CSC411: Assignment #1

Due on Wednesday, February 1, 2017

Yoshiki Shoji (1001277067)

Wednesday, February 1, 2017

Note to the TA: The Procedure to Reproduce the Code is as follows:

Part 1 and 2:

Line 1 to Line 197

This will create the training,validation and test set called called, x_p3, x_p3VAL and x_p3TEST respectively. Furthermore, two folders are required. One called uncropped and the other called cropped to save the gray scale cropped images. Also, faces_subset.txt and subset_actors.txt is required.

Part 3:

Line 200 to Line 241

These lines contain the cost, gradient and gradient descent function. The output has been initialized as y_p5, and the input parameters to the functions are x_p5, y_p5 and the theta term is the one returned by the gradient descent function.

Part 4:

This function requires a folder named part4. It can be executed from Line 248 to Line 266 as can be seen within the python file. Uncomment the section for compiling.

Part 5:

Line 268 to Line 491. The procedure is the same as Part 3. The only difference is the training, validation and test set variables. They have been named x_p5, x_p5VAL and x_p5TEST respectively. The classifier function for this part has been named Classifier_P5. Furthermore, to make the set for act_test, a new file named uncropped2 is required. The variable which stores these images is declared as, x_p5_2. To make the training set for act_set, the faces_subsetP5.txt and subset_actorsP5.txt is required.

Part 6:

Line 494 to Line 570. This part uses the same training set, the variable x_p5 and the output variable as y_p6, the initial theta as theta_p6 and the alpha value as h_p6. The validation and test set has been assigned as the variable, x_p5VAL and x_p5TEST respectively. To find the optimal theta matrix through gradient descent, theta_p6 becomes the input to the function grad_descent_matrix(df_matrix, x, y, init_t, alpha) and as it returns the optimal theta matrix, and this will then become the input to the cost_matrix(x,y,theta) for the theta parameter.

Part 7:

Line 575 to Line 582

Part 8:

Line 585 to Line 623

Part 1

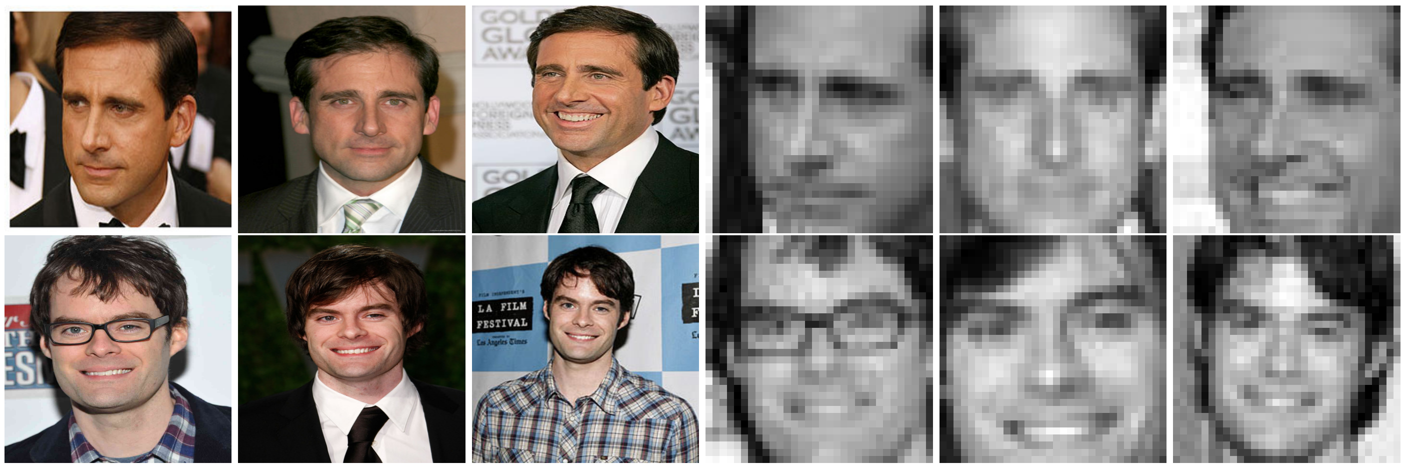


Figure 1: Actual image vs. Cropped image of Steve Carell and Bill Hader

Observing Figure 1, it is evident that the annotations on the actual data set through the process of cropping which yields the 32x32 pixel image is not 100% accruate. For the images of Steve Carell (above), all three cropped images crop a section of the upper head, and this same affect can be seen in Bill Hader as well (below). In some cases, a section of the chin is also cropped out as well. Hence, no accurate overlapping is evident as the cropped images will in most cases contain only majority, but not all sections of the face from the original data set.

