CSC411 Project2

Yifei Ai

Part 1

The dataset is a dictionary consisting of 9 training sets which are just number 0 to 9 and 9 test sets. Each training set is a matrix with $M \times 784$ which M represents the number of handwritten digit images in the dataset and each row is an array of size 1×784 is the flatten array of handwritten digit image, which originally is 28×28.

I randomly pick 10 images for each number, they all looks pretty good. At least I believe they are classified to the correct number.

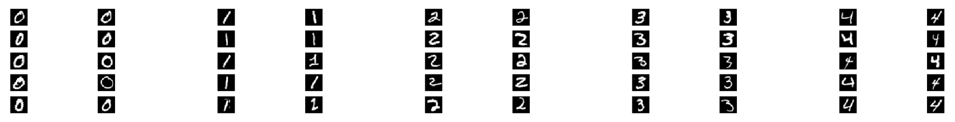


Figure 1: Digits Image 0 to 4

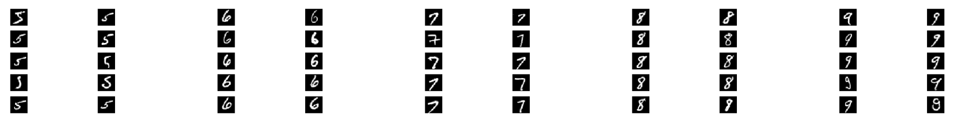


Figure 2: Digits Image 0 to 5

Part 2

Notice to make it convinient, I simply assume we already append the bias vector to Weight matrix

```
    def layer_computation(x, W, b, act_func=lambda x: x):
        L = np.dot(W.T, x) + b
        output = softmax(L)
        return L, output

    def batch_layer_computation(X, W):
        """ X is an input matrix with size NxM, N is the number of input
    units, M is
        the number of training cases. W is a weight matrix already append
    bias vector
        with size (N+1)xL, L is the number of output layers. Notice we need
    extra
        one row for bias, so we should append another row vector [1, 1, 1,
    1, ... ,1]
        to X to do implement the function. """
        one_array = np.ones(X.shape[1])
        X = np.vstack((X, one_array))
        L1 = np.dot(W.T, X)
        Y = softmax(L1)
        return Y
```

Part 3

Suppose X is a matrix with size $N \times M$, where N is the number of input layer, M is the number of training cases. Let W be a matrix be $N \times L$ where L is the number of units in hidden layer. Here L is 10. The output matrix

Y has size of $L \times M$

$$y_j^{(i)} = \frac{e^{o_j^{(i)}}}{\sum_k e^{o_k^{(i)}}} \tag{1}$$

$$o_k^{(i)} = \sum_{p=1}^{N} w_{pk} x_p^{(i)} + b_k \tag{2}$$

$$\mathcal{L}_{CE} = \sum_{i=1}^{M} \sum_{j=0}^{L-1} -t_j^{(i)} log y_j^{(I)} \tag{3}$$

$$= \sum_{i=1}^{M} \sum_{j=0}^{L-1} -t_j^{(i)} [o_j^{(i)} - log(\sum_{k=0}^{L-1} e^{o_k^{(i)}})] \tag{4}$$

$$= \sum_{i=1}^{M} \sum_{j=0}^{L-1} -t_j^{(i)} [\sum_{p=1}^{N} w_{pj} x_p^{(i)} + b_j - log \sum_k (e^{\sum_{p=1}^{N} w_{pk} x_p^{(i)} + b_k})] \tag{5}$$

$$\frac{\partial \mathcal{L}_{CE}}{\partial w_{sm}} = \sum_{i=1}^{M} [(-t_m^{(i)} + \frac{e^{\sum_{s=1}^{N} w_{sm} x_s^{(i)} + b_m}}{\sum_k (e^{\sum_{p=1}^{N} w_{pk} x_p^{(i)} + b_k})} \sum_{j=0}^{L-1} t_j^{(i)}) \ x_s^{(i)}] \tag{6}$$

$$\text{Notice} \sum_{j=1}^{L-1} t_j = 1 \tag{7}$$

$$= \sum_{i=1}^{M} [(-t_m^{(i)} + \frac{e^{o_m^{(i)}}}{\sum_k e^{o_k^{(i)}}}) x_s^{(i)}] \tag{8}$$

