CSC411: Assignment 2

Due on Friday, Feb.23, 2018

Zhongtian Ouyang/Yihao Ni

Problem 1

Dataset description

The pictures are loaded from mnist_all.mat and combined using the showDigits() function. Each line is a digit from 0 to 9. Based on those 10 images for each digit, this dataset includes various styles of writing those digits. Most of the images are great, but some of those images consists of something unrelated. For example, the third image of '1' has a ':' next to it, and the first image of '5' has a meaningless circle.

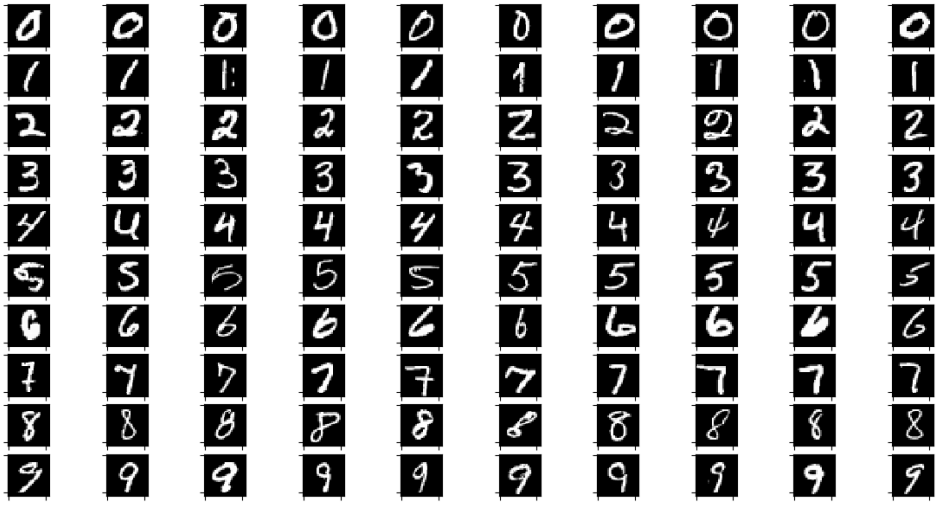


Figure 1: 10 random images for each digit

Problem 2

Neural network

The function forward_p2 first compute o using the formula given, then compute the softmax using the softmax function. The requirements of the inputs are included in the comment. The code is as follows:

```
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 forward_p2(x, w, b):
    '''
    the input x should be 784 x n, the input w should be 784 x 10, the input b
    should be 10 x 1 in our case
    '''
    Os = dot(w.T, x) + b
    result = softmax(Os)
    return result
```

Problem 3

Compute the gradient
Part(a):
The cost function for s sample cases:

$$C = \sum_s (-\sum_k y_k^s log p_k^s)$$

Apply the chain rule to get the gradient of the cost function with respect to the weights:

$$\frac{\partial C}{\partial W_{ij}} = \frac{\partial C}{\partial o_i}\frac{\partial o_i}{\partial w_{ij}} = \sum_s (p_i^s - y_i^s)x_j^s$$

Where $\frac{\partial C}{\partial o_i}$ is computed as follows:

$$\begin{aligned}
\frac{\partial C}{\partial o_i} &= \frac{\partial C}{\partial p_i}\frac{\partial p_i}{\partial o_i} + \sum_{k\neq i}\frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial o_i} \\
&= -\frac{y_i}{p_i}p_i(1-p_i) - \sum_{k\neq i}\frac{y_k}{p_k}(-p_i p_k) \\
&= y_i p_i - y_i + \sum_{k\neq i} p_i y_k \\
&= \sum_k p_i y_k - y_i \\
&= p_i - y_i
\end{aligned} \tag{1}$$

$\frac{\partial o_i}{\partial w_{ij}}$ simply equals to:

$$\frac{\partial o_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\sum_s (w_{i0}x_0^s + w_{i1}x_1^s + ... + w_{ij}x_j^s + ... + b) = \sum_s x_j^s$$

Where $i = 0, ..., 9$ and $j = 0, ..., 783$ in our case, s is the sample size. Therefore $\frac{\partial C}{\partial W_{ij}}$ has a dimension of $10 * 784$.