Part2

The training,validation and the test set were made within the for loop that was given where the data images are being saved. As the loop iterates through each actor within the list, act['Bill Hader','Steve Carell'], there is an if followed by two elif statements. To properly store the images, a list is always appended to an empty list that was first initialized. Hence, when each actor is being stored, the list expands as follows:

$$Initialization : SET = [EMPTY]$$

$$Actor1 : SET = [[IMAGES\ OF\ HADER]]$$

$$Actor2 : SET = [[IMAGES\ OF\ HADER], [IMAGE\ OF\ CARELL]].$$

And so on. Each list, which contains the images of the actors of interest will thereby only interate and store the proper amount of data owing to the if and elif statements. Futhermore, as the images contain 32x32 pixel images, these elements must be utilized further for other continuing parts and thus, numpy reshape function was used to create a (1,2014) dimensional arrays which can be seen within the if/elif statements. Here, a snippet of the code will be given:

```
for a in act:
    name = a.split()[1].lower()
    #print(a)
    #print(name)
    i = 0
    TRAININGSET2.append([])
    VALIDATIONSET2.append([])
    TESTSET2.append([])
    j = len(TRAININGSET2) - 1

    for line in open("faces_subset.txt"):
        if a in line:
    .....
        if (i < 100):
        TRAININGSET['filename'] = im
        TRAININGSET2[j] = TRAININGSET2[j] +
        [np.reshape(TRAININGSET['filename'],(1,2014))

        #VALIDATION SET
        elif (100 < i < 111):
    ....
```

Part3

The hypothesis that was utilized for linear regression was a line within the hyper plane. With this, we construct the cost function which computes the mean squared error given as:

$$J(\Theta) = \frac{1}{2m}\sum_{n=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2,\ where\ 'm'\ corresponds\ to\ the\ number\ of\ images$$

In order to minimize this cost function and hence minimize the amount of error the algorithm will compute for classifying the two actors, Carell and Hader, we take the partial derivative with respect to all the theta's, which in our case will have a dimension (1025,1). Now, take the derivative with respect to our vector, Θ and observe we obtain:

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

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

where our index j corresponds to the Θ element we are taking the partial with respect to. Furthermore, once the gradient of the cost function was implemented, a classifier function was used in order to compute performance. The code is as follows:

```
def classifier(act,x,y,init_t,alpha,df,grad_descent,x_set):
    #what we want to do is compute the value of the percentage between
    #how well the theta was tuned in order for us to obtain the proper actor.
    #if the hyptothesis gives a value greater than 0.5, it will be equal to an
    #output of one. Otherwise, it will be zero.
    a = 0
    b = 0
    theta = grad_descent(df, x, y, init_t, alpha)
    for i in range(0,len(x_set)):
        if i < (len(x_set)/2) and float(dot(theta,x_set[i].T)) > 0.5:
            a += 1
        else:
            if i > (len(x_set)/2) and float(dot(theta,x_set[i].T)) < 0.5:
                b += 1
    print(act[0], float(a)/(len(x_set)/2))
    print(act[1], float(b)/(len(x_set)/2))
    return
```

The Classifier function first observes the first half of the data within the set of interest and then the second half. If our first half computed the dot product between the image and theta vector, if this quantity is greater than 0.5, we classify it as a success value of 1 and therefore, it successfully classifies the actor correctly. If it is less than 0.5 for the bottom half, then it classifies as a success value of 0 and therefore, it classifies the actor correctly. With this, as we keep adding onto our two initialized values of ”a” and ”b”, we print the percentage value at the end.

In order to fine tune the alpha and epsilon parameters, first an arbitrary small value was chosen in order to minimize the chance of approaching an overshoot problem. Furthermore, as it can be seen within the code, a restriction was used in order to set a upper bound to the number of iterations the loop can perform by minimizing the gradient function. Hence, first a value of 1-e8 and 1-e8 was chosen for alpha and epsilon respectively. This however, yielded an output of 0.4656 which is not ideal. To overcome this situation, it was most ideal to decrease the episilon value, which is a value that indicates the norm of the difference between the new and prior theta. Therefore, it was chosen as 1-e10, and alpha to make the running time more efficient, was increased to 1-e6. This yielded a better result, a value of 0.242. By using this tuning method, we then make epsilon smaller again and keeping alpha fixed. This again resulted in a value of 0.242. To observe if this is a good local minimum value on the existing hyperplane, we check to see how well the classifier function does on this particular theta. The result computes 60% for Bill Hader and 60% for Steve Carell. Furthermore, if we continue with this convention in order to fine tune our parameters, it seems the algorithm was not able to compute values better than 60% for each actor. In order to overcome this situation, we now make a drastic increase in the alpha value. The reason for this is to increase the possibility in overcoming the region which our fixed alpha value was contained in giving the same local minimum each time. Therefore, by changing alpha to 1-e4, the algorithm successfully recognized Bill Hader as 90% and 70% for Steve Carell for the validation set and an equal classification percent of 80% for the test set. We note further these results both came with the cost of 0.0385

