CSC411: Assignment 1

Due on Wednesday, February 3, 2017, 11pm

Minh Nguyen

March 18, 2018

Instructions for reproducing the results:
Step 1: Putting all of the following files in the same directory

- *get_data_and_crop.py*
- *auto_pick.py*
- *get_data_and_crop_part5.py*
- *auto_pick_part5.py*
- *faces.py*
- *facescrub_actors.txt*
- *facescrub_actresses.txt*

Step 2: Downloading images

- run *get_data_and_crop.py* to download images of the actors specified in *act*. The script stores the cropped and uncropped images in the folders *cropped/* and *uncropped/*, respectively.
- run *get_data_and_crop_part5.py* to download images of the actors specified in *act_test*. The cropped and uncropped images are stored in the folders *act_test_cropped/* and *act_test_uncropped/*, respectively.
- For either file, after downloading all of the images for male actors in *facescrub_actors.txt*, pleased make small modifcation to download the images for female actors in *facescrub_actresses.txt*.

Step 3: Running the codes

- Each part in the assignment can be tested independently. You can reproduce the results by running the codes in *faces.py*
- The codes in faces.py are divided into sections with comments indicating which parts of the assignment they are belong to. At the end of each section, you will find the function ***def reproduce_part_[i]():***, which is used to reproduce the result for part ***i*** of the assignment. These functions are all self contained.
- For example, you want to check the results of part 7, you first locate the function ***def reproduce_part_7():*** in faces.py, then uncomment the line reproduce_part_7() located right below the function to run it.
- I suggest that you should run only one part at a time, so you will not confuse the results produced by different parts. For instance, if you uncomment the line reproduce_part_7() to read my results for part 7, you better comment out reproduce_part_2(), reproduce_part_3(), ...
- Note that because some of the downloading links are not very stable, you may not have the same set of data that I had, therefore your results may be slightly different from mine.
- For part 5, we are asked to draw plot using the data. My program generated data, and I used Microsoft Excel to draw the graph. This method is accepted by the professor (Piazza @ 276).

Note on LaTeX report: Along with the *faces.tex* files, I submitted a zip file called *full_report_folder.zip*. It stores all of images that can be used to regenerate *faces.tex*.

Part 1

Dataset description:
From the two lists of URLs, a set of of 1095 images were obtained. The number images was less than the number of links, because some of the links were no longer accessible. The set includes 495 images of male actors and 600 images of female actors. These are pictures for the actors specified in the list *act*. Out of the 495 male actor images, I was able to retrieve 487 cropped-out faces. And for the 600 images of the female actors, I got an additional 574 cropped out faces. Not all images were croppable, because some of the downloaded images were broken or corrupted. The script *get_data_and_crop.py* was used to download and crop the images. The script put the cropped and uncropped images in the folders *cropped* and *uncropped*, respectively. The cropped images are all grayscale of size 32×32 pixels.

Examples images from the original data set:

(a) Ferrera

(b) Baldwin

(c) Carell

(d) Carell

(e) Ferrera

(f) Hader

Figure 1

Corresponding cropped out faces using the bounding box defined for the images:

(a) Ferrera

(b) Baldwin

(c) Carell

(d) Carell

(e) Ferrera

(f) Hader

Figure 2

The defined rounding boxes are accurate for the majority of the images. Some exceptions are found in the example above. The bounding boxes for (b) Baldwin and (e) Ferrera did not cover their entire faces. These

are bad samples and should not be used for analysis.
The cropped out faces are not well aligned because the actors show various facial expressions in their images.

Part 2

From part 1, I already got a set of cropped faces stored in the directory *cropped* by running the script *get_data_and_crop.py*. The format of all of the cropped images is [*actor's last name*][*download's order*], in which *download's order* is the order that the original image was downloaded from the web. I therefore could distinguish the actor images using the file names. But I did not use the downloaded orders of the images to select the data, but instead wrote the script *auto_pick.py* that **randomly** picks a sample of 120 images for each of the actor. The first 100 images in the sample are the training data, the next 10 images are the validation data, and the last 10 images are the test data. The script puts all training data into the folder *part2_data/train_data*, all validation into the folder *part2_data/valid_data* and all test data into the folder *part2_data/test_data*. Both the scripts *get_data_and_crop.py* and *auto_pick.py* will be included in my submission.

