arr -- 2D numpy array, of size 32 x 32, that represents the image
    '''
    # reshape x into (1 x 1024) and insert 1 at beginning
    x = np.reshape(arr, 1024)
```

```
    x = np.hstack((array([1]), x))
    return x

def grad_descent(X, y, init_t, alpha, m, vector = False):
    '''Returns the theta parameters for which the cost function is minimized,
    through numerical computation using gradient descent.
    Arguments:
    X -- matrix containing image data
    y -- array containing -1s and 1s or 1 x 6 arrays if vectorized, depending on
         classification
    init_t -- array containing initial theta parameters
    alpha -- parameter used for gradient descent, 'learning rate'
    m -- the number of images
    vector -- set to True to enable vectorized gradient descent
    '''
    EPS = 1e-6
    prev_t = init_t - 10*EPS
    t = init_t.copy()
    max_iter = 100000
    iter = 0

    while norm(t-prev_t) > EPS and iter < max_iter:
        prev_t = t.copy()
        if vector == True:
            t -= alpha*df_v(X, y, t, m)
        else:
            t -= alpha*df(X, y, t, m)
        iter += 1

    if vector == True:
        cost = f_v(X, y, t, m)
    else:
        cost = f(X, y, t, m)

    print "Minimum found at", t, "with cost function value of", cost, "on iteration",
        iter
    return t

def train_hc(train_set, a, m, run_gd = True):
    '''Returns the theta vector for which the cost function is minimized, if
    run_gd is True, otherwise the constructed 2D array X and the array y are
    returned.
    Arguments:
    train_set -- training set for hader/carell classification, if run_gd is
                 False, this set is just the set used to create X and y
    a -- value of alpha for gradient descent
    m -- total number of images
    run_gd -- set to False if gradient descent is not desired
    '''
    X = []
    y = array([])

    for filename in train_set:
```

```
        # hader classified as -1
        if "hader" in filename:
            x = make_row(train_set[filename])
            y_temp = -1
        # carell classified as 1
        elif "carell" in filename:
            x = make_row(train_set[filename])
            y_temp = 1
        else:
            continue

        if X == []:
            X = array([x])
        else:
            X = np.append(X, [x], 0)

        y = np.append(y, y_temp)

    # initial theta
    t0 = array([np.zeros(1025)])

    y = array([y])

    if run_gd == True:
        return grad_descent(X, y, t0, a, m)
    else:
        return X, y

def performance_hc(set, t):
    '''Evaluates the performance of a given set for hader/carell classification.
    Arguments:
    set -- set to be evaluated
    t -- theta vector generated through training
    '''
    right = 0
    wrong = 0

    for filename in set:
        if "hader" in filename:
            x = make_row(set[filename])
            # x*t < 0 -> -1 (hader)
            if dot(x, t.T) < 0:
                right += 1
            else:
                wrong += 1

        if "carell" in filename:
            x = make_row(set[filename])
             # x*t > 0 -> 1 (carell)
            if dot(x, t.T) >= 0:
                right += 1
            else:
                wrong += 1
```

```
    if set == train_set:
        set = "Training set"
    elif set == val_set:
        set = "Validation set"
    else:
        set = "Testing set"

    print set, "is", str(right/((right + wrong)*0.01)) + "% accurate"

# PART 4 ---------------------------------------------------------------------
def get_rn(min, max, n):
    '''Returns array of n random numbers (no repeats) between min and max
    (inclusive).
    Arguments:
    min -- minimum number that can be chosen
    max -- maximum number that can be chosen
    n -- number of numbers chosen
    '''
    arr = np.arange(min, max+1)
    np.random.shuffle(arr)
    return arr[:n]

# PART 5 ---------------------------------------------------------------------
def train_gender(train_set, a, n):
    '''Returns the theta vector for which the cost function is minimized by
    using the training set to train the gender classifier. The number of
    training examples used is specified by n per actor/actress for a total of
    6n.
    Arguments:
    train_set -- training set
    a -- alpha value used for gradient descent
    n -- number of each actor/actress, total size of 6n
    '''

    X = []
    Y = array([])

    # males are classified as -1 and females as 1
    for filename in train_set:
        if "baldwin" in filename and int(filename[-3:])-20 < n:
            x = make_row(train_set[filename])
            y = -1
        elif "hader" in filename and int(filename[-3:])-20 < n:
            x = make_row(train_set[filename])
            y = -1
        elif "carell" in filename and int(filename[-3:])-20 < n:
            x = make_row(train_set[filename])
            y = -1
        elif "drescher" in filename and int(filename[-3:])-20 < n:
            x = make_row(train_set[filename])
            y = 1
        elif "ferrera" in filename and int(filename[-3:])-20 < n:
```

