CSC411: Assignment #2

Due on Friday, February 23, 2018

Yifei Dong, Zifei Han

February 21, 2018

Part 1

Dataset description

The MNIST dataset contains pictures of digits from 0 to 9. The dataset is separated into training set and testing set. The following are 10 sample images from each number from the training set. In order to make the sample unbiased, we picked 10 random images from each number.

Observing the examples, we noticed that some numbers are written in different forms. For example, in Figure 8, one image of number 7 has an across in the middle, but all other images of number 7 does not have the across in middle. Another thing we noticed is that the numbers have slightly different rotations. It is obvious in Figure 2, the images of number 1.

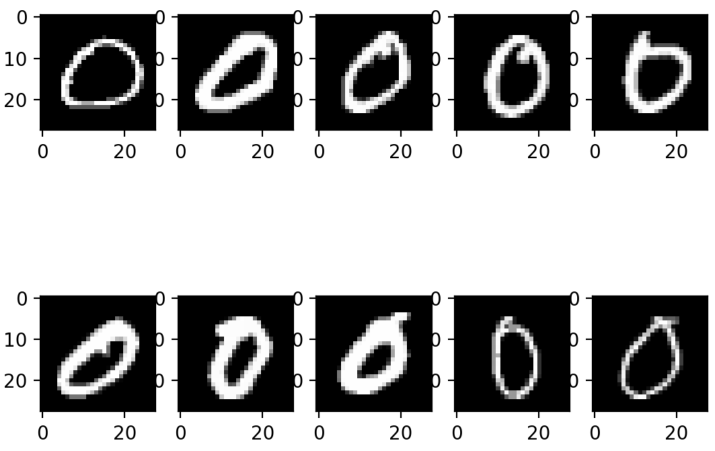


Figure 1: Number 0

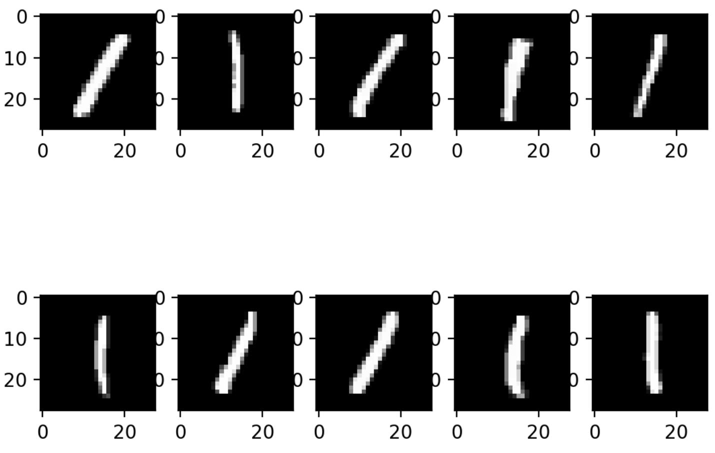


Figure 2: Number 1

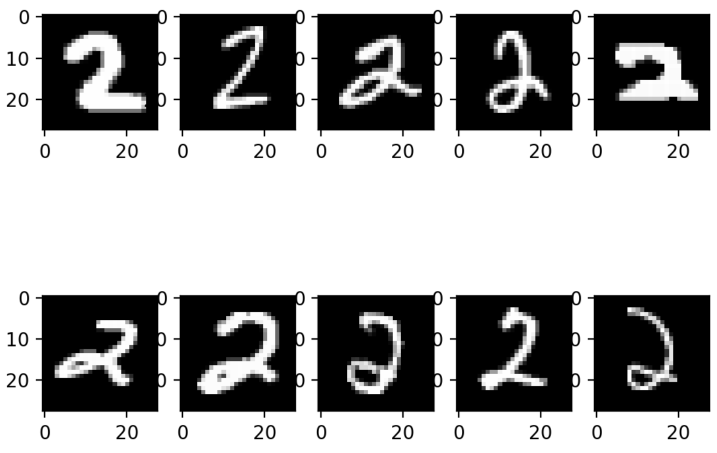


Figure 3: Number 2

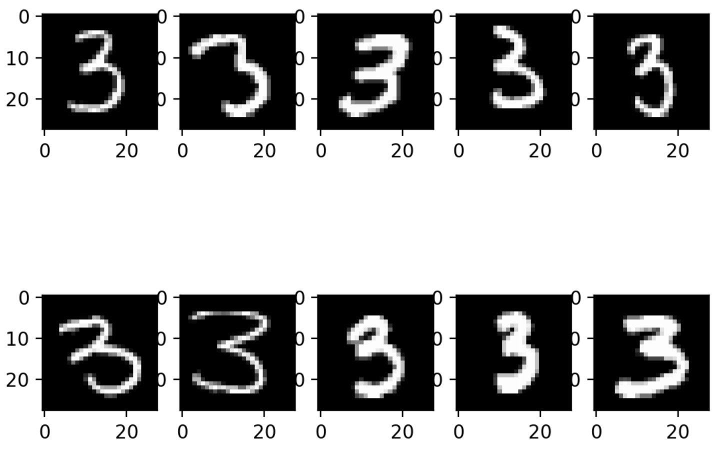