$$= \sum_{i=1}^{M} [(-t_m^{(i)} + y_m^{(i)}) x_s^{(i)}] \tag{9}$$

Therefore, we get the derivative of weight is:

$$\frac{\partial \mathcal{L}_{CE}}{\partial W} = X(Y - T)^T$$

To approximate the gradient, I use the following code:

```
    def approx_CE_dWeight_single(X, W, T, p, q, t):
        """Approximate the derivative of single entry in Jacobian matrix"""
        new_weight = W.copy()
        new_weight[p][q] += t
        Y = batch_layer_computation(X, W)
        Y_h = batch_layer_computation(X, new_weight)
        df = part3_cross_entropy(Y_h, T) - part3_cross_entropy(Y, T)
        df = np.true_divide(df, t)
        return df

    def approx_CE_dWeight(X, W, T, t=1e-4):
        size = W.shape
        df_matrix = np.empty(size, dtype=np.float64)
        for i in range(size[0]):
            for j in range(size[1]):
                df_matrix[i][j] = approx_CE_dWeight_single(X, W, T, i, j, t)
        return df_matrix
```

I wrote some test case to test the function. Notice here W already append the bias vectors.

```
In [130]: X
Out[130]:
array([[1, 5],
       [2, 4]])

In [131]: W
Out[131]:
array([[3., 4., 7.],
       [2., 5., 9.],
       [3., 2., 6.]])

In [143]: Y = ast2.batch_layer_computation(X, W)

In [144]: Y
Out[144]:
array([[7.58255810e-10, 7.09547416e-23],
       [3.05902227e-07, 6.30511676e-16],
       [9.99999693e-01, 1.00000000e+00]])

In [133]: T
Out[133]:
array([[1, 0],
       [0, 1],
       [0, 0]])

In [134]: ast2.CE_dWeight(X, Y, T)
Out[134]:
array([[-1.        , -4.99999969,  5.99999969],
       [-2.        , -3.99999939,  5.99999939],
       [-1.        , -0.99999969,  1.99999969]])


In [135]: ast2.approx_CE_dWeight(ast2.batch_layer_computation, X, W, T)
Out[135]:
array([[-1.        , -4.99999969,  5.99999969],
       [-2.        , -3.99999939,  5.99999939],
       [-1.        , -0.99999969,  1.99999969]])

In [184]: X
Out[184]:
array([[2, 8],
       [1, 9]])

In [185]: W
Out[185]:
array([[1., 7., 2.],
       [0., 7., 5.],
       [6., 8., 4.]])

In [186]: Y = ast2.batch_layer_computation(X, W)

In [187]: Y
Out[187]:
array([[7.58255957e-10, 8.40859712e-50],
       [9.99999887e-01, 1.00000000e+00],
       [1.12535162e-07, 1.18506486e-27]])

In [188]: ast2.CE_dWeight(X, Y, T)
Out[188]:
array([[-2.00000000e+00,  1.99999977e+00,  2.25070324e-07],
       [-9.99999999e-01,  9.99999887e-01,  1.12535162e-07],
       [-9.99999999e-01,  9.99999887e-01,  1.12535162e-07]])

In [189]: ast2.approx_CE_dWeight(X, W, T)
Out[189]:
array([[-2.00000000e+00,  1.99999977e+00,  2.25099939e-07],
       [-9.99999999e-01,  9.99999887e-01,  1.12549969e-07],
       [-9.99999999e-01,  9.99999887e-01,  1.12549969e-07]])
```

Part 4

I just randomly initialize each weight between -0.4 to 0.4. For bias, I just let them be zeros. Then I try different learning rate, seems like learning rate with 10^{-4} to 10^{-5} have best performance. Then I try to display the image of the weights. I find image of weights for learning rate 10^{-5} have many noise there, so it might be overfitting, even though performance for 10^{-5} is 2% higher than performance for 10^{-4}. So I just choose the value between them 5×10^{-5}. We can see that this learning rate works better than 1e-4. The following is my graph.

