CSC411: Assignment 2

Due on Sunday, February 18, 2018

Yian Wu, Zhou Quan

February 24, 2018

Part 1

Describe the dataset of digits. In your report, include 10 images of each of the digits. You may find matplotlib?s subplot useful for displaying multiple images at once.

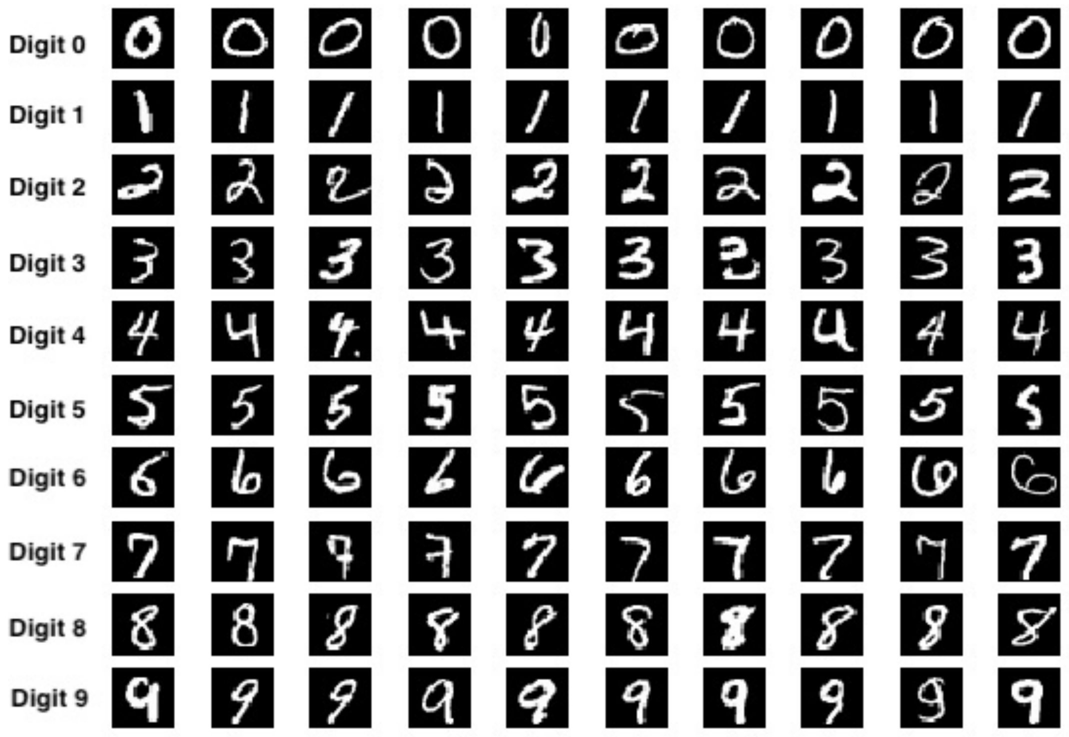


Figure 1: Part 1

We have included 10 images of each of the digits from MNIST dataset in Figure 1.
For each digit, we randomly selected 10 samples by using python standard library random to make sure the selected ones are unbiased random samples.
In samples for each digits, we can see there are many different ones.
There are ones looks faded in some parts or twisted in different directions. There are ones that are written in different style such as 7.

Part 2

Implement a function that computes the network below using NumPy. Include the listing of your implementation in your report for this Part

Listing 1: part2

```
def softmax(y):
    '''Return the output of the softmax function for the matrix of output y. y
    is an NxM matrix where N is the number of outputs for a single case, and M
    is the number of cases'''
    return exp(y)/tile(sum(exp(y), 0), (len(y), 1))

def part2(x):
    # I recommend you divide the data by 255.0
    x /= 255.0
    # initialize Weight and bias matrix to all zero
    W = np.zeros((784, 10))
    b = np.zeros((10, 1))
    return softmax(np.dot(W.T, x) + b)
```

Part 3

We would like to use the sum of the negative log-probabilities of all the training cases as the cost function.

Part 3(a)

Compute the gradient of the cost function with respect to the weight w_{ij}. Justify every step. You need to justify every step there, and that your cost function is the sum over all the training examples.

We want to compute the gradient of the cost function with respect to the weight w_{ij} which is

$$\frac{\partial C}{\partial W_{ij}} = \frac{\partial C}{\partial o_j}\frac{\partial o_j}{\partial w_i j} = \sum_k \frac{\partial C}{\partial o_j^{(k)}}\frac{\partial o_j^{(k)}}{\partial w_{ij}}$$

i: index of input dimension(1-784)

j: index of output dimension(1-10)

k: k^{th} training set

$$\frac{\partial C}{\partial o_j} = \sum_i \frac{\partial C}{\partial p_i}\frac{\partial p_i}{\partial o_j}$$

$$\frac{\partial C}{\partial p_i} = \frac{\partial(-\sum_i y_i log(p_i))}{\partial p_i} = -\frac{y_i}{p_i}$$

$$\frac{\partial p_i}{\partial o_j} = \frac{\partial p_j}{\partial o_j} = p_j(1-p_j) \qquad \text{\# while i = j, refers to Slide 7}$$

$$\frac{\partial p_i}{\partial o_j} = \frac{\partial(\frac{exp(o_i)}{\sum_m o_m})}{\partial o_j} = -p_i p_j \qquad \text{\# while i} \neq \text{j, refers to Slide 7}$$

$$\frac{\partial C}{\partial o_j} = \sum_i \frac{\partial C}{\partial p_i}\frac{\partial p_i}{\partial o_j} = \sum_{i,i\neq j}(-\frac{y_i}{p_i}\cdot(-p_ip_j)) - (i=j)\frac{y_i}{p_j}\cdot p_j(1-p_j)$$

$$\therefore \frac{\partial C}{\partial o_j} = p_j\sum_i y_i - y_j = p_j - y_i$$

$$\frac{\partial o_j}{\partial w_{ij}} = \frac{\partial(\sum_i w_{ij}x_i + b_j)}{\partial w_i j} = x_i$$

$$\therefore \frac{\partial C}{\partial W_{ij}} = \sum_k \frac{\partial C}{\partial o_j^{(k)}}\cdot\frac{\partial o_j^{(k)}}{\partial w_{ij}} = \sum_k (p_j^{(k)} - y_j^{(k)})x_i^{(k)}$$

Part 3(b)

Write vectorized code that computes the gradient of the cost function with respect to the weights and biases of the network. Check that the gradient was computed correctly by approximating the gradient at several coordinates using finite differences. Include the code for computing the gradient in vectorized form in your report.

