CSC411: Assignment #2 Deep Neural Networks

Due on Monday, March 6, 2017

Bowen Chen, Yuan Yao

March 6, 2017

Part 1

Problem Overview - Data Exploration

The handwritten digits images dataset are from $MINST$ dataset and consists of 60000 images in total. By examining the data, all the images are of gray scale and same size. The training set images and the test set images are already separated. The data is saved in a dictionary format, with each digit label as $train/test_number$ format. The examples of images in the dataset are shown below in $Figure$ 1

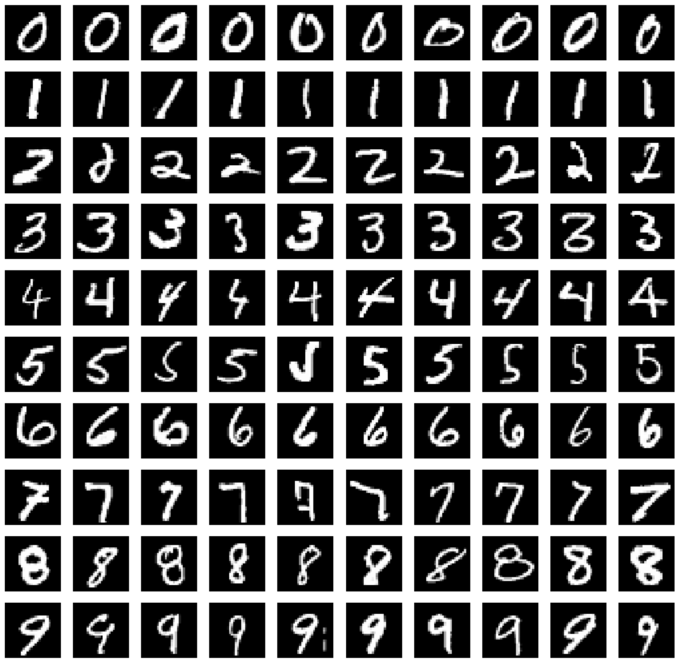


Figure 1: Ten Sample Images per Digits

Part 2

Implementation of Softmax Function

In this part, we implemented the softmax cost function similar to the top layer (output layer) of the neural network. The architecture of the network is shown below

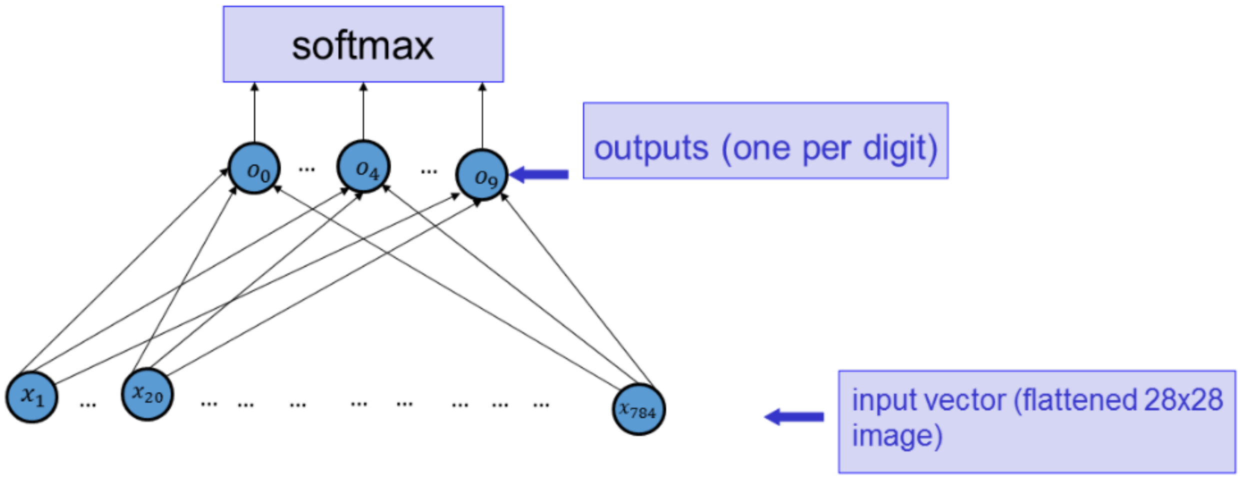


Figure 2: The Neural Network Architecture

The purpose of the softmax function is to transform the output of the layer of the neural network into values of range between 0 and 1. Softmax could also be viewed as a probability of a certain outcome. In the context of our problem, the softmax function is used as a cost function. The code of building such cost function is shown below.

```
    def softmax(z):
        softmax = (np.exp(z).T / np.sum(np.exp(z),axis=1)).T
        return softmax

    def f(x,y,w):
        m = np.shape(x)[1]
        z = np.dot(x,w)
        p = softmax(z)

        return -1.0/m *np.sum(y*np.log(p))
```

Part 3

The Implementation of the Softmax Cost Function on Digits Classification

3(a)

$$\frac{\partial C}{\partial W_{ij}} = \frac{\partial C}{\partial O_i}\frac{\partial O_i}{\partial W_{ij}}$$