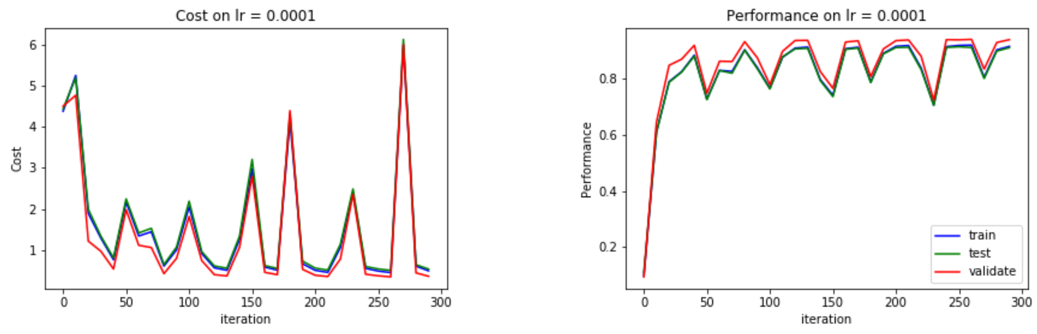


Figure 3: Learning rate $1e-4$

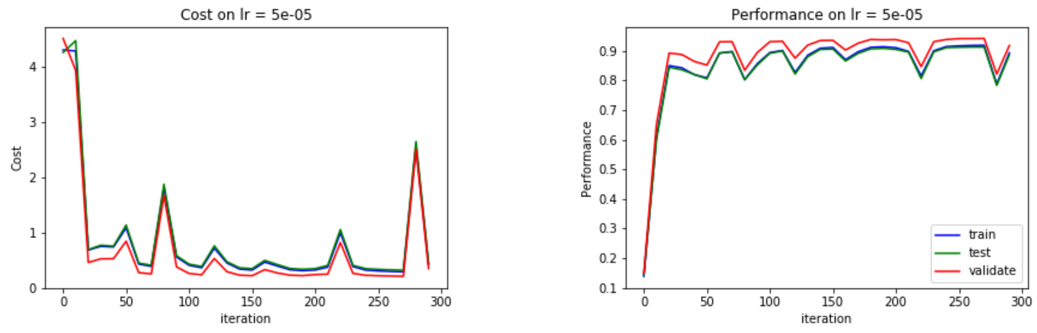


Figure 4: Learning rate $5e-5$

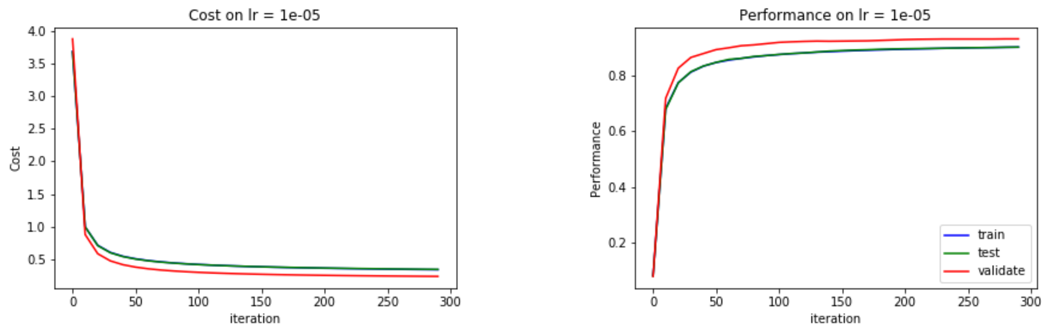


Figure 5: Learning rate $1e-5$

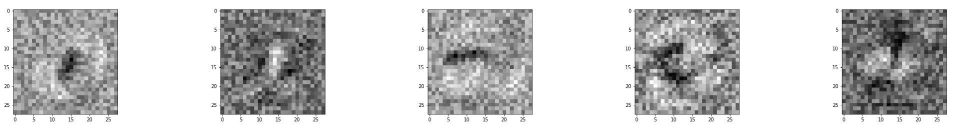


Figure 6: Digits Image 0 to 4

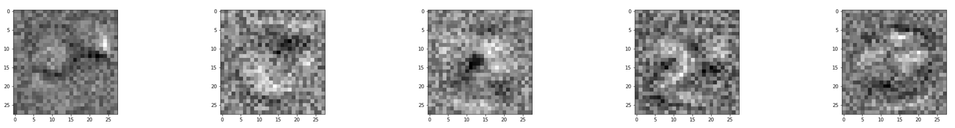


Figure 7: Digits Image 5 to 10

Part 5

I just simply follow the algorithm, momentum is at line 29

```
1  def gradient_descent(X, T, alpha=0.00001, EPS = 1e-5, max_iter = 300, mu =
       0):
2      """ X is the matrix of inputs, NxM, N is the number of input units, M is
        the
3      number of training cases. T is the matrix of results, size: LxM, L is
       the
4      number of output units """
5      # Initialize a weight matrix
6      size = [X.shape[0], T.shape[0]]
7      print "Weight size is:" + str(size)
8      W = initialize_weights(size)
9      b = initialize_bias(T.shape[0])
10     previous_W = W - 10*EPS
11     previous_b = b - 10*EPS
12     count = 0
13     perform_dict = dict()
14
15     summary = np.empty((6, 0), dtype=float)
16
17     X_test, T_test, X_valid, T_valid = seperate_train_valid()
18
19     p = 0
20
21     while norm(W - previous_W)+norm(b - previous_b) > EPS and count <
