CSC411: Assignment #1 Face Recognition

Due on Friday, February 3, 2017

Bowen Chen

February 3, 2017

Part 1

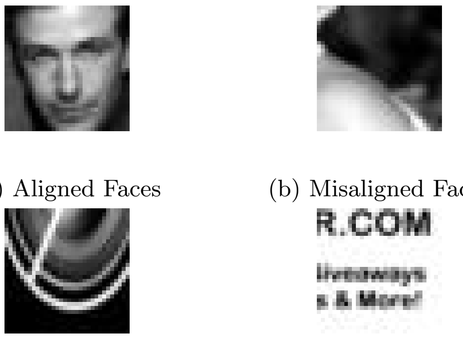
Problem Overview - Data Exploration

The actor images dataset are downloaded from the *FaceScrub* and consists of 1922 images. By examining the data, the images are of different sizes. All the images are three dimensional pixels. Three examples of images in the dataset are shown below in *Figure* 1

(a) (b) (c)

Figure 1: Example Images After Preproccessing (Cropping, Gray Scale)

In order to have a more standardized the data, all the images are pre-processed to gray scale, and the images are cropped and reshaped to 32×32 (Examples are shown in *Figure* 2).After closely examined the data, the pre-processing is not perfect. Most of the images are properly aligned, similar to the image in Panel (a). There exists some miscropped images, which is similar to Panel (b). There are also some images that are not faces at all, Panel (c) (d) for example. (c) shows an image that is a non-face, and exteremly dark, while (d) shows an image that is a non-face and extremely bright. The images were cropped using **update_images** (lines 138 - 153) in **faces.py**.



(a) Aligned Faces (b) Misaligned Faces

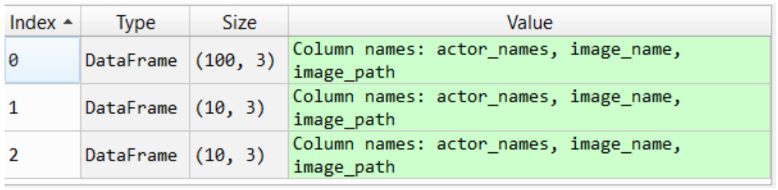
(c) Non Faces - Dark (d) Non Faces - Bright

Figure 2: Example Images After Preproccessing (Cropping, Gray Scale)

Part 2

Prepare Training, Test, Validation Datasets

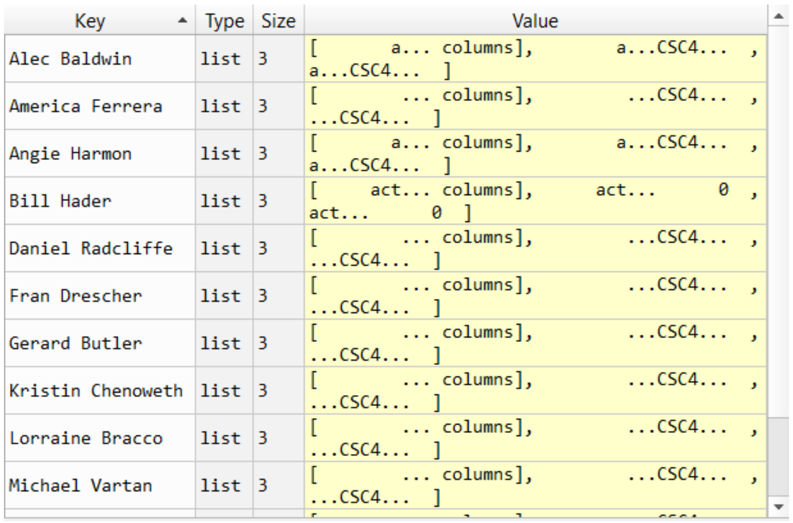
The training, test and validation set are prepared using *PandasDataFrame*.First, the entire dataset of 1922 images is organized into a *Pandas DataFrame* called *actor_dataframe*. *actor_dataframe* contains the *actor_names* and *image_name* columns. Then the *actor_dataframe* is passed into a function named *organize_all_data*. Inside function *organize_all_data*, another function *separate_training_sets* is called, which generates training, test and validation sets, and append them into the list that corresponding to a certain actor. In the DataFrame, only the local paths of the image are stored. Every time when an image is needed, simply call *imread*() and the image could easily be restored. *Figure* 3 shows an example of actor training, test and validation sets.