Listing 2: part3b

```python
def f(x, y):
    p = part2(x)
    return -sum(y * log(p))

# computes the gradient of the cost function with respect to the weights and biases of the network
def df(x, y):
    p = part2(x)
    partial_Co = p - y  # (10, 1)
    x = x[:, np.newaxis]

    # x -> (784, 1) partial_Co -> (1, 10)
    partial_CW = dot(x, partial_Co.reshape((1, 10)))
    partial_Cb = dot(partial_Co, ones((partial_Co.shape[1], 1)))

    return partial_CW, partial_Cb

def cost_part3(x, y, W, b):
    x = x[:, np.newaxis]
    p = softmax(np.dot(W.T, x) + b)
    cost_val = -sum(y * log(p))
    return cost_val

def part3():
    x = M["test0"][10].T/255.  # x -> (784, )
    y = np.zeros((10, 1))  # true value of output
    y[0] = 1

    #np.random.seed(1)
    par_CW, par_Cb = df(x, y)

    for i in range(5):
        m = np.random.randint(0, 784)
        n = np.random.randint(0, 10)
        W = np.zeros((784, 10))
        b = np.zeros((10, 1))
        h = 0.00000001


        # respect to Weights
        weight_h = np.zeros((784, 10))
        weight_h[m][n] = h
        fd_weight = (cost_part3(x, y, W + weight_h, b) - cost_part3(x, y, W, b)) / h

        # respect to Bias
        bias_h = np.zeros((10, 1))
        bias_h[n][0] = h
        fd_bias = (cost_part3(x, y, W, b + bias_h) - cost_part3(x, y, W, b)) / h

        print "================================================================="
        print "at coordinates:(" + str(m) + ", " + str(n) + " )"
        print "finite difference with respect to weight: " + str(fd_weight)
        print "gradient of cost function with respect to the weights: " + str(par_CW[m][n])
        print "finite difference with respect to bias: " + str(fd_bias)
        print "gradient of cost function with respect to the bias: " + str(par_Cb[n][0])
```

===
======================= Computed Output ==========================

At coordinates:(206, 6)
finite difference with respect to **weight**: 0.0976541514319
gradient of cost function with respect to the weights: 0.0976541491177
finite difference with respect to **bias**: 0.100007246928
gradient of cost function with respect to the bias: 0.100007261145

At coordinates:(118, 0)
finite difference with respect to **weight**: 0.0
gradient of cost function with respect to the weights: -0.0
finite difference with respect to **bias**: -0.899857566239
gradient of cost function with respect to the bias: -0.899857563027

At coordinates:(73, 8)
finite difference with respect to **weight**: 0.0
gradient of cost function with respect to the weights: 0.0
finite difference with respect to **bias**: 0.0999656357692
gradient of cost function with respect to the bias: 0.0999656218074

At coordinates:(740, 7)
finite difference with respect to **weight**: 0.0
gradient of cost function with respect to the weights: 0.0
finite difference with respect to **bias**: 0.100030872474
gradient of cost function with respect to the bias: 0.10003091569

At coordinates:(411, 0)
finite difference with respect to **weight**: -0.861040172495
gradient of cost function with respect to the weights: -0.861040177955
finite difference with respect to **bias**: -0.899857566239
gradient of cost function with respect to the bias: -0.899857563027

Part 4

Train the neural network you constructed using gradient descent (without momentum). Plot the learning curves. Display the weights going into each of the output units. Describe the details of your optimization procedure ? specifically, state how you initialized the weights and what learning rate you used.

