CSC411: Project #1

Due on Monday, January 29, 2018

Hunter Richards

January 30, 2018

Part 1

The FaceScrub dataset consists of 3105 URLs from various websites pertaining to images of 12 different celebrity actors and actresses. Filtering all non-responsive and dead links reduced the dataset to 2781 downloaded images of various aspect ratios, resolutions, and formats.

(a) Angie Harmon

(b) Daniel Radcliffe

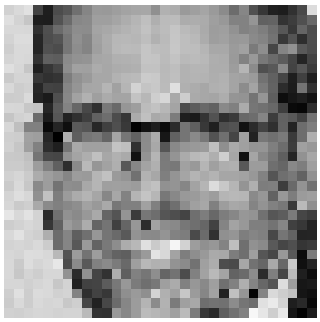
(c) Steve Carell

Figure 1: Sample of unprocessed images

These images were then processed. The processing procedure crops the image to the face, eliminating most noise (i.e. text, shadows, rest of the body, other faces in the image, etc.) while also resizing and converting the image colours to monochrome. The result is a final dataset of 2025 images. The yield is 65.2%.

(a) Angie Harmon

(b) Daniel Radcliffe

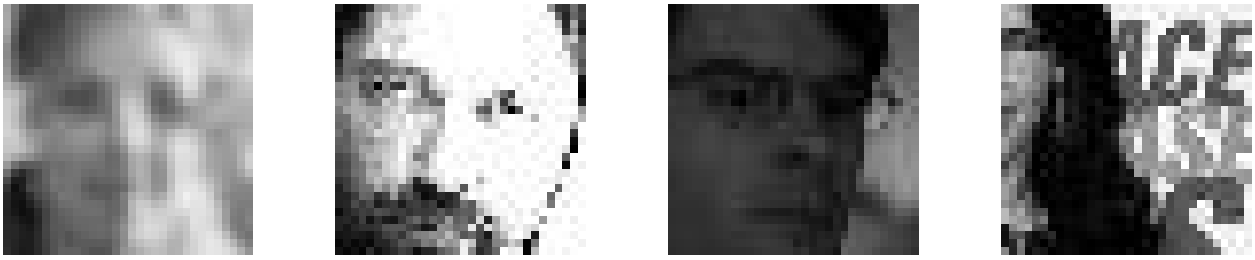


(c) Steve Carell

Figure 2: Image sample post-processing

These images will serve as the main dataset for the project. They are all 32×32 pixels, grayscale, with at least 120 images per actor. The dataset is accurate and high quality, with approximately 98.7% of bounding boxes capturing the face. All other images are either erroneous and not intended to be part of the dataset, or have bounding boxes that completely miss the face. The bounding boxes are also effective at aligning the facial features in images. It can be seen in Figure 2 that the eyes, nose, and mouth are all aligned across the images of multiple actors.

The images capture multiple angles of the actor's face and exposures to different lighting. This allows for a very diverse dataset that more accurately represents practical applications where angle and exposure are not always ideal for facial recognition. Some images, however, are not perfect and have some sort of a defect.



(a) Blurry (b) Over exposure (c) Under exposure (d) Off centre

Figure 3: Various images demonstrating various defects

Fortunately, there are not enough of these images to skew the results. They may, in fact, test the robustness of the facial recognition program and prove desirable.

The downloading and processing of images is performed by the functions `getData()`, lines 25-51, and `timeout()`, lines 5-23, found in `getdata.py`.

Part 2

For splitting the dataset into a training set, validation set, and test set, randomization using the NumPy library is key. A database of file names, implemented as a list in python, is generated from all of the available processed images. A specified number of images are randomly selected without replacement from the database to form a subset. Finally, subsections of the subset are assigned to the training set, validation set, and test set, each implemented as a list, according to a specified ratio. By default this ratio is $100 : 10 : 10$ (given in the project handout), corresponding to 100 images per actor in the training set, and 10 images per actor in the validation and test sets.

The use of randomization without replacement ensures there is no pattern or bias to the images, and no image is used more than once. This allows for the purest sets with which to train, validate, and test the program.

The randomization and splitting of the data into sets is performed by the function `getSets()`, lines 62-78, found in `getdata.py`.