       max_iter:
22         previous_W = W.copy()
23         b = b.copy()
24         Y = batch_layer_computation(X, W, b)
25         Y_test = batch_layer_computation(X_test, W, b)
26         Y_valid = batch_layer_computation(X_valid, W, b)
27         data = calculate_data(Y, T, Y_test, T_test, Y_valid, T_valid)
```

```
        summary = np.hstack((summary, data[:,None]))
        p = mu*p + alpha*CE_dWeight(X, Y, T)
        W = W - p
        b = b - alpha*CE_dBias(Y, T)
        print "Iter: " , count
        print "Weight: " , W
        print "Cost: " , part3_cross_entropy(Y, T)
        if count % (max_iter/5) == 0:
            print "Accuracy on test set: ", calculate_accuracy(Y, T)
            perform_dict[count] = (calculate_accuracy(Y, T), previous_W,
    previous_b)
        count += 1
    if mu == 0:
        np.save('tmp/part4_summary_data'+str(alpha), summary)
    else:
        np.save('tmp/part4_summary_data_momentum'+str(alpha), summary)
    return W, b, perform_dict
```

We can see that gradient descent with momentum converge much faster than the one without using momentum.

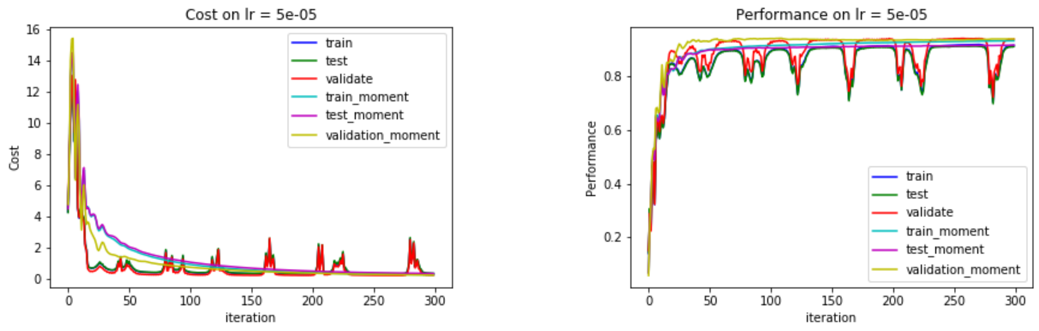


Figure 8: Learning rate 5×10^{-5}

Part 6

(a)