Figure 4: Number 3

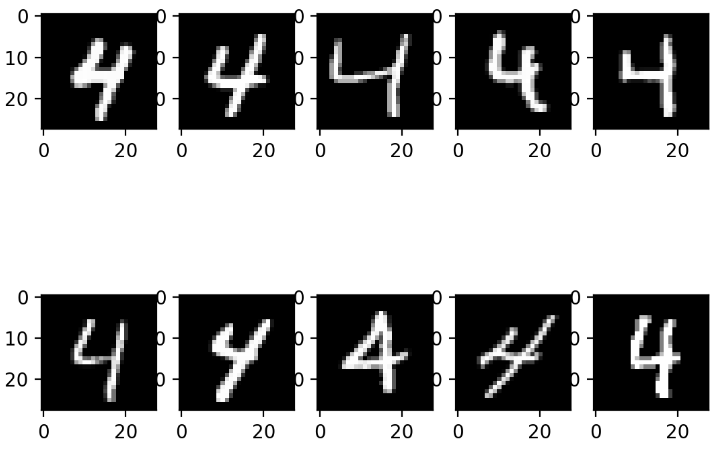


Figure 5: Number 4

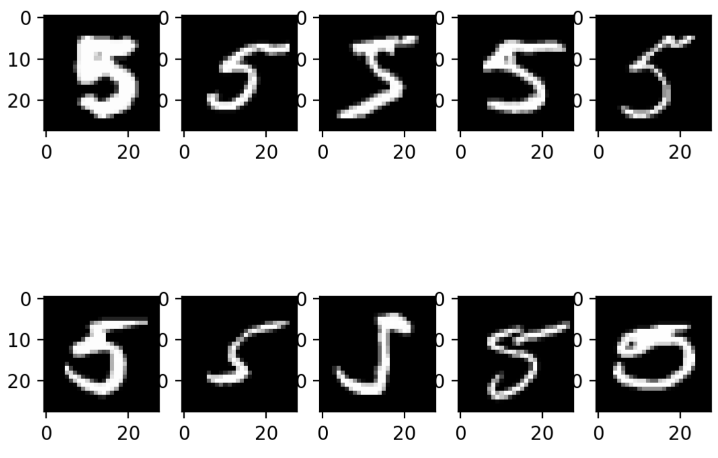


Figure 6: Number 5

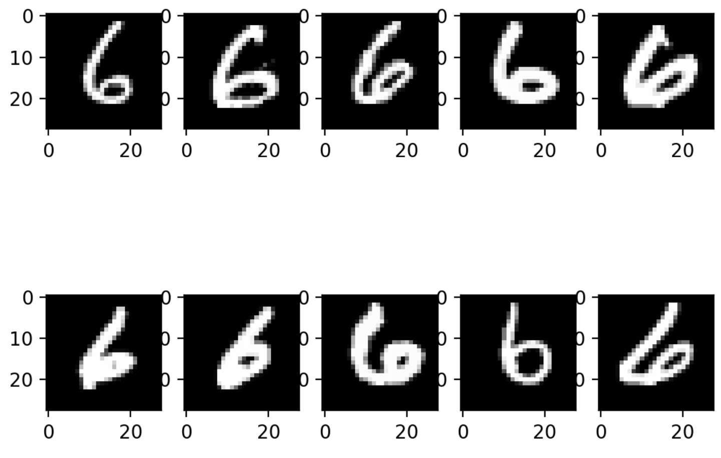


Figure 7: Number 6

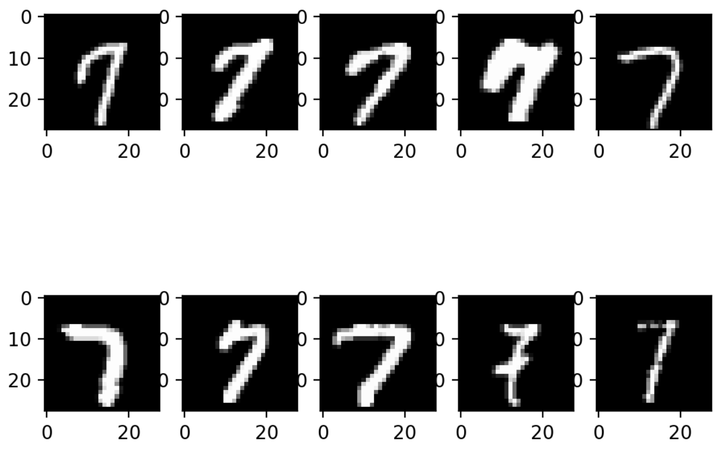


Figure 8: Number 7

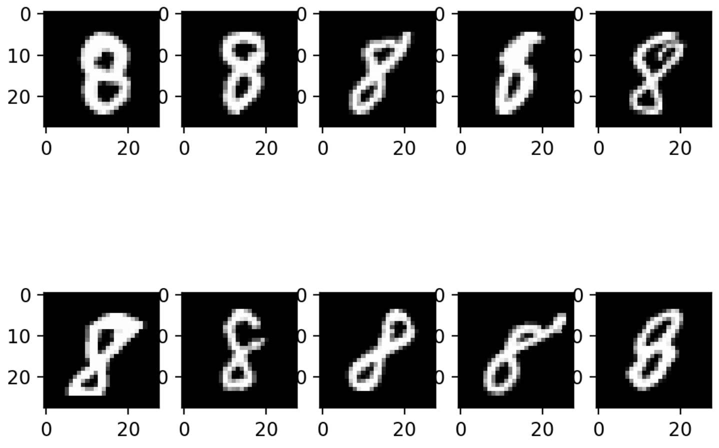


Figure 9: Number 8

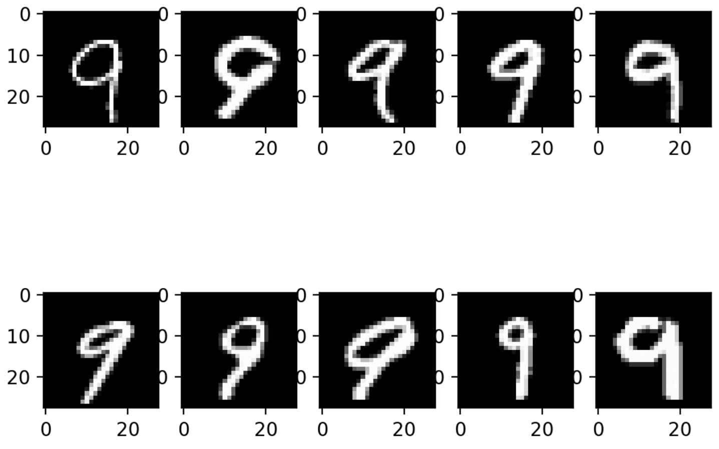


Figure 10: Number 9

Part 2

Implement a function that computes the network in handout using NumPy

```
def calculate_output(X, W):
    X = np.vstack( (ones((1, X.shape[1])), X))
    output = dot(W.T, X)
    return output

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 part_2(X, W):
    y = calculate_output(X, W)
    result = softmax(y)
    return result
```

Since the network does not have a hidden layer and the activation function in the output layer is the identity, the computation of $o_i = \sum_j w_{ji}x_j + b_i$ was vectorized so that the outputs of all images could be done in one matrix multiplication.