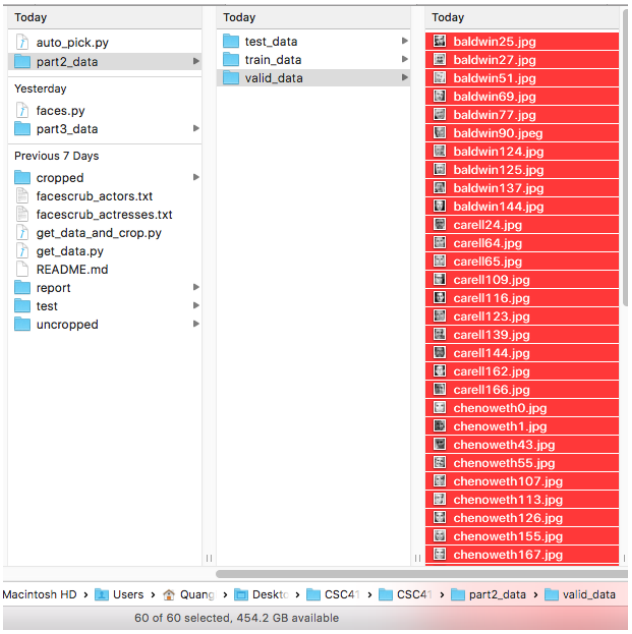


Figure 3: 60 randomly picked images of 6 actors in the validation set

In the sets of images that I picked, few images were not well suited for doing analysis. They were images partitally show the faces of the actors or had inccorect bounding boxes. I replaced these images by other images that had not been picked. Note that because each actor has more than 120 images, there are always some spares that can be used as the substitution. Below are examples of types of images that should not be included.

(a) Original images

(b) Cropped images

Figure 4: Image with incorrect bounding box

(a) Original images

(b) Cropped images

Figure 5: Image shows only one side of the face

(a) Original images

(b) Cropped images

Figure 6: Images show only one side of the face

Part 3

The cost function:

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

Where

- m is the sample's size.
- x is a vector which stores the pixels of the images.
- y is a vector with labels of the images (contains only 0s and 1s).

Cost function and performance report:

- The value of the cost function $f(x)$ for the **validation set** consisting 20 randomly picked images, 10 for each of the two actors is 0.02. The value of the cost function $f(x)$ for the **training set** consisting 200 randomly picked images, 100 for each of the two actors is also 0.02.
- The classifier was built using the test set with 100 randomly picked images for each of the two actors.
- The performance of the classifier on the **validation set** is 17 out of 20 images, a correctness of 85%.
- The classifier correctlly identifies 97.5% of the images in the **training set**.
- On the **test set** with 10 images for each actor, the classifier has a correctness rate of 90%.

Code to compute output of the classifier:

```
def classifier():
    #------------Building the classifier from training data-----------------#
    train_data = os.listdir("part3_data/train/")
    x = array([])
    y = zeros((200,), dtype=np.int)
    i = 0;
    train_size = 0;
    for name in train_data:
        if "carell" in name or "hader" in name:
            train_size = train_size + 1;
            img = imread("part3_data/train/" + name)
            img = rgb2gray(img)
            img = img.flatten()
            if x.size == 0:
                x = vstack([img])
            elif x.size > 0:
                x = vstack([x, img])
            if "hader" in name:
                y[i] = 1

            i = i + 1;
    x = x.T
    theta0 = np.zeros(1025)
    theta = grad_descent(f, df, x, y, theta0, 0.000001, train_size)

```

```
    #--------------Running the classifier to obtain the result---------------#
    test_data = os.listdir("part3_data/test/");
    count_hader = 0; # Number of correct classification for hader.
    count_carell = 0; # Number of correct classification for carell.
    test_size = 0;
    for name in test_data:
        if "carell" in name or "hader" in name:
            test_size = test_size + 1;
            imgtest = imread("part3_data/test/" + name)
            imgtest = rgb2gray(imgtest)
            imgtest = imgtest.flatten()
            result = dot(theta[1:], imgtest) + theta[0] # Hypotheis result
            # print("Hypothesis result: ", result)
            if result > 0.5:
                print('The image {} is classified to be {}'.format(name,'Bill Hader'))
                if "hader" in name: # classification result matches the label
                    count_hader = count_hader + 1;.
            elif result <= 0.5:
                print('The image {} is classified to be {}'.format(name,'Steve Carell'))
                if "carell" in name: # classification result matches the label
                    count_carell = count_carell + 1;
    print(float(count_hader + count_carell)/test_size)*100; # Percentage result
```