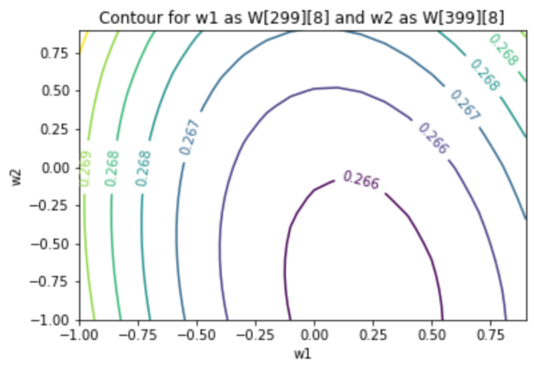


Figure 9: Contour set with w1 and w2 around optimum

(b), (c)

I tried several examples, these two demonstrate the feature of momentum

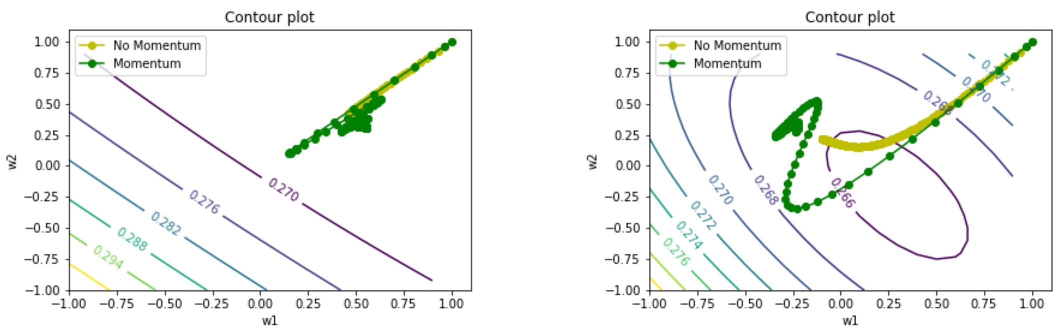


Figure 10: trajectory for gradient descent with moment and without momentum

(d)

The trajectory with momentum goes to the optimum faster, but would get around the optimum in a very small range compare with the trajectory without momentum. This is because we have momentum, that somehow remembers the direction of last update.

(e)

We should choose the pixel in the center of the image. Our image is 28*28, so if we choose some variable on the edge then either the gradient descent on those 2 variables stop on first several steps or have a very slow trajectory. I think this is becuase it is on the edge, so since other entries in W is already close to optimum, so the variable on the edge doesn't really affect the result, in other word, it probably because the input is 0 corresponds to that entry is 0, so it doesn't really affect too much.