Where

$$\frac{\partial C}{\partial P_j} = -\frac{y_j}{P_j}$$

$$\frac{\partial C}{\partial O_i} = \sum_{j=1}^{M}\frac{\partial C}{\partial P_j}\frac{\partial P_j}{\partial O_i}$$

if i = j ,

$$\frac{\partial P_j}{\partial O_j} = \frac{e^{O_j}\cdot\sum_j e^{O_j} - e^{O_j}\cdot e^{O_j}}{(\sum_j e^{O_j})^2}$$

$$= \frac{e^{O_j}(\sum_j e^{O_j} - e^{O_j})}{(\sum_j e^{O_j})^2}$$

$$= \frac{e^{O_j}}{\sum_j e^{O_j}}\cdot\frac{\sum_j e^{O_j} - e^{O_j}}{\sum_j e^{O_j}}$$

$$= P_j\cdot(1-P_j)$$

if i ≠ j ,

$$\frac{\partial P_j}{\partial O_i} = \frac{0\cdot\sum_i e^{O_i} - e^{O_i}\cdot e^{O_j}}{(\sum_i e^{O_i})^2}$$

$$= -\frac{e^{O_j}}{\sum_j e^{O_i}}\cdot\frac{e^{O_j}}{\sum_j e^{O_i}}$$

$$= -P_j\cdot P_i$$

Then ,

$$\frac{\partial C}{\partial O_i} = \sum_{j=1}^{M}\frac{\partial C}{\partial P_j}\frac{\partial P_j}{\partial O_i}$$

$$= \sum_{i=j}^{M}\frac{\partial C}{\partial P_j}\frac{\partial P_j}{\partial O_i} + \sum_{i\neq j}^{M}\frac{\partial C}{\partial P_j}\frac{\partial P_j}{\partial O_i}$$

$$= \sum_{i=j}^{M} -\frac{y_j}{P_j}\cdot P_j\cdot(1-P_j) + \sum_{i\neq j}^{M}(-\frac{y_j}{P_j})\cdot\ (-P_j\cdot P_i)$$

$$= \sum_{i=j} y_j\cdot P_j + \sum_{i\neq j} y_j\cdot P_j - y_i$$

$$= \sum_{i=1}^{M} y_i \cdot P_i - y_i$$

Since there is only 1 value of y_j across j has value 1, then

$$\frac{\partial C}{\partial O_i} = P_i - y_i$$

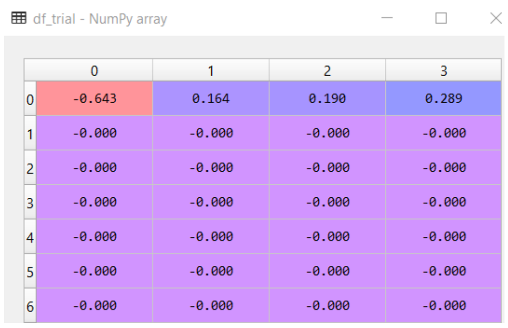
Then ,

$$\frac{\partial C}{\partial W_{ij}} = \frac{\partial C}{\partial O_i} \frac{\partial O_i}{\partial W_{ij}}$$

$$= (P_i - y_i) \cdot X_j$$

3(b)

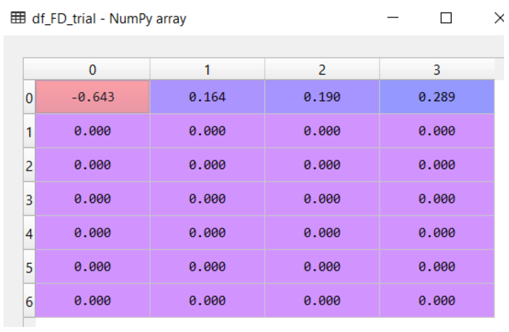
The vectorized gradient of the cost function (function *df*) is verified against the gradient obtained from the finite difference method (function *df_FD*). The error between two result gradient are very small. The calculations are shown below



df_trial - NumPy array

	0	1	2	3
0	-0.643	0.164	0.190	0.289
1	-0.000	-0.000	-0.000	-0.000
2	-0.000	-0.000	-0.000	-0.000
3	-0.000	-0.000	-0.000	-0.000
4	-0.000	-0.000	-0.000	-0.000
5	-0.000	-0.000	-0.000	-0.000
6	-0.000	-0.000	-0.000	-0.000

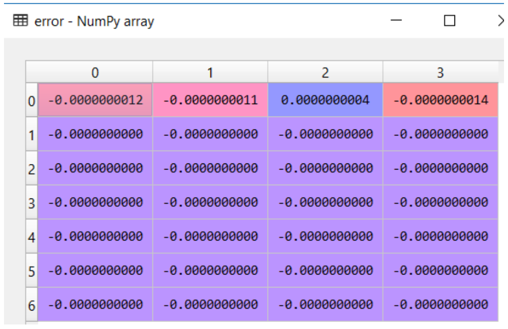
(a) derivative computed from vectorized gradient method



df_FD_trial - NumPy array

	0	1	2	3
0	-0.643	0.164	0.190	0.289
1	0.000	0.000	0.000	0.000
2	0.000	0.000	0.000	0.000
3	0.000	0.000	0.000	0.000
4	0.000	0.000	0.000	0.000
5	0.000	0.000	0.000	0.000
6	0.000	0.000	0.000	0.000

(b) derivative computed from finite difference method



error - NumPy array

	0	1	2	3
0	-0.0000000012	-0.0000000011	0.0000000004	-0.0000000014
1	-0.0000000000	-0.0000000000	-0.0000000000	-0.0000000000
2	-0.0000000000	-0.0000000000	-0.0000000000	-0.0000000000
3	-0.0000000000	-0.0000000000	-0.0000000000	-0.0000000000
4	-0.0000000000	-0.0000000000	-0.0000000000	-0.0000000000
5	-0.0000000000	-0.0000000000	-0.0000000000	-0.0000000000
6	-0.0000000000	-0.0000000000	-0.0000000000	-0.0000000000

