CSC411: Project #1

Due on Monday, January 29, 2018

Tyler Gamvrelis

January 29, 2018

Foreword

In this project, a subset of the FaceScrub dataset is used to train face recognition and gender classification systems. The emphasis of the project is to implement model details "from scratch" as opposed to importing libraries that automate this task. Accordingly, the nature of the project is highly exploratory, challenging students to understand the consequences of each parameter involved in their models.

Parts 1 and 2 detail procedures related to the acquisition of the images and their filtering. Part 3 uses linear regression to build a classifier to distinguish pictures of Alec Baldwin from Steve Carell. Visualizations of the learned parameters from part 3 are presented in part 4. Overfitting is demonstrated in part 5 when the relationship between model performance and training set size is presented for a gender classifier. Part 6 explores the use of one-hot encoding schemes, which generalizes classification problems to an arbitrary number of labels. Additionally, part 6 presents a vectorized form of the cost function along with its gradient, and a finite-difference approximation for the latter. Part 7 explores the use of one-hot encoding for face recognition, and part 8 presents a visualization of the learned parameters from part 7.

System Details for Reproducibility:

- Python 3.6.0
- Libraries:
  - numpy
  - matplotlib
  - time
  - scipy
  - os
  - shutil
  - errno
  - urllib
  - math
  - imghdr

Part 1

The subset of the FaceScrub dataset used in this assignment contained URLs of images of 12 actors (6 male, 6 female) and coordinates for the bounding boxes of their faces. The images were downloaded and processed using a Python script developed in this project. The number of images obtained for each actor varied due to factors such as internet connection speed and image availability (some links were dead or contained invalid images), but exceeded 100 in all cases.

Figure 1: 3 randomly-selected images of actors from the dataset in color. The grayscale images are the same faces after being cropped at the coordinates specified in the dataset.

The majority of the bounding boxes for the faces were reasonably accurate, capturing sufficient detail for the cropped faces to be identified with their original counterparts, as seen in Fig. 1. Perhaps due to the conditions in which the images were captured, the images varied significantly in how the actors were presented. That is, some actors faced left while others faced right, some actors' faces were centred in the bounding boxes while other actors' faces were closer to a corner, etc. Due to this variation, the cropped-out faces cannot be aligned with each other in general. That is, in general, the eyes, noses, mouths, etc. would not be aligned if two random cropped out faces were overlaid.

As was mentioned previously, attempts were made to filter out invalid images. These attempts were made using the `imghdr.what` function to create a list of all the images that were not identifiable as JPEGs so that they could be removed. This action removed 199 "invalid images", approximately 20% of which turned out to be valid PNGs. Since there were 1971 total "valid" images downloaded at this point (165 per actor on average), it was decided that the loss of the PNGs was affordable. Examples of non-ideal images that survived this procedure are given in Fig. 2, below.

Figure 2: Examples of bad images. The two images on the left were supposed to be of Steve Carell, but instead depict messages indicating that the photos are no longer available. Since these images are valid JPEGs, they were not filtered out by `imghdr.what`. The rightmost image is an example of a bounding box that cropped the actor's face in the left half of the image as opposed to the centre.

Part 2

Prior to the images being downloaded, two new directories "uncropped" and "cropped" were created. As the images were downloaded, they were placed into the "uncropped" directory. Each image was then cropped and placed in the "cropped" directory. Then, each image in the "cropped" directory was made grayscale and resized to 32×32.

The idea for how to separate these processed images into 3 non-overlapping sets (training (70 images per actor), validation (10 images per actor), and test (10 images per actor)) was random sampling without replacement. The procedure for this can be found in faces.py, and is presented in pseudocode below:

```
function make-sets:
     Create directories("training", "validation", "test")

     # Initialize a list called pool with all the file names of the processed images
     list pool = cropped.getNames() # pool = ['gilpin0.jpg',..., 'baldwin34.jpg',...]

     list training = new list()
     get-images(training, 70)
     copy-images(training, "training")

     list validation = new list()
     get-images(validation, 10)
     copy-images(validation, "validation")

     list test = new list()
     get-images(test, 10)
     copy-images(test, "test")


function get-images(L, num_img):
     for each actor:
          # Make a list img_list of all the strings in pool containing the name
          # of the current actor
          img_list = [image for image in pool if actor in image]
          while(count != num_img):
               # Sample
               current_img = img_list.getRandom()
               L.append(current_img)

               # Ensure no replacement
               img_list.delete(current_img)
               pool.delete(current_img)
```