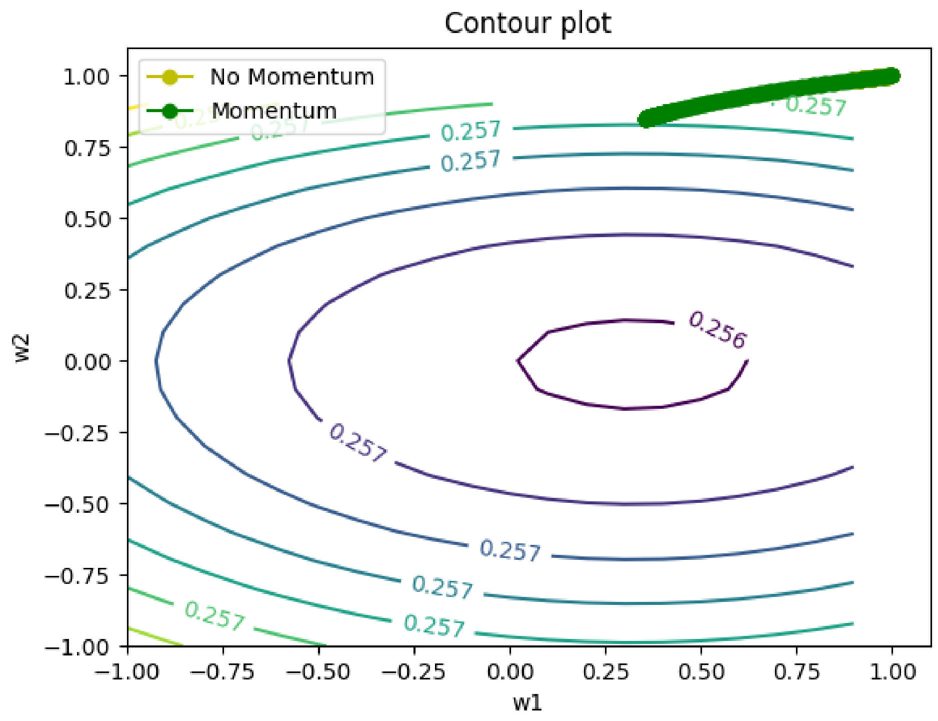


Figure 11: trajectory that doens't demonstrate the benefit of momentum

This is the w1, w2 from W[100][8] and W[200][8], if we check the train set $M['train8'][i][100]$ and $['train8'][i][200]$, most of them are 0. We can see that 2 trajectories are almost the same.

To get a nice example, I simply look for some entry in M['train8'][i] that is not 0, since 8 relatively has more entries be gray compare with other number in my opinion. Then I do the gradient descent to those 2 variables on the W.

Part 7

Suppose we use back propagation, then, denote $T(N)$ as the complexity with N layers.

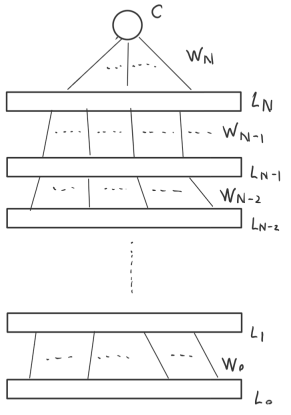


Figure 12: Network with N+1 Layer

With Back Propagation

Consider we have $N+1$ layers, now, if we need to calculate the gradients for all weights, then the first N layer we have calculate actually costs $T(N)$ by our assumption, then for the last layer, we need to know $\frac{\partial C}{\partial W_0}$. Since we use back propagation, we have $\frac{\partial C}{\partial L_2}$, thus we just need to calculate $\frac{\partial L_2}{\partial L_1}$ and $\frac{\partial L_1}{\partial W_0}$. So we will have

$$T(N+1) = T(N) + 2$$

Which means complexity with back propagation is $O(N)$.

Without Back Propagation

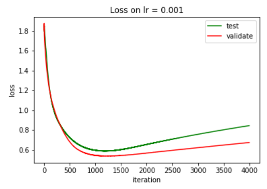
In this case, since we don't store any information, to calculate $\frac{\partial C}{\partial W_0}$, we need to do $\frac{\partial C}{\partial L_N}\frac{\partial L_N}{\partial L_{N-1}}\cdots\frac{\partial L_2}{\partial L_1}\frac{\partial L_1}{\partial W_0}$. So we will have

$$T(N+1) = T(N) + (N+1)$$

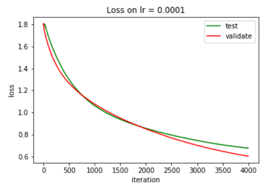
This means the complexity without back propagation is $O(N^2)$

Part 8

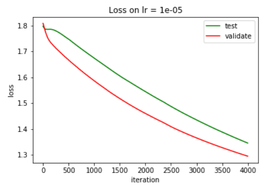
I use a network with 32*32 or 64*64 input units, 6 output units, and one hidden layer. In that hidden layer, there are 12 hidden units, the activate function is ReLU. I tried both 32x32 and 64x64 to train the network. For 32x32 input, I set 64 hidden units, for 64x64 input, I set 32 hidden units. And I decide to use Adam optimizer. Since here the data is not very big, so I just set number of mini-batches as small as possible. Here are the final performance for each learning rate. The final performance on test set is around 75% to 85% if the learning rate is not too bad.



(a) rate 1e-3

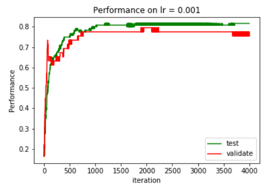


(b) rate 1e-4

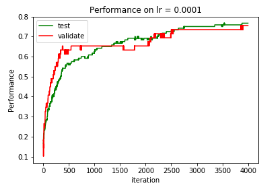


(c) rate 1e-5

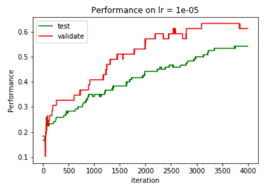
Figure 13: Resolution 32x32, learning curve with rate 0.01, 0.001, 0.0001