(c) Difference between gradient calculated by two methods

Figure 3: 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 cost function is functional.
The code that is used in this part is demonstrated below

```
def f(x,y,w):
  m = np.shape(x)[1]
  z = np.dot(x,w)
  p = softmax(z)

  return -1.0/m *np.sum(y*np.log(p))

def df(x,y,w):
  m = np.shape(x)[1]
  z = np.dot(x,w)
  p = softmax(z)

return -1.0/m * np.dot(x.T,(y-p))


def finite_difference(x, y, w):
  delta_h = 0.0000001
  df_FD = np.zeros(np.shape(w))
  for i in range(np.shape(w)[0]):
  for j in range(np.shape(w)[1]):
  h = np.zeros(np.shape(w))
  h[i][j] = delta_h
  df_FD[i][j] = (f(x, y, w+h) - f(x,y, w-h))/(2*delta_h)
  return df_FD

if __name__ == "__main__":

  random.seed(100)

  x = train_x[0:6, 0:7]
  y = train_y[0:6, 0:4]
  w = np.random.rand(7,4)

  df_trial = df(x,y,w)
  df_FD_trial = finite_difference(x, y, w)

  print ('The derivative of cost funtion computated by df_multi_class is \n',
       df_trial)
  print ('The derivative of cost funtion computated by finite difference is \n',
      df_FD_trial)
  error = df_trial - df_FD_trial
  print ('The difference between two methods is \n',error)
```

Part 4

Network Training Results

The network in part 2 is implemented and trained with gradient descent algorithm.

Key Decisions in the implementations:

Learning rate(alpha) was adjusted to speed up the convergence process. since the total number of training set is relatively larger compare to previous project. However, when alpha is too large, the gradient starts to diverge. So the final alpha is set at 0.001. Given the learning rate fixed, the maximum number of iteration is set to 100000. However, after the iteration reaches 3000, the performance does not change even if iteration continues to increase. So the maximum number of iteration is set to 3000. The initial_w (weights) is set to 0 because this not a huge neural network, this network has no hidden layer, and 0 is likely to be close to final result.

The learning curve is shown below in the *Figure* 4

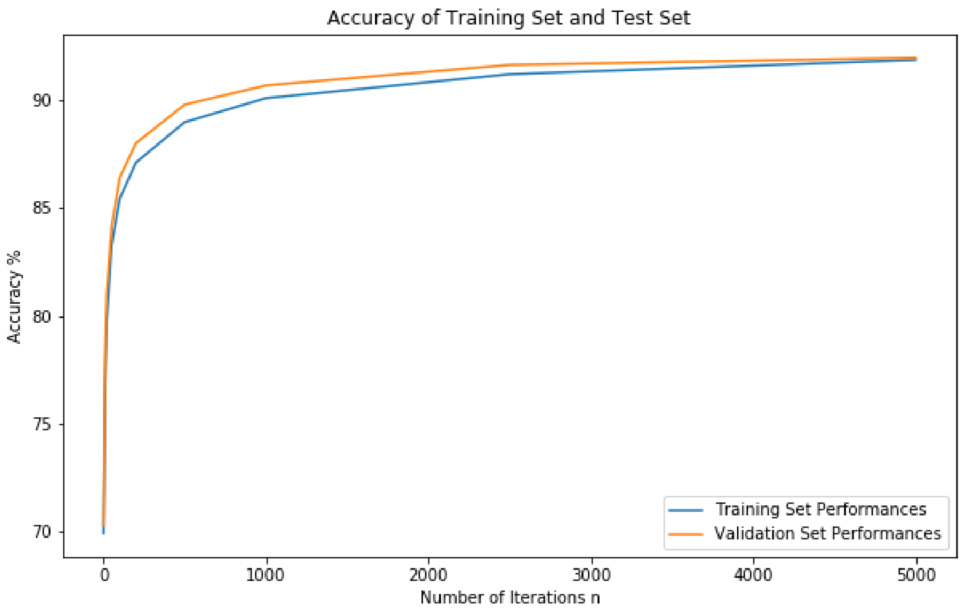


Figure 4: The Learning Curve for the Part 2 Network with 5000 Training Iterations

The change of each set of weights going into each unit for different iterations is shown below in *Figure* 5, the weights are recorded at iteration number 1, 10, 20, 50, 100, 200, 500, 1000, 2500, 5000. From *Figure* 5, each set of weight is plotted vertically (weights corresponding to digits of 0 to 9). Along the rows shows the change of weights at those 10 different iterations.

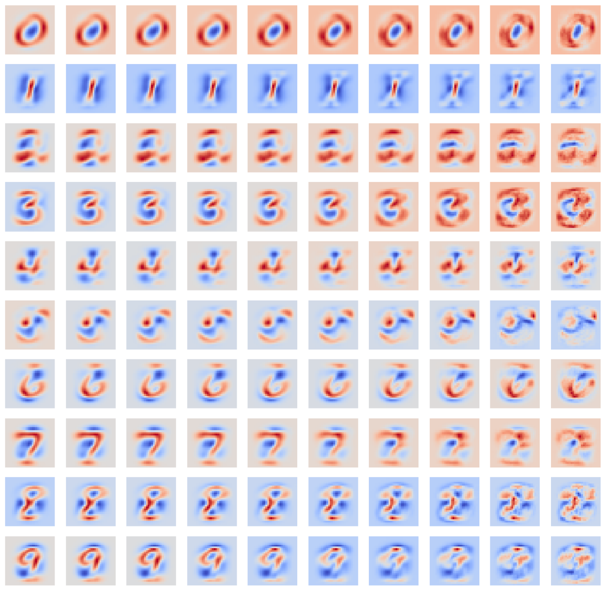


Figure 5: The Change of Weights Visualization for Each Number With Respect to Iterations

Part 5

Multi-nominal Logistic vs Linear Regression

In order to demonstrate the performance of using Logistic method is significantly better than using linear method, x was randomly generated, and y was calculated using a initial_w + error term. The error term was generated using normal distribution with large sigma. With a large sigma, the possibility of having a data point that have big impact on cost function is large for linear regression method, which uses Euclidean distances as cost function. On the contrary, the logistic regression with softmax cost function will be less affected by the large error term since the logistic cost function transforms the large outputs to range between 0 and 1. So the performance using logistic method will be better than performance using linear method on test set. As shown below, the performance on training for both method is great, so the training is at least not under fit. The linear method is trained fully and stop training but logistic is not even trained fully.