Part4

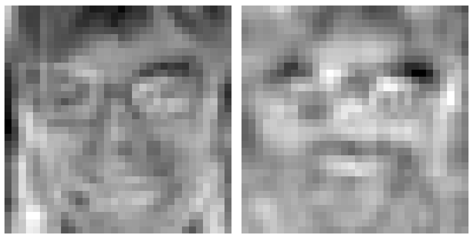


Figure 2: 2 images per actor vs. 100 images per actor of Steve Carell and Bill Hader

The left image gives the result for 2 images per actor and the right image gives the image for 100 images per actor. It is evident that the left one looks more closely to an image of a face, whereas the right one yields almost a blur. This is to be expected, as the images within the training set increases, potential noise increases and hence, disturbs the image. The converse may be said with 2 images per actor. If we decrease the amount of data within the training set, the linear model overfits to a line that lies almost in between the data points, giving almost overlapping images of the two actors. The code is given below:

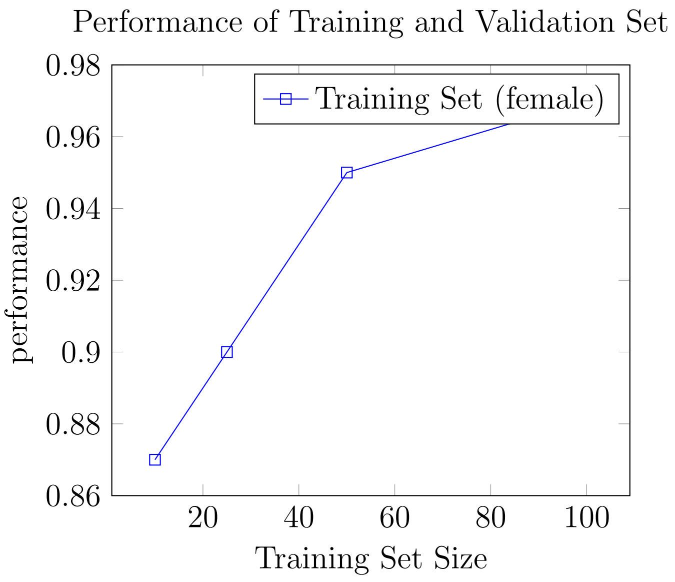
```
#As we already have the theta’s stored through the Grad.Descent Function, we will
#assign the variable and save the image within the working directory folder named
#part 4

t_100 = grad_descent(df,x_p3,y_p3,theta0,h)
t_100 = np.delete(t_100,0)
t_100 = np.reshape(t_100,(32,32))
imsave("part4/"+filename,t_100)

#To create the array which contains the four images, two per actor,
#we will assign it to a variable called t_2

t_2 = np.array((4,1025))
y_p3_2 = np.array((4,1))
t_2 = np.vstack((x_p3[0],x_p3[1],x_p3[100],x_p3[101]))
y_temp1_2 = np.ones((2,1))
y_temp2_2 = np.zeros((2,1))
y_p3_2 = np.vstack((y_temp1_2,y_temp2_2))
t_2 = grad_descent(df,t_2,y_p3_2,theta0,h)
t_2 = np.delete(t_2,0)
t_2 = np.reshape(t_2,(32,32))
imsave("part4/"+filename,t_2)
```