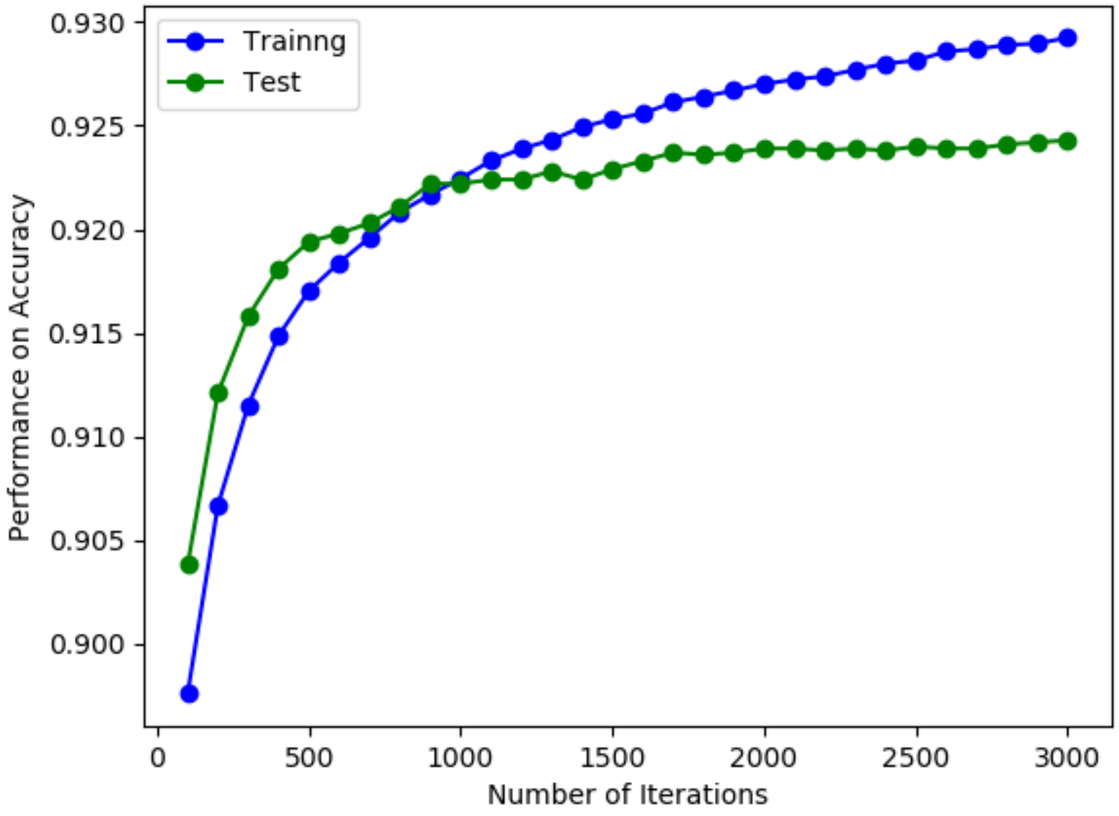


Figure 2: The Learning Curves

This is how we initialize our weights. We use np.random.randn to random generate a 784x10 matrix which the samples from a mean zero. Then we times the matrix with 0.01 to make all the number smaller. Since the activiation function that we are using is tanh function. As the absolute value of the input increases, the output of the tanh function will become more smooth, and gradient will be close to zero. Then the tanh function will be saturate and make the neural network hard to learn. Thus, we should choose the intputs which are having samll absolute values, in other word, the input should be close to zeros. In order to do so, we should initialize the weights to be numbers close to zero. However, we didn't set the weight to be all zero to avoid the same update of the weights.
For the learning rate we used in our code was 10e-5.

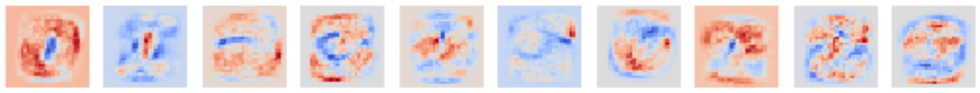


Figure 3: The Weights going into each of the output units

Part 5

In Part 4, we used (vanilla) gradient descent, without momentum. Write vectorized code that performs gradient descent with momentum, and use it to train your network. Plot the learning curves. Describe how your new learning curves compare with gradient descent without momentum. In your report, include the new code that you wrote in order to use momentum.

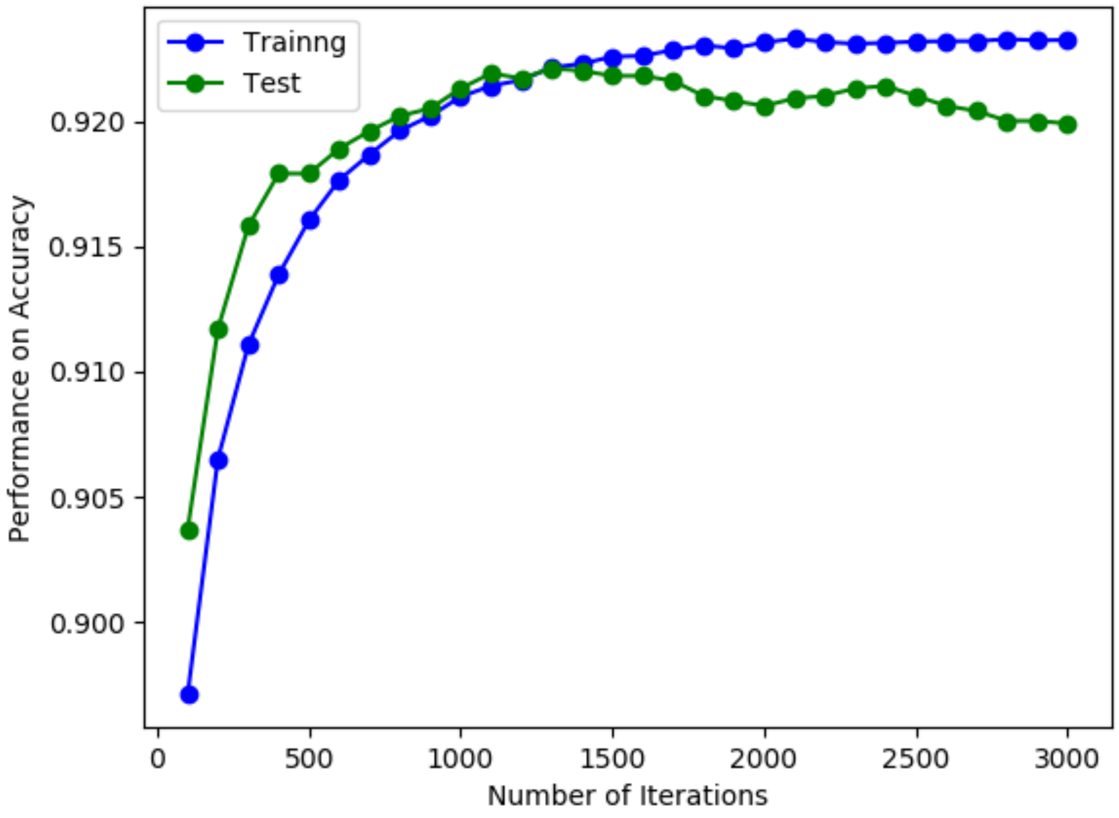


Figure 4: The Learning Curves

From Figure 4 we can see that, it reachs a higher accuracy at earlier iterations than in Figure 2(gradient descent without momentum).
This is because we use Momentum in this part.
Momentum helps accelerates. For gradient descent without momentum, the updating is quite slow, and is hesitant when closing to the lowest point.
W_cp -= alpha * partial_CW + momentum * W2
A value (momentum) is added when doing updates. Since the adjacent value of two updates are towards different direction, the momentum will let the gradient point in same direction increases and gradient point to opposite direction decreases.
In addition, when gradient approaches zero after many iterations, the momentum allows the updates to be greater, jump out of the ”trap” and find the lowest point. Thus, push the value of gradient faster to reach the lowest point.