Index	Type	Size	Value
0	DataFrame	(100, 3)	Column names: actor_names, image_name, image_path
1	DataFrame	(10, 3)	Column names: actor_names, image_name, image_path
2	DataFrame	(10, 3)	Column names: actor_names, image_name, image_path

Figure 3: Alec Baldwin Training, Test Validation Sets Generated By `separate_training_sets`.

The function *organize_ data* then generates a dictionary of 12 lists, with the actor name being the key to the dictionary. *Figure* 4 shows the constructed dictionary of lists.



Key	Type	Size	Value
Alec Baldwin	list	3	[a... columns], a...CSC4... , a...CSC4...]
America Ferrera	list	3	[... columns], ...CSC4... , ...CSC4...]
Angie Harmon	list	3	[a... columns], a...CSC4... , a...CSC4...]
Bill Hader	list	3	[act... columns], act... 0 , act... 0]
Daniel Radcliffe	list	3	[... columns], ...CSC4... , ...CSC4...]
Fran Drescher	list	3	[... columns], ...CSC4... , ...CSC4...]
Gerard Butler	list	3	[... columns], ...CSC4... , ...CSC4...]
Kristin Chenoweth	list	3	[... columns], ...CSC4... , ...CSC4...]
Lorraine Bracco	list	3	[... columns], ...CSC4... , ...CSC4...]
Michael Vartan	list	3	[... columns], ...CSC4... , ...CSC4...]

Figure 4: The `organized_data` dictionary generated by `organize_all_data`

Part 3

Linear Regression Classifier on Two Actors - Bill Hader and Steve Carell

A linear regression model is built to classify two actors - *Bill Hader* and *Steve Carell*. I represent *Bill* as 0 and *Steve* as 1. The cost function I minimized is the sum of square function

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

Note: Since the image data are already normalized, the fraction component $\frac{1}{2m}$ *is omitted from the cost function above .*

The initial estimation of θ is set to be a 1025 $\times$ 1 vector with all components being 0. The step size α was chosen to be 0.00001.The model is trained with gradient descent algorithm with maximum number of iterations of 30000 in the function *train_regression_model* and evaluated by the function $evaluate\_performance$ under $Part\_$ 3 of **faces.py**. The performance of the classifier is measured by the *accuracy* of the model, which is defined as

$$accuracy = \frac{number\ of\ correctly\ images}{total\ number\ of\ images\ supplied\ to\ the\ classifier}$$

The accuracy on the training set is found to be 100%, the cost function value on training set is 0.304
The accuracy on the validation set is found to be 90% , the cost function value on validation set is 1.377

Note: All the results were generated from Linux, python 2.7.13. I have slightly different performance results in Windows.

Choice of step size α: The choice of the step size is crucial to the performance of gradient descent algorithm. When the step size α is set to be too large, the gradient descent algorithm will diverge as the function value will go to ∞. When the step size is set to be too small, the gradient descent algorithm will take a rather large number of iterations to converge. With the maximum number of iteration fixed, the program will terminate before the local minimum is found. This will lead to a decrease in performance.