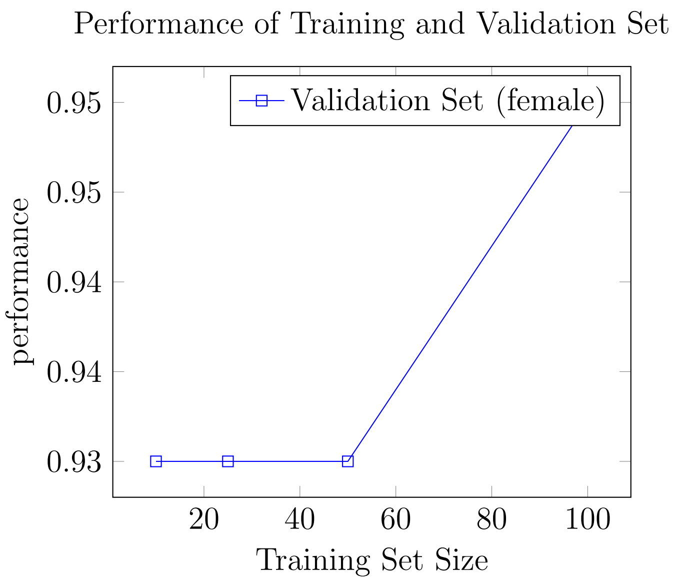
Part5

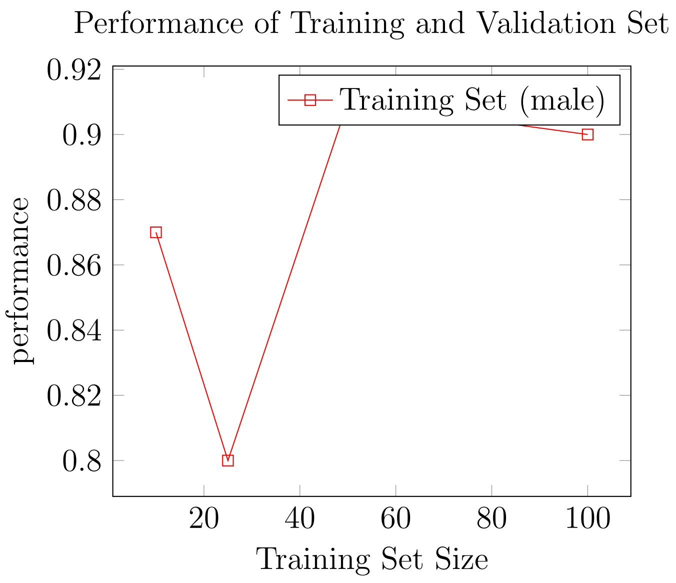
To test each performance, a training set of 100,50,25 and 10 images per actor was used. This then was used to compute the performance on the same validation set containing 10 images per actor. Observing table 1, it was noticeable that the performance was proportional to the number of images, as can be seen within the graphs below. Furthermore, for the performance on a set of different actors, act_test which contained 100 images per actor, the male performance was given to be 87.33% and the female to be 86.73%.

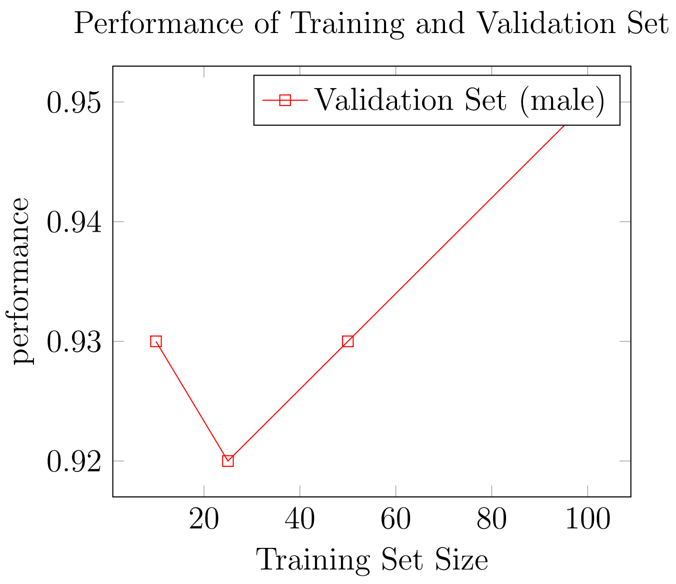
Training Set Size (images per actor)	Validation Set Size (images per actor)	Performance of Classification on Validation Set	Performance of Classification on Training set	Cost	Epsilon Value	Alpha Value
100 images per actor	10	male: 90% female: 97%	male: 95% female: 96%	0.03356	1e-10	1e-4
50 images per actor	10	male: 91% female: 95%	male: 93% female:93%	0.0377	1e-10	1e-4
25 images per actor	10	male: 80% female: 90%	male: 92% female: 93%	0.04159	1e-10	1e-4
10 images per actor	10	male: 87% female: 87%	male: 93% female: 93%	0.04184	1e-10	1e-4