Listing 3: part3b

```
def grad_descent_m(x, y, W, b, alpha, itr, momentum):
    EPS = 1e-5
    prev_w = W - 10 * EPS
    W_cp = W.copy()
    b_cp = b.copy()
    i = 0
    results = []
    while norm(W_cp - prev_w) > EPS and i < itr:
        i += 1
        prev_w = W_cp.copy()
        partial_CW, partial_Cb = df(x, y, W_cp, b_cp)

        W2, b2 = W.copy(), b.copy()
        W_cp -= alpha * partial_CW + momentum * W2
        b_cp -= alpha * partial_Cb + momentum * b2

        if i % 100 == 0:
            print "Iteration: ", i
            curr_W, curr_b = W_cp.copy(), b_cp.copy()
            print "=======In while======="
            print curr_W, curr_b
            results.append([i, curr_W, curr_b])

    return results, W_cp, b_cp


def part5():

    W = np.random.randn(784, 10) * 10e-5
    b = np.zeros((10, 1))

    training = np.empty((784, 0))
    tests = np.empty((784, 0))
    y_training = np.empty((10, 0))
    y_test = np.empty((10, 0))

    for i in range(0, 10):
        training = np.hstack((training, M["train"+str(i)].T/255.0))
        tests = np.hstack((tests, M["test"+str(i)].T/255.0))
        training_size = len(M["train"+str(i)])
        test_size = len(M["test"+str(i)])
        # one hot vector
        o_vector = np.zeros((10, 1))
        o_vector[i] = 1
        y_training = np.hstack((y_training, tile(o_vector, (1, training_size))))
        y_test = np.hstack((y_test, tile(o_vector, (1, test_size))))

    # calculating performance
    alpha = 0.00001
    momentum = 0.99
    plot_data, end_W, end_b = grad_descent_m(training, y_training, W, b, alpha, 3000, momentum)

    print "===========plot data============"
    print plot_data

    train_accuracy, test_accuracy, itr_idx = [], [], []
    for i in plot_data:
        itr_idx.append(i[0])
        curr_w, curr_b = i[1], i[2]
        training_size, test_size = training.shape[1], tests.shape[1]

        train_correct, test_correct = 0, 0
        train_x = part2(training, curr_w, curr_b)
        test_x = part2(tests, curr_w, curr_b)

        for j in range(training_size):
            if y_training[:, j].argmax() == train_x[:, j].argmax():
                train_correct += 1
        test_correct = 0
        for k in range(test_size):
            if y_test[:, k].argmax() == test_x[:, k].argmax():
                test_correct += 1
        train_accuracy.append(float(train_correct)/float(training_size))
        test_accuracy.append(float(test_correct)/float(test_size))


    plt.figure()
    plt.plot(itr_idx, train_accuracy, color='blue', marker='o', label="Trainng")
    plt.plot(itr_idx, test_accuracy, color='green', marker='o', label="Test")
    plt.legend(loc="best")
    plt.xlabel("Number of Iterations")
    plt.ylabel("Performance on Accuracy")
    savefig('part5_learning_curves')
```