To build the classifier, y is set to 1 if the image is of actor Bill Hader, and 0 if the image is of actor Steve Carell. Gradient descent algorithm is then used to find the θ. The classifier do the classification based on the value returned by the hypothesis function $h(x) = \theta^T x$. If the result is > 0.5 the image is classified to be of Bill Hader, else if the result is < 0.5, the image is classified to be of Steve Carell. The full code can be found in the faces.py

Things I considered:
To make the algorithm works, I need an α level that is neither too big or too small. α as defined in class is the step size used in gradient descent algorithm to find the local minimum. We don't want α to be too small, because it would take too many iterations to reach the local minimum. In constrast, if the step size is too big, the algorithm would bypass the optimal point, and diverge to infinity. I also need an intital theta, which is sufficiently small. The inital theta was set to be a vector with only zeros.

Each gray scale images is represented by a numby array in 2-D, but the atrix x in the cost function $f(x)$ only takes 1-D vector for each training sample. So I have to flatten the image by combining all rows in the matrices into one single row, forming a new 1-D vector. This is done by calling *img.flatten()*.

Part 4

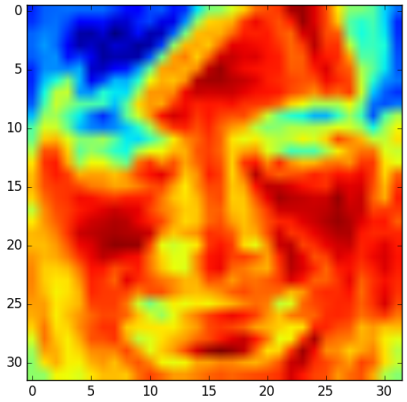


Figure 7: The *theta* image built using a training set contains 4 images, 2 for each of the two actors Steve Carell and Bill Hander.

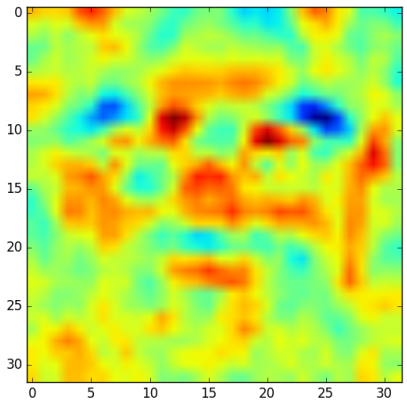


Figure 8: The *theta* image built using a training set contains 200 images, 100 for each of the two actors Steve Carell and Bill Hander.

Part 5

Picking the data:
I created a new script *auto_pick_part5.py* that automatically pick a train set, a validation set and a test set of the sizes I specified. This is the more advanced version of *auto_pick.py* that I created in Part 2. Below is the signature of the function in the script used to pick the images.

```
def pickrandom_p5(actor, source_dir, training_dir, valid_dir, test_dir,
                  train_size, valid_size, test_size):
```

Where:

- **actor**: is the list of actor to be picked from.
- **source_dir**: is the source directory path of th images to be picked from.
- **training_dir**, **valid_dir**, **test_dir** are path to the destination for the training, validation and test sets respectively.
- **train_size**, **valid_size**, **test_size** are sizes of the training, validation and test sets respectively (number of images per actor in the set).

For instance, to randomly pick a training set that has 10 images for each of the actors from the cropped images stored in folder *cropped/*, we do

```
pickrandom_p5(act, "cropped/", "part5_data/train_data_10", "", "", 10, 0, 0); \\
```

Plot the performance of the classifiers on the training and validation sets: The code for this part can be found in *faces.py*. I built the the classifiers using different training set size and recorded the performance of the classifiers on the training and validation sets. The table below shows the recorded data.

Training set size	Performance on the training set (%)	Performance on the validation set (%)
2	66.67	61.66
5	83.30	58.33
10	93.33	70.00
20	95.60	88.30
30	97.70	75.50
40	97.50	85.00
50	97.33	95.00
60	98.05	93.33
70	98.10	86.67
80	98.30	90.00

I then used the data to plot a graph that demonstrates the result of overfitting, that is building a classifier that works well on training data but does not work equally well on unseen data. Note that the training set size is the number of images for each actor in the set.