```
            x = make_row(train_set[filename])
            y = 1
        elif "chenoweth" in filename and int(filename[-3:])-20 < n:
            x = make_row(train_set[filename])
            y = 1
        else:
            continue

        if X == []:
            X = array([x])
        else:
            X = np.append(X, [x], 0)

        Y = np.append(Y, y)

    # make y correct dimension (1 x 6n)
    Y = array([Y])

    # initialize theta (1 x 1025)
    t0 = array([np.zeros(1025)])

    # Note: X has dimensions 6n x 1025

    return grad_descent(X, Y, t0, a, 6*n)

def performance_gender(set, t):
    '''Evaluates the performance of a given set for gender classification.
    Arguments:
    set -- set to be evaluated
    t -- theta vector generated through training
    '''
    right = 0
    wrong = 0

    for filename in set:
        # male
        if "baldwin" in filename or "hader" in filename or "carell" in filename or "
            butler" in filename or "radcliffe" in filename or "vartan" in filename:
            x = make_row(set[filename])
            # x*t < 0 -> -1 (male)
            if dot(x, t.T) < 0:
                right += 1
            else:
                wrong += 1

        else:
            x = make_row(set[filename])
            # x*t > 0 -> 1 (female)
            if dot(x, t.T) >= 0:
                right += 1
            else:
                wrong += 1
```

```
    if set == train_set:
        set = "Training set"
    elif set == val_set:
        set = "Validation set"
    else:
        set = "Testing set"

    print set, "is", str(right/((right + wrong)*0.01)) + "% accurate"

# PART 6 ---------------------------------------------------------------------
def f_v(X, Y, THETA, m):
    '''Returns the result of the cost function. The cost function is
    (THETA.T*X-Y)^2 (element wise square) summed vertically and then
    horizontally.
    Arguments:
    X -- matrix of size n x m containing image data
    Y -- array of size k x m containing classification data
    THETA -- array of size n x k containing theta parameters to be minimized
    where:
    n -- number of pixels per image + 1
    m -- number of training examples
    k -- number of possible labels
    '''
    return (1./(2*m))*sum(sum((dot(THETA.T, X) - Y)**2, 0))

def df_v(X, Y, THETA, m):
    '''Returns the vector gradient of the cost function.
    Arguments:
    X -- matrix of size n x m containing image data
    Y -- array of size k x m containing classification data
    THETA -- array of size n x k containing theta parameters to be minimized
    where:
    n -- number of pixels per image + 1
    m -- number of training examples
    k -- number of possible labels
    '''
    return (1./m)*dot(X, (dot(THETA.T, X) - Y).T)

def finite_diff(X, Y, THETA, m, h):
    '''Returns the finite difference calculation of the gradient of the cost
    function defined by f_v.
    X -- matrix of size n x m containing image data
    Y -- array of size k x m containing classification data
    THETA -- array of size n x k containing theta parameters to be minimized
    h -- step size
    where:
    n -- number of pixels per image + 1
    m -- number of training examples
    k -- number of possible labels
    '''
    G = np.zeros((1025, 6))
    for i in range(1025):
        for j in range(6):
```

```
            H = np.zeros((1025, 6))
            H[i,j] = h
            # compute the partial derivative of f with respect to theta_ij and
            # save it in the gradient G
            G[i,j] = (f_v(X, Y, THETA + H, m) - f_v(X, Y, THETA, m))/(h*1.)
    return G

# PART 7 -------------------------------------------------------------------------
def train_fr(train_set, a):
    '''Returns the theta matrix for which the cost function is minimized using
    the training set to train the face recognition classifier.
    Arguments:
    train_set -- training set
    a -- value of alpha for gradient descent
    '''
    X = []
    Y = []

    for filename in train_set:
        if "drescher" in filename:
            x = make_row(train_set[filename])
            y = array([1,0,0,0,0,0])
        elif "ferrera" in filename:
            x = make_row(train_set[filename])
            y = array([0,1,0,0,0,0])
        elif "chenoweth" in filename:
            x = make_row(train_set[filename])
            y = array([0,0,1,0,0,0])
        elif "baldwin" in filename:
            x = make_row(train_set[filename])
            y = array([0,0,0,1,0,0])
        elif "hader" in filename:
            x = make_row(train_set[filename])
            y = array([0,0,0,0,1,0])
        elif "carell" in filename:
            x = make_row(train_set[filename])
            y = array([0,0,0,0,0,1])
        else:
            continue

        if X == []:
            X = array([x])
        else:
            X = np.append(X, [x], 0)

        if Y == []:
            Y = array([y])
        else:
            Y = np.append(Y, [y], 0)