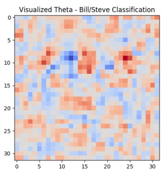
The code that are used in building the linear regression classifier is shown below.The cost function, derivative of cost function and gradient descent function not shown. The functions codes are from lines 217 to 302, and the main function codes are from lines 547 to 557 of **faces.py**

```
def add_Bill_label(Bill_data):
# Take in the dataframe that corresponds to Bill
# and add a column true label of 1
    for i in Bill_data:
        i['label'] = 0
    return Bill_data

def add_Steve_label(Steve_data):
# Take in the dataframe that corresponds to Steve
# and add a column true label of 0
    for i in Steve_data:
        i['label'] = 1
    return Steve_data

def build_regression_sets_two_actors(Bill_Data,Steve_Data):
# Take in the Bill and Steve Data (with true labels) and build a list called
# regression_sets [training_set, test_set, validation_set]
    training_set =pd.concat([Bill_Data[0], Steve_Data[0]], axis=0)
    test_set = pd.concat([Bill_Data[1], Steve_Data[1]], axis=0)
    validation_set = pd.concat([Bill_Data[2], Steve_Data[2]], axis=0)
    regression_sets = [training_set, test_set, validation_set]
    return regression_sets

def reshape_image(image_path):
    image = imread(image_path)/255.0
    reshpaed_image = np.reshape(image,(1,1024))
    return reshpaed_image

def get_regression_parameters(sets):
# Get x and y that will be used for linear regression and put in a list [x, y]
    x = np.ones((1,1024))
    for j in sets['image_path']:
        reshaped_image = reshape_image(j)
        x = np.vstack((x, reshaped_image))
    A = [1] * np.shape(x[1:,:])[0]
    x= np.vstack((A, (x[1:,:]).T))
    y = sets['label']
    set_inputs = [x, y]

    return set_inputs

def train_regression_model(training_sets, f, df, alpha):
# The function that trains the linear regression model
    [train_x,train_y] = get_regression_parameters(training_sets)
    train_y = train_y.reshape(1,len(train_y))
    initial_theta  = np.array([0.0]*1025).reshape(1025,1)
    max_iter = 30000
    theta = gradient_descent(f, df, train_x, train_y, initial_theta, alpha,\
                                            max_iter)[0]
    return theta

def evaluate_performace(test_set,theta):
# returns accuracy of the input set and predicted y from the model
  [test_x,test_y] = get_regression_parameters(test_set)
  hypothesis_y_test = dot(theta.T,test_x)
  hypothesis_y_test[hypothesis_y_test>= 0.5] = 1
  hypothesis_y_test[hypothesis_y_test< 0.5] = 0

  test_y = test_y.reshape(1,len(test_y))
  total_correct = 0

  for i in range(np.shape(test_y)[1]):
      if hypothesis_y_test[0][i] == test_y[0][i]:
          correct = 1
      else:
                correct = 0
            total_correct += correct

        return [total_correct*1.0/np.shape(test_y)[1], hypothesis_y_test]

if __name__ == "__main__":
     #    organize_all_data from part 2
    organized_data = organize_all_data(act, actor_dataframe, image_name_list, \
                                  cropped_path)
    Bill_Data = add_Bill_label(organized_data['Bill Hader'])
    Steve_Data = add_Steve_label(organized_data['Steve Carell'])
    regression_sets = build_regression_sets_two_actors(Bill_Data, Steve_Data)
    initial_theata = np.array([0]*1025).reshape(1025,1)

    alpha = 0.00001
    # the theta that is generated using 100 images
    theta_full = train_regression_model(regression_sets[0], f, df, alpha)
    # predicted y and true
      hypothesis_y = evaluate_performace(regression_sets[1],theta_full)[1]
    true_y = get_regression_parameters(regression_sets[1])
```

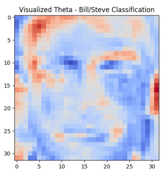
Part 4

Visualize θ with a model trained by 100 images and a model train by 2 images

The weights of the regression θ is shown excluding θ_0 could be visualized as images. Since the vector $\theta_{adjusted} = (\theta_1, ... \theta_{1024})$ is currently a 1024×1 vector, $\theta_{adjusted}$ need to be reshaped to 32×32 in order to be shown as image. The visualized theta for a model trained by 100 images and a model trained by 2 images are shown below in $Figure$ 5



(a) n = 100



(b) n = 4

Figure 5: Visualized θ for models trained by different sizes of training set (n)

Discussion: when the training set only contains 2 images, it is relatively easy for the model to fit all the data points. Therefore, the visualized θ will pickup some important features of the two faces (as shown in (b)). If the training set is large, the model will pick up more features from different faces, which makes the visualized θ image less close to a face