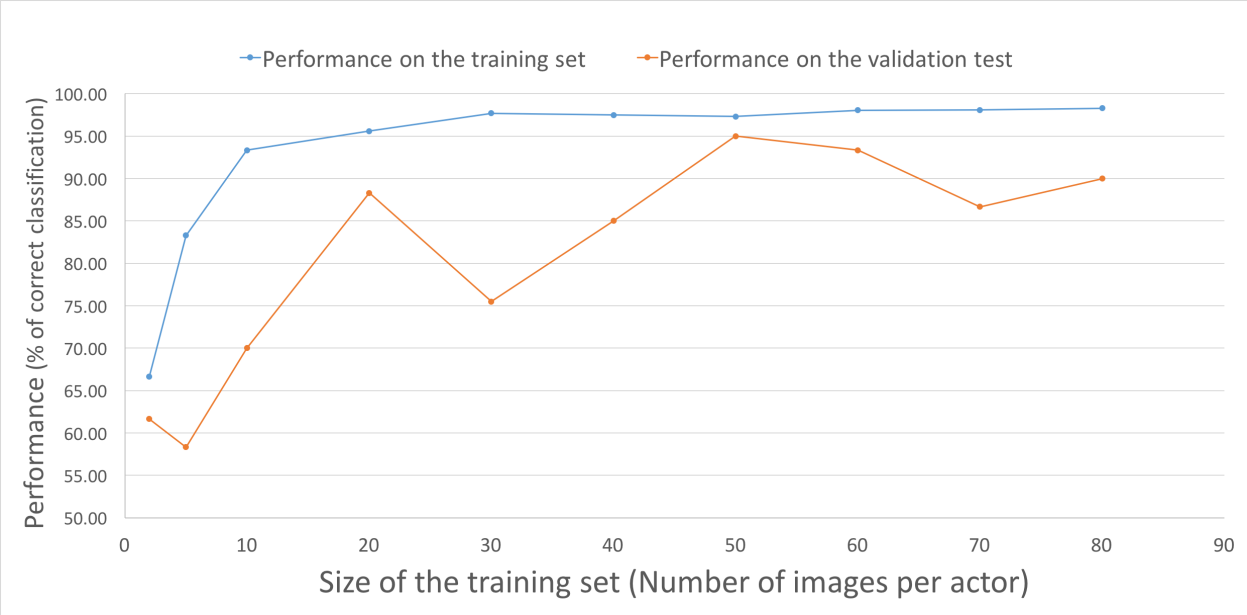


Figure 9: The performance on the training data is always better than on the validation data. However, the performance gap decreases as the the size of the training samples increases. When the training size increases from 20 to 30 images per actor, the peformance on the training set increases whereas the performance on the validation set decreases, an indication that overfitting has happened.

Performance of the classifier on actors in act_test

I modified the script *get_data_and_croppy* created in part 1, to download images of the actors in *act_test* and do face cropping.There were only minor modifications on the names of the destination folders and the list of name of actors to be download. I saved the modified version with the new name *get_data_and_crop_part5.py*, just in case you want to reproduce the data.

I ran the classifier with the training set of size 70 created in the first part on the entire set of cropped images of the actors in *act_test*. The correctness rate was 85.74 %, a 10 % lower than the performance of the classifier on *act*.

Part 6

(a)

We have $\theta_{pq} = \theta_{qp}^T$, so we have the following equality:

$$\sum_i^m (\sum_j^k (\theta^T x^{(i)} - y^{(i)})_j^2) = \sum_i^m \sum_j^k \left(\begin{bmatrix} \theta_{11}^T & \dots & \dots & \dots & \theta_{1n}^T \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \theta_{qp}^T & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_{k1}^T & \dots & \dots & \dots & \theta_{kn}^T \end{bmatrix} \times \begin{bmatrix} x_1^{(i)} \\ \vdots \\ x_p^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} - \begin{bmatrix} y_1^{(i)} \\ \vdots \\ y_q^{(i)} \\ \vdots \\ y_k^{(i)} \end{bmatrix} \right)_j^2$$

