CSC411: Project 2

Due on Monday, February 26, 2018

Anas Al-Raheem

Part 1

The data-set consists of about 6000 images for each digit, representing 0 to 9, each image is of the size 28×28-pixel. For each digit there is a huge variety of styles of writing, different in size, shape and alignment of the digit. We can see that most of the digits have some unique parts that allows the program to work properly and be able to build a classifier. Below is a sample of the digits:

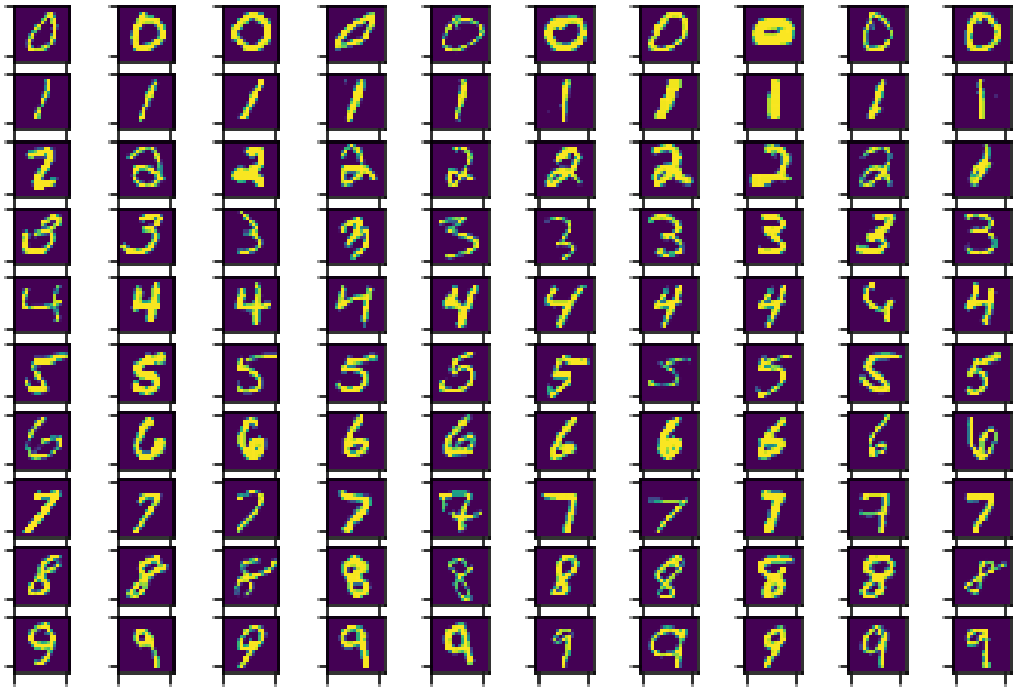

Figure 1: Digits Sample

Part 2

The code below represent the required neural network:

```
def softmax(y):
    return exp(y) / tile(sum(exp(y), 0), (len(y), 1))

def forward(x, W0, b0):
    L0 = dot(W0.T, x) + b0
    output = softmax(L0)
    return L0, output

def get_performance(x, y, W0, b0):
    L0, output = forward(x, W0, b0)
    expected = np.array([i.tolist().index(1) for i in y.T])
    actual = np.array([i.tolist().index(max(i)) for i in output.T])
    results = (expected == actual)
    true_num = results.tolist().count(True)
    performance = true_num / float(output.shape[1])
    return performance * 100

def cost(x, W0, b0, y):
    L0, p = forward(x, W0, b0)
    while p.min() < 1e-323:
        ind = np.unravel_index(np.argmin(p, axis=None), p.shape)
        p[ind] = 1e-323
    return -np.sum(y * np.log(p))

def get_prediction(W0, b0, x):
    x = test_x
    L0, output = forward(x, W0, b0)
    y = np.argmax(output, 0)
    print y
    return y
```

Part 3

A):