Part 6

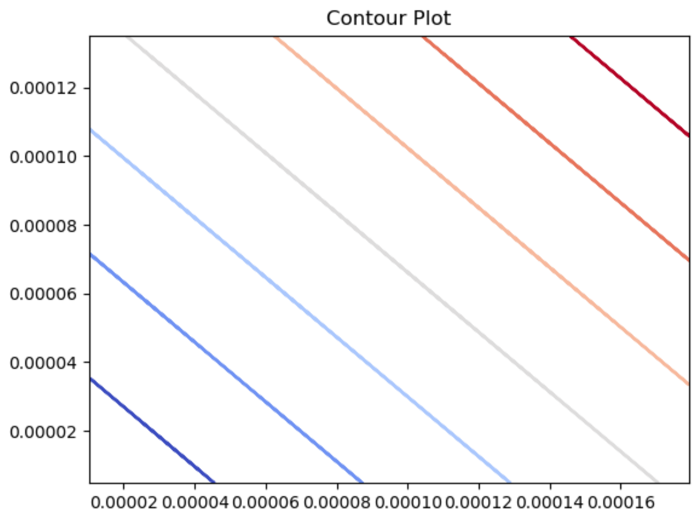


Figure 5: 6(a) w1, w2 take values by random uniform between 0 and 2

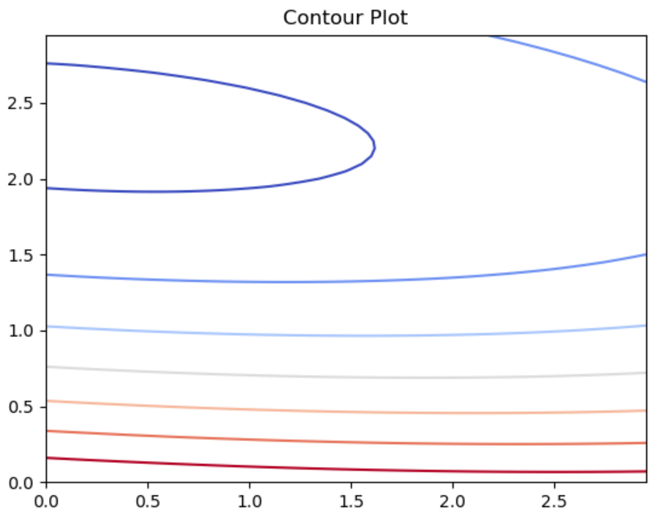


Figure 6: 6(a)w1, w2 take values between 0 to 3

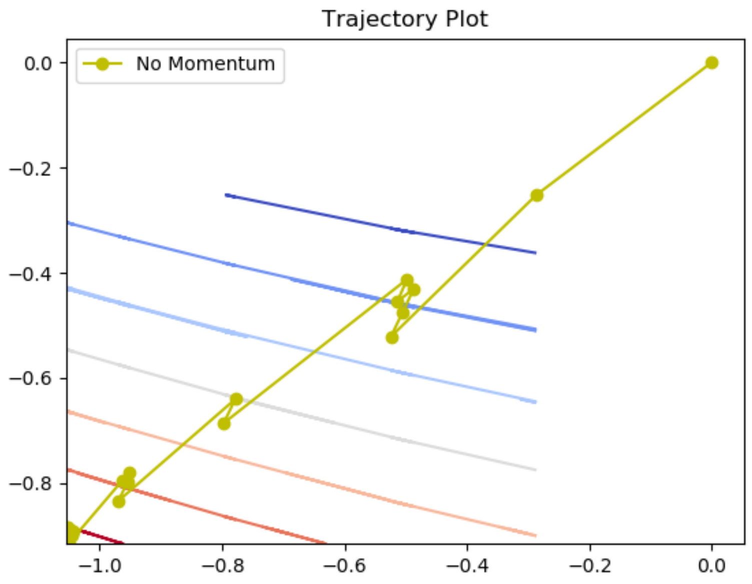


Figure 7: 6(b)

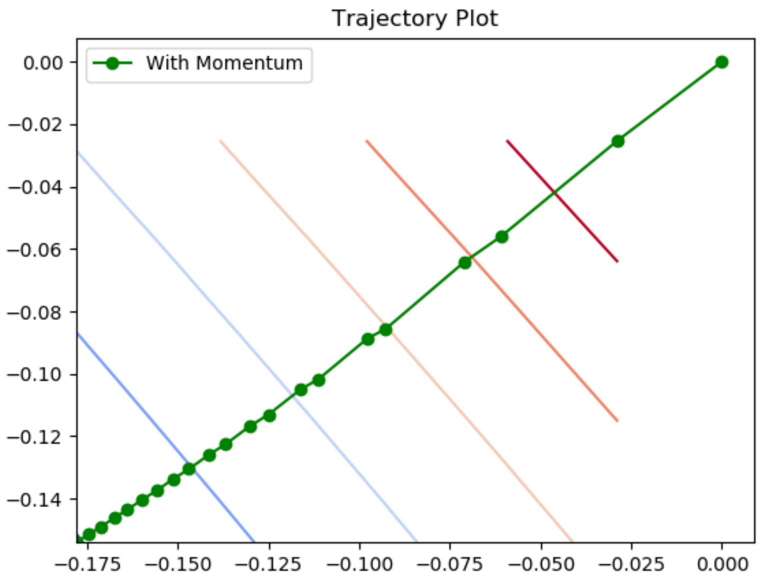


Figure 8: 6(c)

Part6(d)

Describe any differences between the two trajectories. Provide an explanation of what caused the differences.

Trajectory Plot with Momentum is more stable and smooth. Using momentum in order to reach the peak faster.

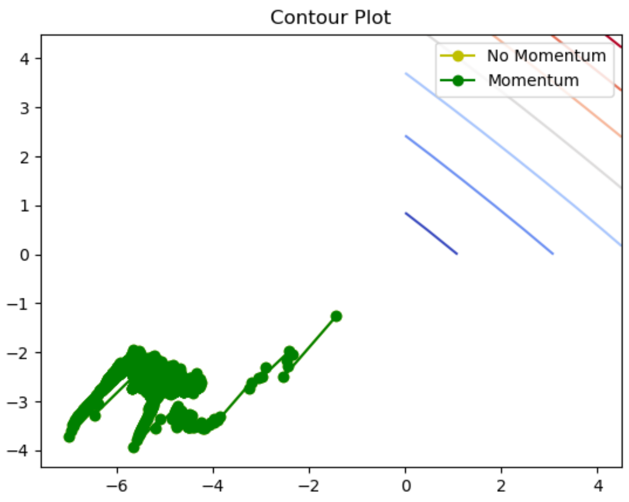


Figure 9: 6(e).2

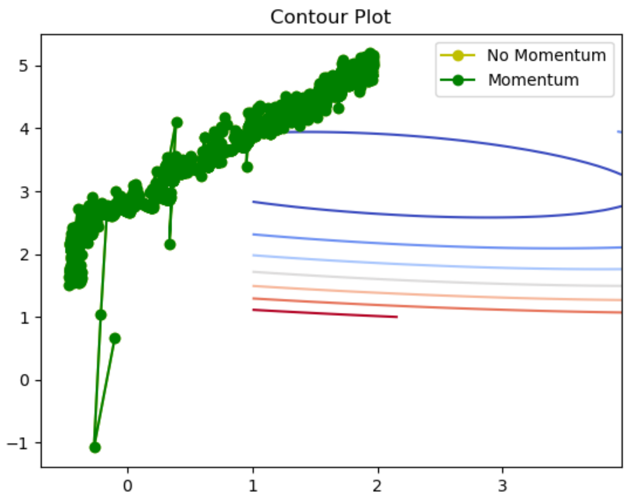


Figure 10: 6(e).1

Part6(e)

The things that caused difference are:

1. where did we pick w1, w2
2. what range did we pick for w1, w2 values

Part 7

Backpropagation can be seen as a way to speed up the computation of the gradient. For a network with N layers each of which contains K neurons, determine how much faster is (fully-vectorized) Backpropagation compared to computing the gradient with respect to each weight individually, without caching any intermediate results. Assume that all the layers are fully-connected. Show your work. Make any reasonable assumptions (e.g., about how fast matrix multiplication can be peformed), and state what assumptions you are making.
Hint: There are two ways you can approach this question, and only one is required. You may analyze the algorithms to describe their limiting behaviour (e.g. big O). Alternatively, you may run experiments to analyze the algorithms empirically.