The performance on training set using logistic method is : 86.25%
The performance on test set using logistic method is : 83.3%

The performance on training set using linear method is : 73.2833%
The performance on test set using linear method is : 71.7%

The code used to conduct this experiment is shown below,

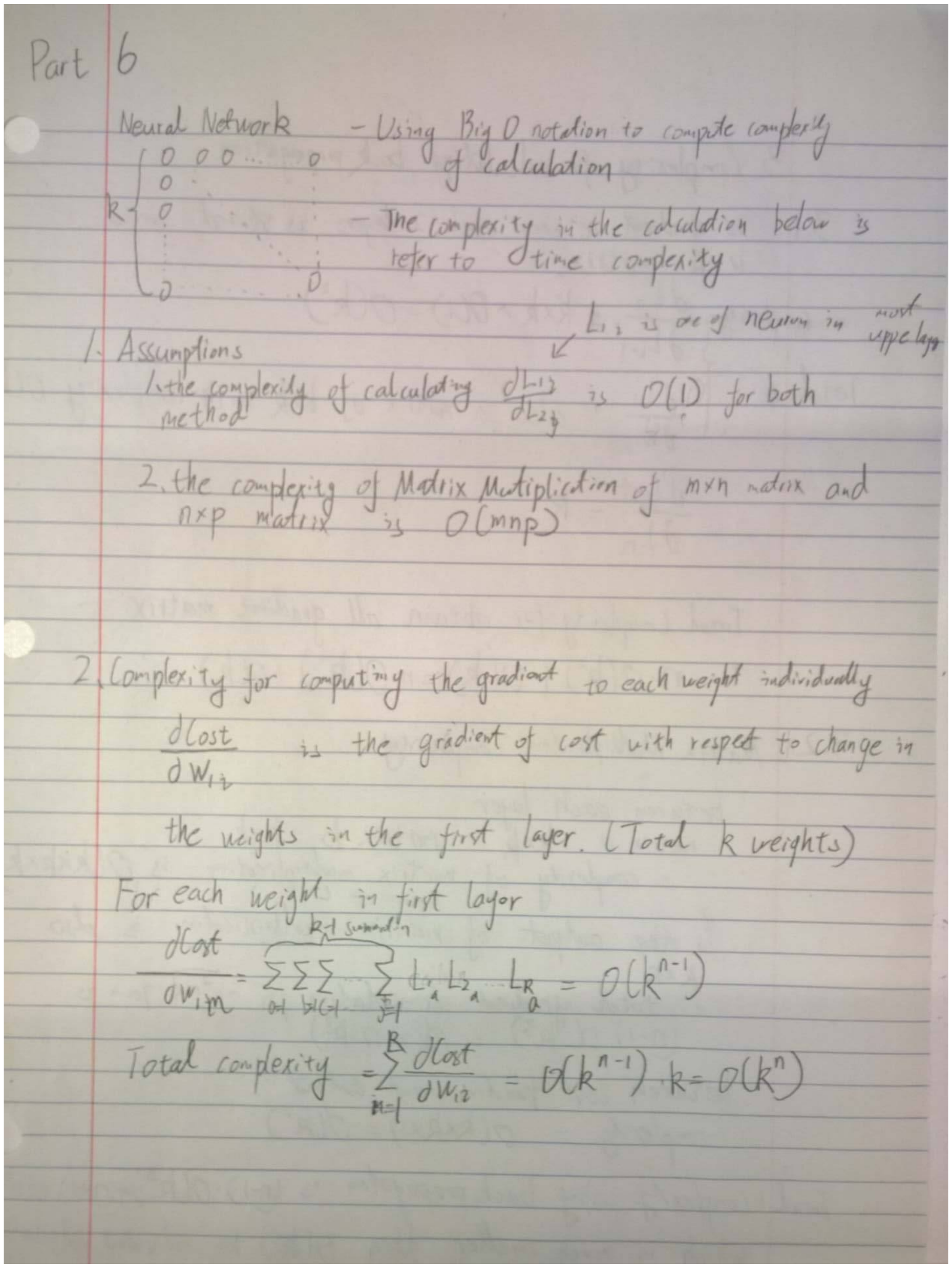
```
def simulation_trial_two_models():
    sigma = 0.1

    w_size = (785,10)
    part5_w = np.random.random(w_size)
    train_x_size = (12000, 785)
    train_y_size = (12000, 10)
    test_x_size = (1000, 785)
    test_y_size = (1000, 10)

#   Normal
    simu_train_x = np.random.random(train_x_size)
    simu_train_y = softmax(np.dot(simu_train_x, part5_w)
                + scipy.stats.norm.rvs(scale = sigma, size=train_y_size))
    simu_test_x = np.random.random(test_x_size)
    simu_test_y = softmax(np.dot(simu_test_x, part5_w)
                + scipy.stats.norm.rvs(scale = sigma, size=test_y_size))

    alpha_softmax = 0.0005
    max_iter_soft = 20000
    w_softmax = train_model (simu_train_x,simu_train_y, f, df,
                            alpha_softmax, max_iter_soft)
    alpha_linear = 0.00001
    max_iter_linear = 100000
    w_linear = train_model(simu_train_x,simu_train_y, f_SSE, df_SSE,
                        alpha_linear, max_iter_linear)

    multi_performance_train = evaluate_model_trials (simu_train_x,simu_train_y,
                            w_softmax)
```

```
      multi_performance_test = evaluate_model_trials (simu_test_x,simu_test_y,
                               w_softmax)


      linear_performance_train = evaluate_model_trials (simu_train_x,simu_train_y,
                                   w_linear)
      linear_performance_test = evaluate_model_trials (simu_test_x,simu_test_y,

      print ('Training Set performance using multinomial method is:'
              + str(multi_performance_train *100) + "% \n" )
      print ('Test Set performance using multinomial method is:'
              + str(multi_performance_test *100) + "% \n")
      print ('Training Set performance using linear method is:'
              + str(linear_performance_train *100) + "% \n")
      print ('Test Set performance using linear method is:'
          + str(linear_performance_test *100) + "% \n")

     return [multi_performance_train, multi_performance_test, linear_performance_train,
linear_performance_test]

    if __name__ == "__main__":

      np.random.seed(0)
      multi_performance_train, multi_performance_test,
      linear_performance_train,  linear_performance_test = simulation_trial_two_models()
```

Part 6

Computation time complexity between calculate individual weights and backpropagation

As shown in the picture below, the time computation complexity is of $O(k^n)$ for backpropagation and $n \times O(k^3)$.