Part 3

In this section, linear regression was used to build a classifier to distinguish pictures of Alec Baldwin from pictures of Steve Carell. The cost function minimized was the quadratic cost:

$$J(\theta) = \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2, \quad where \quad \theta, x^{(i)}, y^{(i)} \in \mathbb{R}^{n \times 1}$$

The performance obtained, using $\epsilon = 1 \times 10^{-6}$, $\alpha = 5 \times 10^{-7}$, and a maximum number of iterations of $300,000$ was:

- Average cost of 1.21×10^{-2} and a correct classification percentage of 100 % on the training set
- Average cost of 9.16×10^{-2} and a correct classification percentage of 95 % on the validation set

The function used to compute the output of the classifier is in Part3.py, and is shown below in Fig. 3. This function simply computed the hypothesis $\theta^T x \in [0, 1]$ for normalized input images x, and rounded the result to the nearest integer. The input image x was predicted to be Steve Carell when 1 was returned, and Alec Baldwin when 0 was returned.

```
1  def part3_classifier(theta, x):
2      '''
3      part3_classifier returns 1 when x is hypothesized to be an image of Steve
4      Carell, and 0 when x is hypothesized to be an image of Alec Baldwin.
5
6      Arguments:
7          theta -- the vector of learned parameters
8          x -- the input image (flattened vector with normalized entries)
9      '''
10
11     return 1 if np.dot(theta.T, x) >= 0.5 else 0
```

Figure 3: Function used to compute the output of the classifier in part 3.

The cost function was minimized using gradient descent, the initial parameters for which were arbitrarily chosen to be $\epsilon = 1 \times 10^{-5}$, $\alpha = 1 \times 10^{-6}$, and $300,000$ iterations as a maximum. These parameters resulted in 96.4 % correct classification on the training set, and 85 % correct classification on the validation set. At first, it was thought that increasing α would cause the algorithm to converge to the solution more quickly, but it was discovered that increasing α too much (e.g. 5×10^{-5}) caused the cost to diverge to infinity; thus, α was bounded above by such values. Furthermore, increasing α to values near the upper bound such as 4×10^{-5} actually resulted in the same classification performance as the initial guess, albeit with greater cost incurred on average. Also, it was noted that when gradient descent terminated the cost was around 1.90, so ϵ was decreased to 1×10^{-6}. Another problem that was encountered for a while was that the percentage of correct classifications on the training set was only reaching 99%. Upon further investigation, the cause of this was traced back to the fact that one of the invalid images of Steve Carrel depicted in Fig. 2 had made its way into the training set. Since this image was so much different than the others, it was not able to be classified correctly within $300,000$ iterations of gradient descent. To overcome this, the seed for the random number generator was changed from 2 to 3 so that this image of Steve Carell was not selected.

In summary, the method employed to optimize theta was a form of trial and error. After finding an upper bound for α, α was decreased such that the gradient descent algorithm would converge to a solution that balanced cost incurred on training data predictions with validation data predictions. ϵ was decreased to force

gradient descent to run for more iterations, thus fitting θ more tightly to the training data. The performance on the training and validation sets was then tested, and α was incrementally decreased as necessary. As a result of this procedure, the final parameters were $\epsilon = 1 \times 10^{-6}$, and $\alpha = 5 \times 10^{-7}$, resulting in the acceptable performance reported at the beginning of this section.

Part 4

In part 3, the learned parameter vector $\theta \in \mathbb{R}^{n \times 1}$ was determined using gradient descent. In this section, visualizations of θ are presented along with descriptions for how different θs can be achieved. Visualizations were accomplished by resizing θ from a $n \times 1$ vector into a $\sqrt{n-1} \times \sqrt{n-1}$ array which could be displayed as an image. The images were plotted using gaussian interpolation to blur the pixel boundaries.

(a) The θ obtained using the full training set and gradient descent parameters described in part 3 is shown in the left image in Fig. 4 below. When the model was trained using only 2 images from each actor, θ strongly resembled a face, as seen in the rightmost image in Fig. 4.