Part 3

Part A

Compute the gradient of the cost function C.
let w represent the weight. i represents the unit x_i and j represents the jth output of the network.

$$\frac{\partial C}{\partial w_{ij}} = \sum_k \left(\frac{\partial C}{\partial o_j} \frac{\partial o_j}{\partial w_j} \right) \quad \text{where } C = -\Sigma_j y_j \log p_j$$

$$p_i = \frac{e^{o_i}}{\Sigma_j e^{o_j}}$$

compute $\frac{\partial C}{\partial o_j}$ first:

$$\frac{\partial C}{\partial o_j} = \frac{\partial C}{\partial p_k} \frac{\partial p_k}{\partial o_j}$$

$$= \frac{\partial}{\partial p_k}\left(-\Sigma_j y_j \log p_j\right) \frac{\partial}{\partial o_j} \frac{e^{o_k}}{\Sigma_{j'} e^{o_{j'}}}$$

$$= \begin{cases} -\frac{y_k}{p_k} \cdot p_j(1-p_j) & \text{when } k=j \\ -\frac{y_k}{p_k} \cdot -p_k p_j & \text{otherwise} \end{cases}$$

$$\frac{\partial o_j}{\partial w_{ij}} = \left(-\frac{y_0}{p_0}(-p_0 p_j) - \frac{y_1}{p_1}(-p_1 p_j) - \dots - \frac{y_j}{p_j}(-p_j p_j) - \dots - \frac{y_9}{p_9}(-p_9 p_j) \right) x_i$$

there are 10 digits.

$$= (p_j(y_1 + \cdots + y_9) - y_j) x_i$$

$$= (p_j - y_j) x_i \quad \text{\# } y_1 + \cdots + y_9 = 1 \text{ b/c one-hot encoding}$$

$$\frac{\partial C}{\partial w_{ij}} = \sum_k \left(\frac{\partial C}{\partial o_j} \frac{\partial o_j}{\partial w_j} \right) = \sum_{k=1}^{m} (p_j^{(k)} - y_j^{(k)}) x_i^{(k)}$$

Nealy Hand written above.

Part B

vectorized code that computes the gradient of the cost function with respect to the weights and biases of the network:

```
def f_p3(x, y, w):
    """
    Use the sum of the negative log-probabilities of all the training cases
    as the cost function.
    """
    return -sum(y * log(part_2(x, w)))

def df_p3(x, y, w):
    p = part_2(x, w)
    x = np.vstack( (ones((1, x.shape[1])), x))
    return dot(x, (p - y).T)
```

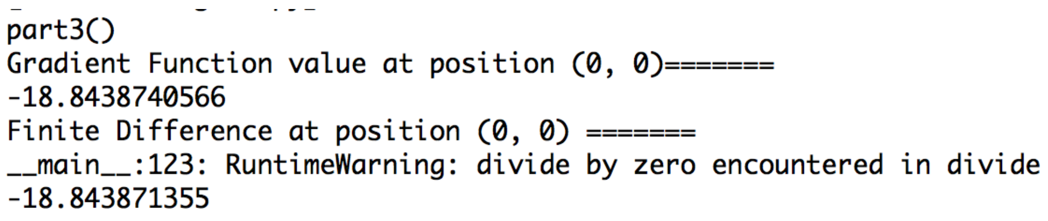
Check that the gradient was computed correctly by approximating the gradient at several coordinates using finite differences:

```
def part3():
    random.seed(0)
    x = reshape(random.rand(784 * 20), (784, 20))
    y = zeros((10, 20))
    y[0, :] = 1
    w = reshape(random.rand(785 * 10), (785, 10))

    h = zeros((785, 10))
    h[0, 0] = 1e-5

    print("Gradient Function value at position (0, 0)=======")
    print(str(df_p3(x, y, w)[0][0]))
    print("Finite Difference at position (0, 0) =======")
    print(str(((f_p3(x, y, w+h) - f_p3(x, y, w))/(h))[0][0]))
```

The result of above code is in the picture as follows:



```
part3()
Gradient Function value at position (0, 0)=======
-18.8438740566
Finite Difference at position (0, 0) =======
__main__:123: RuntimeWarning: divide by zero encountered in divide
-18.843871355
```

Figure 11

Conclusion: The above result shows that there are little difference between computation of gradient using finite different and computation of gradient funciton.

Part 4

Part A

To train neural network using gradient descent without momentum.
1. The optimization procedure are as follows:

- *init_t* - initialized weights using all zeros of vector.
- α - tried different alphas including 0.00001, 0.001, 0.0001
  selected smaller alpha learning rates the smaller one was chosen for smaller rates
  0.001 has accuracy rate of: 90% long time to run potential overfitting
  0.0001 has accuracy rate of: 90%
  0.00001 has accuracy rate of: 86%
  0.0000001 has accuracy rate of: 77%
  so an alpha was choosen to be 0.00001.
- ϵ - EPS used as 1e-5
- max iteration is chosen to be 30000

2. Plot learning curves

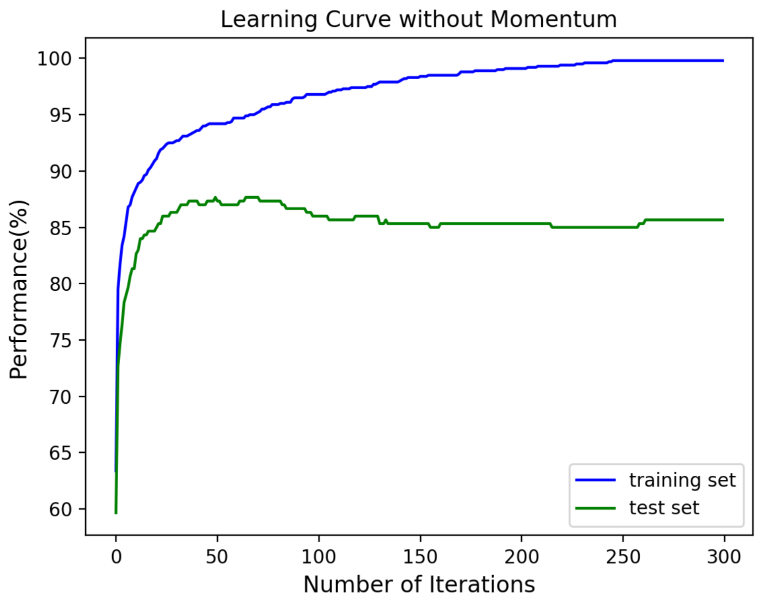


Figure 12

Part B

Display weights going into each output units as figure below:

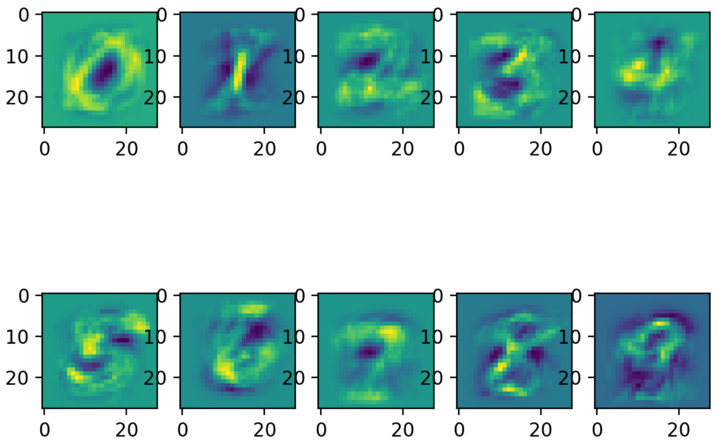
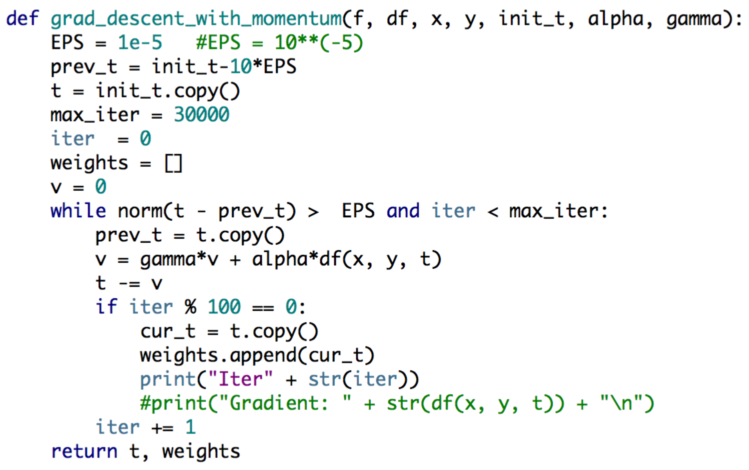


Figure 13: Number 9

Part 5

1. Train the dataset with momentum.

The dataset is trained with the new gradient descent function below:



```
def grad_descent_with_momentum(f, df, x, y, init_t, alpha, gamma):
    EPS = 1e-5   #EPS = 10**(-5)
    prev_t = init_t-10*EPS
    t = init_t.copy()
    max_iter = 30000
    iter  = 0
    weights = []
    v = 0
    while norm(t - prev_t) >  EPS and iter < max_iter:
        prev_t = t.copy()
        v = gamma*v + alpha*df(x, y, t)
        t -= v
        if iter % 100 == 0:
            cur_t = t.copy()
            weights.append(cur_t)
            print("Iter" + str(iter))
            #print("Gradient: " + str(df(x, y, t)) + "\n")
        iter += 1
    return t, weights
```

Figure 14

The gradient descent function is implemented with additional weight update function, which is
v $\leftarrow \gamma$*v + α*df
weight $\leftarrow$ weight - v
where,

- v is initialized as zero
- the initial weight is set as a vector of zero
- momentum gamma is used as 0.99 and 0.9

2. Plot the learning curves of the function with momentum.

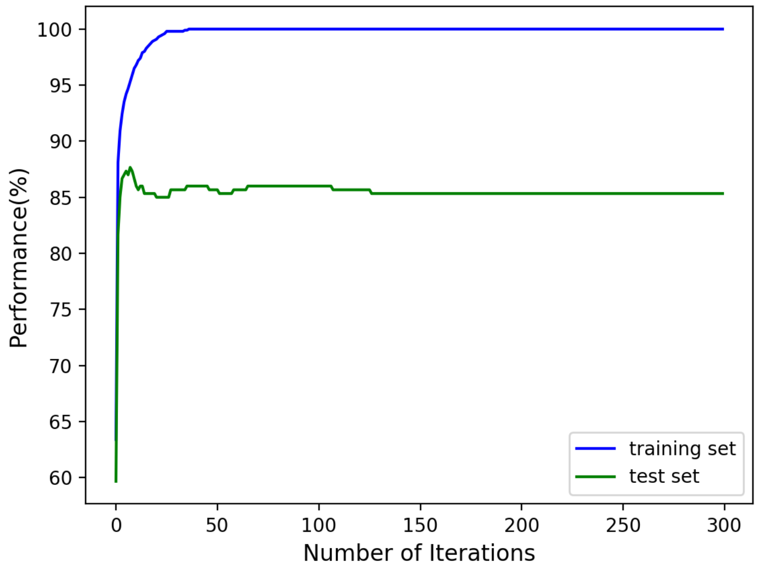


Figure 15: learning curve with momentum trained

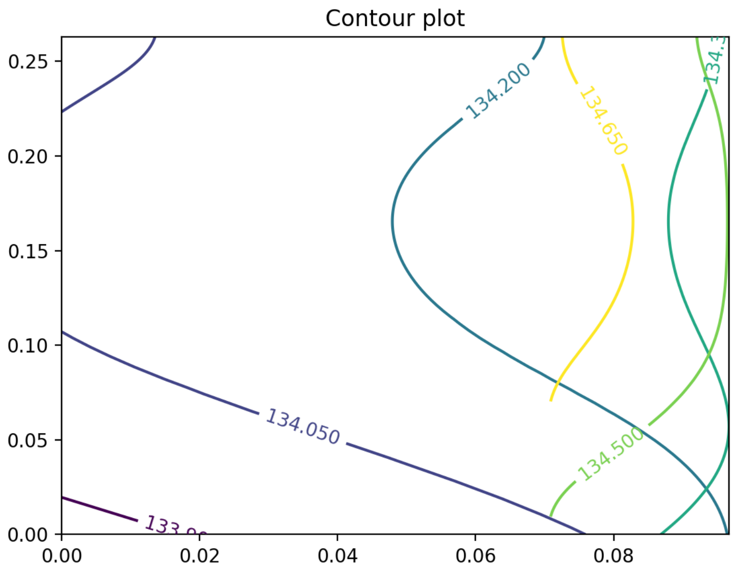
3. There are two major difference we spotted through comparing the performance with and without momentum.