Part 6

Neural Network

- Using Big O notation to compute complexity of calculation
- The complexity in the calculation below is refer to time complexity

L_{1i} is one of neuron in most upper layer

1. Assumptions

1. the complexity of calculating $\frac{dL_{1i}}{dL_{2j}}$ is $O(1)$ for both method
2. the complexity of Matrix Mutiplication of $m \times n$ matrix and $n \times p$ matrix is $O(mnp)$

2. Complexity for computing the gradient to each weight individually

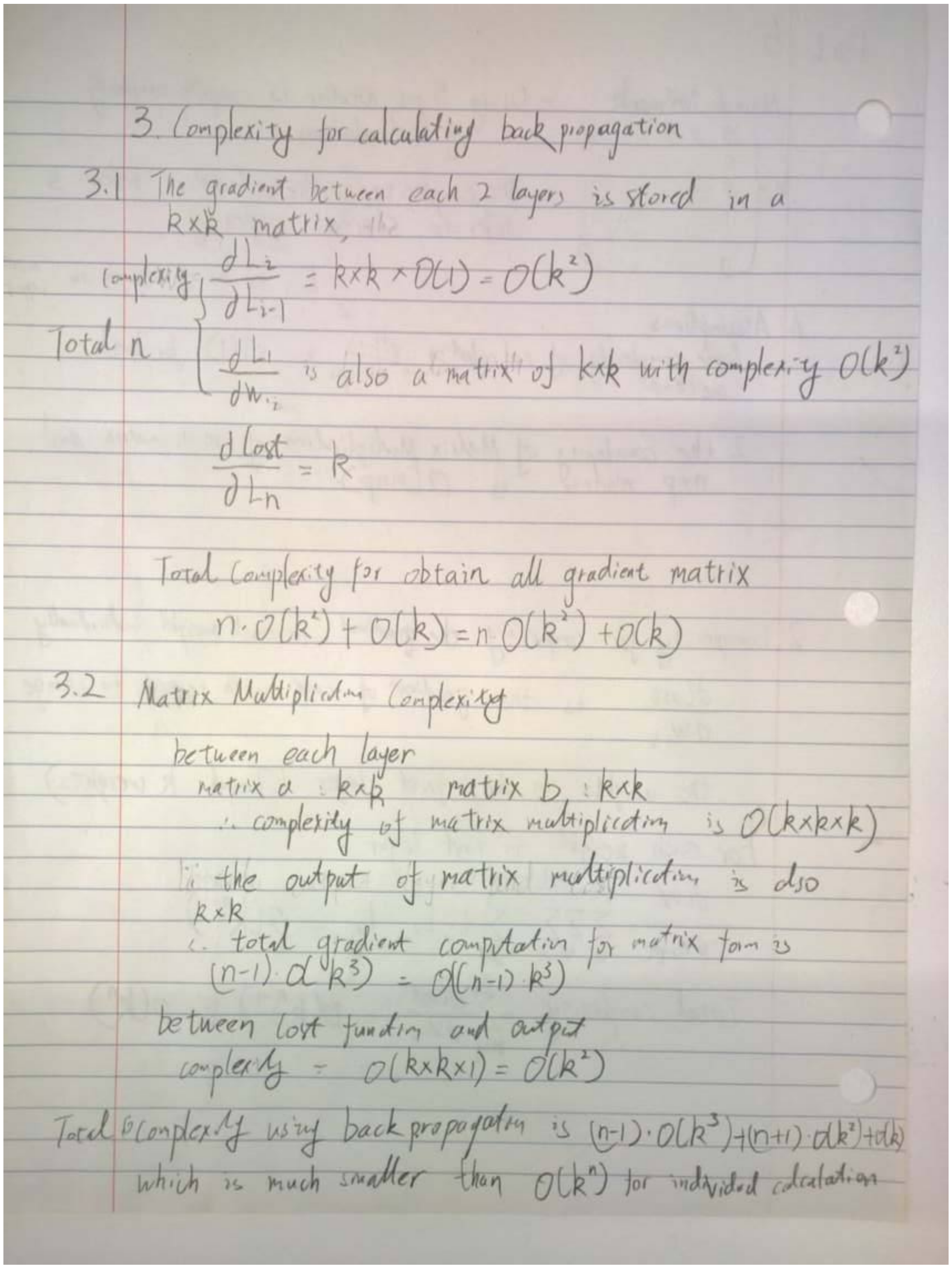
$\frac{d\,Cost}{d\,W_{12}}$ is the gradient of cost with respect to change in the weights in the first layer. (Total k weights)

For each weight in first layor

$$\frac{d\,Cost}{d\,w_{1m}} = \overbrace{\sum_{a=1}\sum_{b=1}\cdots\sum_{j=1}}^{k-1 \text{ summation}} L_{1a} L_{2a} \cdots L_{ka} = O(k^{n-1})$$

$$\text{Total complexity} = \sum_{m=1}^{k} \frac{d\,Cost}{d\,W_{12}} = O(k^{n-1}) \cdot k = O(k^n)$$

Figure 6: First part of part 6



3. Complexity for calculating back propagation

3.1 The gradient between each 2 layers is stored in a $k \times k$ matrix,

complexity $\frac{\partial L_i}{\partial L_{i-1}} = k \times k \times O(1) = O(k^2)$