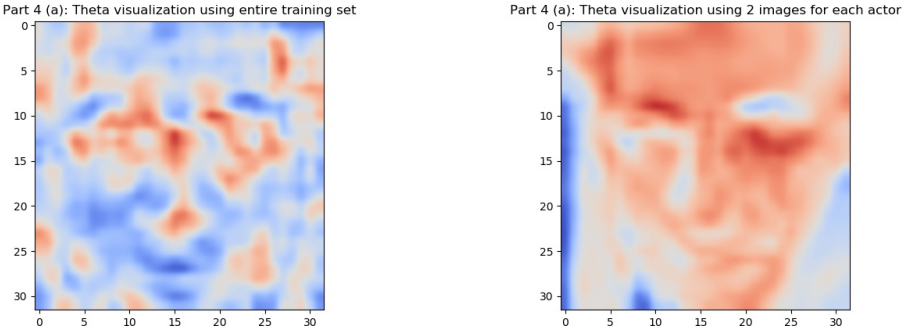

Figure 4: θ visualizations using various training set sizes.

(b) Obtaining θ that does not resemble a face was done using the full training set by forcing ϵ to be very small (about 2×10^{-6} is sufficient to see the result), thus forcing θ to be more closely fit to all the peculiarities in the images before gradient descent terminated. The result is seen in the left image in Fig. 5.

Obtaining θ that resembles a face was done using the full training set by increasing ϵ to 1×10^{-3}. This larger ϵ resulted in gradient descent stopping earlier, thus more closely resembling the features in the training set images nearest to the initial θ vector. In this case, this means that the peculiarities of the training set have not yet been incorporated into the θ vector, so the weighted sum of all the face images still resembles a face.

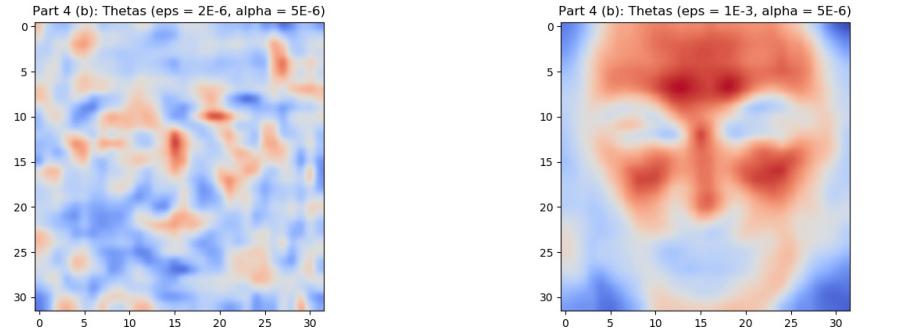

Figure 5: θ visualizations using the full training set and various gradient descent parameters.

Part 5

In this part, overfitting is demonstrated for a gender classifier. Overfitting occurs when a model's fit to the training set is improved at the expense of the model's performance on data outside the training set. That is, the model takes on features (or peculiarities) of the training set that are not found in the representative data in general.

To build the gender classifier, a training set consisting of images of 6 actors (3 male, 3 female) was created, with 70 images for each actor. Subsets of this training set were then used to train a classifier. The first subset consisted of 1 image per actor, the second subset consisted of 2 images per actor, and so on until all 70 images per actor were used. Furthermore the subsets obeyed the relation that if S_1 and S_2 the first and second subsets, then $S_1 \subset S_2$. This was done for programming simplicity. For each of the 70 subsets used to train, the classifier's performance was reported for the similarly-sized training set and 2 validation sets. That is, when the classifier had been trained using 2 images per actor, the training/validation sets that tested its performance consisted of 2 images per actor. One validation set consisted of different images of the 6 actors used to train, and the other validation set consisted of images of 6 other actors which the model had never seen before. Let us refer to the former validation set as *V1*, and the latter as *V2*. These results are reported in Fig. 6 below.

The gradient descent parameters used for this part were $\epsilon = 5 \times 10^{-7}$, $\alpha = 1.6 \times 10^{-5}$, and a maximum number of iterations of $600,000$. α was selected to be the greatest value to 1 decimal place that resulted in convergence, and ϵ was chosen to be sufficiently small that gradient descent would run for the maximum number of iterations as the training set size increased. The maximum number of iterations was increased from $300,000$ in part 3 to $600,000$ in this part to further minimize the error, thereby fitting θ tighter to the training set.