Assumption:

- Let A be a matrix of size a × b, and B is a a matirx of b × c.
  The matrix multiplication complexity of A and B is O(abc) since the total steps of calculation is abc.
- All the layers are fully-connected.

Assume a network with N layers each of which contains K neurons.
l is each layer ∈ [1, ..., N]. j, i is neurons ∈ [1, ..., K]

$$\frac{\partial C}{\partial W^{(l,j,i)}} = \frac{\partial C}{\partial h^{(l,i)}} \cdot \frac{\partial h^{(l,i)}}{\partial W^{(l,j,i)}} = (\sum_i \frac{\partial C}{\partial h^{(l+1,i)}} \cdot \frac{\partial h^{(l+1,i)}}{\partial W^{(l,j,i)}}) \cdot \frac{\partial h^{(l,i)}}{\partial W^{(l,j,i)}}$$

$(\sum_i \frac{\partial C}{\partial h^{(l+1,i)}} \cdot \frac{\partial h^{(l+1,i)}}{\partial W^{(l,j,i)}})$ while i sum to K. Since we already know, i ∈ [1, ..., K]
Therefore, the complexity is O(K) for one layer.
Expected we can continue expand the formula: using the chain rule

$$= (\sum_i (\sum_i \frac{\partial C}{\partial h^{(l+2,i)}} \cdot \frac{\partial h^{(l+2,i)}}{\partial h^{(l+1,i)}}) \cdot \frac{\partial h^{(l+1,i)}}{\partial W^{(l,j,i)}}) \cdot \frac{\partial h^{(l,i)}}{\partial W^{(l,j,i)}}$$

$$= (\sum_i (\sum_i (\sum_i ...) \cdot \frac{\partial h^{(l+2,i)}}{\partial h^{(l+1,i)}}) \cdot \frac{\partial h^{(l+1,i)}}{\partial W^{(l,j,i)}}) \cdot \frac{\partial h^{(l,i)}}{\partial W^{(l,j,i)}}$$

We have a network with N layers, formula got expand N times. The complexity of the whole one is $O(K^N)$

When computing vectorized expresion of $\frac{\partial C}{\partial W}$, we can get following:

$$\frac{\partial C}{\partial W^(l)} = \delta^{(l)} \cdot (o^{(l-1)})^T$$

l states a specific layer, δ is a size of k × 1 matrix, o is a 1 × k matrix. (o^T is 1 × k)
Therefore, the multiplication takes k × k steps.
From above, we can conclude that the back-propagation complexity is $O(k \times k)$, while computing with respect to each weight individually is $O(K^N)$.

Part 8

We have seen PyTorch code to train a single-hidden-layer fully-connected network in this tutorial Modify the code to classify faces of the 6 actors in Project 1.

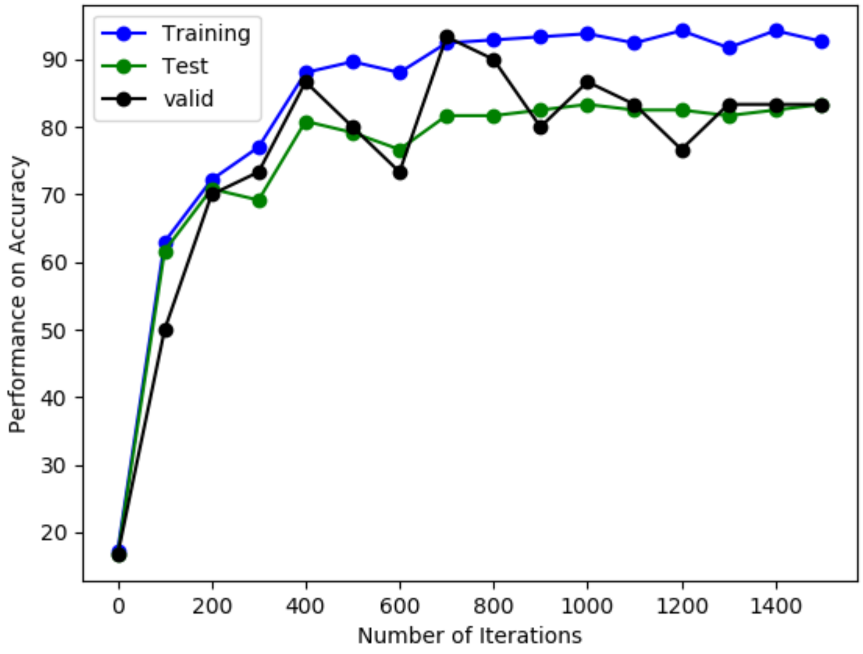


Figure 11: part8

First, we process the import by using get_data. We download all the images in cropped directory. We use SHA256 to check, if the actual hash value is equal to the hash value given in facescrub_actors.txt and facescrub_actresses.txt. Then we removed image that does not match the hash value. After that, we hard code the indices of images that we do not want. Then we resize the image into 32 x 32 (We found the resolution has little effect on the final result). We use rgb images because we found this performs better result (higher performance). We have 32 images for training for gilpin and 60 for each other actors/actresses. Some of the images are originally gray-scaled, so we change its shape from (32, 32) to (32, 32, 3).