Total n $\left\{ \frac{\partial L_i}{\partial W_i} \right.$ is also a matrix of $k \times k$ with complexity $O(k^2)$

$\frac{\partial \text{Cost}}{\partial L_n} = k$

Total Complexity for obtain all gradient matrix

$n \cdot O(k^2) + O(k) = n\,O(k^2) + O(k)$

3.2 Matrix Multiplication Complexity

between each layer

matrix a : $k \times k$ matrix b : $k \times k$

$\therefore$ complexity of matrix multiplication is $O(k \times k \times k)$

$\because$ the output of matrix multiplication is also $k \times k$

$\therefore$ total gradient computation for matrix form is $(n-1) \cdot O(k^3) = O((n-1) \cdot k^3)$

between cost function and output

complexity $= O(k \times k \times 1) = O(k^2)$

Total complexity using back propagation is $(n-1) \cdot O(k^3) + (n+1) \cdot O(k^2) + O(k)$ which is much smaller than $O(k^n)$ for individual calculation

Figure 7: Second part of part 6

Part 7

Single Layer Neural Network for face classifications with Tensorflow

Part 7 and below were written with Tensorflow 1.0 and python 2.7.13 Anaconda Distributions

In this part, we utilized Tensorflow to build a fully connected neural network to build a classification algorithm that could the following listed actors, similar to Project 1.

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

We also checked the hash value of the downloaded images to ensure the network is train only with the correct images. In particular, we saved all of the correct hash information for each image to a csv file upon downloading. We then ran through the uncropped folder and calculate all the hash values for all images. As long as we saw an image with mismatched hash values, we removed it from the uncropped folder. After this operation, we were left with 1653 images.

After the images were preprocessed, we converted all images to gray scale and reshaped them to 64×64.

The following parameter have to be decided for the training the neural network:

1. number of hidden units in hidden layer

We built a Neural Network with single hidden layer that has 100 hidden units. We experimented with 20, 50, 100, 1000 and 4096 hidden units, and found out that 100 was best choice. Using less than 100 hidden units would sometimes cause decrease in performances when the dataset contains more abnormal features (since there were not enough units to detect all the features), while using more than 1000 hidden units would reduce the speed of training by a large amount, but with no obvious performance improvements (Since matrix size is really large).

2. number of iteration

For number of iteration of gradient descent, we have experimented 500,1000,2000,5000 iterations. After 1000 iterations, the marginal improvement for addition of more iterations is getting smaller and smaller. However, if the maximum iteration is too small, then the training is too fully completed.

3. Learning rate

For learning rate α, due to large size of training set, we decided to increase its value to make it learn faster. After experiments, the final value is set at 0.01.

The architecture of the neural network is shown below in $Figure$ 8.

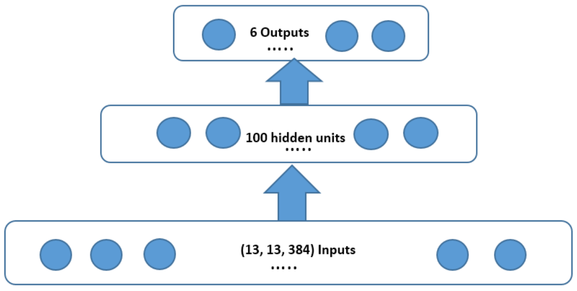


Figure 8: The Neural Network Architecture Used in Part 7

The number of iterations was chosen to be 1000, The accuracy on the training set is found to be 99.85%
The accuracy on the test set is found to be 87.22%
The accuracy on the validation set is found to be 94.99%

The learning curve of the neural network is shown below in $Figure$ 9

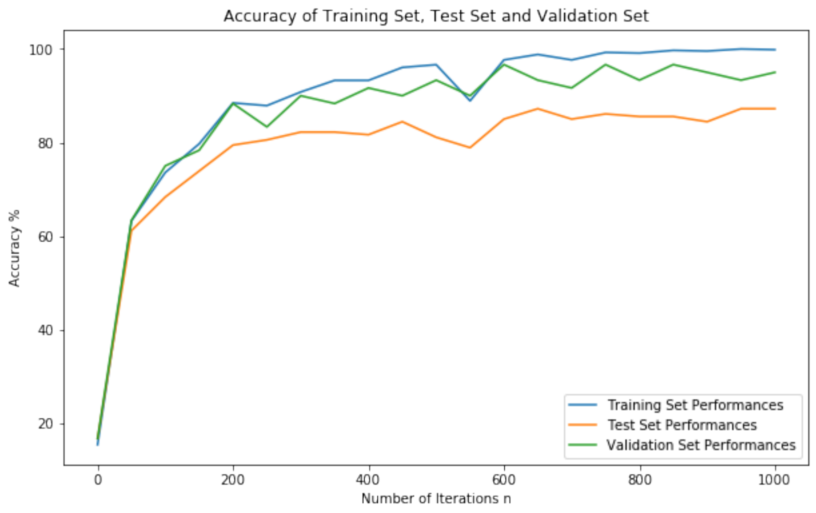


Figure 9: The Learning Curve of the Neural Network in part 7

Part 8

Use of Regularizations
One of reason to use regularization is to prevent overfiting. Given large number of features, the smaller training set size the higher chance to get overfit. Regularization helps on this situation by penalize the large weights. In our training case, we used L2-regularization. This means that we encourage little use of all feature and push all weights toward 0 ultimately. A scenario that will be benefit from regularization is that small training set with large features which is a scenario that very likely to be overfitting.

On the other hand, introducing the regularization will increase the amount of bias added into the model. This type of bias may be lead to unwanted result, so the parameter λ cannot be too large. Also, if the λ is too large, the weights will all extremely close to 0. However, if the λ is too small, the penalty applied will not affect the weights much because large weights will not be penalized enough.