Because we are doing paritial deriative for $\theta_{pq} = \theta_{qp}^T$, we only need to take the deriative of the component where θ_{qp}^T presents.

$$\begin{aligned} \frac{dJ}{d\theta_{pq}} \sum_i^m (\sum_j^k (\theta^T x^{(i)} - y^{(i)})_j^2) &= \frac{dJ}{d\theta_{qp}^T} \sum_i^m (\sum_j^k (\theta^T x^{(i)} - y^{(i)})_j^2) \\ &= \sum_i^m \frac{d}{d\theta_{qp}^T} (\theta_{q1}^T x_1^{(i)} + \theta_{q1}^T x_1^{(i)} + \theta_{q2}^T x_2^{(i)} + ... + \theta_{qp}^T x_p^{(i)} + ... + \theta_{qn}^T x_n^{(i)} - y_q^{(i)})^2 \\ &= 2 \sum_i^m (x_p^{(i)} (\theta_{q1}^T x_1^{(i)} + \theta_{q1}^T x_1^{(i)} + \theta_{q2}^T x_2^{(i)} + ... + \theta_{qp}^T x_p^{(i)} + ... + \theta_{qn}^T x_n^{(i)} - y_q^{(i)})) \\ &= 2 \sum_i^m x_p^{(i)} (\theta_q^T x^{(i)} - y_q^{(i)}) \end{aligned} \tag{2}$$

This function is verified to be correct in part (d).

(b)

$$\frac{dJ}{d\theta} = \begin{bmatrix} \frac{dJ}{d\theta_{11}^T} & \dots & \dots & \dots & \frac{dJ}{d\theta_{1n}^T} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{dJ}{d\theta_{k1}^T} & \dots & \dots & \dots & \frac{dJ}{d\theta_{kn}^T} \end{bmatrix} = \begin{bmatrix} 2\sum_i^m x_1^{(i)} (\theta_1^T x^{(i)} - y_1^{(i)}) & \dots & \dots & \dots & 2\sum_i^m x_n^{(i)} (\theta_1^T x^{(i)} - y_1^{(i)}) \\ \vdots & & \vdots & \vdots & \vdots \\ \vdots & & \vdots & \vdots & \vdots \\ 2\sum_i^m x_1^{(i)} (\theta_k^T x^{(i)} - y_k^{(i)}) & \dots & \dots & \dots & 2\sum_i^m x_n^{(i)} (\theta_k^T x^{(i)} - y_k^{(i)}) \end{bmatrix}$$

$$= 2 \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \dots & \dots & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \dots & \dots & x_n^{(m)} \end{bmatrix} \times \begin{bmatrix} \sum_i^m (\theta_1^T x^{(i)} - y_1^{(i)}) & \dots & \dots & \dots & \sum_i^m (\theta_1^T x^{(i)} - y_1^{(i)}) \\ \vdots & & \vdots & \vdots & \vdots \\ \vdots & & \vdots & \vdots & \vdots \\ \sum_i^m (\theta_k^T x^{(i)} - y_k^{(i)}) & \dots & \dots & \dots & \sum_i^m (\theta_k^T x^{(i)} - y_k^{(i)}) \end{bmatrix}^T$$

$$= 2X \times \begin{bmatrix} (\theta_1^T x^{(1)} - y_1^{(1)}) + \dots + (\theta_1^T x^{(m)} - y_1^{(m)}) & \dots & \dots & \dots & \theta_1^T x^{(1)} - y_1^{(1)}) + \dots + \theta_1^T x^{(m)} - y_1^{(m)}) \\ & & \vdots & \vdots & \vdots & \\ & & \vdots & \vdots & \vdots & \\ (\theta_k^T x^{(1)} - y_k^{(1)}) + \dots + (\theta_k^T x^{(m)} - y_k^{(m)}) + & \dots & \dots & \dots & (\theta_k^T x^{(1)} - y_k^{(1)}) + \dots + (\theta_k^T x^{(m)} - y_k^{(m)}) \end{bmatrix}^T$$

$$= 2X(\theta^T X - Y)^T \tag{3}$$

Where

- θ is an $n \times k$ matrix, in which n is the number of pixels in an images plus 1 (1025), and k is the number of possible labels. Then θ^T is a matrix of size $k \times n$.
- X is the input matrix of size $n \times m$, n again is the number of pixels plus 1 (1025), and m is the size of the training samples.
- Y is the output label matrix of size $k \times m$, that is the number of availables labels $\times$ the size of the training samples.
- $\theta^T X$ is the hypothesis matrix of size $k \times m$. Therefore, $\theta^T X - Y$ also has size $k \times m$.
- $2X$ has size $n \times m$, multiplying by the $m \times k$ matrix $(\theta^T X - Y)^T$ generate the cost matrix $2X(\theta^T X - Y)^T$ that has size $n \times k$.