We have the following functions:
The cost function

$$C = -y_i \log P_i$$

The probability

$$P_i = \frac{e^{o_i}}{\sum_k e^{o_k}}$$

The output unit

$$o_i = \sum_j w_{ji} x_j + b_i$$

The gradient of the cost function in respect to w_{ij} (*):

$$\frac{\partial C}{\partial w_{ij}} = \sum_k \frac{\partial C}{\partial P_k} \frac{\partial P_k}{\partial o_i} \frac{\partial o_i}{\partial w_{ij}}$$

We need to calculate each part of the derivative:

1:

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

2:

$$\frac{\partial P_i}{\partial o_i} = \frac{(\sum_k e^{o_k}) e^{o_i} - e^{o_i} * e^{o_i}}{(\sum_k e^{o_k})^2}$$

$$\frac{\partial P_i}{\partial o_i} = \frac{e^{o_i} (\sum_k e^{o_k} - e^{o_i})}{(\sum_k e^{o_k})^2}$$

$$\frac{\partial P_i}{\partial o_i} = \frac{e^{o_i}}{\sum_k e^{o_k}} \frac{\sum_k e^{o_k} - e^{o_i}}{\sum_k e^{o_k}}$$

$$\frac{\partial P_i}{\partial o_i} = P_i (1 - P_i)$$

3:

$$\frac{\partial P_j}{\partial o_i} = \frac{(\sum_k e^{o_k}) 0 - e^{o_j} * e^{o_i}}{(\sum_k e^{o_k})^2}$$

$$\frac{\partial P_j}{\partial o_i} = -\frac{e^{o_j} * e^{o_i}}{(\sum_k e^{o_k})^2}$$

$$\frac{\partial P_j}{\partial o_i} = -\frac{e^{o_j}}{\sum_k e^{o_k}} \frac{e^{o_i}}{\sum_k e^{o_k}}$$

$$\frac{\partial P_j}{\partial o_i} = -P_j P_i$$

4:

$$\frac{\partial o_i}{\partial w_{ij}} = x_j$$

Now we substitute in (*):

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial P_i} \frac{\partial P_i}{\partial o_i} \frac{\partial o_i}{\partial w_{ij}} + \sum_{k \neq i} \frac{\partial C}{\partial P_k} \frac{\partial P_k}{\partial o_i} \frac{\partial o_i}{\partial w_{ij}}$$

$$\frac{\partial C}{\partial w_{ij}} = -\frac{y_i}{P_i} P_i(1 - P_i) x_j + \sum_{k \neq i} \frac{y_k}{P_k} P_i P_k x_j$$

$$\frac{\partial C}{\partial w_{ij}} = -y_i x_j + y_i P_i x_j + P_i x_j \sum_{k \neq i} y_k$$

$$\frac{\partial C}{\partial w_{ij}} = -y_i x_j + P_i x_j (y_i + \sum_{k \neq i} y_k)$$

$$\frac{\partial C}{\partial w_{ij}} = -y_i x_j + P_i x_j (\sum_k y_k)$$

We are using one hot encoding thus all y values will be 0 except 1, thus the sum is 1.

$$\frac{\partial C}{\partial w_{ij}} = -y_i x_j + P_i x_j$$

$$\frac{\partial C}{\partial w_{ij}} = x_j (P_i - y_i)$$

That was the gradient with one image, for m number of images we will have:

$$\frac{\partial C}{\partial w_{ij}} = \sum_m x_j^m (P_i^m - y_i^m)$$

B):

Below is the code for the computation:

```
# gradient of the cost function with respect to the weights
def dw_cost(x, W0, b0, y):
    L0, p = forward(x, W0, b0)
    dCdL0 = p - y
    dCdW0 = dot(x, dCdL0.T)
    return dCdW0

# gradient of the cost function with respect to the biases
def db_cost(x, W0, b0, y):
    L0, p = forward(x, W0, b0)
    dCdL0 = p - y
    dCdb0 = np.sum(dCdL0, 1)
    return dCdb0.reshape(10, 1)
```