Part(b):
The code for computing the gradients is as follows:

```
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 forward_p2(x, w, b):
    '''
    the input x should be 784 x n, the input w should be 784 x 10, the input b
    should be 10 x 1 in our case
    '''
    Os = dot(w.T, x) + b
    result = softmax(Os)
    return result
```

```
def grad_p3(x, w, b, y):
    '''
    return the gradient of weights and the gradient of biases
    '''
    p = forward_p2(x, w, b)
    dc_do = p - y
    return np.dot(dc_do, x.T).T, np.dot(dc_do, np.ones((x.shape[1], 1)))
```

Parameter dc_do in function grad_p3 is $\frac{\partial C}{\partial o}$ in part(a). Following is obtained by approximating the gradient at several coordinates using finite differences with a random generated 25-sample set:

when h = 0.1:
The sum of the absolute value of difference for weight is 0.11572252765064116
The average of the absolute value of difference for weight is 1.4760526486051168e-05
The sum of the absolute value of difference for bias is 0.0005599162912244271
The average of the absolute value of difference for bias is 5.599162912244271e-05

When h = 0.001:
The sum of the absolute value of difference for weight is 1.1595024297009595e-05
The average of the absolute value of difference for weight is 1.4789571807410197e-09
The sum of the absolute value of difference for bias is 5.597965579973163e-08
The average of the absolute value of difference for bias is 5.597965579973163e-09

When h = 0.0001:
The sum of the absolute value of difference for weight is 3.1475914028065034e-06
The average of the absolute value of difference for weight is 4.0147849525593155e-10
The sum of the absolute value of difference for bias is 3.5902092410111663e-09
The average of the absolute value of difference for bias is 3.5902092410111663e-10

It is clear that when h gets smaller, the difference gets smaller as well. Therefore if h gets arbitrarily small, the difference approaches zero, which indicates our gradient is computed correctly.

Problem 4

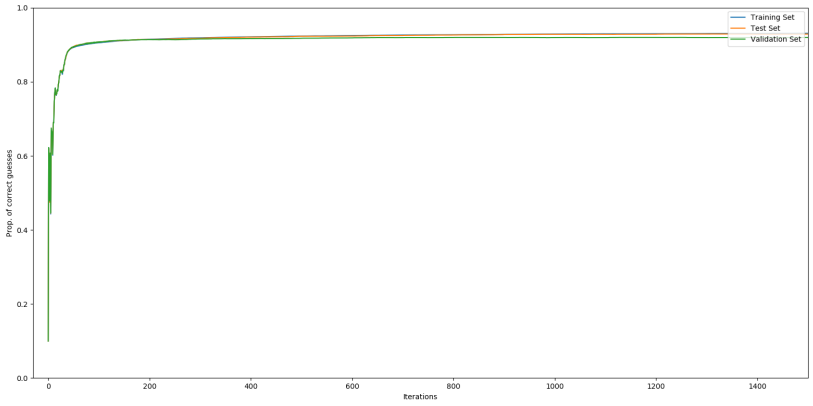
Gradient descent without momentum

The sets are obtained through the get_sets() function. To get the validation and training set. we shuffled the training set from the data set and use the first 4000 as the training set and the following 1000 as the validation set. For the test set, we shuffle the set from data and take the first 850 cases.

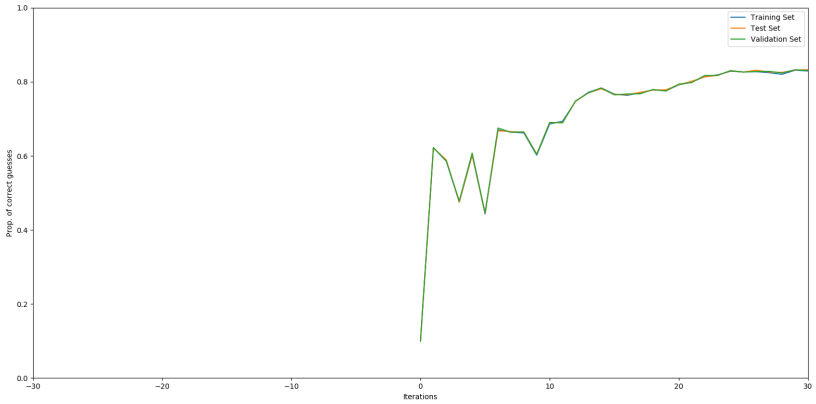
We had tried to initialize the weight with random numbers, normal distributed normal numbers and all zeros. We choose the weight to be instialized as all zeros because that would keep the weights correspond irrelevant pixels, which are the pixels that are always black in all cases, keep at 0. While the weights for relevant point would have a non zero weight and thus become non-zero.