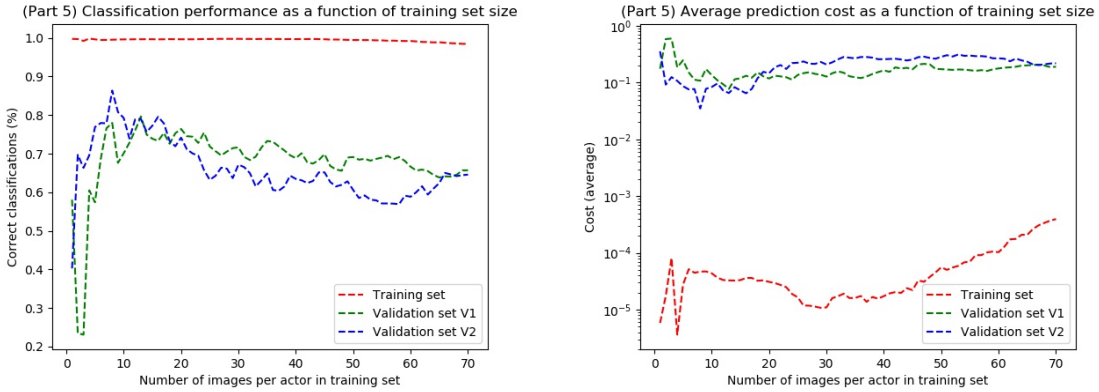

Figure 6: Model performance for various training set sizes. Recall that *V1* denotes the validation set consisting of images of actors that the model had trained with (**NB:** not images used in training), and *V2* denotes the validation set consisting of images of different actors entirely.

As seen in Fig. 6 above, performance on the training set remained near 100 % for all iterations, dipping only slightly near the end as the training set grew very large. Note that for these large training sets, gradient descent was terminating due to the maximum number of iterations being reached as opposed to convergence. From Fig. 6 it can also be observed that performance on *V1* settles between 65 and 75 % once there are about 20 images per actor in the training set. The validation set *V2* shows a decreasing performance trend with training set size after about 10 images, which exemplifies overfitting since it shows that θ loses its

generality to data of faces it has never seen before as it becomes fit more tightly to the training data. The performance on *V2* is likely worse than the performance on *V1* due to the fact that the training data was only trained on images of the same actors whose images are in *V1*, meaning that even when overfitting, some general trends may persist.

Part 6

6. (a) We want to compute $\frac{\partial J}{\partial \theta_{pq}}$, where the cost function is the sum of squared differences, defined as $J(\theta) = \sum_i (\sum_j (\theta^T x^{(i)} - y^{(i)})_j^2)$, where $\theta \in \mathbb{R}^{n \times k}$, $x^{(i)} \in \mathbb{R}^{n \times 1}$, and $y^{(i)} \in \mathbb{R}^{k \times 1}$. Note that $x^{(i)}$ and $y^{(i)}$ are columns of $X \in \mathbb{R}^{n \times m}$ and $Y \in \mathbb{R}^{k \times m}$, where m is the number of training examples. The dimension n is interpreted as 1 greater than the number of pixels per image, and the dimension k as the number of labels (actors in the training set). The (i) superscript on x and y specifies the i$^{\text{th}}$ training example (i.e. i$^{\text{th}}$ image flattened into a column vector), and the θ indices p and q specify rows and columns of θ, respectively. Physically, each column of the theta matrix defines a vector in n-space that represents a "feature template" to which we compare x using the inner product. Thus, the operation $\theta^T x^{(i)}$ is interpreted as computing the similarity of the image $x^{(i)}$ to the "feature template" in each column of θ. The result of this matrix-vector multiplication is a column vector whose k$^{\text{th}}$ entry is a real number between 0 and 1 that represents the model's prediction about the image $x^{(i)}$ being the k$^{\text{th}}$ actor (1 implies very likely, 0 implies unlikely). Subtracting $y^{(i)}$ from this quantity produces the cost associated with the prediction of the image being the k$^{\text{th}}$ actor. Thus, $\frac{\partial J}{\partial \theta_{pq}}$ represents the change in the cost function due to adjusting the weight for the p^{th} pixel in template q.

We begin by making the summation indices explicit for clarity, where m and k are as defined above.