We can see the finite differences using the function $test\_gradient(W0, b0, x, y)$:

```
def test_gradient(W0, b0, x, y):
    h = 1e-5

    c1 = cost(x, W0, b0, y)
    W02 = W0.copy()
    W02[127][0] += h
    c2 = cost(x, W02, b0, y)
    diff = (c2 - c1) / h
    dc = dw_cost(x, W0, b0, y)
    print "weight gradient1:", dc[127][0], diff
    W02 = W0.copy()
    W02[272][7] += h
    c2 = cost(x, W02, b0, y)
    diff = (c2 - c1) / h
    dc = dw_cost(x, W0, b0, y)
    print "weight gradient2:", dc[272][7], diff

    b02 = b0.copy()
    b02[0] += h
    c2 = cost(x, W0, b02, y)
    diff = (c2 - c1) / h
    dc = db_cost(x, W0, b0, y)
    print "bias gradient1:", dc[0], diff
    b02 = b0.copy()
    b02[8] += h
    c2 = cost(x, W0, b02, y)
    diff = (c2 - c1) / h
    dc = db_cost(x, W0, b0, y)
    print "bias gradient2:", dc[8], diff
```

and the out put is:

```
weight gradient1: -0.17840041969105483 -0.178400400407952
weight gradient2: 0.10674925863689534 0.10674973540680809
bias gradient1: [-0.8920021] -0.8920016167479615
bias gradient2: [0.10321209] 0.10321255605738598
```

Part 4

The code used to initialize the weights and biases:

```
np.random.seed(0)
W0 = np.random.uniform(-0.01, 0.01, (784, 10))
b0 = np.random.uniform(-0.01, 0.01, (10, 1))
alpha = 1e-5
```

I noticed that a learning rate greater than $10^{-}5$ either would crash the program or would cause lower performance, probably due to the high learning rate we are skipping over the global minimum repeatedly. Smaller learning rates caused slow convergence and would require more iterations to reach the same performance. Regarding the weights and biases, they are randomly taken from a uniform distribution. I had some problems with log(0) and as the prof advised on Piazza I kept the numbers close to 0, in addition I added a condition to prevent log(0) from happening, by changing the values that cause this error to 10^{-323} (As I saw the error was caused by values smaller).

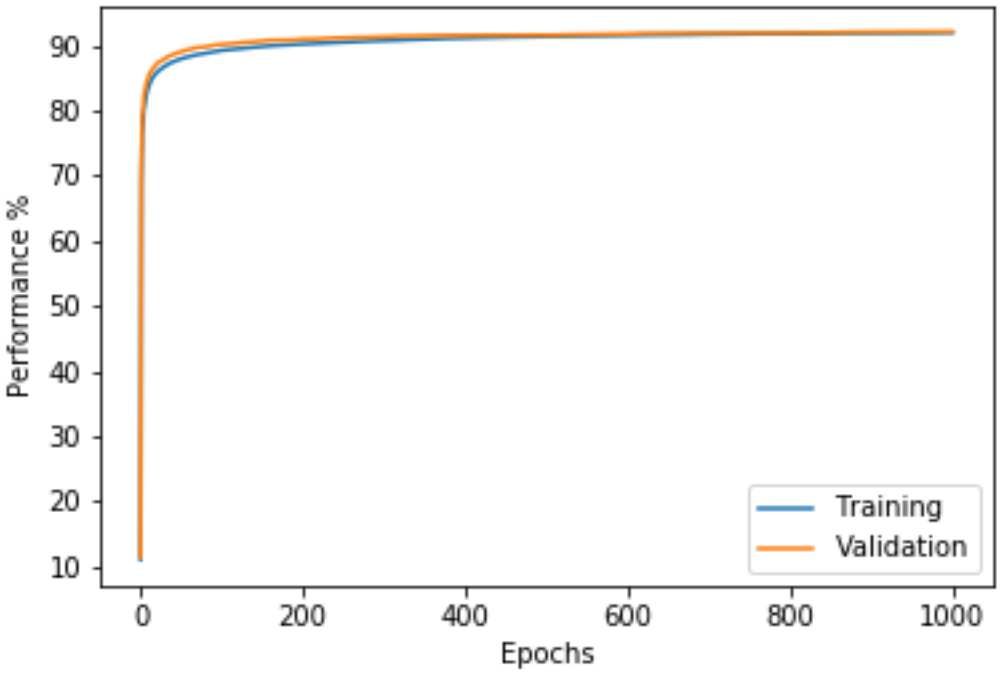

Figure 2: Learning Curve Without Momentum

Below are the weights:

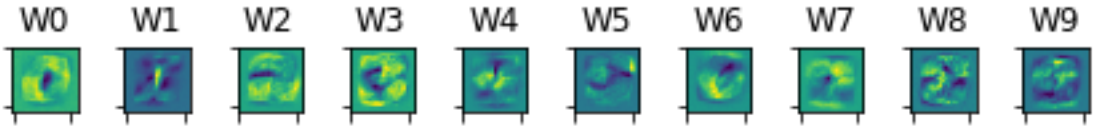

Figure 3: Part 4, Weights For Each Digit

Part 5

We can see from the learning curve below, that using momentum we reach the same performance but with only 300 iterations vs. 1000 iteration when running the gradient descent without momentum. Note that all other parameters are the same for both runs.

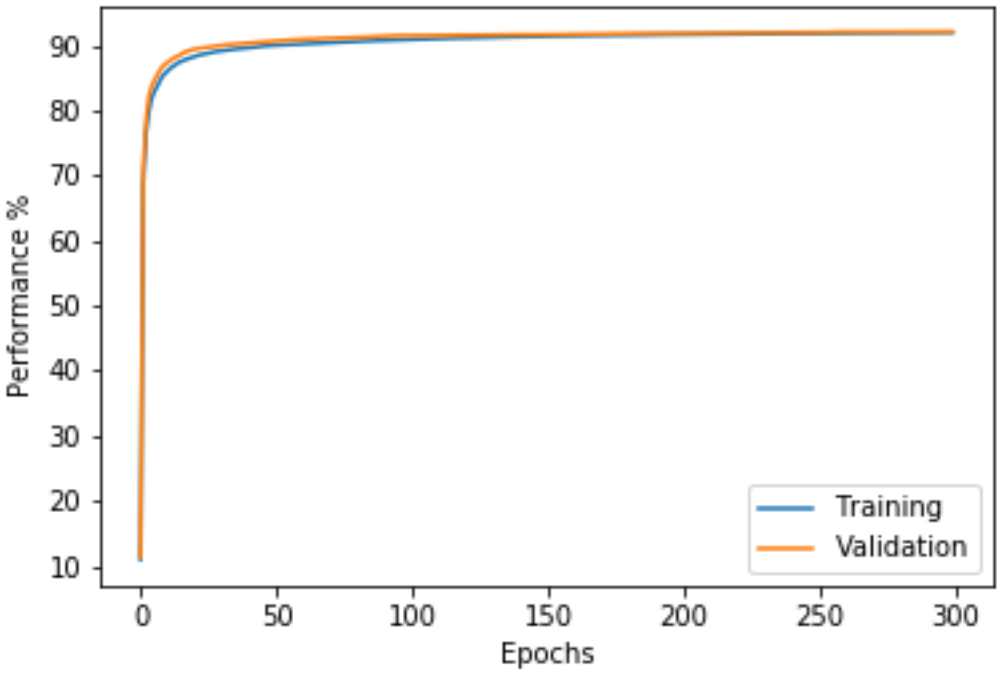

Figure 4: Learning Curve With Momentum

The same function was used for both part4 and part5. The function has an argument "momentum" that enables momentum gradient descent. Note: the function updates the biases as well, as the profs wanted (based on a Piazza post).

```
def gradient_descent(x, y, validation_x,validation_y, init_W, b0, alpha, max_iter = 2500, r
    EPS = 1e-5  # EPS = 10**(-5)
    prev_W = init_W - 10 * EPS
    W0 = init_W.copy()

    if momentum != 0:
        v = 1
        vb = 1
        name = "part5"
    else:
        name = "part4"

    results = {}
    iter = 0
    while norm(W0 - prev_W) > EPS and iter < max_iter:
        prev_W = W0.copy()
```

```
        c0 = cost(x, W0, b0, y)
        results[iter] = [get_performance(x, y, W0, b0), get_performance(validation_x, valid
        new_W = dw_cost(x, W0, b0, y)
        new_b = db_cost(x, W0, b0, y)
        if momentum != 0:
            v = momentum * v + (alpha * new_W)
            vb = momentum * vb + (alpha * new_b)
            W0 -= v
            b0 -= vb
        else:
            W0 -= (alpha * new_W)
            b0 -= (alpha * new_b)
        if iter % 100 == 0:
            print "Iter", iter
            print "cost = ", c0
        iter += 1
    print "Iter", iter
    print "cost = ", c0
    learning_curve(results, name)
    if momentum == 0:
        W_images(W0, name)
    return W0
```