Part 5

Overfiting

The important question for every machine learning model is whether the model has been *overfitted*. *Overfiting* is defined as the excessive amount of features relative to the number of data points in the data set. To demonstrate the effect of textitoverfiting, a gender classification model trained using a training set that consists of 6 different actors is evaluated. The validation set is also generated from the same 6 actors.

```
        act =['Fran Drescher', 'America Ferrera', 'Kristin Chenoweth',
               'Alec Baldwin', 'Bill Hader', 'Steve Carell']
```

The size of training set ranges from 2 to 802. A plot of different size of training set (n) vs performances (accuracy) on training set and validation set are shown below in *Figure* 6.

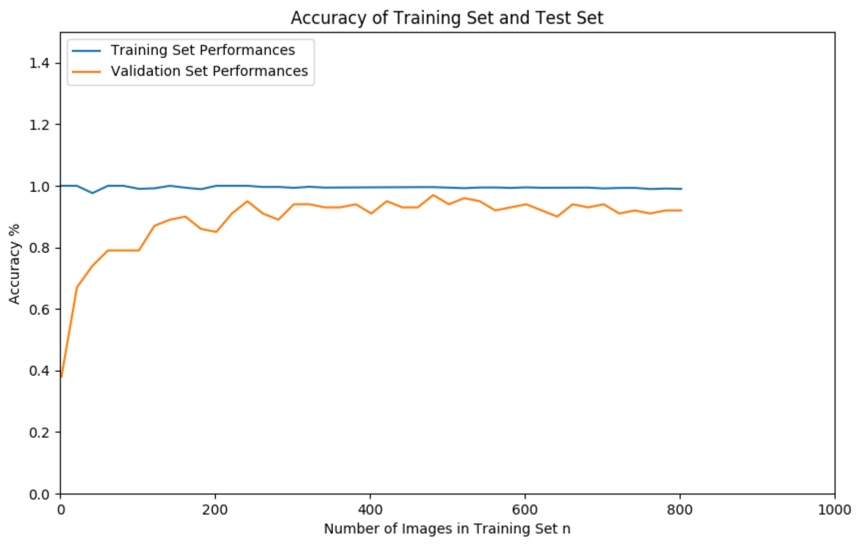


Figure 6: Performances vs Different Training Set Size (n)

The regression model contains 1024 features, which can not be modified. In changing the training set size n, we are shifting the difference between number of training data in the training set and number of features. The smaller n is, the more overfitted the model is. From *Figure* 6, we can see that for smaller training set size n, both the training set and validation set performances decreases. As training set size gets larger, the model is less overfitted than the case when training set is small. The training set could only get 50% accuracy when n = 2, and it have a great improvement on accuracy when the training set is larger than 20. The validation set performance gradually increase as the training set size increases.

The model is also tested on the other 6 actors who do not belong to the act.

```
        act_test = ['Gerard Butler', 'Daniel Radcliffe', 'Michael Vartan',
                     'Lorraine Bracco', 'Peri Gilpin', 'Angie Harmon']
```

The accuracy on the validation set that are from act_test is found to be 88.6%

Part 6

One-Hot Encoding in Preparation of Multi-Class Classifications

6(a) The cost function for multi-class classification is

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

y is an m×k matrix, with k representing the number of different categories in the classification model. x dimension stays the same, which was m×n. In order to match dimensions over the equation y = $\theta^T x$, θ need to have dimensions of k×n.