Table 1: Performance of training set and validation set of various sizes on gender classification




Performance of Training and Validation Set
Validation Set (female)
performance
0.95
0.95
0.94
0.94
0.93
20
40
60
80
100
Training Set Size


Performance of Training and Validation Set
Training Set (male)
performance
0.92
0.9
0.88
0.86
0.84
0.82
0.8
20
40
60
80
100
Training Set Size


Performance of Training and Validation Set
Validation Set (male)
performance
0.95
0.94
0.93
0.92
20
40
60
80
100
Training Set Size

Part6

(a) We are given the following cost function:

$$J(\Theta) = \sum_i (\sum_j (\theta^T x^{(i)} - y^{(i)})_j^2)$$

We first recognize that the inner index corresponds to how many elements are within the row of our θ matrix. If we were to build a classifier with only 4 actors as given in the assignment example, $j = 1, .., 4$. our outer index will correspond to how many images we have within our respective sets ie. training,test,validation. With this in mind, we now take the partial using indical notation:

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

Observing the above, we then recognize that if p and q are of equal indical notation, the partial derivative inside the sum terms will become, $x_{pq}^{(i)}$, where the "x" terms will vanish whenever p≠q. With this, we conclude that the following partial derivative is:

$$2 \sum_i (\sum_j x_{pq}^{(i)} (\theta^T x^{(i)} - y^{(i)})_j)$$

(b) We are given the following equation that claims to compute the gradient when the theta is not a column vector, but instead a matrix:

$$\frac{\partial J}{\partial \theta_{pq}} = 2X(\theta^T X - Y)^T$$

Before we proceed further, let us observe the dimensions of each matrix:

θ: (n x k) matrix

X: (n x 1025) matrix

Y: (k x 1025) matrix

Furthermore, rather than using indical notation to find if the equation does compute the gradient, we first observe it in the matrix form and compare it to the equation given:

Step 1: Observe the inner term, a (k x 1025) matrix

$$\begin{bmatrix} \theta_1^T x_1 & \theta_2^T x_1 & \theta_3^T x_1 & \theta_4^T x_1 & \dots & \theta_k^T x_1 \\ \theta_1^T x_2 & \theta_2^T x_2 & \theta_3^T x_2 & \theta_4^T x_2 & \dots & \theta_k^T x_2 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_1^T x_n & \theta_2^T x_n & \theta_3^T x_n & \theta_4^T x_n & \dots & \theta_k^T x_{1025} \end{bmatrix}$$

where θ_j, j = 1...k, represents each row of the θ matrix, and x_i, i = 1...1025, represents the column of the X matrix.

Rewriting the inner matrix, using $\overline{J}(\theta_{ij}) = \theta_j^T x_i - y_{ij}$, where y_{ij} represents the entries within the Y matrix, we obtain the following by also taking the transpose, noting further that it is of dimension (1025 x k):

$$\begin{bmatrix} \overline{J}(\theta_{11}) & \overline{J}\theta_{12}) & \overline{J}(\theta_{13}) & \overline{J}(\theta_{14}) & \dots & \overline{J}(\theta_{1k}) \\ \overline{J}(\theta_{21}) & \overline{J}(\theta_{22}) & \overline{J}(\theta_{23}) & \overline{J}(\theta_{24}) & \dots & \overline{J}(\theta_{2k}) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \overline{J}(\theta_{1025,1}) & \overline{J}(\theta_{1025,2}) & \overline{J}(\theta_{1025,3}) & \overline{J}(\theta_{1025,4}) & \dots & \overline{J}(\theta_{1025k}) \end{bmatrix}$$

Dropping the constant 2 for simplicity, we see that the only term left is through the product between the $\overline{J}(\theta)$ matrix and the X matrix. Following the same procedure above, if we observe matrix multiplication in terms of rows and columns, we obtain the following, noting that we obtain a (n x k) matrix, where we remind ourselves "n" is the total number of images and "k" is the total number of labels, or actors.

$$\begin{bmatrix} \frac{\partial J}{\partial \theta_{11}} & \frac{\partial J}{\partial \theta_{12}} & \frac{\partial J}{\partial \theta_{13}} & \frac{\partial J}{\partial \theta_{14}} & \dots & \frac{\partial J}{\partial \theta_{1k}} \\ \frac{\partial J}{\partial \theta_{21}} & \frac{\partial J}{\partial \theta_{22}} & \frac{\partial J}{\partial \theta_{23}} & \frac{\partial J}{\partial \theta_{24}} & \dots & \frac{\partial J}{\partial \theta_{2k}} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial \theta_{n1}} & \frac{\partial J}{\partial \theta_{n2}} & \frac{\partial J}{\partial \theta_{n3}} & \frac{\partial J}{\partial \theta_{n4}} & \dots & \frac{\partial J}{\partial \theta_{nk}} \end{bmatrix}$$

Hence, we interpret the result as follows: Each entry of the matrix above is the cost associated with the actor. If one actor obtains a label, [1,0,0,0], then each of these entries have a cost associated with them to compute the hyperlinear model.