Part 6

For parts A,B and C they are all plotted on the figure below:

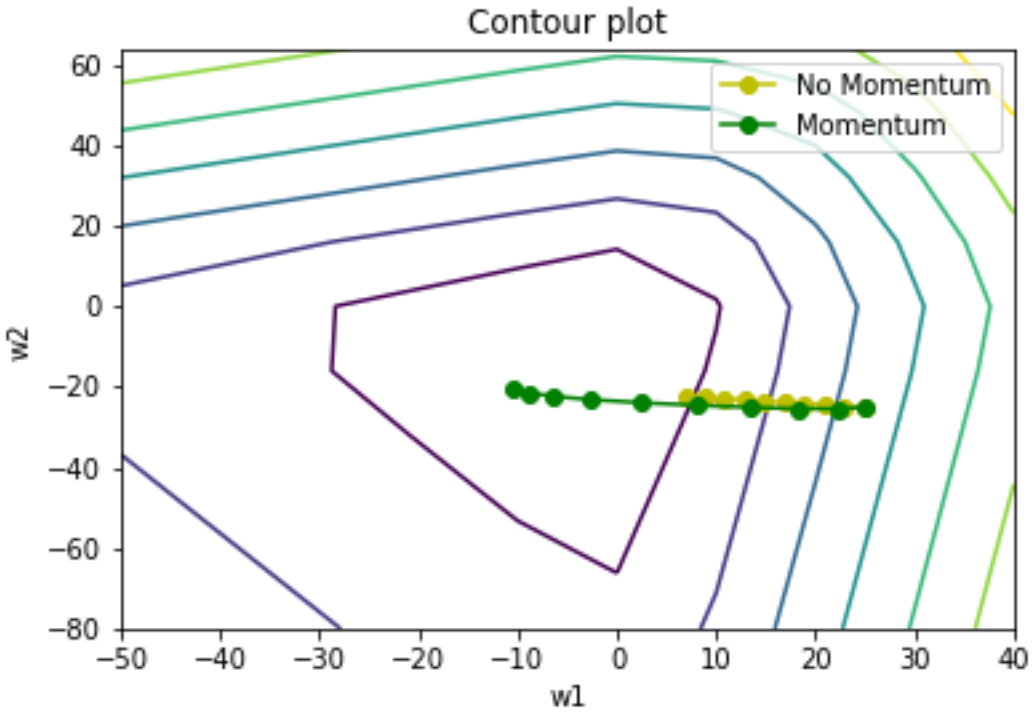

Figure 5: Contour Plot

D):
Both trajectories on the contour plot were run with the same parameters, and we can see that momentum trajectory reaches the desired performance faster than no momentum trajectory. The plot displays the benefits of momentum in decreasing required time to reach the minimum (this can also be seen from the learning curves in part4 and part5).

E):
For this part different values for w1, w2 were attempted and based on the contour plots I selected the best values for w1, w2. I noticed values that are away from the edges provided better plots, while those close to the edge didn't produce a contour sometime (result in warning:"No contour levels were found"). I Also experimented with the range of values to replace w1, w2 with. For bigger ranges I noticed the trajectories were to small to be easily distinguishable, for smaller ranges the trajectories will be out of scope.

Part 7

For this part we assume we have N hidden layers without the input or output layer. If we were to calculate the gradient with respect each unit without caching: We have N layers, each with K units.
For the top layer for each unit we need to calculate the gradient with respect to K units in each layer underneath. Thus N * K for each unit in the top layer, therefore we will have N * K * K for the top layer.
For the second layer from the top, for each unit (N-1) * K, Thus for all the layer we do (N-1) * K * K.