    X = X.T  # n x m = 1025 x 600
    Y = Y.T  # k x m = 6 x 600
```

```
    T0 = np.zeros((1025, 6))  # n x k = 1025 x 6

    return grad_descent(X, Y, T0, a, 600, True), X, Y, T0

def performance_fr(set, T):
    '''Evaluates the performance of a given set for facial recognition
    classification.
    Arguments:
    set -- set to be evaluated
    T -- theta matrix generated through training
    '''
    right = 0
    wrong = 0

    for filename in set:
        # drescher
        if "drescher" in filename:
            x = make_row(set[filename])
            # (T.T*X)[0] is max in T.T*X ->  (drescher)
            if max(dot(T.T, x.T)) == dot(T.T, x.T)[0]:
                right += 1
            else:
                wrong += 1
        # ferrera
        elif "ferrera" in filename:
            x = make_row(set[filename])
            # (T.T*X)[1] is max in T.T*X ->  (ferrera)
            if max(dot(T.T, x.T)) == dot(T.T, x.T)[1]:
                right += 1
            else:
                wrong += 1
        # chenoweth
        elif "chenoweth" in filename:
            x = make_row(set[filename])
            # (T.T*X)[2] is max in T.T*X ->  (chenoweth)
            if max(dot(T.T, x.T)) == dot(T.T, x.T)[2]:
                right += 1
            else:
                wrong += 1
        # baldwin
        elif "baldwin" in filename:
            x = make_row(set[filename])
            # (T.T*X)[3] is max in T.T*X ->  (baldwin)
            if max(dot(T.T, x.T)) == dot(T.T, x.T)[3]:
                right += 1
            else:
                wrong += 1
        # hader
        elif "hader" in filename:
            x = make_row(set[filename])
            # (T.T*X)[4] is max in T.T*X ->  (hader)
            if max(dot(T.T, x.T)) == dot(T.T, x.T)[4]:
                right += 1
```

```
            else:
                wrong += 1
        # carell
        elif "carell" in filename:
            x = make_row(set[filename])
            # (T.T*X)[5] is max in T.T*X ->  (carell)
            if max(dot(T.T, x.T)) == dot(T.T, x.T)[5]:
                right += 1
            else:
                wrong += 1
        else:
            continue

    if set == train_set:
        set = "Training set"
    elif set == val_set:
        set = "Validation set"
    else:
        set = "Testing set"

    print set, "is", str(right/((right + wrong)*0.01)) + "% accurate"


#-----------------------------------MAIN CODE-------------------------------------
act = ["Fran Drescher", "America Ferrera", "Kristin Chenoweth", "Alec Baldwin", "Bill
    Hader", "Steve Carell"]
act_test = ["Gerard Butler", "Daniel Radcliffe", "Michael Vartan", "Lorraine Bracco",
    "Peri Gilpin", "Angie Harmon"]

# PART 1 ---------------------------------------------------------------------
print "Running Part 1..."
if not os.path.exists("act_set.npy"):
    act_set = create_set(act)
    np.save("act_set.npy", act_set)

else:
    act_set = np.load("act_set.npy").item()
print "Part 1 Complete!\n"

# PART 2 ---------------------------------------------------------------------
print "Running Part 2..."
train_set, val_set, test_set = partition(act, act_set)
print "Part 2 Complete!\n"

# PART 3 ---------------------------------------------------------------------
print "Running Part 3..."
t = train_hc(train_set, a=1e-7, m=200)
X, y = train_hc(train_set, a=1e-7, m=20, run_gd=False)

print "Cost for the validation set is:", f(X, y, t, m=20)

performance_hc(train_set, t)
performance_hc(val_set, t)
```

```
performance_hc(test_set, t)
print "Part 3 Complete!\n"

# PART 4 ------------------------------------------------------------------------
print "Running Part 4..."
# training using the full training set
# first remove theta_0 at front of t
t = t[:,1:]
t = np.reshape(t, (32, 32))
imsave("part4_1.jpg", t)

# training using 2 images of each actor, randomly selected
rn = get_rn(20, 119, 4)
train_set = {"hader"+str(rn[0]).zfill(3):train_set["hader"+str(rn[0]).zfill(3)],
             "hader"+str(rn[1]).zfill(3):train_set["hader"+str(rn[1]).zfill(3)],
             "carell"+str(rn[2]).zfill(3):train_set["carell"+str(rn[2]).zfill(3)],
             "carell"+str(rn[3]).zfill(3):train_set["carell"+str(rn[3]).zfill(3)],}

t = train_hc(train_set, a=1e-8, m=4)
t = t[:,1:]
t = np.reshape(t, (32, 32))
imsave("part4_2.jpg", t)
print "Part 4 Complete!\n"