- the accuracy reaches steady faster with fewer iterations with momentum
- the performance is better (higher accuracy rate) generally with momentum

Therefore, I can conclude that momentum is a better training algorithm with this training set.

Part 6

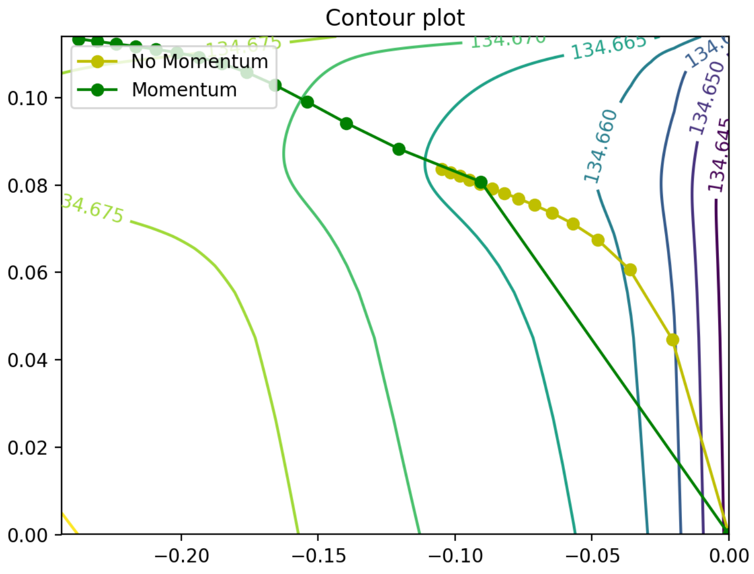
a)



w1 = 378, w2 = 322

378 = 28*13 + 14 and 322 = 28 * 11 + 14 are chosen for digit 5. (why we chose w1 and w2 like this are explained in part e.

b,c)



w1 = 377, w2 = 325

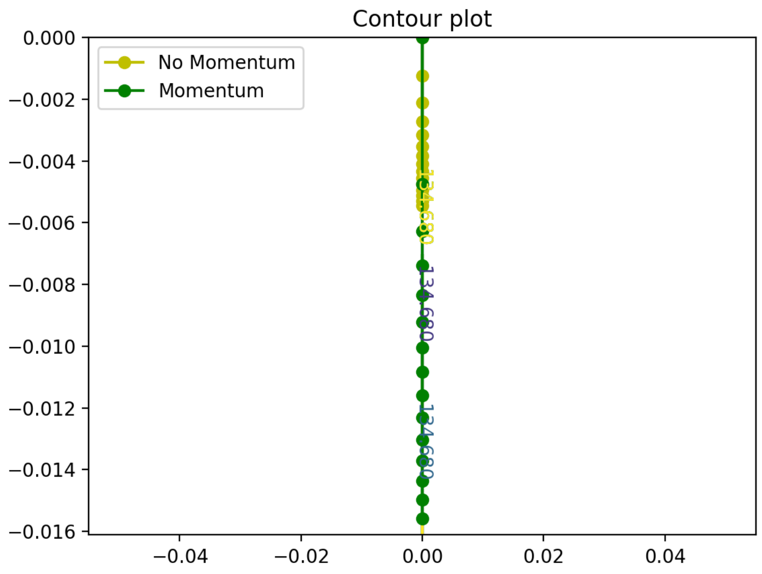
378 = 28*13 + 13 and 322 = 28 * 11 + 17 are chosen for digit 1.

d)

The trajectory of the gradient descent with momentum performs better than the one without momentum. The gradient consistently points into one direction, and the one with momentum goes faster in the direction.

This is because a weight $\gamma * v$ is added to the theta every iteration. Instead of updating theta as $\theta = \theta - \alpha \frac{\partial c}{\partial W}$, the gradient descent with momentum is updating theta as $\theta = \theta - (\gamma * v + \alpha \frac{\partial c}{\partial W})$.

e)
I found the appropriate w_1 and w_2 by trying to make then one of the center pixels of an image. For the handwritten digits, the numbers are in the middle of the image most of the time. Therefore, w_1 and w_2 should be one of the pixels within the number to converge properly. Since the images are 28×28, I found w_1 and w_2 by calculating $28 * a + b$, where a, b can be any number between 10 - 18.
For settings that do not demonstrate the benefits using momentum, we made $w_1 = 60$ and $w_2 = 577$. The following is the graph of the counter plot with momentum, the trajectory with momentum and the trajectory without momentum.