Thus for all the layers: $N * K^2 + (N-1) * K^2 + ... + 1 * K^2 \implies K^2 * (N + (N-1) + ... + 1) \implies K^2 * \frac{N*(N-1)}{2} \implies O(K^2 * N^2)$

For Back propagation, we have a matrix of size K * K, we need to multiply it by a matrix of size K * K we then multiply the result with a matrix of size 1 * K. this is done for each layer.
if $K > n$ then total complexity is: $O(n * K^2)$.

Part 8

Performance on test set: %0.833333333333

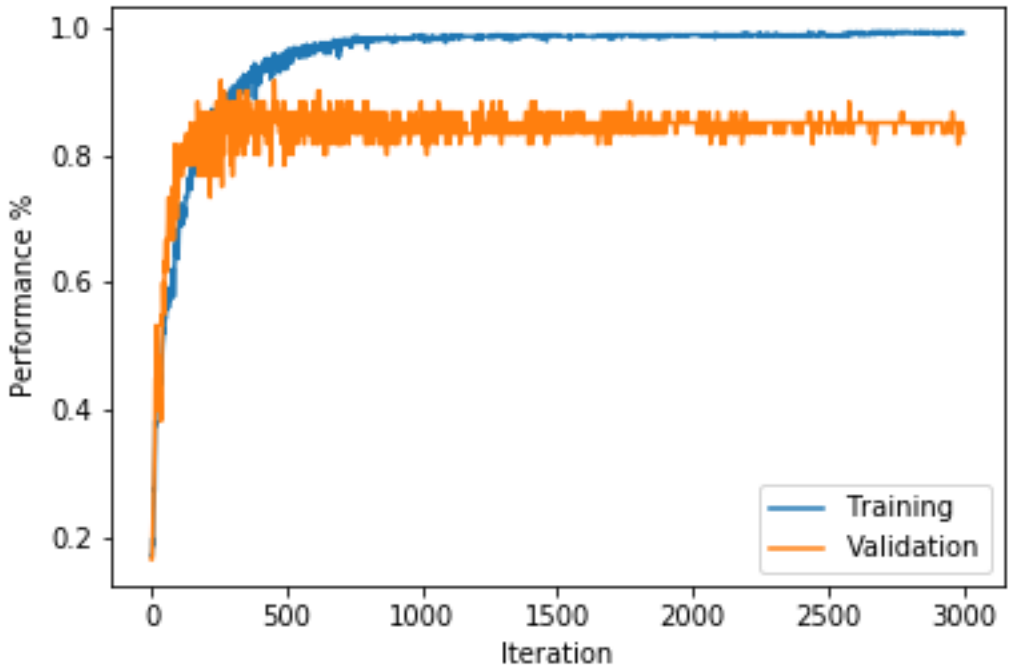

Figure 6: Learning Curve

For this neural network, I used gray images (the same used for Project 1) of the size 32×32. Thus the input layer was of size 32×32. For the hidden layer I used 12 units and the output is 6 units. I noticed that increasing the number of hidden units sometimes slightly improve the performance (depending on the learning rate) but also increased computation time. I decided to choose 12 hidden units as it provided the highest performance and the computation ran quickly. The weights were initialized by the default pytorch

module. I used ReLU activation as it gave me slightly better results than Tanh. For the batch size, 16 image per actor produced the best performance.

Part 9

For this part I choose baldwin and bracco, I performed a forward pass with the test data for each actor (20 images per actor) then I ran the loss function and performed back propagation. Looking at the gradient of the weights connecting the hidden units to the output layer for baldwin or bracco (connecting to output units: 0 for baldwin and 3 for bracco), if the gradient is not zero that means the hidden unit is useful in classifying the actor. Below are the visualizations of the hidden units