Our strategy is randomly selecting a small set of images (120 images, with no balance of class) and set the image size as 64×64. With 4096 inputs and a small training set size, the model without regularizations will easily overfit and produce a degraded performance

Various learning rate have been experimented, and α is set as 0.0001. The maximum iteration is set at 1500, because the performance after this iteration will not improve anymore.

The parameter that gives the optimized improvement is 0.25
The performance on test set without regularization is 73.3%
The performance on test set with regularization is 80.0%

The performance difference between cost function with or without regularization is approximately 7%. Both method are checked to make sure it is fully trained, so the difference clearly demonstrate regularization improve the performance when the dataset is less than number of features.
The comparison of the learning curves of the two methods are shown below in *Figure* 10

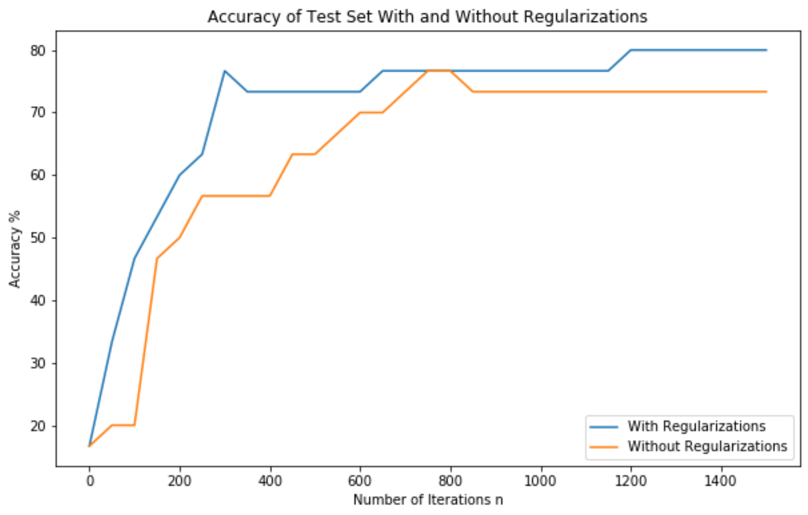


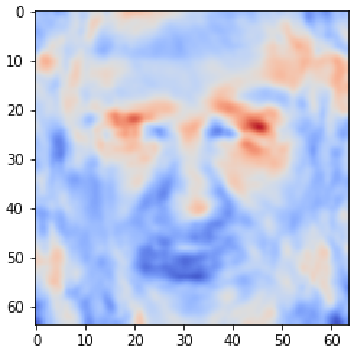
Figure 10: The Comparison of the Learning Curves With and Without Regularizations

Part 9

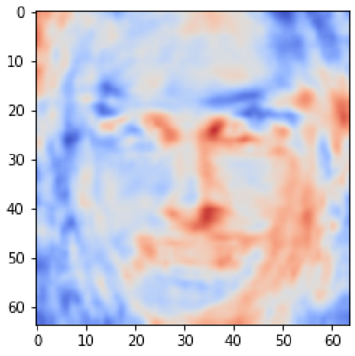
Visualizations of Weights for a Certain Actor

Among all hidden units, some of the hidden units are responsible for classifying a certain actor. It is not obvious to tell by purely looking at the weights. We decide to implement a strategy to find out which units are responsible for classifying America Ferrera and Alec Baldwin.

First, we constructed a true label of either Ferrera as [0, 1, 0, 0, 0, 0] or Baldwin as [1, 0, 0, 0, 0, 0]. Then we used the last W_0 matrix (generated from part 7 after 1000 iterations) as input features. We calculated the cost of all 100 hidden units by calling *session.run* against cost function, with $feed\_dict$ as $x = W_0$ and $y = [0, 1, 0, 0, 0, 0]$ (or [0, 1, 0, 0, 0, 0]) . Each neuron is represented by each row of the inputing W_0 matrix. We found the unit with the lowest cost and visualize that row weights as a 64 × 64 image. The two images are shown below in $Figure$ 11.



(a) Alec Baldwin Visualized Weights



(b) America Ferrera Visualized Weights

Figure 11: Visualizations of Weights that are Responsible for Classifying a Certain Actor

Part 10

Face Classification with AlexNet Weights

In this part, we extracted the pre-trained AlexNet convolution layer 4 weights and use them as input of a new neural network with similar structure as the network in Part 7.

To begin with, we reshaped all of the uncropped images to be $227 \times 227 \times 3$ to fit in the input scheme of the AlexNet. We used *session.run* against the layer called *conv*4, with all of the $227 \times 227 \times 3$ images as *feed_dict* parameters. The dimensionality of the activation layer was found to be (13, 13, 384). We then transformed the (13, 13, 384) to a 1 dimensional array as a (1, 64896). After that we were able to build a neural network with 400 hidden units (since the input is much larger in this case). *Figure* 12 shows an illustration of such system.

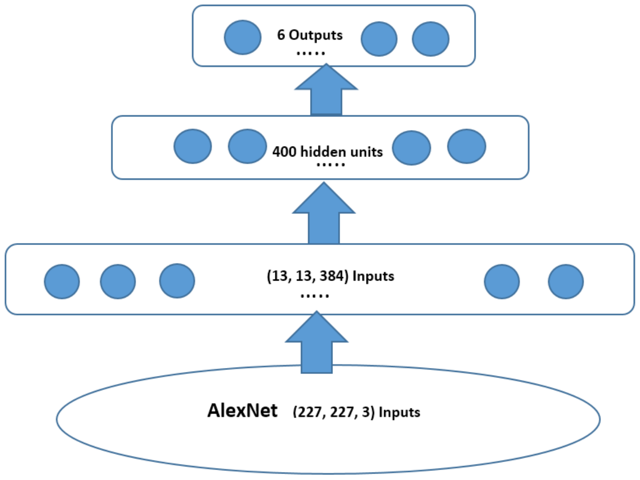


Figure 12: The Neural Network Architecture With AlexNet

The result of the classification is spectacular,
The accuracy on the training set is found to be 98.83%
The accuracy on the test set is found to be 96.11%
The accuracy on the validation set is found to be 100.0%