In matrix form,

$$\theta = \begin{bmatrix} \theta_{01} & \theta_{11} & \theta_{21} & \dots & \theta_{k1} \\ \theta_{02} & \theta_{12} & \theta_{22} & \dots & \theta_{k2} \\ \vdots & \vdots & \vdots & \boldsymbol{\theta_{pq}} & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_{0n} & \theta_{1n} & \theta_{2n} & \dots & \theta_{kn} \end{bmatrix}$$

For a single element in θ matrix θ_{pq}, the partial derivative of cost function J(θ) with respect to θ_{pq} is

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

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

$$= \sum_{i=1}^{n}\sum_{j=1}^{k} 2x^{(i)}(\theta^T x^{(i)} - y^{(i)})_j$$

Note: When computing $\frac{\partial}{\partial \theta_{pq}}(\theta^T x^{(i)})$, for any element in the matrix θ_{ij}, $\frac{\partial}{\partial \theta_{pq}}\theta^T = 1$ for the element $i = p$ ***and*** *$j = q$, $\frac{\partial}{\partial \theta_{pq}}\theta^T = 0$ for any element where $i \neq p$* ***or*** *$j \neq q$, Therefore,*

$$\frac{\partial J(\theta)}{\partial \theta_{pq}} = \sum_{i=1}^{n} 2 \cdot 1 \cdot x^{(p)}(\theta_{pq}x^{(p)} - y^{(p)})_q + \sum_{i \neq p}\sum_{j \neq q} 2 \cdot 0 \cdot x^{(i)}(\theta^T x^{(i)} - y^{(i)})_j$$

$$= \sum_{i=1}^{n} 2x^{(i)}(\theta_{iq}x^{(i)} - y^{(i)})_q + 0$$

$$= \sum_{i=1}^{n} 2x^{(i)}(\theta_{iq}x^{(i)} - y^{(i)})_q$$

6(b) For any single element θ_{pq} in matrix θ, we have $\frac{\partial J(\theta)}{\partial \theta_{pq}} = \sum_{i=1}^{n} 2x^{(p)}(\theta_{pq}x^{(p)} - y^{(p)})_q$, if we organize all of the $\frac{\partial J(\theta)}{\partial \theta_{pq}}$ into a matrix $\frac{\partial J(\theta)}{\partial \theta}$, the matrix $\frac{\partial J(\theta)}{\partial \theta}$ will have the same dimensions as the matrix θ at k×m.

In particular,

$$\frac{\partial J(\theta)}{\partial \theta} = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_{01}} & \frac{\partial J(\theta)}{\partial \theta_{11}} & \frac{\partial J(\theta)}{\partial \theta_{21}} & \cdots & \frac{\partial J(\theta)}{\partial \theta_{k1}} \\ \frac{\partial J(\theta)}{\partial \theta_{02}} & \frac{\partial J(\theta)}{\partial \theta_{12}} & \frac{\partial J(\theta)}{\partial \theta_{22}} & \cdots & \frac{\partial J(\theta)}{\partial \theta_{k2}} \\ \vdots & \vdots & \vdots & \boldsymbol{\frac{\partial J(\theta)}{\partial \theta_{pq}}} & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J(\theta)}{\partial \theta_{0n}} & \frac{\partial J(\theta)}{\partial \theta_{1n}} & \frac{\partial J(\theta)}{\partial \theta_{2n}} & \cdots & \frac{\partial J(\theta)}{\partial \theta_{kn}} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{i=1}^{n} 2x^{(i)}(\theta_{01}x^{(i)} - y^{(i)})_1 & \cdots & \sum_{i=1}^{n} 2x^{(i)}(\theta_{k1}x^{(i)} - y^{(i)})_1 \\ \sum_{i=1}^{n} 2x^{(i)}(\theta_{02}x^{(i)} - y^{(i)})_2 & \cdots & \sum_{i=1}^{n} 2x^{(i)}(\theta_{k2}x^{(i)} - y^{(i)})_2 \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} 2x^{(i)}(\theta_{01}x^{(i)} - y^{(i)})_n & \cdots & \sum_{i=1}^{n} 2x^{(i)}(\theta_{kn}x^{(i)} - y^{(i)})_n \end{bmatrix}$$