We pick 0.00003 as learning rate. Bigger values such as 0.01 will lead to overflow, and values like 0.0001 will cause the cost to increase in some iterations ,showing that the weight is moving back and forth around the center. The full learning curve runs for 1500 iteration. To make it clear, we also include the curve with only 30 iterations. Following is the curve.

NOTICE: the lines are pretty much identical. But if observed carefully, you can see three seperated lines.



(a) Full learning curve



(b) learning curve for first 30 iterations

Figure 2: Part4

The weights connecting to each output are as follows:

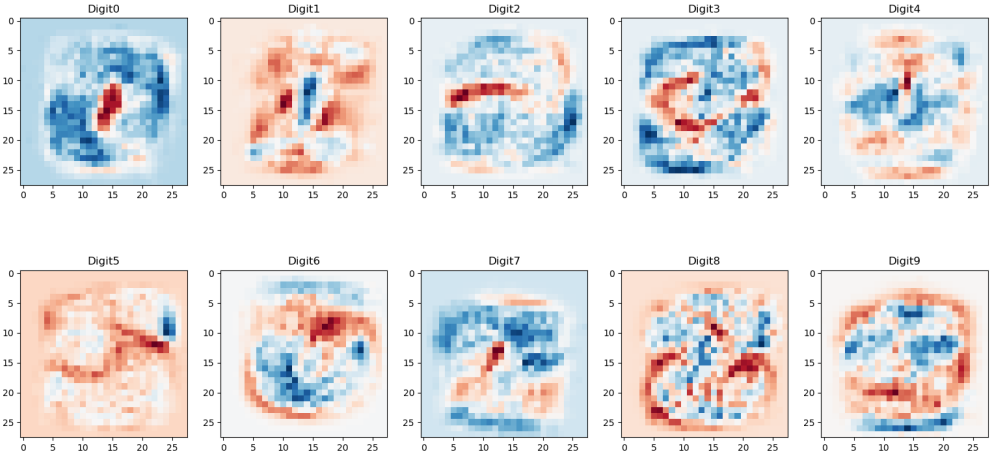


Figure 3: Visualization of weights

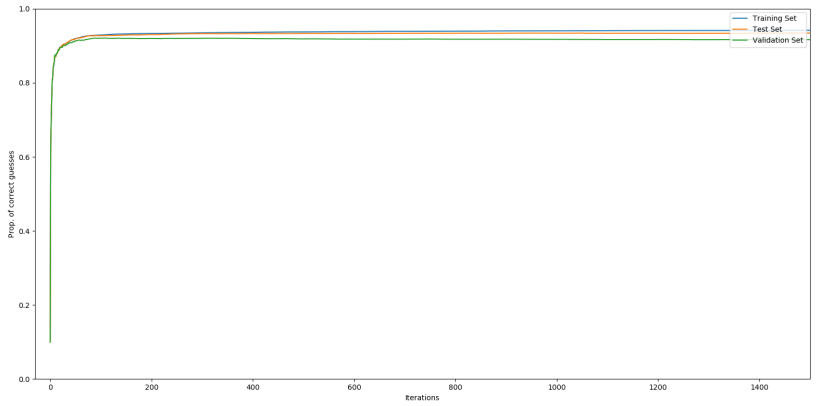
Problem 5

Gradient descent with momentum

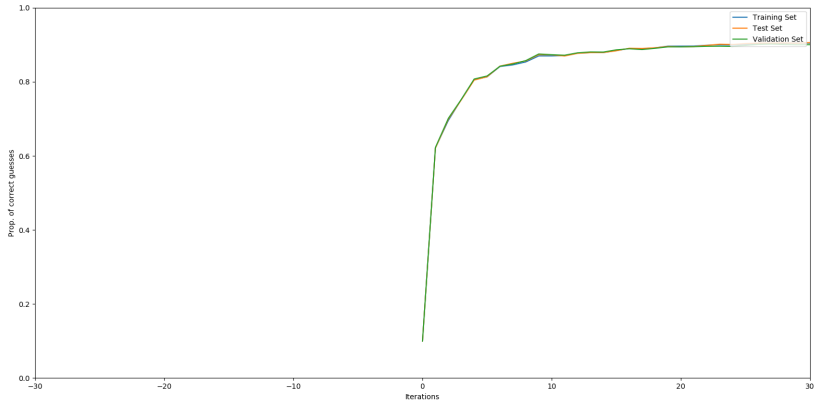
We use the same learning rates and iteration as problem 4. And we use 0.9 as the gamma value. All the sets are the same as previous