Part 3

The quadratic cost function

$$J(\boldsymbol{\theta}) = \frac{1}{2N} \sum_i \left(\sum_j \theta_j {x_j}^{(i)} - y^{(i)} \right)^2$$

was minimized using gradient descent to obtain optimal parameters $\boldsymbol{\theta}$ for building a classifier to differentiate images of Alec Baldwin from images of Steve Carell. The bias term θ_0 is included in the weight parameter vector and a dummy value of $x_0 = 1$ is appended to each vector $\mathbf{x}^{(i)}$, where $\mathbf{x}^{(i)}$ represents a $1 \times n$ row vector of pixels corresponding to a specific training example i.

This cost function was implemented as a vectorized function in python. The vectorized quadratic cost function is given by

$$J(\boldsymbol{\theta}) = \frac{1}{2N} ||\mathbf{X}\boldsymbol{\theta} - \mathbf{y}||^2$$

as per Roger Grosse's notes on linear regression, where $\mathbf{X}$ is an $m \times n$ matrix representing the entire training set of m images and $\mathbf{y}$ is an $m \times 1$ vector of data labels. Each row is a training example $\mathbf{x}^{(i)}$.

To perform gradient descent, finding the right value for the learning rate α was crucial. Because the training set is small and gradient descent runs quickly, the procedure to find an optimal α was through trial and error. If the algorithm did not converge, then α was too large and was passing over the minimum. Smaller and smaller values of α were tested until the algorithm converged on a solution. Finally, α was then increased by very small increments to get close to the threshold where the algorithm stops converging again. It is desired to have as large an α as possible because too small an α causes the algorithm to converge unnecessarily slow.

In addition to finding an appropriate learning rate, the maximum number of iterations was chosen such that it was not too small, forcing the algorithm to end too early, and not too large to cause overfitting. Also, the tolerance for when $\boldsymbol{\theta}$ is no longer increasing (the loop invariant) can not be too large or the algorithm will terminate prematurely and not too small or the algorithm will take too long and may overfit. The values for maximum iterations and tolerance are 15000 and 10^{-5} respectively, and were determined using trial and error.

Using this cost function and these parameters, a cost of $J(\boldsymbol{\theta}) = 1.2459 \times 10^{-3}$ was obtained for the training set. A very small value. Conversely, the cost for the validation set was $J(\boldsymbol{\theta}) = 4.1995 \times 10^{-2}$, an order of magnitude larger. While still a very small value this demonstrates that there is some overfitting using the specified set size ratio of 100 : 10 : 10 and number of maximum iterations. After performing gradient descent and calculating $\boldsymbol{\theta}$, the function `classify`, lines 53-59 in `regression.py`, checks $\boldsymbol{\theta}$ against an image:

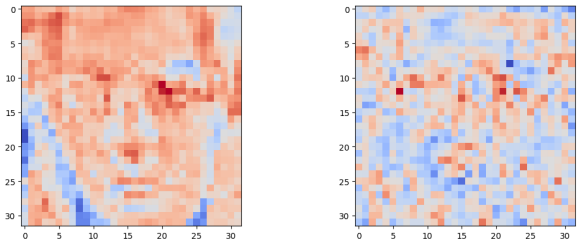
```
def classify(filename, W, y):
    img = Image.open("processed/" + filename)
    X = np.array(img).flatten() / 255.
    res = dot(W.T, np.append(X, 1))
    th = max(res) if isinstance(res, list) else TSH_HOLD
    res = (res >= th).astype(int)
    return norm(res-y) == 0
```

This function handles a weight parameter `W` (i.e. $\boldsymbol{\theta}$) and data label `y` of any dimension (useful for Part 7). The result from applying the weights to the pixels of an image is converted to a binary number based upon a set threshold. (If `res` is multidimensional, then the largest value is set to `1` and the rest to `0`.) This is then compared to the data label. Whether the program predicted correctly or not is returned by the function.

The accuracy of the classifier is much more promising. It scores 100% accuracy on the training set (as should be expected since the classifier was built from this set) and 96% accuracy on the validation set. It is perhaps due to the simplicity of this problem, distinguishing between only two specific people, that the consequences of overfitting are not apparent. Note that because the sets are randomized these values are an average over 10 runs. This ensures they are accurate.