# PART 5 ------------------------------------------------------------------------
print "Running Part 5..."
for i in range(10):
    train_set, val_set, test_set = partition(act, act_set, 10*(i+1))
    t = train_gender(train_set, a=1e-7, n=10*(i+1))
    print "For training set of size", str(10*(i+1)) +":"
    performance_gender(train_set, t)
    performance_gender(val_set, t)
    performance_gender(test_set, t)
    print

# test on set of other actors
if not os.path.exists("act_test_set.npy"):
    act_test_set = create_set(act_test)
    np.save("act_test_set.npy", act_test_set)

else:
    act_test_set = np.load("act_test_set.npy").item()

train_set, val_set, test_set = partition(act, act_set, 100)
t = train_gender(train_set, a=1e-7, n=100)
performance_gender(act_test_set, t)
print "Part 5 Complete!\n"

# PART 6 ------------------------------------------------------------------------
print "Running Part 6..."
train_set, val_set, test_set = partition(act, act_set)
T, X, Y, T0 = train_fr(train_set, a=1e-7)
for i in range(5):
```

```
    X_test = X
    Y_test = Y
    T_test = T0
    grad_analytic = df_v(X_test, Y_test, T_test, m=600)
    grad_fd = finite_diff(X_test, Y_test, T_test, m=600, h=10**-(5+i))
    print "For h =", str(10**-(5+i)) + ":"
    print "The norm of the difference between the analytical and finite difference
        gradient is:", norm(grad_fd-grad_analytic)
    print
print "Part 6 Complete!\n"

# PART 7 -------------------------------------------------------------------------
print "Running Part 7..."
performance_fr(train_set, T)
performance_fr(val_set, T)
print "Part 7 Complete!\n"

# PART 8 -------------------------------------------------------------------------
print "Running Part 8..."
for i in range(6):
    t = T[1:,i]
    t = np.reshape(t, (32, 32))
    imsave("part8_"+str(i+1)+".jpg", t)
print "Part 8 Complete!\n"
```

Appendix B: Results

```
Running Part 1...
Part 1 Complete!

Running Part 2...
Part 2 Complete!

Running Part 3...
Minimum found at [[  6.80763797e-07   3.01956922e-05   1.04435244e-04 ...,
    -4.48283445e-05
    1.98460589e-04  -2.38103923e-04]] with cost function value of 0.0419727337978 on
        iteration 3369
Cost for the validation set is: 0.419727337978
Training set is 100.0% accurate
Validation set is 95.0% accurate
Testing set is 75.0% accurate
Part 3 Complete!

Running Part 4...
Minimum found at [[ -1.80449643e-07  -5.38147815e-06  -8.38475676e-06 ...,
    1.90012646e-04
    1.82475301e-04   1.06938873e-04]] with cost function value of 0.0206196529755 on
        iteration 469
Part 4 Complete!

Running Part 5...
Minimum found at [[ -3.03474137e-06   4.55245129e-04   4.47290202e-04 ...,
    -4.56429193e-04
   -2.75588940e-04  -3.26083029e-04]] with cost function value of 0.0125160995913 on
       iteration 2348
For training set of size 60:
Training set is 100.0% accurate
Validation set is 90.0% accurate
Testing set is 75.0% accurate

Minimum found at [[ -3.21151352e-06   4.70412415e-04   3.07065090e-04 ...,
    -3.45775522e-04
   -1.39476073e-04  -7.24854449e-05]] with cost function value of 0.0364053788428 on
       iteration 3612
For training set of size 120:
Training set is 99.1666666667% accurate
Validation set is 96.6666666667% accurate
Testing set is 85.0% accurate

Minimum found at [[ -3.66443770e-06   4.56268461e-04   2.92813951e-04 ...,
    -4.95099320e-04
   -8.06166283e-05   1.87309981e-04]] with cost function value of 0.0547274681447 on
       iteration 3465
For training set of size 180:
Training set is 99.4444444444% accurate
Validation set is 98.3333333333% accurate
Testing set is 83.3333333333% accurate
```

```
Minimum found at [[ -4.42673049e-06   4.19265197e-04   2.99696579e-04 ...,
    -4.34780186e-04
   -1.22231671e-04   2.21818762e-06]] with cost function value of 0.0639899318247 on
       iteration 3333
For training set of size 240:
Training set is 99.1666666667% accurate
Validation set is 98.3333333333% accurate
Testing set is 85.0% accurate