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

Next, we expand the matrix multiplication in $J(\theta)$ as summation using the dummy index l:

$$\frac{\partial J}{\partial \theta_{pq}} = \frac{\partial}{\partial \theta_{pq}} \sum_{i=1}^{m} \sum_{j=1}^{k} (\sum_{l=1}^{n} \theta_{lj} x_l^{(i)} - y_j^{(i)})^2$$

We can pass the differential operator through the sum from $i = 1$ to m without concern, but when passing it through the sum from $j = 1$ to k, we need to add a Dirac delta $\delta(j - q)$ to sift out the entries involving q. This is because only the q^{th} row of $\theta^T x^{(i)}$ will be affected by a changed in θ_{pq}.

$$\begin{aligned}\frac{\partial J}{\partial \theta_{pq}} &= \sum_{i=1}^{m} \sum_{j=1}^{k} \delta(j - q) \frac{\partial}{\partial \theta_{pq}} (\sum_{l=1}^{n} \theta_{lj} x_l^{(i)} - y_j^{(i)})^2 \\ &= \sum_{i=1}^{m} \frac{\partial}{\partial \theta_{pq}} (\sum_{l=1}^{n} \theta_{lq} x_l^{(i)} - y_q^{(i)})^2\end{aligned}$$

We now make a variable substitution. Let $u(\theta_{pq}) = \sum_{l=1}^{n} \theta_{lq} x_l^{(i)} - y_q^{(i)}$. Then $\frac{\partial u(\theta_{pq})}{\partial \theta_{pq}} = x_p^{(i)}$. Rewriting $J(\theta)$ in terms of u, and applying the chain rule, we obtain:

$$\frac{\partial J}{\partial \theta_{pq}} = \sum_{i=1}^{m} \frac{\partial}{\partial \theta_{pq}} u(\theta_{pq})^2 = \sum_{i=1}^{m} 2u(\theta_{pq}) \frac{\partial u(\theta_{pq})}{\partial \theta_{pq}}$$

We then substitute the full expression for u and its derivative:

$$\frac{\partial J}{\partial \theta_{pq}} = \sum_{i=1}^{m} 2 \sum_{l=1}^{n} (\theta_{lq} x_l^{(i)} - y_q^{(i)}) x_p^{(i)}$$

Using the commutativity of real numbers, we can re-arrange the order of the terms into the final expression:

$$\frac{\partial J}{\partial \theta_{pq}} = 2 \sum_{i=1}^{m} x_p^{(i)} (\sum_{l=1}^{n} \theta_{lq} x_l^{(i)} - y_q^{(i)})$$

6. (b) In this part, we want to show that $\frac{\partial J}{\partial \theta}$ can be written in matrix form as $2X(\theta^T X - Y)^T$. A thorough explanation of the dimensions of the matrices and the meaning of their dimensions is given in part (a), but a summary is provided as follows:

- $\frac{\partial J}{\partial \theta} \in \mathbb{R}^{n \times k}$ contains the differential change in cost with respect to each learned parameter
- $X \in \mathbb{R}^{n \times m}$ contains all the input training data, with a row of ones stacked ontop
- $Y \in \mathbb{R}^{k \times m}$ contains the labels (i.e. expected output) for the training data
- $x^{(i)} \in col(X)$ represents the i$^{\text{th}}$ flattened image, and $y^{(i)} \in col(Y)$ represents the i$^{\text{th}}$ label
- $\theta \in \mathbb{R}^{n \times k}$ contains the learned parameters which form "templates" to compare the input data to
- m is the number of training examples
- n is 1 greater than the number of pixels per image
- k is the number of labels (i.e. the number of classification "bins")
- The θ indices p and q specify rows and columns of θ, respectively

From (a), we obtained:

$$\frac{\partial J}{\partial \theta_{pq}} = 2\sum_{i=1}^{m} x_p^{(i)}(\sum_{l=1}^{n} \theta_{lq} x_l^{(i)} - y_q^{(i)})$$

We recognize this as the pq^{th} entry of the derivative matrix $\frac{\partial J}{\partial \theta}$. If we can show that this quantity is equal to the pq^{th} entry of $2X(\theta^T X - Y)^T$, we will be done.

We begin by noticing that $2X(\theta^T X - Y)^T = 2X((\theta^T X)^T - Y^T) = 2X(X^T\theta - Y^T)$. The pq^{th} entry of this is:

$$(2X(\theta^T X - Y)^T)_{pq} = 2\sum_{i=1}^{m} X_{pi}(X^T\theta - Y^T)_{iq}$$

We can dissect the $(X^T\theta - Y^T)_{iq}$ term by finding the expression for the iq^{th} element of $X^T\theta$ and the iq^{th} element of Y^T:

$$(X^T\theta)_{iq} = \sum_{l=1}^{n} X_{li}\theta_{lq} = \sum_{l=1}^{n} \theta_{lq} X_{li}$$

$$(Y^T)_{iq} = Y_{qi}$$

Thus,

$$(X^T\theta - Y^T)_{iq} = \sum_{l=1}^{n} \theta_{lq} X_{li} - Y_{qi}$$

Substituting this into the expression for $(2X(\theta^T X - Y)^T)_{pq}$, we obtain:

$$(2X(\theta^T X - Y)^T)_{pq} = 2\sum_{i=1}^{m} X_{pi}(X^T\theta - Y^T)_{iq} = 2\sum_{i=1}^{m} X_{pi}(\sum_{l=1}^{n} \theta_{lq} X_{li} - Y_{qi})$$