Part 4

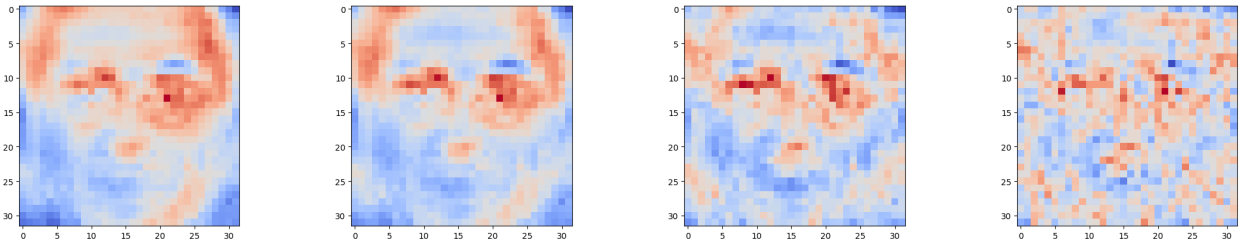
a) The following images were produced by plotting $\boldsymbol{\theta}$ from the gradient descent algorithm:



(a) Two images per actor (b) Full training set

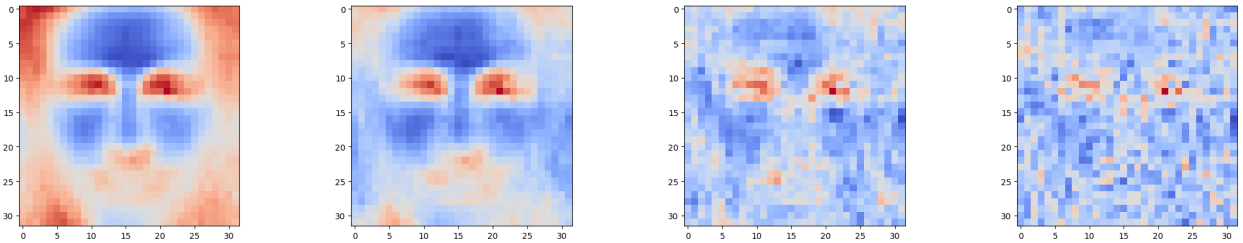
Figure 4

b) Using the full training set and various methods, the following images were produced:



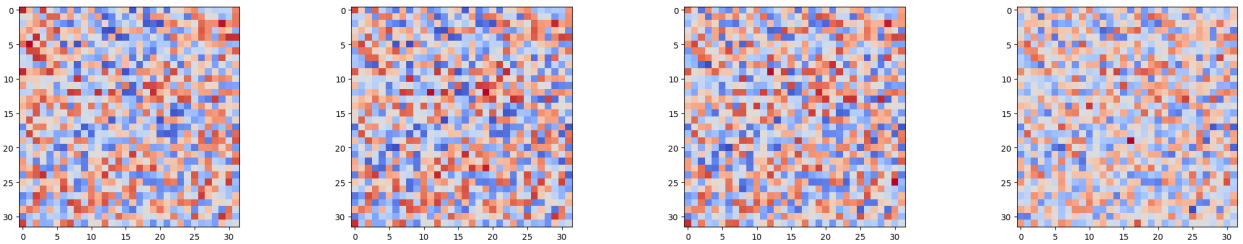
(a) 10 iterations (b) 100 iterations (c) 1000 iterations (d) 10000 iterations

Figure 5: $\boldsymbol{\theta}$ initialized as all zeros



(a) 10 iterations (b) 100 iterations (c) 1000 iterations (d) 10000 iterations

Figure 6: $\boldsymbol{\theta}$ initialized as all ones



(a) 10 iterations (b) 100 iterations (c) 1000 iterations (d) 10000 iterations

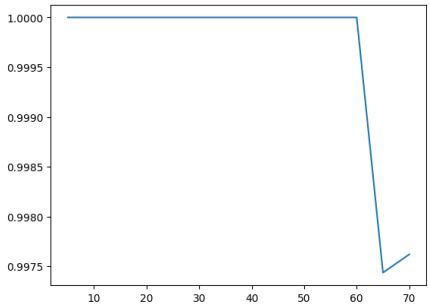
Figure 7: $\boldsymbol{\theta}$ initialized as random numbers $\in [0, 1]$

It can be seen that stopping the training set early produces an image that more closely resembles a face. The more iterations the algorithm performs, the less the image resembles a face. This agrees with the result from 4(a) because a training set with only two images per actor, and therefore less iterations to be performed, produced an image that resembled a face. The opposite can be said for the full training set.