Following is the code that performs gradient descent with momentum and the learning curves. We can observe that although the full learning curve looks similar to the one without momentum, if we focus on the first 30 iterations, the one with momentum is much more smooth and more effecient. It reaches 0.8 with only about 5 iterations, while the one without momentum needs over 10 iterations. And the smooth curve shows that we are keep running toward the center.

```
def grad_descent_m(f, df, x, y, init_t, init_b, alpha, gamma, max_iter, test,
                   t_answer, validation, v_answer, plotCurve=True):
    EPS = 1e-5  # EPS = 10**(-5)
    prev_t = init_t - 10 * EPS
    t = init_t.copy()
    prev_b = init_b - 10 * EPS
    b = init_b.copy()
    vt = 0
    vb = 0
    iter = 0
    perfomance_x = [perform(x, t, b, y)]
    perfomance_t = [perform(test, t, b, t_answer)]
    perfomance_v = [perform(validation, t, b, v_answer)]
    iterations = [0]
    print('Doing gradient Descent')
    while (np.linalg.norm(t - prev_t) + np.linalg.norm(
                b - prev_b)) > EPS and iter < max_iter:
        prev_t = t.copy()
        prev_b = b.copy()
        grad_t, grad_b = df(x, t, b, y)
        vt = gamma * vt + alpha * grad_t
        vb = gamma * vb + alpha * grad_b
        t -= vt
        b -= vb
        if iter % 100 == 0:
            print("Iter", iter)
            print("f(x) = %.2f" % (f(forward_p2(x, t, b), y)))
            # print("Gradient: ", df(x, t, b, y), "\n")
        if plotCurve:
            perfomance_x.append(perform(x, t, b, y))
            perfomance_t.append(perform(test, t, b, t_answer))
            perfomance_v.append(perform(validation, t, b, v_answer))
            iterations.append(iter + 1)
        iter += 1
    if plotCurve:
        plt.plot(iterations, perfomance_x)
        plt.plot(iterations, perfomance_t)
        plt.plot(iterations, perfomance_v)
        plt.legend(['Training Set', 'Test Set', 'Validation Set'],
                   loc='upper right')
        plt.axis([-30, max_iter, 0, 1])
        plt.ylabel('Prop. of correct guesses')
        plt.xlabel('Iterations')
        plt.show()
    print('Final Cost is ' + format(f(forward_p2(x, t, b), y), '.2f'))
    return t, b
```



(a) Full learning curve



(b) learning curve for first 30 iterations

Figure 4: Part5

Problem 6

Momentum performance

Part(a)

Following is the contour plot of the cost function from Part 5. The two weights are chosen to be $11 * 28 + 11$(pixel[11,11]) and $17 * 28 + 17$(pixel[17,17]), since pixels in the center of the image is more likely to be part of the digit, and thus does not has weight zero. The center is around (-0.5, -0.6).