Minimum found at [[ -5.11140580e-06   5.70335105e-04   4.82430743e-04 ...,
    -4.45895396e-04
   -2.62862385e-04   4.32900853e-05]] with cost function value of 0.0727375175841 on
       iteration 3331
For training set of size 300:
Training set is 98.6666666667% accurate
Validation set is 96.6666666667% accurate
Testing set is 86.6666666667% accurate

Minimum found at [[ -5.22418648e-06   5.78117066e-04   5.02516805e-04 ...,
    -4.64446519e-04
   -2.81956497e-04   5.85839228e-06]] with cost function value of 0.0755921666359 on
       iteration 3248
For training set of size 360:
Training set is 98.3333333333% accurate
Validation set is 98.3333333333% accurate
Testing set is 90.0% accurate

Minimum found at [[ -4.89571401e-06   4.47790123e-04   4.28131125e-04 ...,
    -3.94585249e-04
   -2.41023302e-04   4.08865817e-05]] with cost function value of 0.0801034201224 on
       iteration 3210
For training set of size 420:
Training set is 97.8571428571% accurate
Validation set is 98.3333333333% accurate
Testing set is 91.6666666667% accurate

Minimum found at [[ -3.59944050e-06   4.65126774e-04   3.93377435e-04 ...,
    -3.40651486e-04
   -1.97472794e-04   6.63474698e-05]] with cost function value of 0.0877833662506 on
       iteration 3153
For training set of size 480:
Training set is 97.7083333333% accurate
Validation set is 98.3333333333% accurate
Testing set is 90.0% accurate

Minimum found at [[ -3.44472179e-06   4.87683710e-04   3.83120205e-04 ...,
    -3.11265623e-04
   -1.62909563e-04   9.71856382e-05]] with cost function value of 0.0904268591769 on
       iteration 3028
For training set of size 540:
Training set is 97.4074074074% accurate
Validation set is 98.3333333333% accurate
```

```
Testing set is 93.3333333333% accurate

Minimum found at [[ -3.39417859e-06   5.09513578e-04   3.93590314e-04 ...,
    -2.62134832e-04
   -1.20781613e-04   6.89684937e-05]] with cost function value of 0.0927739673196 on
       iteration 3047
For training set of size 600:
Training set is 97.8333333333% accurate
Validation set is 98.3333333333% accurate
Testing set is 90.0% accurate

Minimum found at [[ -3.39417859e-06   5.09513578e-04   3.93590314e-04 ...,
    -2.62134832e-04
   -1.20781613e-04   6.89684937e-05]] with cost function value of 0.0927739673196 on
       iteration 3047
Testing on the set of other actors:
Testing set is 87.9768786127% accurate
Part 5 Complete!

Running Part 6...
Minimum found at [[  8.74465206e-07  -1.57996022e-07   7.05656037e-07   7.31652456e-08
    1.71712721e-06   3.31238945e-06]
 [  1.18494190e-04  -7.69766075e-05   1.98933784e-04  -2.30340196e-04
   -6.70514431e-05   5.93270740e-06]
 [ -5.83798104e-06  -3.24165650e-06   1.81479542e-04  -1.44162263e-04
   -4.94730215e-05  -3.32330542e-05]
 ...,
 [ -1.47842411e-04  -1.03129578e-05   2.48570298e-05  -5.86192390e-05
    1.12243084e-04   1.07930302e-04]
 [ -8.40001324e-05  -7.32417267e-05   8.98281409e-05  -1.61615206e-04
    6.60622128e-05   1.69758296e-04]
 [ -2.57448163e-06  -1.54162389e-04   1.81924672e-04  -1.77410961e-04
    1.17079758e-04   1.69324980e-05]] with cost function value of 0.116199948422 on
       iteration 3550
For h = 1e-05:
The norm of the difference between the analytical and finite difference gradient is:
    8.79824978211

For h = 1e-06:
The norm of the difference between the analytical and finite difference gradient is:
    0.879824980611

For h = 1e-07:
The norm of the difference between the analytical and finite difference gradient is:
    0.0879825181068

For h = 1e-08:
The norm of the difference between the analytical and finite difference gradient is:
    0.00879848669397

For h = 1e-09:
The norm of the difference between the analytical and finite difference gradient is:
    0.000882300676455
```

```
Part 6 Complete!

Running Part 7...
Training set is 97.0% accurate
Validation set is 95.0% accurate
Part 7 Complete!

Running Part 8...
Part 8 Complete!
```