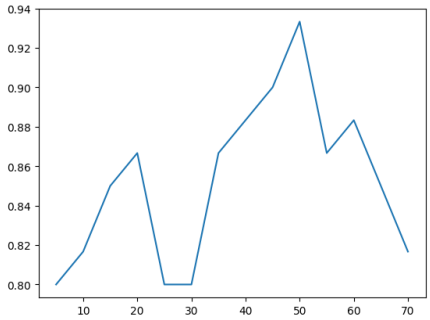
Initializing $\boldsymbol{\theta}$ as different values produces an interesting result. Despite the clear differences between the images at 10 iterations, the images become more and more similar as the number of iterations increases, regardless of what value $\boldsymbol{\theta}$ is initialized as. This is due to the fact that linear regression with a quadratic cost function is a convex problem. There is a single global minimum and this guarantees gradient descent can converge on this minimum. However, it also appears that randomizing the initial value of $\boldsymbol{\theta}$ slows down convergence and impairs the effectiveness of the algorithm. The randomized image in Figure 7(d) does not look as similar to Figures 5(d) and 6(d) as they do to each other. This may be because it is harder for the algorithm to find a minimum when provided a scrambled input as opposed to a uniform input.

Part 5

The following graphs were obtained from plotting the percent accuracy on the training and validation set (labelled along the y-axis) with respect to various training set sizes (labelled along the x-axis):



(a) Performance on training set



(b) Performance on validation set

It can been seen that as more images are included in the training set overfitting starts to occur. The performance of the classifier on the validation set drops because the weight parameters cannot generalize effectively.

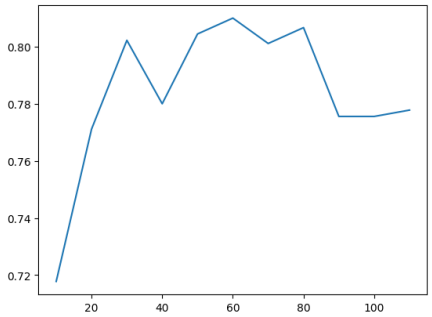


Figure 8: Performance of male/female classifier on actors not in `act`

The performance of the classifier on actors not in `act` is quite high at 81%. This demonstrates the effectiveness of the algorithm to classify images it has never seen before when care is taken to avoid overfitting. However, the accuracy drops to 77% when too many training examples are used. The accuracy of classifying images of actors not in act are more severely affected by overfitting because it is harder to generalize to images that lie completely outside of the domain of the training set (i.e. there are no images of Daniel Radcliffe in the training set, for example).

Part 6

a) Taking the derivative of J with respect to θ_{pq}

$$\frac{\partial J}{\partial \theta_{pq}} = \sum_i \sum_j \frac{\partial}{\partial \theta_{pq}} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - \mathbf{y}^{(i)} \right)_j^2$$

$$= \sum_i \sum_j \frac{\partial}{\partial \theta_{pq}} \left(\begin{bmatrix} \theta_{11} & \cdots & \theta_{1k} \\ \vdots & \ddots & \vdots \\ \theta_{n1} & \cdots & \theta_{nk} \end{bmatrix}^T \begin{bmatrix} x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} - \begin{bmatrix} y_1^{(i)} \\ \vdots \\ y_k^{(i)} \end{bmatrix} \right)_j^2$$

$$= \sum_i \sum_j \frac{\partial}{\partial \theta_{pq}} \left(\begin{bmatrix} \theta_{11} & \cdots & \theta_{n1} \\ \vdots & \ddots & \vdots \\ \theta_{1k} & \cdots & \theta_{nk} \end{bmatrix} \begin{bmatrix} x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} - \begin{bmatrix} y_1^{(i)} \\ \vdots \\ y_k^{(i)} \end{bmatrix} \right)_j^2$$