(c) The Cost function is given as follows:

```
def cost_matrix(x,y,theta):
    return float(sum(sum( (y.T - dot(theta.T,x.T)) ** 2),0))/len(x)
```

The Gradient function is given as follows:

```
def df_matrix(x,y,theta):
    return 2* dot(x.T,(dot(theta.T,x.T) - y.T).T)
```

side note: For the "x" matrix, the line which appends the ones column to the (n,1024), where 'n' is the number of images and 1024 is the 32x32 pixel image dimensional matrix as seen within the CSC411 website is neglected as it is done outside the loop. Furthermore, the Transposes may not be exactly positioned the exact same as the equations that were given, however, that is due to the fact within the actual algorithm, I have stored the each element in such a way. Nevertheless, we obtain the 6x1025 Matrix for df_matrix function and a value for the $cost_matrix$ function.

(d) For the finite differences, we use the following definition of a Partial Derivative nothing that we take the partial of the given Cost function, denoted as $J(\theta)$:

$$\frac{\partial J}{\partial \theta} = \frac{J(\theta + h) + J(\theta)}{h}$$

where "h" is to be made arbitrarily small and also taking into account θ term is a matrix. Hence we develop the algorithm as follows:

```
def finite_difference(df_matrix,cost_matrix2,x,y,theta,h):
    #To evaluate the cost function, we use the fact of computing
    #finite differences: (f(x+h)-f(x))/h, where "h" is an arbitrary
    #small constant
    Fdiff = [ ]
    DFreal = (df_matrix(x,y,theta)).T
    for i in range(0,1025):
        Fdiff.append([])
        for j in range(0,6):
            H = np.zeros((1025,6))
            H[i][j] = h
            Fdiff[i] = Fdiff[i] + [(cost_matrix_2(x,y,theta + H)
            - cost_matrix_2(x,y,theta)) / (h)]


    Fdiffnew = np.asarray(Fdiff)
    Fdiffnew = Fdiffnew.T
    Delta = norm((DFreal - Fdiffnew))
    return Delta
```

The finite difference function works by first initializing an empty list. Within the first for loop, an empty list is added a total of 1025 times at the end of the execution, and the second nested loop for a total of 6 times. This yields a matrix of 1025x6. To make the matrices the right dimension, the two matrices returned by the gradient function in part (c) and the one computed within the finite difference function have both been transposed, giving the correct 6x1025 dimension. Hence, to compare whether the vectorized gradient and the computed numerical one are both nearly identical to one another, the normalized distance between the difference of these two matrices have been computed. The values are given within the table below:

"h" value	Norm (finite difference - vectorized gradient)
1e-5	0.153
1e-6	0.0153
1e-7	0.00153
1e-8	0.000306

SideNote : If the "h" value was made too arbitrarily small, the performance on the norm started to become worse. This was noticeble when the "h" value was less than 1e-10.

Part 7

Actor	Performance on Training Set (100 images per actor)	Performance on Validation Set (10 images per actor)
Fran Dresher	90%	70%
America Ferrera	80%	50%
Kristin Chenoweth	80%	80%
Alec Baldwin	80%	70%
Bill Hader	90%	60%
Steve Carell	70%	70%

Table 2: Performance of training set and validation set of various sizes on six different actors, given its own respective labels.