(c)

Vectorized gradient function:

```
def vectorized_gradient(x, y, theta):
    x = vstack((ones((1, x.shape[1])), x))
    return dot(dot(2,x), (dot(theta.T, x) - y).T)
```

Cost function:

```
def cf_part6(x, y, theta):
    x = vstack((ones((1, x.shape[1])), x))
    return sum((dot(theta.T,x) - y) ** 2)
```

(d)

Gradient Component Function: The *finite_difference_gradient* function below takes in the matrices x, y and theta, it computes the component (p, q) of the gradient of the cost function. This function is the code implementation of the gradient formula derived in part (a).

```
# Computing the component (p, q) of the gradient matrix.
def finite_difference_gradient(x, y, theta, p, q):
    x = vstack((ones((1, x.shape[1])), x))
    sum = 0;
    for i in range(x.shape[1]):
        sum = sum + x[p][i] * (dot(theta.T[q],x[:,i]) - y[q][i]);
    return 2 * sum;
```

Testing: I create two matrices x and y, compute the gradient using both methods, and compare if the two results are the same.

```
def test_part6():
    # Say each images has 5 pixels (not include 1 on top)
    # 2 images [1,1, 1, 1,5] ,[2, 1, 1, 1, 1]
    # 3 possible labels [1, 0, 0], [0, 1, 0], [0, 0, 1]
    # x = (5 + 1) x 2 matrix
```

```
    # y = 3 x 2 matrix
    # theta.T = 3 x (5 + 1) matrix

    # Form an x
    img1 = array([1,1, 1, 1,5])
    img2 = array([2, 1, 1, 1, 1])
    img = vstack([img1, img2])
    x = img.T # x is in n * m, where n is number of pixels and m is number of samples.
    print("------ x is (without 1 on top yet) -------")
    print(x)

    # Form a theta
    theta0 = zeros((6, 3))
    print("---------- theta zero is -------------")
    print(theta0) # theta is in n * k, where n is number of pixels and k is number of labels..

    # Form a y
    y = array([[1, 0, 0], [0, 0, 1]]);
    y = y.T; # y is in k * m, where k is number of labels and m is the sample size.
    print("---------------- y is -----------------")
    print(y)

    print("---------vectorized gradient-----------")
    print(vectorized_gradient(x, y, theta0))

    print("------gradient computed using finite diffeence------")
    g = zeros_like(vectorized_gradient(x, y, theta0))

    # Compute every componento f the gradient using fitenite difference method.
    for p in range(x.shape[0] + 1):
        for q in range(y.shape[0]):
            g[p, q] = finite_difference_gradient(x, y, theta0, p, q)
    print(g)
```

Output:

```
------------ x is (without 1 on top yet) --------------
[[1 2]
 [1 1]
 [1 1]
 [1 1]
 [5 1]]
---------- theta zero is -------------
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
---------------- y is -----------------
[[1 0]
 [0 0]
 [0 1]]
---------vectorized gradient-----------
```

```
[[ -2.   0.  -2.]
 [ -2.   0.  -4.]
 [ -2.   0.  -2.]
 [ -2.   0.  -2.]
 [ -2.   0.  -2.]
 [-10.   0.  -2.]]
------gradient computed using finite diffeence------
[[ -2.   0.  -2.]
 [ -2.   0.  -4.]
 [ -2.   0.  -2.]
 [ -2.   0.  -2.]
 [ -2.   0.  -2.]
 [-10.   0.  -2.]]
```

Conclusion: There is no difference between the gradient matrices computed using the two techniques. Therefore, the deriative function derived in part (a) is verified.

Part 7

Codes:
The code for this part can be found in *faces.py* and *auto_pick_part5.py*.