Recognizing that X_{pi} and X_{li} are entries in the i$^{\text{th}}$ column of X, and Y_{qi} is the q$^{\text{th}}$ entry in the i$^{\text{th}}$ column of Y, we can rewrite this expression using $x^{(i)}$ and $y^{(i)}$ to represent the i$^{\text{th}}$ columns of X and Y, respectively:

$$2\sum_{i=1}^{m} X_{pi}(\sum_{l=1}^{n} \theta_{lq} X_{li} - Y_{qi}) = 2\sum_{i=1}^{m} x_p^{(i)}(\sum_{l=1}^{n} \theta_{lq} x_l^{(i)} - y_q^{(i)})$$

But this is the same expression found from (a). Therefore, since for arbitrary p and q, the pq^{th} entry of $\frac{\partial J}{\partial \theta_{pq}}$ is equal to the pq^{th} entry of $2X(\theta^T X - Y)^T$, we must have that the derivative of $J(\theta)$ with respect to all the components of θ can be written in matrix form as $2X(\theta^T X - Y)^T$, as desired.

6. (c) The cost function from part (a) and its vectorized gradient function are presented in Python code in Fig. 7, below (source: Part6.py).

```
def part6_J(theta, X, Y):
    '''
    part6_J returns the cost associated with using the (n x k) parameter matrix
    theta to fit the data X to the corresponding labels Y.

    Arguments:
        theta -- (n x k) matrix of learned parameters
        x -- (n x m) matrix whose columns correspond to images (data from which
        to make predictions)
        Y -- (k x m) matrix whose columns corrspond to the actual/target outputs
        (labels)
    '''

    return np.sum(np.square(np.dot(theta.T, X) - Y))

def part6_grad_J(theta, X, Y):
    '''
    part6_grad_J returns the (n x k) derivative matrix of the cost function J
    defined in part6_J with respect to the learned parameter matrix theta.

    Arguments:
        theta -- (n x k) matrix of learned parameters
        x -- (n x m) matrix whose columns correspond to images (data from which
        to make predictions)
        Y -- (k x m) matrix whose columns corrspond to the actual/target outputs
        (labels)
    '''

    return 2 * np.dot(X, np.transpose((np.dot(theta.T, X) - Y)))
```

Figure 7: Vectorized gradient and cost function.

6. (d) To demonstrate that the vectorized gradient function presented in Fig. 7 works, 5 components of the gradient were computed using the finite-difference approximation presented below, implemented in Python in Fig. 9.

$$\frac{\partial J(\theta)}{\partial \theta_{pq}} \approx \frac{J\left\{\begin{bmatrix} \theta_{11} & \theta_{12} & \dots & \dots & \theta_{1k} \\ \theta_{21} & \ddots & \dots & \dots & \theta_{2k} \\ \vdots & \vdots & \theta_{pq}+h & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_{n1} & \theta_{n2} & \dots & \dots & \theta_{nk} \end{bmatrix}\right\} - J\left\{\begin{bmatrix} \theta_{11} & \theta_{12} & \dots & \dots & \theta_{1k} \\ \theta_{21} & \ddots & \dots & \dots & \theta_{2k} \\ \vdots & \vdots & \theta_{pq} & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_{n1} & \theta_{n2} & \dots & \dots & \theta_{nk} \end{bmatrix}\right\}}{h}$$

To select an appropriate value for h, the finite-difference approximation was compared to the vectorized computation using the routine presented in Fig. 10. The first h-value tried was 1×10^{-1} which resulted in an average percentage error of about 1×10^{-2} %. The value of h was then decreased by a factor of 10 until the error began increasing. Using this procedure, the optimal h-value was determined to lie between 1×10^{-4} and 1×10^{-6}. At the mean point, $h = 1 \times 10^{-5}$, the average percentage error was 7.21×10^{-7} %. Note that the error metric was the percentage error defined as follows:

$$error = \frac{|(vectorized\ computation) - (finite\ difference\ computation)|}{vectorized\ computation}$$