$$= \sum_i \sum_j \frac{\partial}{\partial \theta_{pq}} \left(\begin{bmatrix} \theta_{11} x_1^{(i)} + \cdots + \theta_{n1} x_n^{(i)} - y_1^{(i)} \\ \vdots \\ \theta_{1k} x_1^{(i)} + \cdots + \theta_{nk} x_n^{(i)} - y_k^{(i)} \end{bmatrix} \right)_j^2$$

$$= \sum_i \sum_j \begin{bmatrix} 0 \\ \vdots \\ 2x_p^{(i)} \left(\theta_{1q} x_1^{(i)} + \cdots + \theta_{nq} x_n^{(i)} - y_q^{(i)} \right) \\ \vdots \\ 0 \end{bmatrix}_j$$

$$= \sum_i \left[2x_p^{(i)} \left(\theta_{1q} x_1^{(i)} + \cdots + \theta_{nq} x_n^{(i)} - y_q^{(i)} \right) \right]$$

$$= 2 \sum_i x_p^{(i)} \left(\sum_j \theta_{jq} x_j{}^{(i)} - y_q^{(i)} \right)$$

Therefore, we have

$$\frac{\partial J}{\partial \theta_{pq}} = 2 \sum_i x_p^{(i)} \left(\sum_j \theta_{jq} x_j{}^{(i)} - y_q^{(i)} \right)$$

or more succinctly

$$\frac{\partial J}{\partial \theta_{pq}} = 2 \sum_i x_p^{(i)} \left(z_q^{(i)} - y_q^{(i)} \right)$$

where

$$z_q^{(i)} = \sum_j \theta_{jq} x_j{}^{(i)}$$

b) Taking the derivative of J with respect to $\boldsymbol{\theta}$ and substituting the result from 6(a)

$$\begin{aligned}
\frac{dJ}{d\boldsymbol{\theta}} &= \begin{bmatrix} \frac{\partial J}{\partial \theta_{11}} & \cdots & \frac{\partial J}{\partial \theta_{1k}} \\ \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial \theta_{n1}} & \cdots & \frac{\partial J}{\partial \theta_{nk}} \end{bmatrix} \\
&= 2 \begin{bmatrix} \sum_i x_1^{(i)} \left(z_1^{(i)} - y_1^{(i)} \right) & \cdots & \sum_i x_1^{(i)} \left(z_k^{(i)} - y_k^{(i)} \right) \\ \vdots & \ddots & \vdots \\ \sum_i x_n^{(i)} \left(z_1^{(i)} - y_1^{(i)} \right) & \cdots & \sum_i x_n^{(i)} \left(z_k^{(i)} - y_k^{(i)} \right) \end{bmatrix} \\
&= 2 \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_n^{(i)} & \cdots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} z_1^{(1)} - y_1^{(1)} & \cdots & z_k^{(1)} - y_k^{(1)} \\ \vdots & \ddots & \vdots \\ z_1^{(m)} - y_1^{(m)} & \cdots & z_k^{(m)} - y_k^{(m)} \end{bmatrix} \\
&= 2\mathbf{X} \begin{bmatrix} z_1^{(1)} - y_1^{(1)} & \cdots & z_1^{(m)} - y_1^{(m)} \\ \vdots & \ddots & \vdots \\ z_k^{(1)} - y_k^{(1)} & \cdots & z_k^{(m)} - y_k^{(m)} \end{bmatrix}^T \\
&= 2\mathbf{X} \left(\begin{bmatrix} z_1^{(1)} & \cdots & z_1^{(m)} \\ \vdots & \ddots & \vdots \\ z_k^{(1)} & \cdots & z_k^{(m)} \end{bmatrix} - \begin{bmatrix} y_1^{(1)} & \cdots & y_1^{(m)} \\ \vdots & \ddots & \vdots \\ y_k^{(1)} & \cdots & y_k^{(m)} \end{bmatrix} \right)^T \\
&= 2\mathbf{X} \left(\mathbf{Z} - \mathbf{Y} \right)^T
\end{aligned}$$