Training set size: 60 for Gilpin, 75 for other actors/actresses, since Gilpin has only 86 images.
Test set size: 20 for each actors/actresses

The architecture of the network is :

```
dim_x = 32 * 32 * 3
dim_h = 300
dim_out = 6
```

We have 3072 pixels per sample, 300 units for single hidden layer, 6 units for output layer. The activation function is ReLU. We use mini-batch to compute the updates of gradient by using a smaller size of training size. The batch size we use is 20. This helps the program run faster and avoid local minimum.

The optimizer we choose is Adam. We tried SGD, SGD with momentum, RMSprop and Adam, and found that Adam performs the best. SGD is the most basic method, but it takes the longest time to reach the target. Momentum performs better than SGD, it accelerates the updates. RMSprop considered AdaGrad method, which add a resistant when updates goes to wrong direction, however it does not include momentum. Adam condidered both Momentum and AdaGrad method, and it performs better than RMSprop.

We intialize weight use default weight setting. From the source code of pytorch, we know that the default weight is initialized using a uniform distribution, and its values is between -1/sqrt(in_features) and 1/sqrt(in_features)

```
source code: http://pytorch.org/docs/0.3.0/_modules/torch/nn/modules/linear.html#Linear
```

The learning rate we choose is 0.001. We found if this value become higher or lower will cause low accuracy. We iterates 1500 times. The more iteration we have, the better results we obtained.
The final result we obtained is:

```
training set accuracy: 92.6436781609
testing set accuracy:  83.3333333333
validation set accuracy:  83.3333333333
```

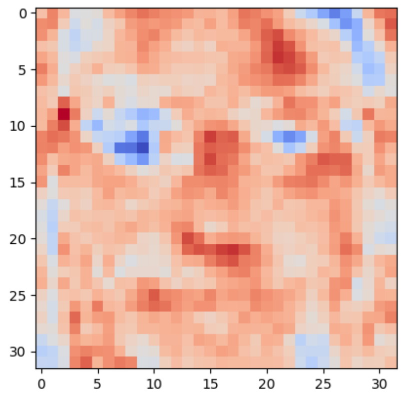
Part 9

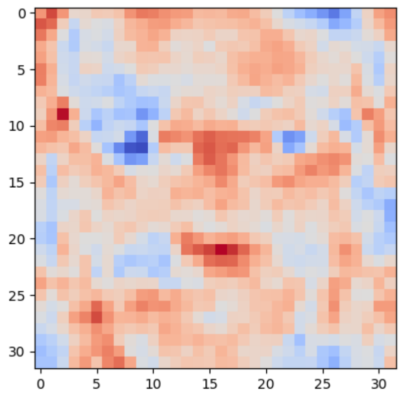
Select two of the actors, and visualize the weights of the hidden units that are useful for classifying input photos as those particular actors. Explain how you selected the hidden units.

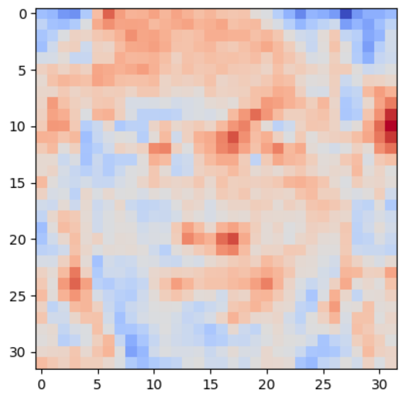
We first find the weight that is associated most with activation, which is the weights of the hidden units that are useful for classifying input photos, by using argsort(), which sorted the object. It returns an array of indices that index data along the given axis in sorted order.
Since we are using the rgb images. We then:

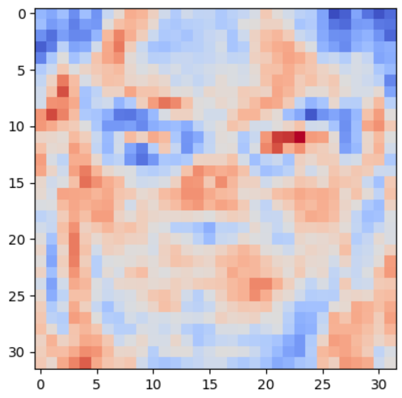
```
W = (W[:,:,0] + W[:,:,1] + W[:,:,2])
```

Angie Harmon:

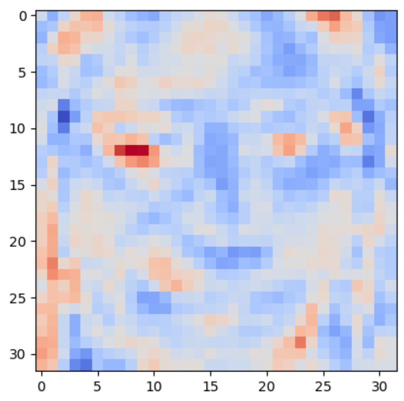
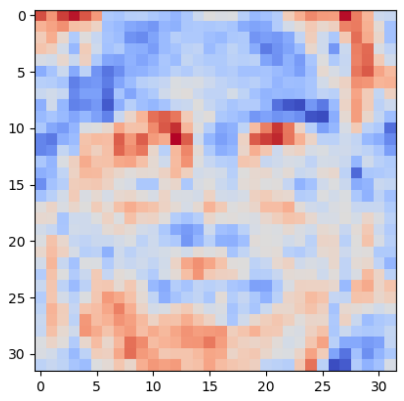
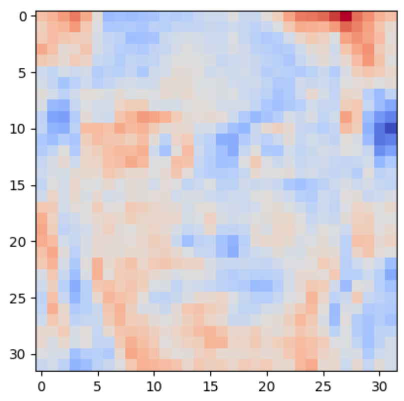
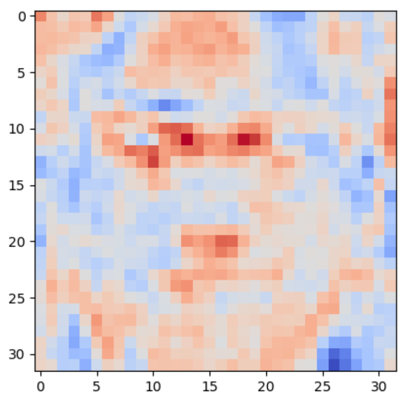