The result of running the routine in Fig. 10 is shown below in Fig. 8. Note that a "dummy" θ matrix was initialized with all entries equal to 0.5 for this part, and X and Y were chosen to be the matrices of actor images and their labels from part 5.

```
>>> (executing lines 386 to 410 of "faces.py")
p:  565 | q:  4 | diff. apprx:  155310.44900417328 | vectorized comp. :
155310.44899653975
p:  298 | q:  3 | diff. apprx:  109200.6206512451 | vectorized comp. :
109200.62056132256
p:  915 | q:  5 | diff. apprx:  129328.45950126646 | vectorized comp. :
129328.46077662437
p:  129 | q:  1 | diff. apprx:  64631.259441375725 | vectorized comp. :
64631.25868512109
p:  53 | q:  3 | diff. apprx:  132697.8087425232 | vectorized comp. :
132697.8105497885
Avg. error:  7.210894004722594e-07
```

Figure 8: Finite-difference approximation and vectorized gradient computation comparison for 5 pq-coordinates, using $h = 1 \times 10^{-5}$.

```python
def part6_grad_J_finite_diff(theta, X, Y, p, q, h):
    '''
    part6_grad_J_finite_diff returns a finite difference approximation of the
    gradient of the cost function define in part6_J.

    Arguments:
        theta -- (n x k) matrix of learned parameters
        x -- (n x m) matrix whose columns correspond to images (data from which
        to make predictions)
        Y -- (k x m) matrix whose columns corrspond to the actual/target outputs
        (labels)
        p -- the row of the theta matrix whose q-th entry will be adjusted
        q -- the column of the theta matrix whose p-th entry will be adjusted
        h -- the differential quantity
    '''

    # Idea: increase the pq-th entry of the theta matrix by a small amount h,
    # then evaluate J at this theta. From this quantity, subtract J evaluated
    # at the original theta, then divide that difference by the differential
    # quantity h. This will give the partial derivative of J with respect
    # to the pq-th entry of the theta matrix.

    new_theta = theta.copy()
    new_theta[p, q] += h
    return (part6_J(new_theta, X, Y) - part6_J(theta, X, Y)) / h
```

Figure 9: Function used to compute finite differences of gradient components. Implementation in Part6.py.

```
# Load the image data into numpy arrays
(actMatrix, actLabels, val_actMatrix, val_actLabels) = p6.part6_getActorLists()

# Initialize a dummy theta array (all entries = 0.5)
dummy_theta = np.full(shape = (actMatrix.shape[0], actLabels.shape[0]),
                      fill_value = 0.5)

# Compute the derivative matrix for the cost. Store results in vcomp
vcomp = p6.part6_grad_J(dummy_theta, actMatrix, actLabels)

# Select 5 random entries in the derivative matrix to compute using the
# finite difference approximation. Print the entry index, the finite
# difference approximation, and the vectorized computation.
rand.seed(3)
for i in range(5):
     p = round(actMatrix.shape[0] * rand.random())
     q = round(actLabels.shape[0] * rand.random())
     apprx = p6.part6_grad_J_finite_diff(dummy_theta, actMatrix, actLabels,
                                         p, q, 1E-5)
     error += abs(vcomp[p, q] - apprx) / vcomp[p, q]
     print("p: ", p, "| q: ", q, "| diff. apprx: ", apprx,
           "| vectorized comp. : ", vcomp[p, q])
print("Avg. error: ", error / 5 * 100)
```

Figure 10: Routine in the Part 6 section of faces.py to compare the vectorized gradient computation to the finite difference approximation.

Part 7

In this part, gradient descent was run on 6 actors in order to perform face recognition.
The performance obtained, using $\epsilon = 6 \times 10^{-5}$, $\alpha = 1 \times 10^{-6}$, and a maximum number of iterations of $15,000$ was:

- Average cost of 7.24 and a correct classification percentage of 92.6 % on the training set
- Average cost of 7.16 and a correct classification percentage of 81.7 % on the validation set

The aforementioned gradient descent parameters seem to make sense because the classification performance on the validation set is well-balanced with the classification performance on the training set. Since θ is not perfectly fit to the training set (e.g. 100 % performance), the peculiarities in the training set have not affected the model significantly for this choice of ϵ. Thus, the model should generalize well to similarly-prepared images of the same actors. Further insight is provided in Part 8, where it can be seen that the rows of the θ matrix resemble faces with unique characteristics reminiscent of the actors they represent.