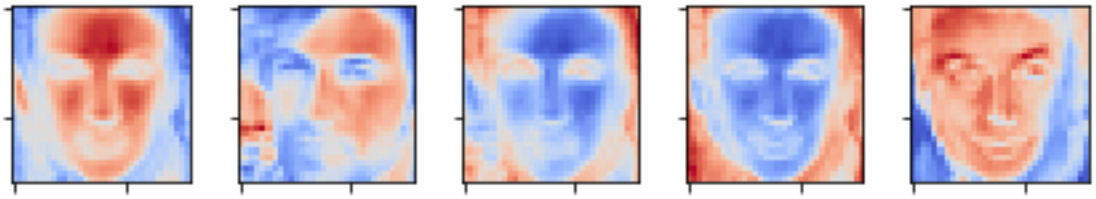

Figure 7: baldwin weights

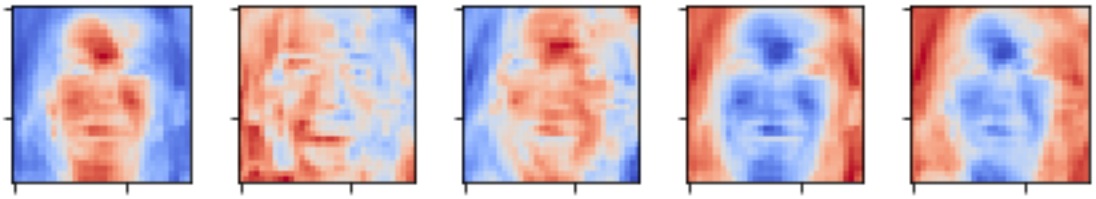

Figure 8: bracco weights

Part 10

In order to extract the activations I changed myalexnet.py file to only run the network up until Conv4 layer, this was done by removing the classifier method call in the forward function. The output is then used as an input in a neural network similar to the one built in part 8. The input is images of size 9216, that are connected to 64 hidden unit, and then 6 output units. I also attempted to extract a different layer , layer Conv3, by removing the computation that followed from the model, but I got worse performance then when the same training was done on Conv4 activations (validation set performance $< 45\%$). I tried using a ReLU activation function, but noticed that Tanh provided a higher performance with the same settings.

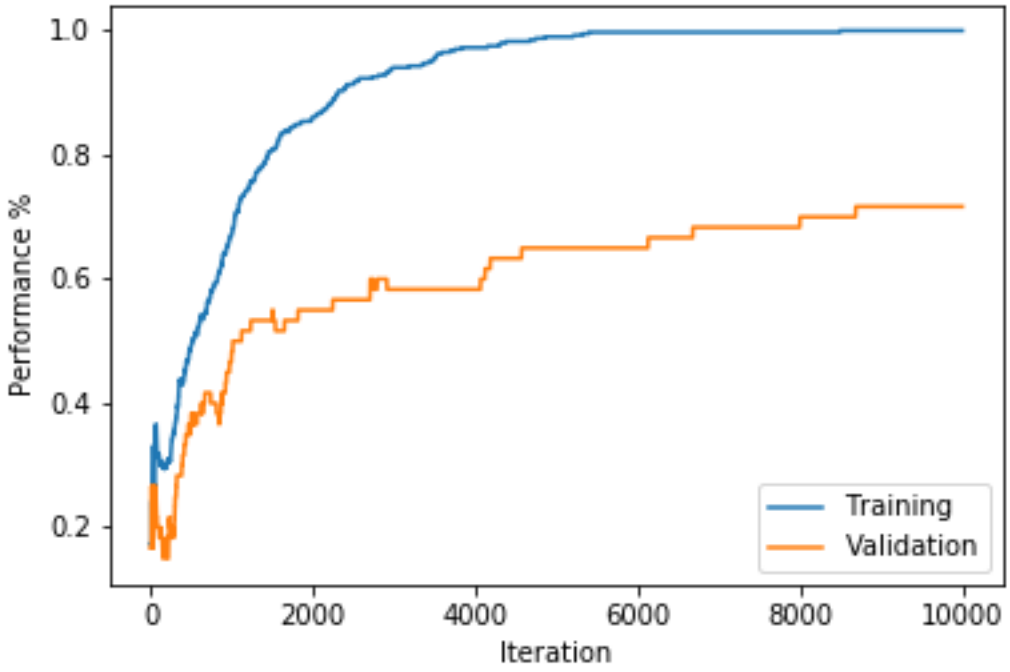

Figure 9: Learning Curve