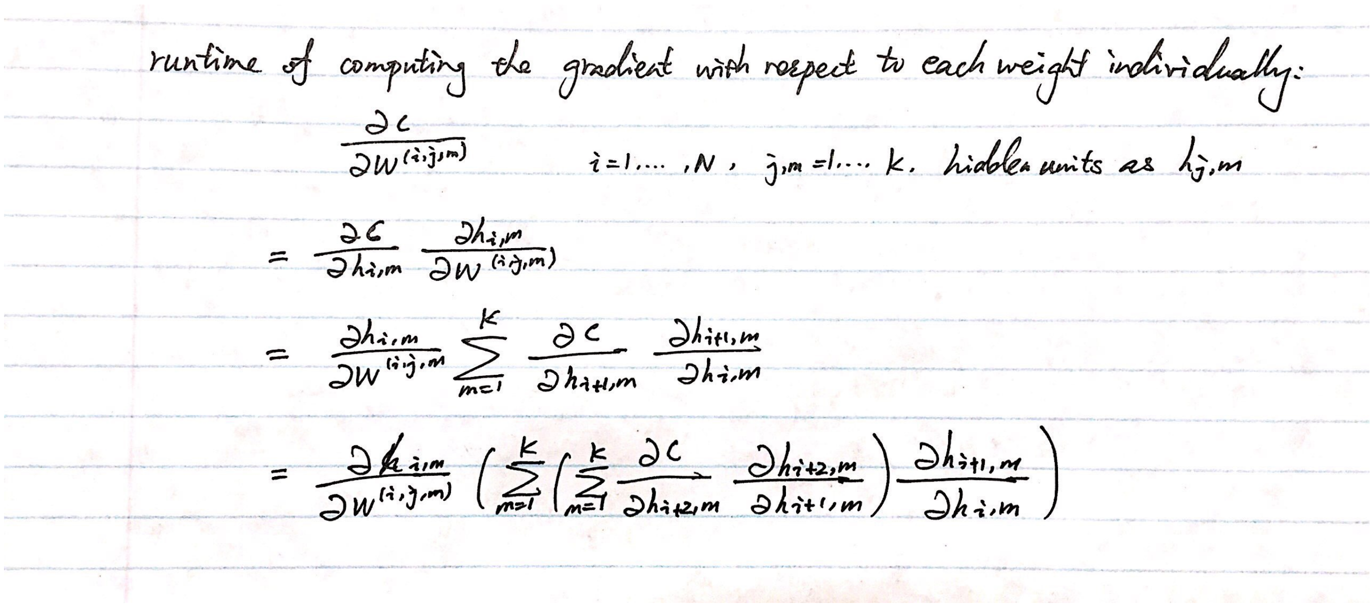
Both w_1 and w_2 are pixels at the side of an image.
Therefore, the settings from part c and d work for producing a good visualization, while the ones we used in this part do not.

Part 7

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.

Solution:
We computed the runtime in two cases separately to do the comparison.

1. Runtime of computing the gradient with respect to each weight individually:



runtime of computing the gradient with respect to each weight individually:

$\frac{\partial C}{\partial W^{(i,j,m)}}$ $i=1,\dots,N$, $j,m=1,\dots,K$. hidden units as $h_{j,m}$

$$= \frac{\partial C}{\partial h_{i,m}} \frac{\partial h_{i,m}}{\partial W^{(i,j,m)}}$$

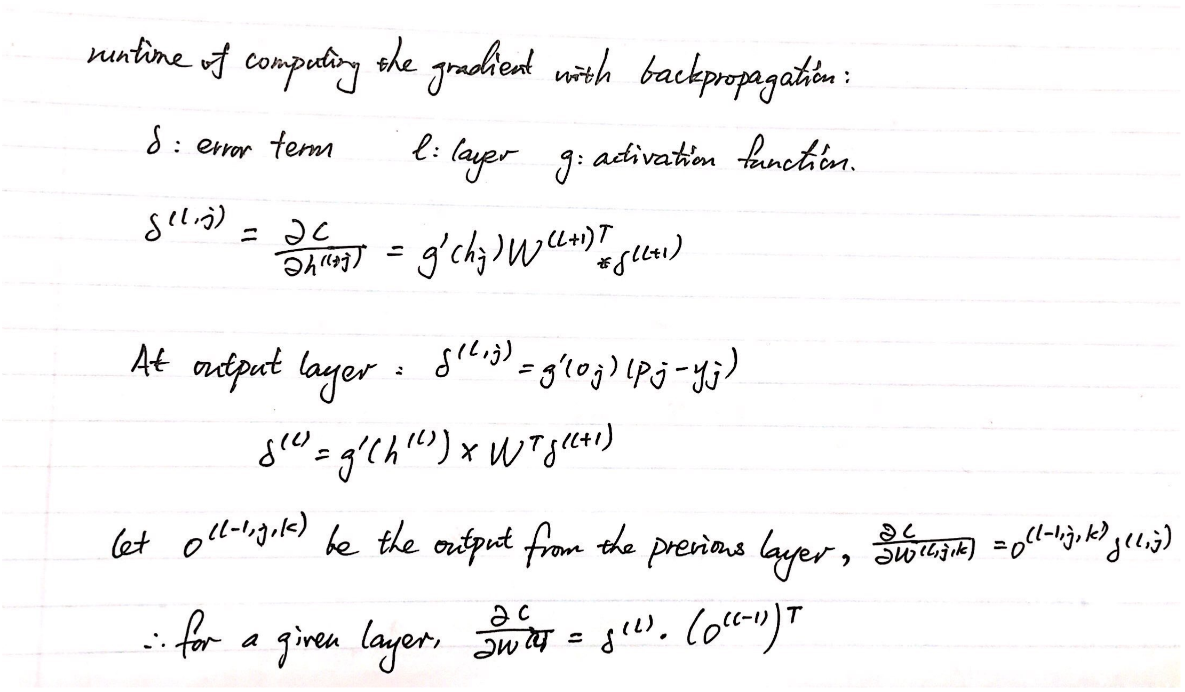
$$= \frac{\partial h_{i,m}}{\partial W^{(i,j,m)}} \sum_{m=1}^{K} \frac{\partial C}{\partial h_{i+1,m}} \frac{\partial h_{i+1,m}}{\partial h_{i,m}}$$

$$= \frac{\partial h_{i,m}}{\partial W^{(i,j,m)}} \left(\sum_{m=1}^{K} \left(\sum_{m=1}^{K} \frac{\partial C}{\partial h_{i+2,m}} \frac{\partial h_{i+2,m}}{\partial h_{i+1,m}} \right) \frac{\partial h_{i+1,m}}{\partial h_{i,m}} \right)$$

Figure 16

When we only have two partial terms, the run time is $O(2)$. For N-1 layer, it will be $O(K-1)$. Suppose we have N layers, the runtime will be $O(K^N)$, which is observed by the nested summation.