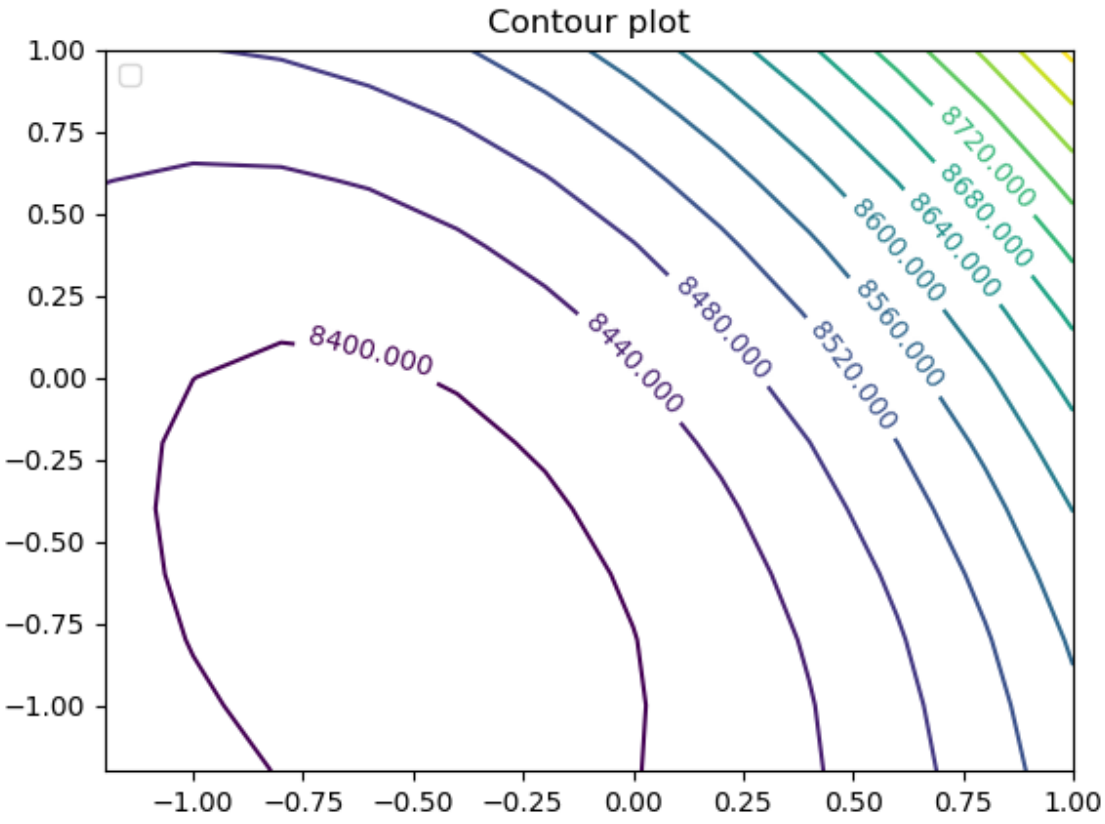


Figure 5: contour plot of the cost function

Part(b)(c)(d)

Starting point is (0.8, 0.8). Learning rate is chosen to be 0.003 for both trajectories, 100 times as it was in Part5, this is because the max iteration here is just 20 to visualize all points on the graph. Gamma is 0.7, smaller than that in Part(5), since larger gamma such as 0.9 chosen in Part(5) would make it cross the center point to the lower left side, which is too much. Green dots are gradient descent with momentum, while yellow ones are that without momentum. It can be visualized in this graph that the green dots approaches