For all elements in matrix $\frac{\partial J(\theta)}{\partial \theta}$, we can write the elements of $x_j^{(i)}s$ as matrix X, the elements of $y_j^{(i)}s$ as matrix Y, the elements of θ_{ij} as a matrix θ, then the derivative of cost function J(θ) could be written as the matrix multiplication forms, after the dimensions are matched, we have,

$$= 2X(\theta^T x - Y)^T$$

- X is the matrix that contains the training (testing/validation) datasets, in other words, input variable matrix. X will have dimensions of m×n
- Y is the matrix that contains the true label on different classes. Y will have dimensions of m×k
- θ is the weights that each feature will be allocated in the classification model. θ will have dimensions of k×n

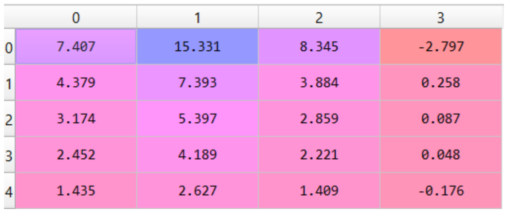
6(c) According to 6(b), the new cost function for the multi-class classification model could be implemented in Python.

```python
def f_multi_class(x, y, theta):
# Cost function J
    return  sum((y - dot(theta.T,x)) ** 2)

def df_multi_class(x, y, theta):
# Derivative of Cost function J
    return 2 * dot(x,(dot(theta.T, x) - y).T)
```

```
  def finite_difference(x, y, theta):
  # Derivative of cost function J computed by finite difference methods
      delta_h = 0.0000001
      df_FD = np.zeros(np.shape(theta))
      for i in range(np.shape(theta)[0]):
          for j in range(np.shape(theta)[1]):
              h = np.zeros(np.shape(theta))
              h[i][j] = delta_h
              df_FD[i][j] = (f_multi_class(x, y, theta+h) \
                  - f_multi_class(x,y, theta-h))/(2*delta_h)
      return df_FD

if __name__ == "__main__":

    gradient_trial_x = training_x_multi[0:5,0:6]
    gradient_trial_y = training_y_multi[0:4,0:6]
    gradient_trial_theta = np.random.rand(5,4)
    df = df_multi_class(gradient_trial_x, gradient_trial_y,\
                                 gradient_trial_theta)
    df_FD = finite_difference(gradient_trial_x, gradient_trial_y, \
                                 gradient_trial_theta)
```

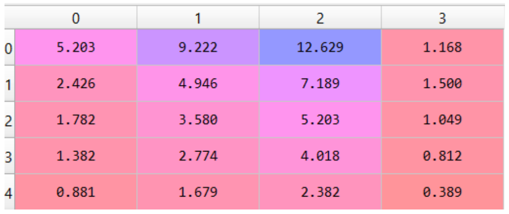
6(d) In order to prove the function to prove the vectorized gradient function is functional, the result return by that function is compared against the result returned by finite differences. Since the image data set is large and finite difference method in computing the gradient would be quite time consuming. I selected only the top left corner components of x, y and randomly generated θ and calculate the grcost function value. The results are shown in below *Figure* 7.

Note: Below figure is generated from Windows. In Linux, we will have two slightly different matrices (a), (b), but their difference (c) will be really similar



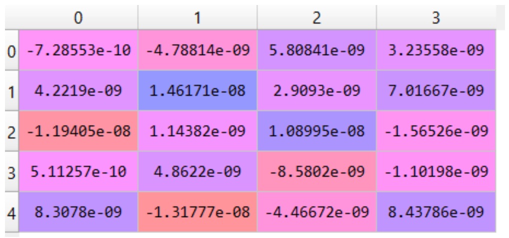
	0	1	2	3
0	7.407	15.331	8.345	-2.797
1	4.379	7.393	3.884	0.258
2	3.174	5.397	2.859	0.087
3	2.452	4.189	2.221	0.048
4	1.435	2.627	1.409	-0.176

(a) derivative computed from vectorized gradient function



	0	1	2	3
0	5.203	9.222	12.629	1.168
1	2.426	4.946	7.189	1.500
2	1.782	3.580	5.203	1.049
3	1.382	2.774	4.018	0.812
4	0.881	1.679	2.382	0.389