0
5
10
15
20
25
30


0
5
10
15
20
25
30

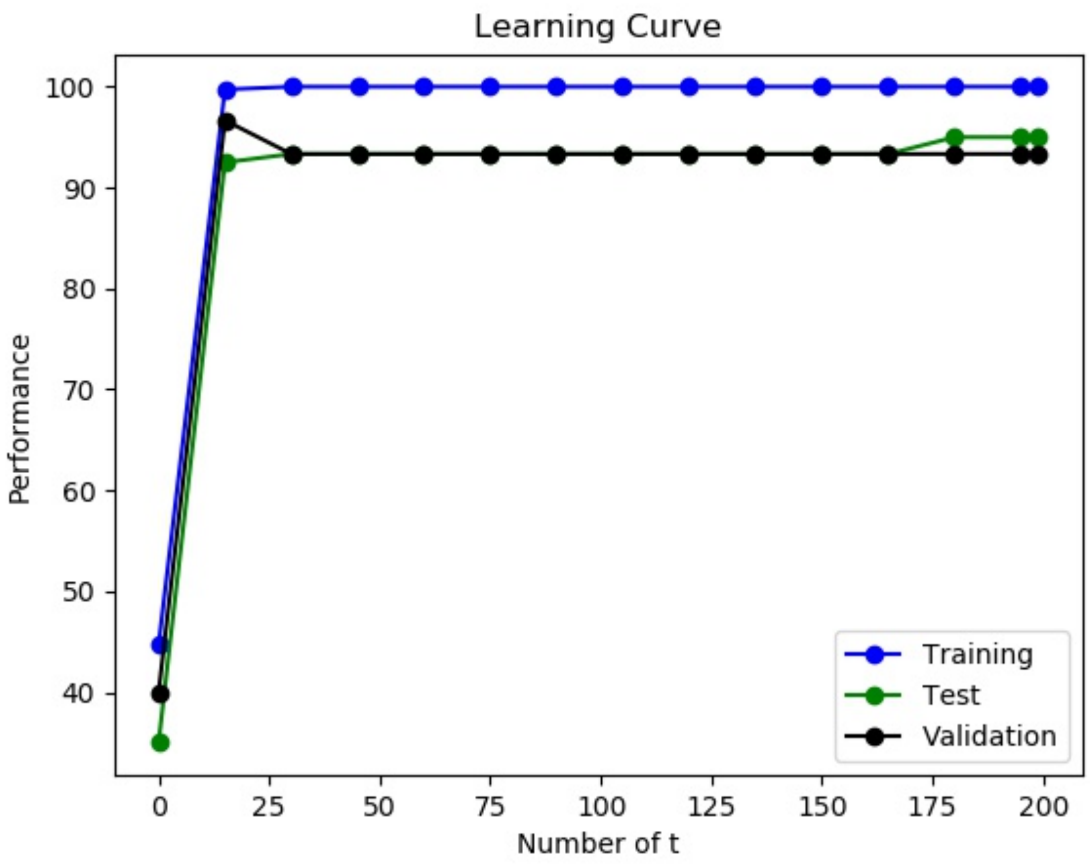

0
5
10
15
20
25
30


0
5
10
15
20
25
30

Lorraine Bracco:

Part 10



We used modified get_data.py to get images (cropped_rgb227_1 and cropped_rgb227_2) which are all resized to 277 x 277. AlexNet will work better if the image size is around 227x227, this is because the size of images in its training set is 277. This is bit different from part 8. We have 60 images for the training set for each actors/actresses. We modified the given function 'def forward(self, x)' inside class MyAlexNet. In order to extract the values of the activations of AlexNet on the face images in a particular layer.

We first use get_set() to divide data into training set, validation set, and test set. Then we use the code provided to deal with the images. We modified the given function 'def forward(self, x)' inside class MyAlexNet, in order to extract the values of the activations of AlexNet on the face images in a particular layer. We extract the values of the activations of AlexNet on the face images in Conv4. We modified the given function 'def forward(self, x)' inside class MyAlexNet to return the activations from the Conv4 layer.

```
def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        return x
```

```
model.forward(imv).data.numpy()
```

We used functions from project 1 get_set() and get_several_y() with a bit modification. get_several_y() helps us find the y labels. We obtained training set and y labels for training by that. Then we followed the given code and train data with AlexNet.
The architecture of the network is:

```
dim_x = 256 * 6 * 6
dim_h = 300
dim_out = 6
```

The dimension of input x is 9216. The number of hidden layer we use is 400, and there are 6 units for output layer. The network is fully connected. We have tried different number number of hidden layer, 200, 300, 400, and found that 400 performs the best. The learning rate is 0.0001, and we iterates for 200 times. The batch size is 20.
The result we obtained, is better then we obtained in part8. It reaches a higher performance at an earlier iteration.
The final performance is:

```
train_perform 100.0
test_perform 94.1666666667
valid_perform 93.3333333333
```