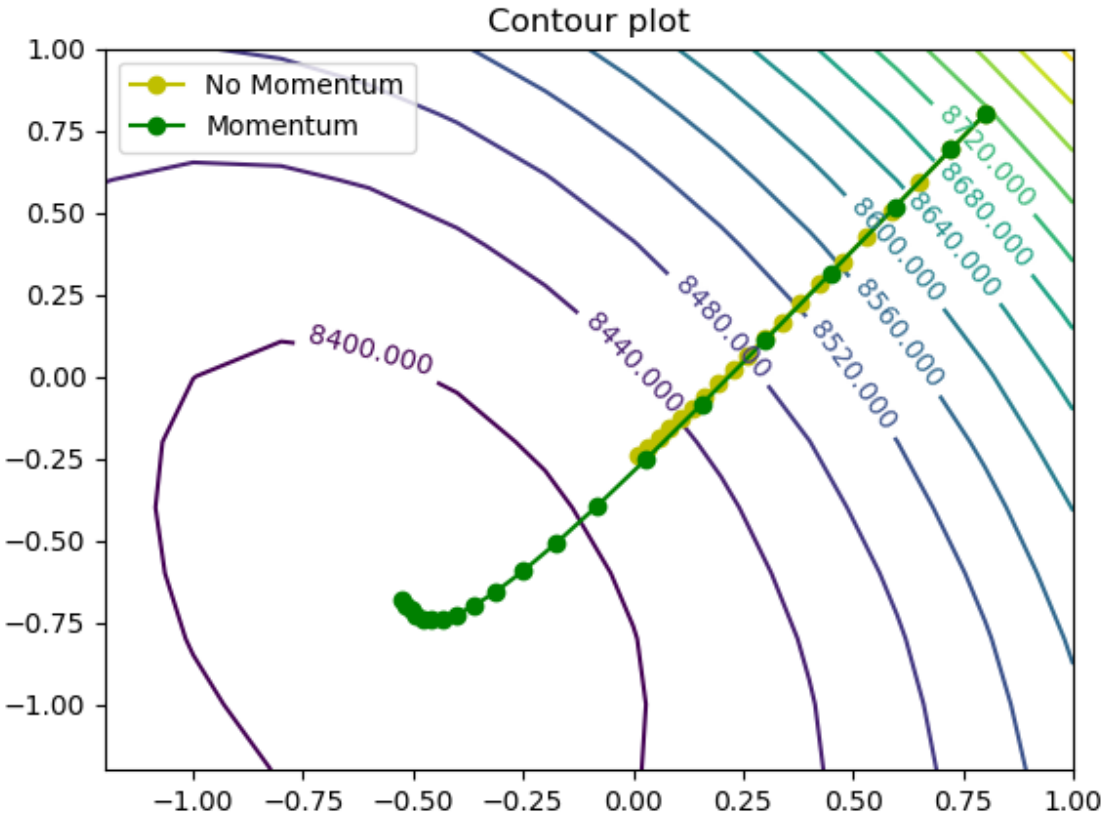


Figure 6: contour plot of the cost function with two trajectories

the center much faster than yellow dots do. This is because when we use gradiant descent with momentum, each step is increased with the amount calculated by last step, so that it approaches the center point faster.

Part(e):
This is an example of bad choice of settings and weights. The two chosen pixels are the upper-left one and the bottom-right one, in the corners, which is more likely to have nothing to do with the digits. It can be observed that nothing is on the graph except for a green point (the yellow one might be covered by it). The contour line does not appear since all points on the graph have a same (or similar) value, thus no contour lines to be shown. If one of the point is bad, the graph of the contour map would be either all horizontal lines or all vertical lines.

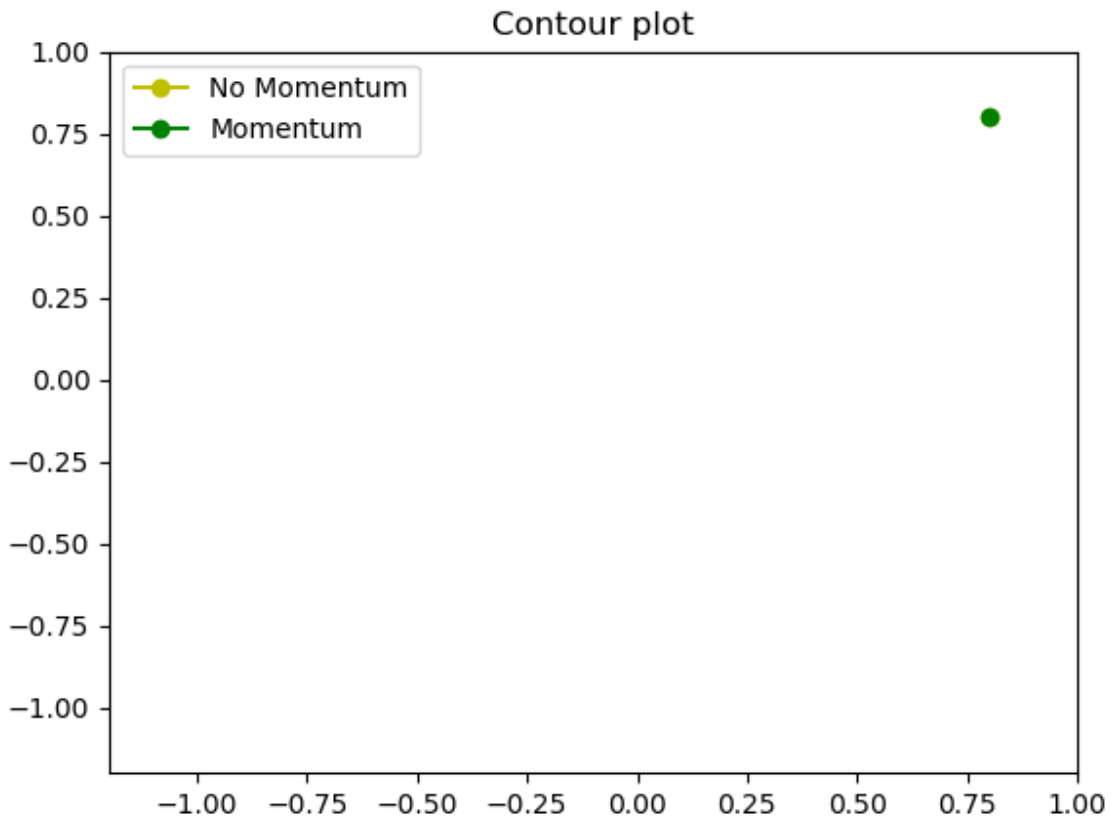


Figure 7: Inappropriate choise of settings and weights

Problem 7

Backpropagation for a network with NK neurons

For a network with N layers each of which contains K neurons, there are K*N neurons in total.