To obtain the label from the output of the model, the maximum element was located in the prediction vector $\theta^T X$. The entry of the maximum element was then set to 1 while all the other entries were cleared to 0.

Part 8

In this part, the θ vector is visualized as 6 images corresponding to the rows of θ. Gaussian interpolation was used in the plot function.

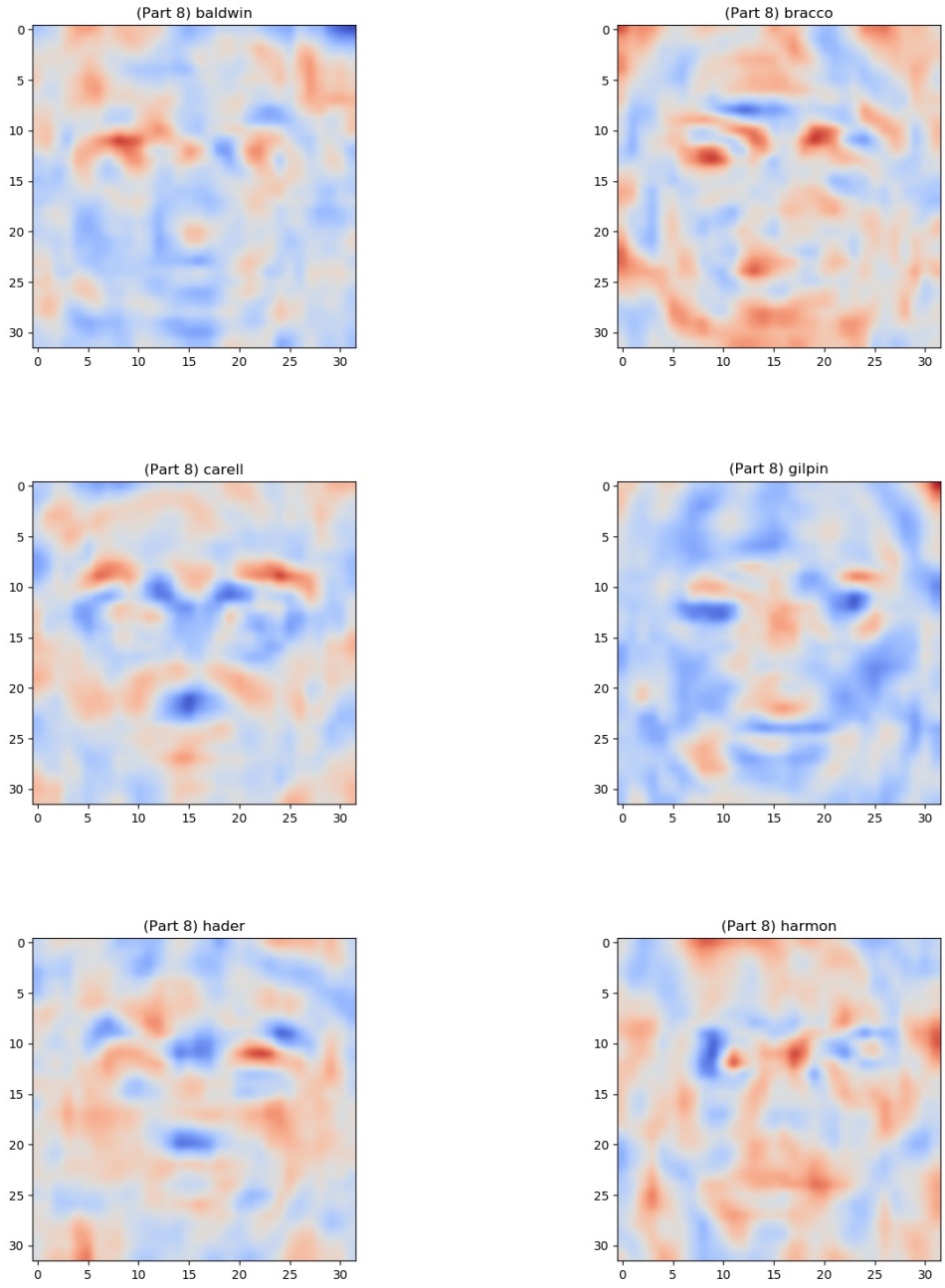

Figure 11: θ visualization from gradient descent run in part 7.