(b) derivative computed from finite difference method



	0	1	2	3
0	-7.28553e-10	-4.78814e-09	5.80841e-09	3.23558e-09
1	4.2219e-09	1.46171e-08	2.9093e-09	7.01667e-09
2	-1.19405e-08	1.14382e-09	1.08995e-08	-1.56526e-09
3	5.11257e-10	4.8622e-09	-8.5802e-09	-1.10198e-09
4	8.3078e-09	-1.31777e-08	-4.46672e-09	8.43786e-09

(c) Difference between gradient calculated by two methods

Figure 7: Evaluating Vectorized Gradient Function

(a) (b) shows the gradient calculated by two different methods.(c) shows the difference among all components between two methods.It is obvious that all components in (c) is very small, which proves that the vectorized gradient function is functional.

Part 7

Multi-class Classification

In *part* 6, we examined the means of encoding different classes of data with One-Hot Encoding technique. With this method, a model of classifying the six actors that belongs to act is evaluated.

```
act =['Fran Drescher', 'America Ferrera', 'Kristin Chenoweth',
      'Alec Baldwin', 'Bill Hader', 'Steve Carell']
```

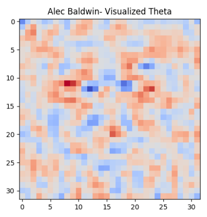
The step size α choice is particularly tricky in this case. I experimented 5 different αs and found out that $\alpha = 0.000001$ is the best choice. With this particular step size, the gradient descent algorithm converges relatively fast, and the cost function could be effectively prevented from going to ∞. The maximum number of iterations are set to be 50000, which could produce a relatively high accuracy on the validation set, and also prevent the model from being overfitted.

The accuracy on the training set is found to be 97.0%
The accuracy on the validation set is found to be 79.2%

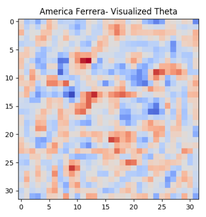
Part 8

Visualize θ for Multi-class Classification

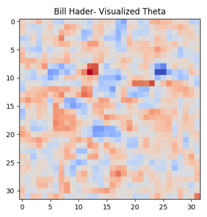
Same as the techniques we used in *part* 4, the rows of θ matrix generated from regression could also be visualized. By removing the column of that corresponds to θ_0s, the resulting matrix has dimension of 1024×6. Since each row of the θ corresponds to a single actor, each row of θ could be reshaped to a 32×32 image. The generated 6 visualized θs of 6 actors are shown below in *Figure* 8.



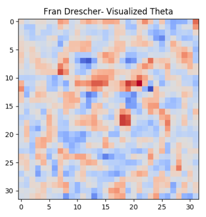
(a) Alec Baldwin



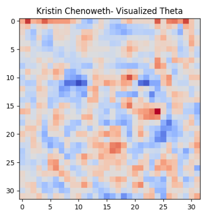
(b) America Ferrera



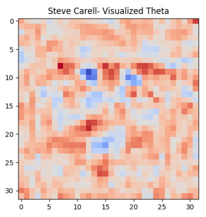
(c) Bill Hader



(d) Fran Drescher



(e) Kristin Chenoweth



(f) Steve Carell

Figure 8: Visualized Theta for Multi-Class Classfications - 6 Actors

Part 9

Appendix - Instructions on how to run the code

My code utilizes the file $faces_subset.txt$, the program will generate one csv file called $crop_coordinates.csv$, which stored the coordinates for the each image in the dataset in order to crop out the face. There are two ways to run the program.

1. **Run from the beginning**uncomment $Part$ 1 and first half of $Part$ 2, and put **faces_subset.txt** under same directory with **faces.py**.The code will download images and proceed with all the processes, but it will likely take a long time.
2. **Run from crop images** Comment $Part$ 1 and first half of $Part$ 2 until the line included

   ```
   update_images(image_list_uncroped,crop_coordinates)
   ```

   faces_subset.txt should be extracted under the same directory as **faces.py**. I have also attached the zip file for all cropped images, and the zip file should be extracted under the same directory as **faces.py**. The program will continue on from part 2 and proceed with all the parts.