Cost for fully vectorized backpropagation:

Using the cache values, the derivative with respect to each weight between layers can be calculated by just one matrix multiplication. Given each layer contains K neurons, the matrices are K by K. Multiply two K by K matrices has complexity $O(n^3)$. Therefore, the complexity of vectorized backpropogation is $O(NK^3)$.

Cost for computing gradient individually:

For a specific weight into layer N (the upper one) the complexity would be O(1), and for all the weights the complexity is $O(K^2)$, and for layer N-1 (the second upper one) the complexity would be $O(n^3)$. Since we are computing those values individually, those complexity have to be summed up to get the total cost, and it would be: $O(K^2) + O(K^3) + ... + O(K^N)$, which is complexity $O(K^N)$.

For large values of K and N, using fully vectorized backpropagation change the complexity from $O(K^N)$ to $O(NK^3)$, which is a big improvemnet.

Problem 8

Single-hidden-layer fully-connected network with pytorch
The performance of the network on test set is 88.13%, on validation set is 89.83%, on training set is 100%
The images are downloaded and cropped using get_data.py. They are resized to 32*32 when loaded from cropped folder
The network has a bottom layer with 3072 units, a top layer with 6 units and one fully connected hidden layer with 12 units. The activation function for the hidden layer is the ReLU function. I had also tried Tanh and Sigmoid as activation function, but they both yield worse results.
The cost function I used is the CrossEntropyLoss function. Since this loss function already include softmax, I didn't add a softmax layer in the network
And I use Adam as an optimizer.The max learning rate is 1e-3 for the optimizer.
The weights for the two linear layers are initialized randomly with normal distribution with a mean at 0 and a variance at 0.01. During testing, when I was using tahn with variance at 1, I find out performance drop by about 5%. The reason is that when the variance is too large, some weights is initialized too far away from the ideal with almost zero gradient for a tahn function as it flatten out on the sides.
I initialized the bias to 1 when using ReLU because gradient for a ReLU is 0 when x ¡ 0
The samples are divided into minibatches. I first choose batch size to be 10, and gradually increase it to 50, 75, 100, 200. Batch size 100 yields the best result among the choices. I stop the training at 900 epochs to prevent overfitting the model too much and decrease the performance on the validation set.
And with some testing using 12,15,18,21,24 neurons, all yield a similar result on the validation set. And since 12 neuron is least expensive to calculate, I choose to use 12 neurons.
Also, to avoid overfitting, I applied L2 regularization by setting the weight_decay for Adam optimizer to 1e-5
Conclusion:12 neurons in the hidden layer; ReLU activation function; bias is init to 1; weight is init to normal with mean = 0, variance = 0.01; The batch_size is 100; stopped at 900 epochs