where $\mathbf{Z}$ and $\mathbf{Y}$ are both $k \times m$ matrices. Now, we recognize that

$$\begin{aligned}
\mathbf{Z} &= \begin{bmatrix} \sum_j \theta_{j1} {x_j}^{(1)} & \cdots & \sum_j \theta_{j1} {x_j}^{(m)} \\ \vdots & \ddots & \vdots \\ \sum_j \theta_{jk} {x_j}^{(1)} & \cdots & \sum_j \theta_{jk} {x_j}^{(m)} \end{bmatrix} \\
&= \begin{bmatrix} \theta_{11} & \cdots & \theta_{n1} \\ \vdots & \ddots & \vdots \\ \theta_{1k} & \cdots & \theta_{nk} \end{bmatrix} \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_n^{(i)} & \cdots & x_n^{(m)} \end{bmatrix} \\
&= \boldsymbol{\theta}^T \mathbf{X}
\end{aligned}$$

Therefore, we have

$$\frac{dJ}{d\boldsymbol{\theta}} = 2\mathbf{X} \left(\boldsymbol{\theta}^T \mathbf{X} - \mathbf{Y} \right)^T$$

as required.

c) The vectorized form of the given cost function is given by

$$J = \sum_i \left(\sum_j \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - \mathbf{y}^{(i)} \right)_j^2 \right) = ||\boldsymbol{\theta}^T \mathbf{X} - \mathbf{Y}||_F^2$$

where $||\cdot||_F$ is the Frobenius norm of a matrix and is defined as

$$||\mathbf{A}||_F = \sqrt{\sum_i \sum_j |a_{ij}|^2}$$

This is the default norm used by NumPy on a matrix. The following code implements this result and the result from 6(b) in python:

```
def J(X, W, Y):
    return norm(dot(W.T, X)-Y)**2
def dJ(X, W, Y):
    return 2*dot(X, (dot(W.T, X)-Y).T)
```

These functions are located on lines 10-13 from `regression.py`.

d) To check that this result is correct, the following code, located on lines 44-47 from `regression.py`, was used to calculate a finite difference approximation of several components of the gradient:

```
def finiteDiff(f, X, W, Y, p, q, h=DIV):
    E = np.zeros((len(W), len(W[0])))
    E[p][q] = h
    return (f(X, W+E, Y) - f(X, W, Y)) / h
```

The approximated values were compared to 5 elements of the gradient by calculating the percent difference of each and taking the average. These points where chosen at random. The following code demonstrates this procedure:

```
def part6():
    # Some code ...
    res = 0
    for i in range(0, 5):
        p, q = np.random.randint(0, VEC_SIZE + 1), np.random.randint(0, k)
        approx = finiteDiff(J, X, W, Y, p, q)
        res += percentDiff(grad[p][q], approx)
    print "Percent difference =", res/5., "%"
    return

def percentDiff(a, b):
    return 2 * abs(a-b) / float(a+b) * 100
```

The function `part6()` is on lines 81-97 from `faces.py` and `percentDiff()` is on lines 49-50 from `regression.py`. The average percent difference between 5 random values calculated by the gradient function, and that which was calculated using a finite difference approximation, was determined to be 2.0227×10^{-6} %. The difference is very small, proving this implementation of the gradient to be correct. A value of $h = 10^{-5}$ was used. This value was chosen because it is large enough for approximations to consistently be nonzero. The value is also as small as possible to get the most accurate approximation. This was confirmed using trial and error.