Comparing the performance on the training and validation set, it is noticeable the validation set gave a success rate. This can be explained as the hyperlinear model potentially overfitted, performing exceptionally well with the training set but otherwise with the validation set. If a better result is required on the validation set, an optimal number for the images within the training set should be found giving a consistent reading for each actor. The code for the performance is given below:

```
def classifier_P8(act,x,y,init_t,alpha,df_matrix,grad_descent_matrix,x_set):
    #what we want to do is compute the value of the percentage between
    #how well the theta was tuned in order for us to obtain the proper actor.
    #if the hyptothesis gives a value greater than 0.5, it will be equal to an
    #output of one. Otherwise, it will be zero.
    a = 0
    b = 0
    c = 0
    d = 0
    e = 0
    f = 0
    theta = grad_descent_matrix(df_matrix, x, y, init_t, alpha)

    for i in range(0,len(x_set)):
        THETA_X = (dot(theta.T,x_set[i].T))
        if (0 < i < 10) and np.amax(THETA_X) == THETA_X[0]:
            a += 1
        elif (10 < i < 20) and np.amax(THETA_X) == THETA_X[1]:
            b += 1
        elif (20 < i < 30) and np.amax(THETA_X) == THETA_X[2]:
            c += 1
        elif (30 < i < 40) and np.amax(THETA_X) == THETA_X[3]:
            d += 1
        elif (40 < i < 50) and np.amax(THETA_X) == THETA_X[4]:
            e += 1
        elif (50 < i < 60) and np.amax(THETA_X) == THETA_X[5]:
            f += 1
    else:
        pass

print(actP5[0], float(a)/(len(x_set)/len(actP5)))
print(actP5[1], float(b)/(len(x_set)/len(actP5)))
print(actP5[2], float(c)/(len(x_set)/len(actP5)))
print(actP5[3], float(d)/(len(x_set)/len(actP5)))
print(actP5[4], float(e)/(len(x_set)/len(actP5)))
print(actP5[5], float(f)/(len(x_set)/len(actP5)))
```