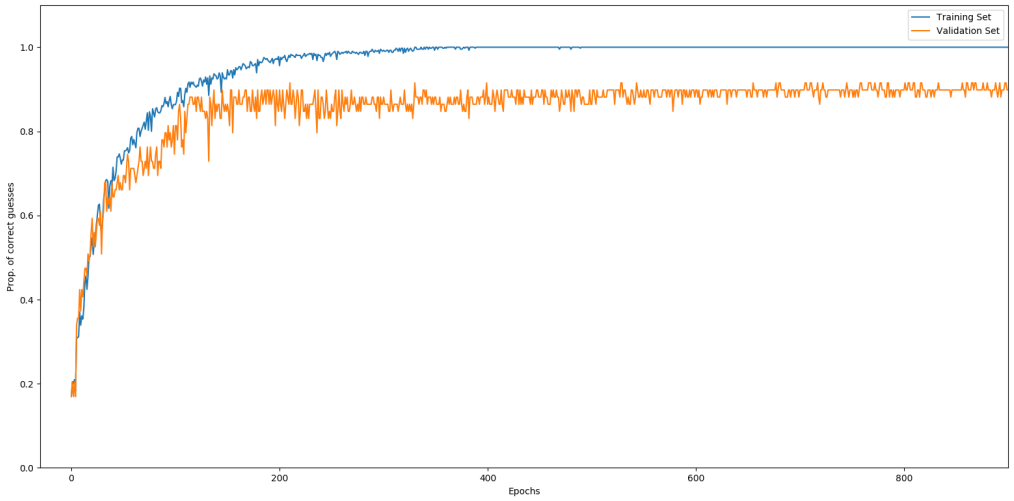


Figure 8: Learning curve

Problem 9

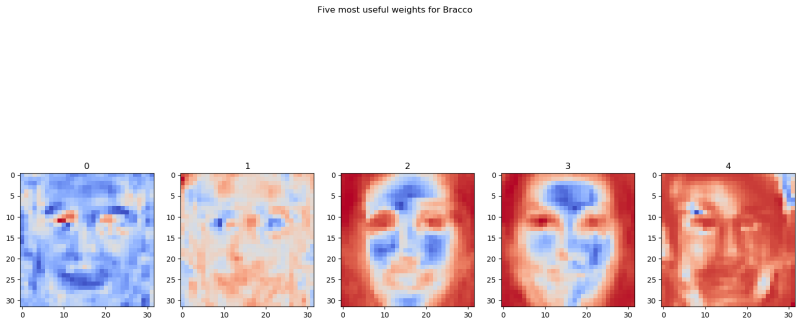
Useful neurons to classify two specific actors

The two actors I choose are Bracco and Baldwin. Below are the visualization of the weights of the 5 most useful neurons to classify each actor. Here are the steps to find out the most useful neurons:

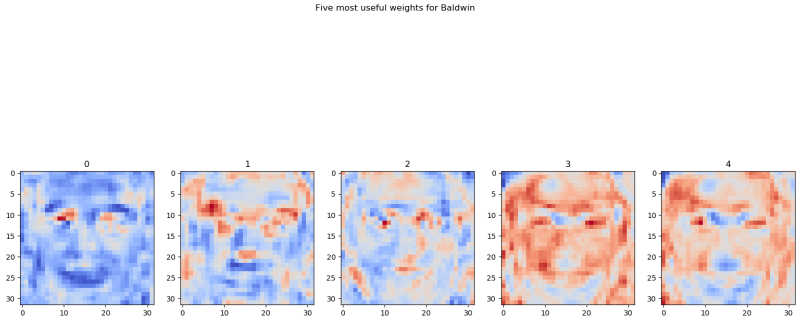
1. Get the weights w of the 6-neuron top layer of the network. The shape of the weights would be 6 * number of hidden neurons in our case. 2. Each row of w would be how much each hidden-layer neuron contribute to classifying a specific actor. For example, The first row would be for the actor with y being [1, 0, 0, 0, 0, 0], Bracco in our case. And the value at index k of that row shows how the kth neuron contribute to classifying a picture as Bracco. If the value is positive, it means activating the neuron suggest that this is a picture of Bracco. Vice Versa.

3. Find the indexes of the five largest numbers in the actors' row. A greater number means the evidence is stronger.

4. Visualize the weights of the neurons in the hidden layer that correspond to the five indexes we find in previous step.



(a) Neurons supporting Bracco



(b) Neurons supporting Baldwin

Figure 9: Part9

Problem 10

Single-hidden-layer network with AlexNet

I load the pictures from cropped folder, resized them to 227 * 227, do the prepocess and group them into sets in get_sets function.

I extract the value from the last MaxPool2d layer. I do that by modifying the forward function to following. If I want to use another layer, I would just keep that layer and the layers before it in the features. When I passing in the values to Alexnet, I passed in 40 pictures a time so that the ram used is around 2 GB. Then I convert the outputs of the function to matrixs with Variable.data.numpy() and stacks them together to get the result. Finally, using the result as the input for my network.

The structure of my network is consist of a input layer with 9216 units, a hidden layer with 12 neurons, and an output layer with 6 units. The output would be the one-hot coding of the actors. The activation function is ReLU. The bias is init to 1. The weight is init to normal with mean = 0, variance = 0.01. The batch_size is 100. It is stopped at 1000 epochs.

The performance of the network is 96.61% on test set, 100% on training set, 94.91% on validation set.

Compare to the performance in part8, we reduce the error rate by more than 50%

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