The learning curve of the system is show below in $Figure$ 13

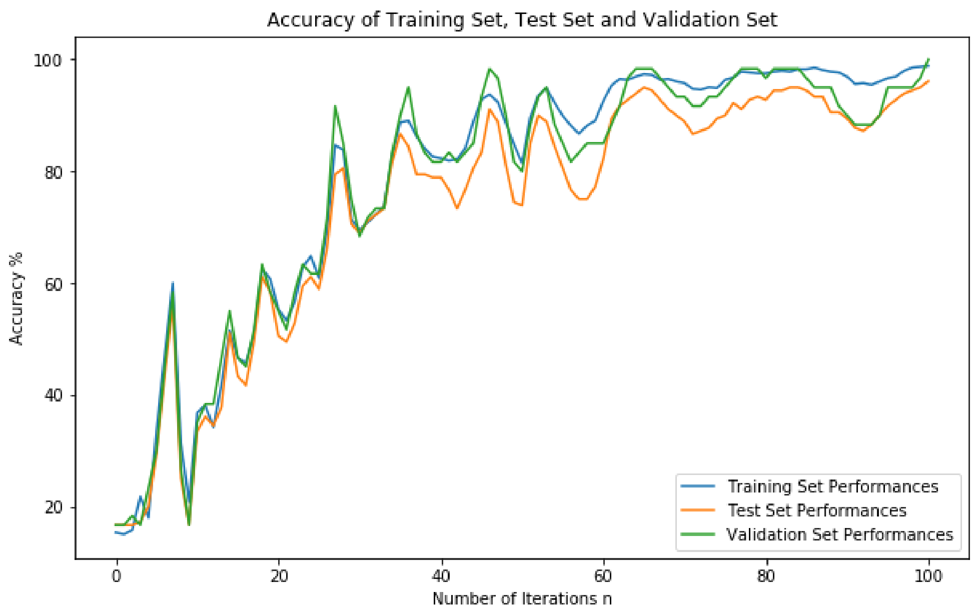


Figure 13: The Learning curve of the Neural Network with AlexNet

Part 11

Bonus - Visualizations of Convolution Layer 4

In this part we attempted to visualize part of the feature maps in the convolution layer 4 of AlexNet. In the convolution layer 4, a certain unit will attempt to detect the presence of a specific feature. We selected 5 different kinds of feature maps, each has 5 examples of the detection. The visualization of these 3 feature maps are shown below in *Figure* 14

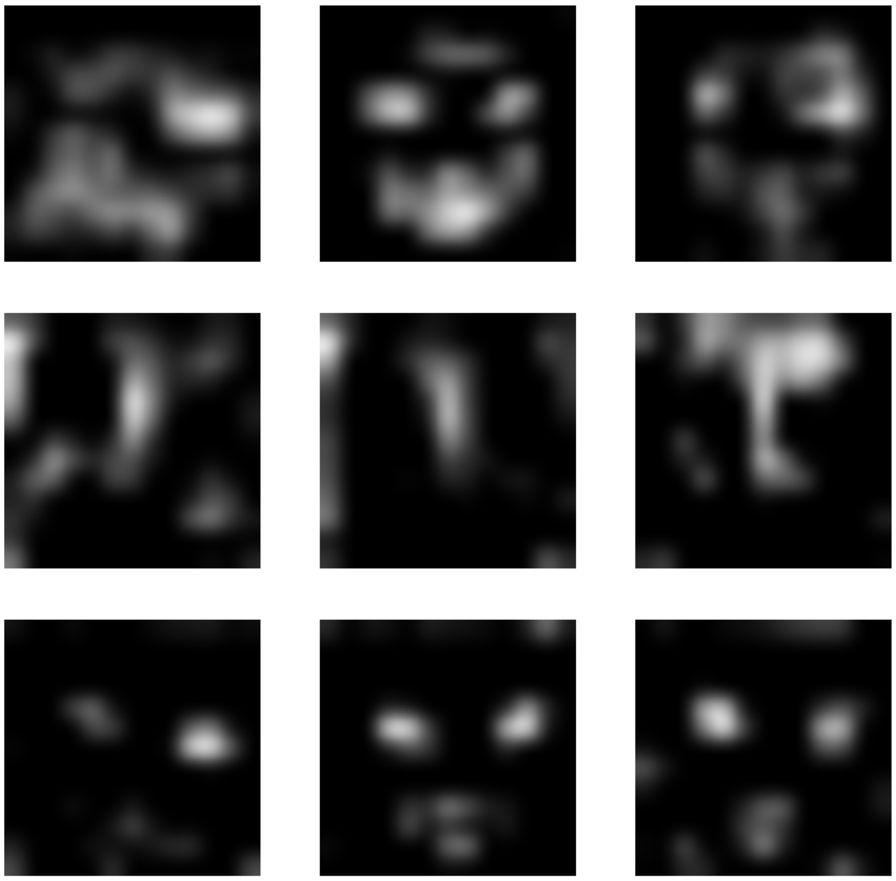


Figure 14: Convolution Layer 4 Feature Maps Visualizations

(1) The feature maps showing in the first row are detecting the smiley facial expressions. The image in the middle in fact looks like a carved pumpkin lamp. The image on the left and right represents the face image is taken when the actor is smiling, but facing left.
(2) The feature maps showing in the second row is responsible for detecting nose-like features. The image

on the right of second row shows a relatively clear image of a nose.
(3) The feature maps showing in the third row is responsible for detecting the layouts and orientations of the actor's face. The image on the left of the third row shows the face is facing left and the next two are facing the right.