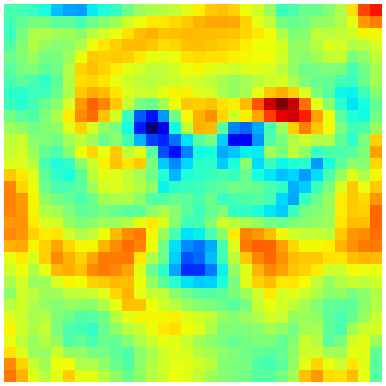
Picking The Data:
I used the script *auto_pick_part5.py* (created in part 5) to randomly select 100 training images and 10 validating images for every actor from the folder cropped that stores all cropped images of the actors. So in total, I have 600 images in the training set, and 60 images in the validation set.

Classifier:
I set $\theta_{initial}$ to be a zero matrix of size $n \times k$. The step size α for the gradient descent algorithm was set to be 10^{-6}, this was the result of a trial and error process. I tried with bigger α 10^{-5}, 10^{-4}, but this resulted in NaN values returned by the computation for some components for theta. It was because the large step size caused some gradient matrix's compoenents to bypass their local minimums and diverge. Else if I got α set to smaller numbers, like 10^{-7} or 10^{-8}, gradient descent took huge number of iterations to give a decent correctness rate.

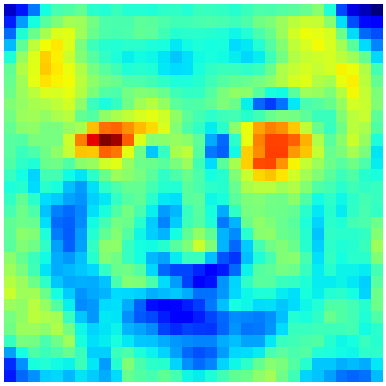
With the step size $\alpha = 10^{-6}$ and $maxnumberofiterations = 50000$ for the gradient descent algorithm, the classifier gives the correctness rate of 99.3% for the training set and 93.3% for the validation test. Because of the large number of iterations, you may have to wait few minutes before the program finishes executing.

Part 8

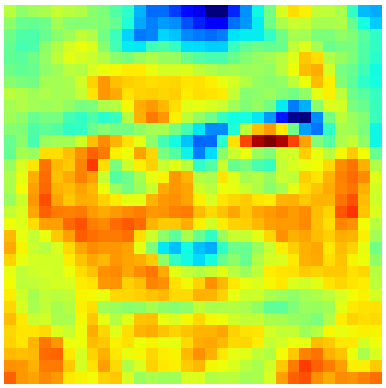
Codes: The code for this part is in *faces.py*.
Images:



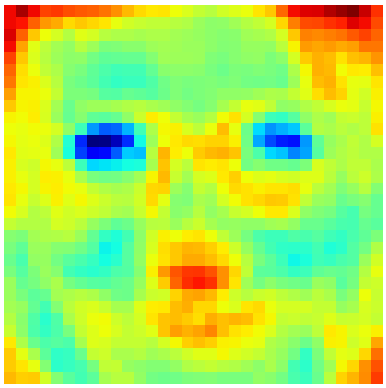
(a) Steve Carell



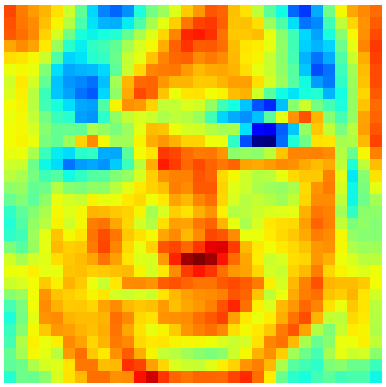
(b) Alec Baldwin



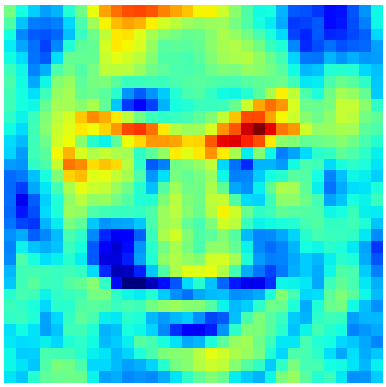
(c) Bill Hader



(d) Kristin Chenoweth



(e) America Ferrera



(f) Fran Drescher