2. Run-time of computing the gradient with back propagation:



runtime of computing the gradient with backpropagation:

δ : error term l: layer g: activation function.

$$\delta^{(l,j)} = \frac{\partial C}{\partial h^{(l,j)}} = g'(h_j) W^{(l+1)T} * \delta^{(l+1)}$$

At output layer : $\delta^{(l,j)} = g'(o_j)(p_j - y_j)$

$$\delta^{(l)} = g'(h^{(l)}) \times W^T \delta^{(l+1)}$$

let $o^{(l-1,j,k)}$ be the output from the previous layer, $\frac{\partial C}{\partial W^{(l,j,k)}} = o^{(l-1,j,k)} \delta^{(l,j)}$

$\therefore$ for a given layer, $\frac{\partial C}{\partial W^{(l)}} = \delta^{(l)} \cdot (o^{(l-1)})^T$

The last function is the gradient of a cost function for weight matrix of size $k * k$. From the above equations, we know the runtimw will be $O(k^3)$.

Therefore, we know that computing the gradient using backpropagation is faster when number of layers is greater than 3.

Part 8

Section A - Set up data

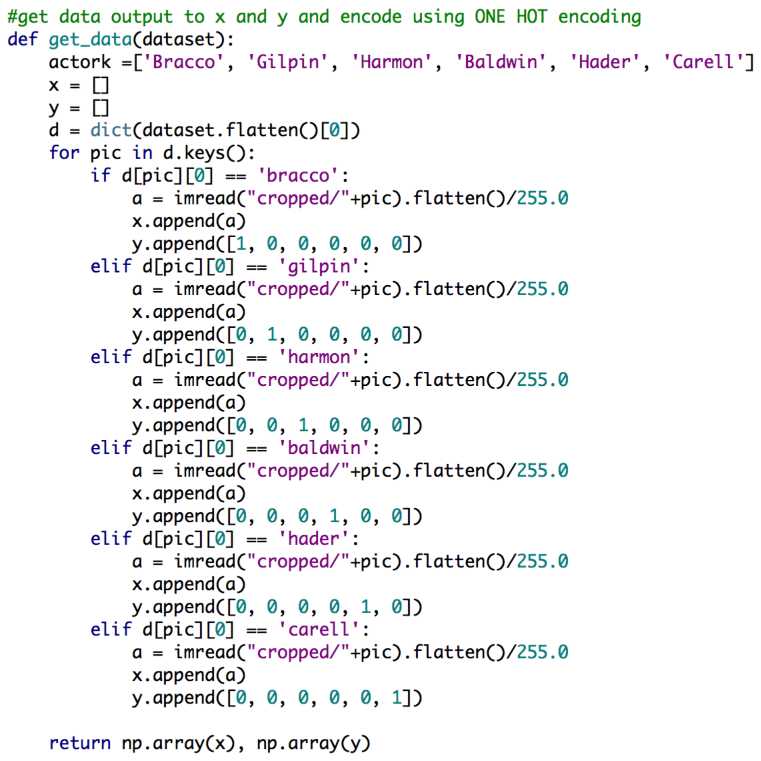
- x - is carved face compiled to be grey-scaled with size (32*32). Then it is transformed to be a vector of size 2014.
- y - is encoded to a vector of size 6 using one-hot coding. The output value are assigned as follows:

```
'bracco':  [1,0,0,0,0,0]
'gilpin':  [0,1,0,0,0,0]
'harmon':  [0,0,1,0,0,0]
'baldwin': [0,0,0,1,0,0]
'hader':   [0,0,0,0,1,0]
'carell':  [0,0,0,0,0,1]
```

- size of each set are generated as below:

```
training set: each actor has a size of 70 images, except "gilpin" has size 56 becaus
validation set: each actor has a size of 10 images
tet set: each actor has size of 20
```

Data formation is as follows:



```python
#get data output to x and y and encode using ONE HOT encoding
def get_data(dataset):
    actork =['Bracco', 'Gilpin', 'Harmon', 'Baldwin', 'Hader', 'Carell']
    x = []
    y = []
    d = dict(dataset.flatten()[0])
    for pic in d.keys():
        if d[pic][0] == 'bracco':
            a = imread("cropped/"+pic).flatten()/255.0
            x.append(a)
            y.append([1, 0, 0, 0, 0, 0])
        elif d[pic][0] == 'gilpin':
            a = imread("cropped/"+pic).flatten()/255.0
            x.append(a)
            y.append([0, 1, 0, 0, 0, 0])
        elif d[pic][0] == 'harmon':
            a = imread("cropped/"+pic).flatten()/255.0
            x.append(a)
            y.append([0, 0, 1, 0, 0, 0])
        elif d[pic][0] == 'baldwin':
            a = imread("cropped/"+pic).flatten()/255.0
            x.append(a)
            y.append([0, 0, 0, 1, 0, 0])
        elif d[pic][0] == 'hader':
            a = imread("cropped/"+pic).flatten()/255.0
            x.append(a)
            y.append([0, 0, 0, 0, 1, 0])
        elif d[pic][0] == 'carell':
            a = imread("cropped/"+pic).flatten()/255.0
            x.append(a)
            y.append([0, 0, 0, 0, 0, 1])

    return np.array(x), np.array(y)
```

Figure 17: X and Y formation Code

Section B - Algorithm Architecture

- x - is wrapped in variable object of float tensor
- y - is wrapped in variable object of long tensor
- initial weights - used pytorch nn model package's default initial weight
- activation function - I used ReLU function for the hidden layer. x and y layer is linear.
- cost function - is the loss function was initially used as CrossEntropyLoss
- gradient descent - was used without momentum
- max iteration - of 10000
- alpha - learning rate of 1e-2
- optimizer - used Adam optimizer

The Data reformat to pytorch and subset is as below:

```
dtype_float = torch.FloatTensor
dtype_long = torch.LongTensor

#formulate dataset
train_x, train_y = get_data(train)
test_x, test_y = get_data(test)
validate_x, validate_y = get_data(validate)

#wrap datasets in pytorch variable object
x = Variable(torch.from_numpy(train_x)).type(dtype_float)
y_classes = Variable(torch.from_numpy(np.argmax(train_y,1)), requires_grad=False).type(dtype_long)
x_test = Variable(torch.from_numpy(test_x), requires_grad=True).type(dtype_float)
x_validate = Variable(torch.from_numpy(validate_x), requires_grad=True).type(dtype_float)

#Subsample the training set for faster training - MINIBATCH
train_idx = np.random.permutation(range(train_x.shape[0]))[:300]
x = Variable(torch.from_numpy(train_x[train_idx]), requires_grad=False).type(dtype_float)
y_classes = Variable(torch.from_numpy(np.argmax(train_y[train_idx], 1)), requires_grad=False).type(dt
ycompare = train_y[train_idx]
```