Part 7

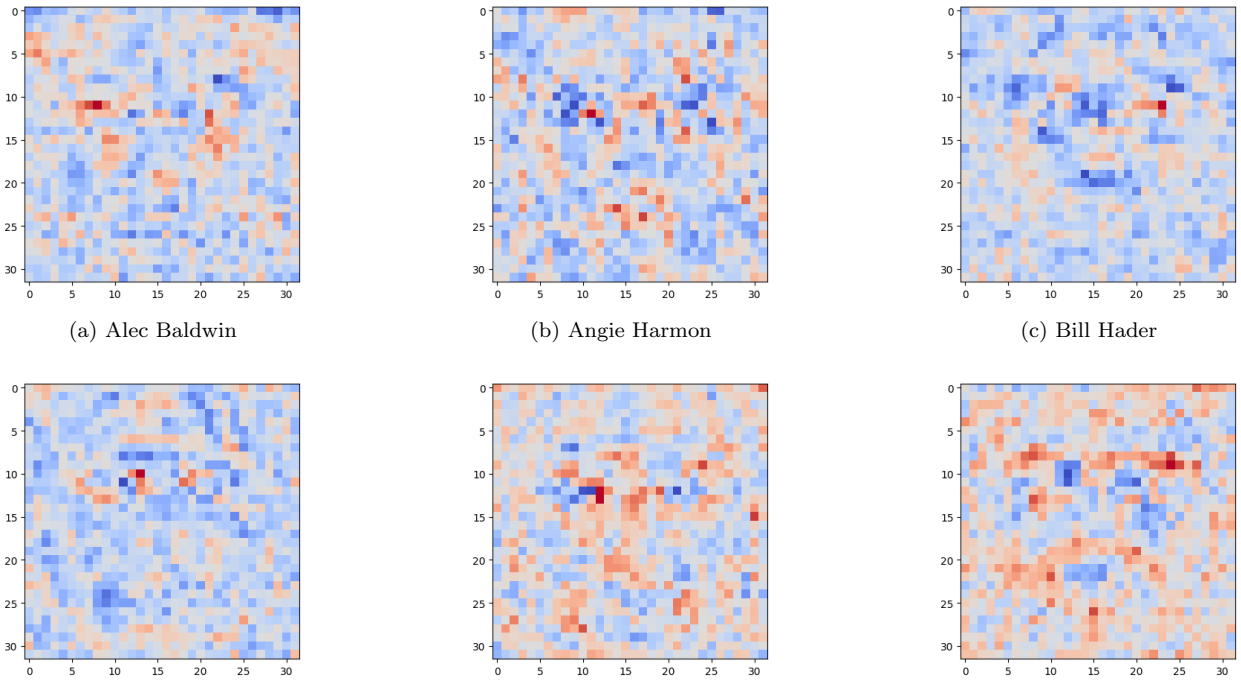
A randomized training and validation set were used to test the new gradient function for facial recognition. The percent accuracy on the training set was 94.7% and 70% on the validation set. The training set accuracy is still very high as it should be. The accuracy on the validation set, however, is considerably lower than what was reported in previous parts. This is likely due to the higher complexity of the problem making it more sensitive to factors such as overfitting.

When choosing parameters for the gradient descent algorithm, the task was much more difficult. The cost function is no longer being normalized, so a learning rate three orders of magnitude is required to prevent the algorithm from diverging. Because the complexity of the problem has increased, the maximum number of iterations has been reduced to 10000. This is implemented as a form of regularization to prevent overfitting by ending the algorithm early. The tolerance remains at 10^{-5} or else the algorithm will diverge.

To obtain the data label from the model, the result of $\boldsymbol{\theta}^T\mathbf{x}$ is first converted to a binary array. The largest value in the array is converted to a `1` and the rest to `0`. This is to isolate the most likely prediction made by the program. Taking the 2-norm (or Frobenius norm if $\boldsymbol{\theta}^T\mathbf{X}$ is a matrix, but this is not the case here) of the difference between the binary array and the data label vector $\mathbf{y}$ will return either `0` if the vectors are identical, or `1` otherwise. Whether this operation returns a `0` or `1` allows one to determine the data label of the image being tested. A return value of `0` indicates a correct prediction and vice versa.

Part 8

The following images of $\boldsymbol{\theta}$ were obtained from the result of part 7:



(a) Alec Baldwin (b) Angie Harmon (c) Bill Hader

(d) Lorraine Bracco (e) Peri Gilpin (f) Steve Carell

Figure 9

While not very representative of the actor's faces, this is to be expected. It has been demonstrated in part 4 that as the algorithm tends towards a solution, the image obtained from $\boldsymbol{\theta}$ looks less and less like a face. Some prominent features, however, are still barely visible. Most notably the eyes and mouths.