Part 8

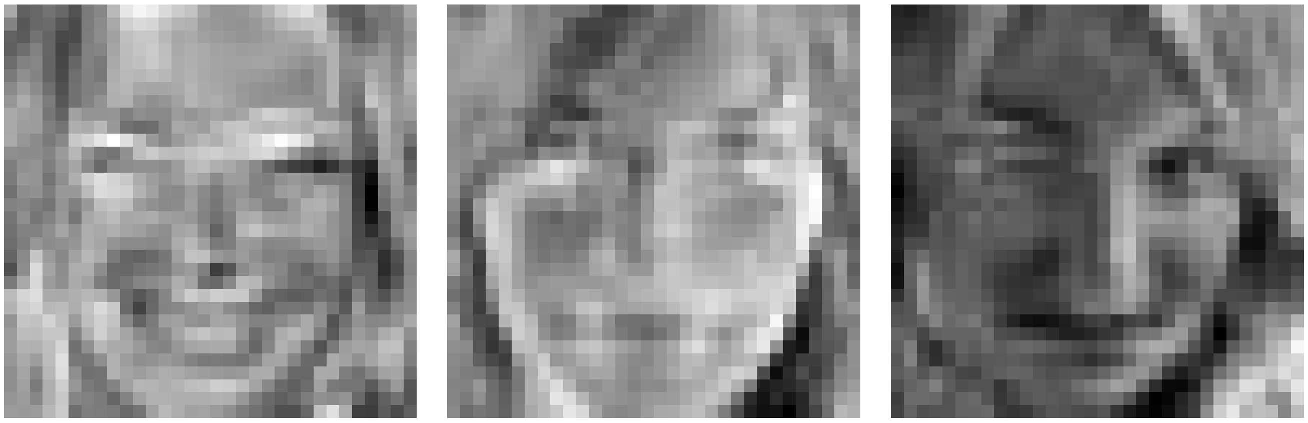


Figure 1: Drescher, Ferrera, Chenoweth with 100 images each respectively

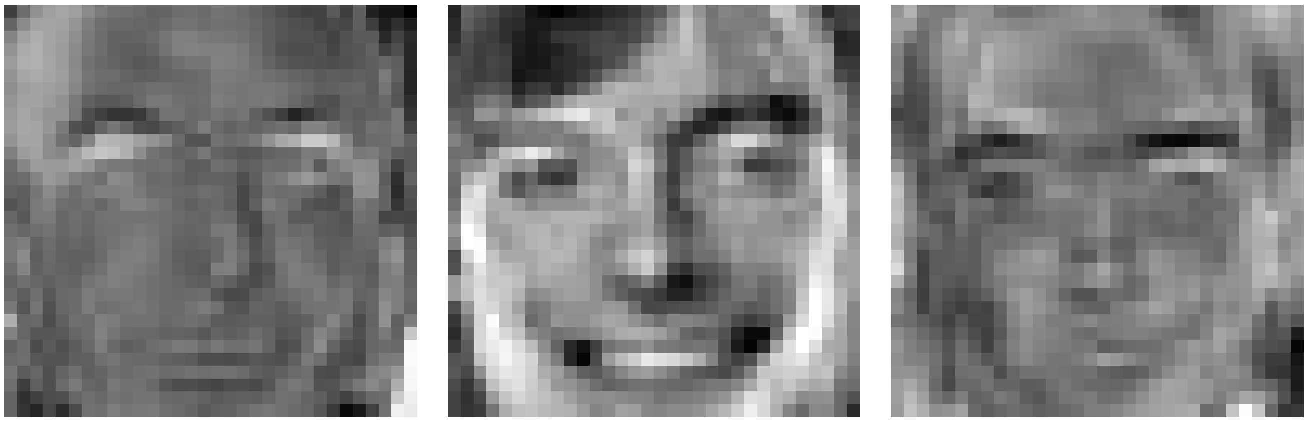


Figure 2: Baldwin, Hader, Steve with 100 images each respectively

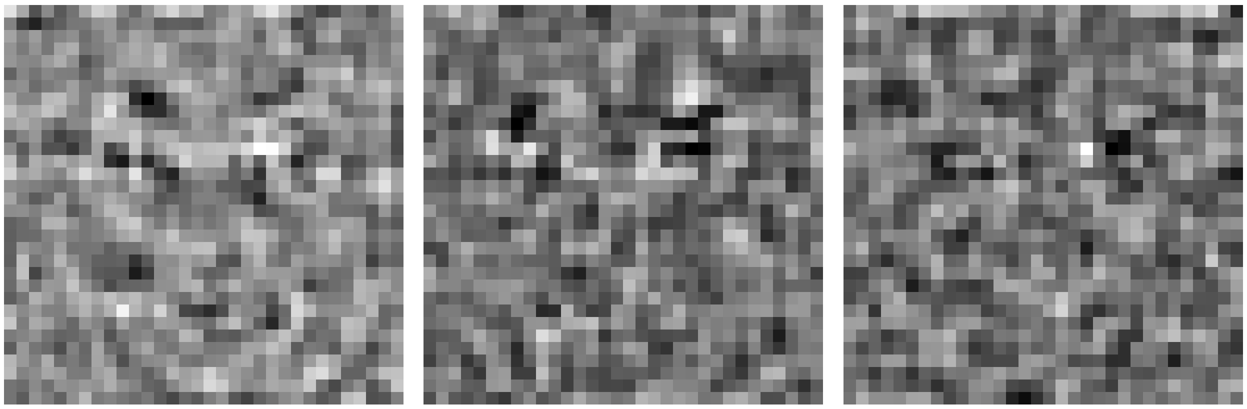


Figure 3: Drescher, Ferrera and Chenoweth with 2 images each respectively

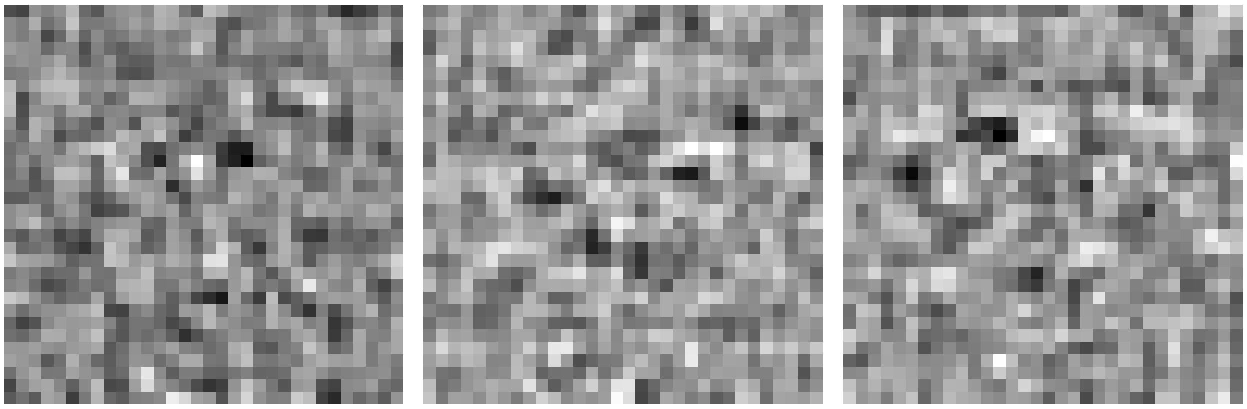


Figure 4: Baldwin, Hader, Steve with 4 images each respectively