(a) rate 1e-3

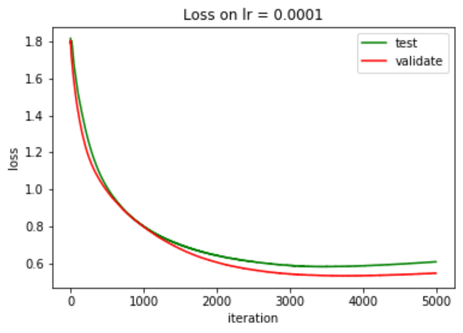


(b) rate 1e-4

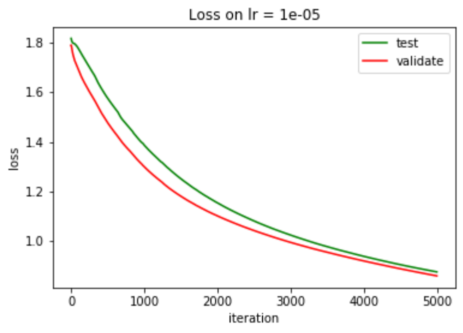


(c) rate 1e-5

Figure 14: Resolution 32x32, learning curve with rate 0.01, 0.001, 0.0001

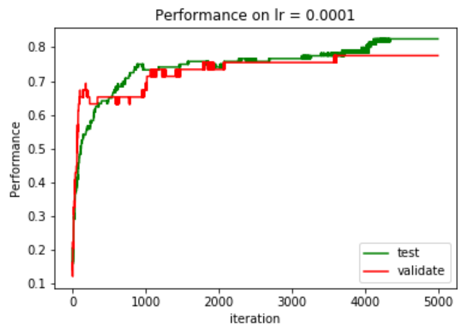


(a) rate 1e-4

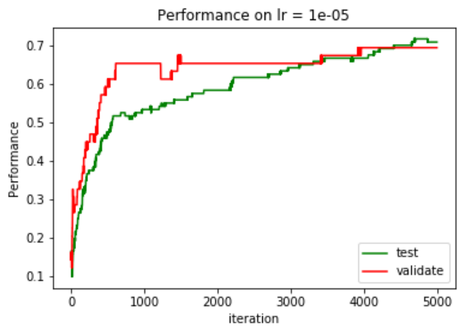


(b) rate 1e-5

Figure 15: Resolution 64x64, learning curve with rate $10^{-4}, 10^{-5}$



(a) rate 1e-4



(b) rate 1e-5

Figure 16: Resolution 64x64, learning curve with rate $10^{-4}, 10^{-5}$

Part 9

In this section, I actually change the number of mini-batches to 1. I also use the weight from the network for 64x64 size images cuz that somehow give more information. Then the strategy to find the most useful weights is the following.

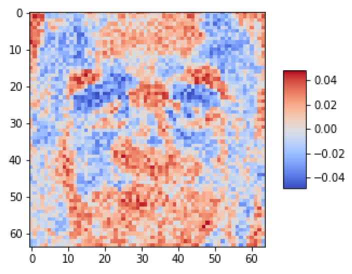
1. Choose 2 actors, and look at the output nodes with respect to that actors
2. For each actor, inspect the weights connected from hidden layer to that node

3. Find the weight with hightest value, get the index of that weight, say it is k

4. Inspect the k^{th} node in the hidden layer, get the weight matrix from input layer to that node

5. Convert that matrix to the image

Here is my result, I choose gilpin and carell


(a) gilpin

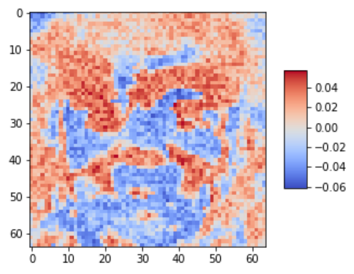

(b) carell

Figure 17: Image of Weights

Part 10

I construct a network whose input size matches the output from AlexNet.feature which is 256*6*6, I use 12 hidden units in the hidden layer, and the activate function is ReLU. I use Adam optimizer, with learning rate 1e-5, get 91.6% performance on test set.