The training code is as below:

```
#dimensions
dim_x = 1024
dim_h = 300
dim_out = 6

#pytorch nn model
model = torch.nn.Sequential(
    torch.nn.Linear(dim_x, dim_h),
    torch.nn.ReLU(),
    torch.nn.Linear(dim_h, dim_out),
)

#initial trial loss function
loss_fn = torch.nn.CrossEntropyLoss()

#train the algorithm to classify faces
learning_rate = 1e-2
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(10000):
    y_pred = model(x)
    loss = loss_fn(y_pred, y_classes)

    model.zero_grad()  # Zero out the previous gradient computation
    loss.backward()    # Compute the gradient
    optimizer.step()   # Use the gradient information to
                       # make a step
    if t%100 == 0:
        print("iteration--" + str(t))

#compute test data result
y_pred = model(x_test).data.numpy()
#get performance
print(np.mean(np.argmax(y_pred, 1) == np.argmax(test_y, 1)))
```

Section C - Experiment Settings

Experiment with different settings to obtain best performance. Tried different settings of (a) learning rate (b) hidden layer dimension (c) activation function (d) loss function

- hidden layer dimension - tried different values [20, 300, 600] resulted different accuracy rate
- alpha - tried different learning rate [0.001, 0.0001, 0.00001]
- batch size [100, 210, 300]
- tried different size and their different combinations to get the best result

Table 1: Different settings with results

Alpha	hidden layer dimension	batch size	accuracy rate
0.001	20	100	0.56
0.0001	20	100	0.7
0.00001	20	100	0.75
0.001	20	210	0.78
0.0001	20	210	0.83
0.00001	20	210	0.81
0.001	20	300	0.78
0.0001	20	300	0.85
0.00001	20	300	0.85
0.001	300	100	0.75
0.0001	300	100	0.7
0.00001	300	100	0.68
0.001	300	210	0.83
0.0001	300	210	0.78
0.00001	300	210	0.75
0.001	300	300	0.86
0.0001	300	300	0.85
0.00001	300	300	0.83
0.001	600	100	0.65
0.0001	600	100	0.76
0.00001	600	100	0.7
0.001	600	210	0.73
0.0001	600	210	0.75
0.00001	600	210	0.8
0.001	600	300	0.85
0.0001	600	300	0.83
0.00001	600	300	0.85

Section D Results

- learning curve

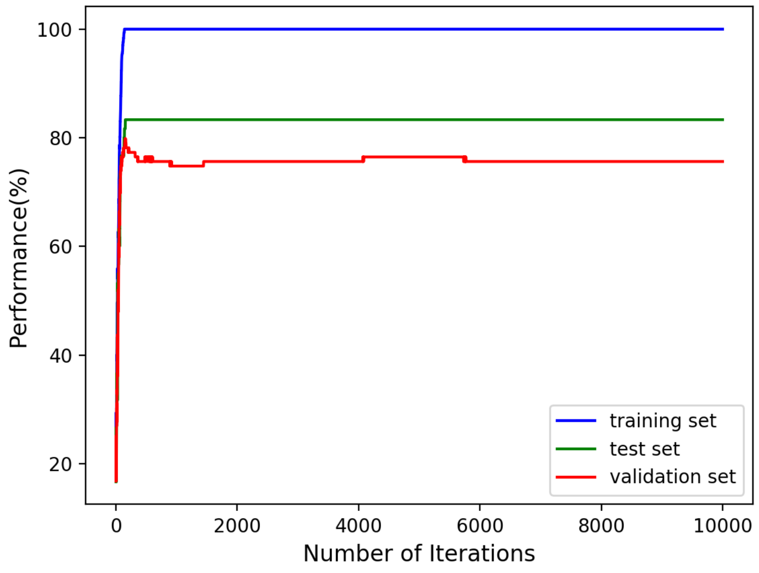


Figure 18: Learning curve

- performance classification: to conclude, according to table1 above we choose settings the settings with the highest accuracy rate which is (a) alpha=0.001 (b)hidden layer dimension=300 and (c) batch size=300 to obtain the accuracy rate of 86%.
  The accuracy rate may vary with different random batch selections. However, consider the run time 300 batch size and 300 hidden layer dimension may be a better choice than 600, because the algorithm will run faster.

Part 9

Selected two actors from index0 and index1 from one hot encoding. They are bracco and gilpin.

To find the weights of the hidden units that are useful for classifying input photos as those particular actors. I first subset the x's that predict these two actors. Then perform forward propagation using these two subsets to calculate the hidden weights of these subsets.
After that, I find the index of the largest hidden weights. Although they change for different x's. I selected the most frequent index that produced largest hidden weights. The code is shown as follows:

```
def draw_weights():
    y_pred = model(x_test).data.numpy()
    y_index = np.argmax(y_pred, 1)

    #select the index that predict actor1-gliphin and actor0-bracco
    #subset the x that predicted actor1 and actor0
    x1_index = []
    x0_index = []
    for i in range(len(y_index)):
        print(i)
        if y_index[i] == 1:
            x1_index.append(i)
        elif y_index[i] == 0:
            x0_index.append(i)

    x1 = x_test[x1_index]
    x0 = x_test[x0_index]

    #find the hidden layer using forward propagation
    h1 = dot(model[0].weight.data.numpy(), x1.data.numpy().T)
    h0 = dot(model[0].weight.data.numpy(), x0.data.numpy().T)

    #find the index larges value of the hidden layers
    argmax(h1.T[0])
    argmax(h1.T[1])
    argmax(h1.T[2])
    argmax(h1.T[3])
    argmax(h1.T[4])
    #We can see that for actor1 all of the max hidden layer value is at index 152 so
    plt.imshow(model[0].weight.data.numpy()[152, :].reshape((32, 32)), cmap=plt.cm.coolwarm)
    show()

    argmax(h0.T[0])
    argmax(h0.T[1])
    argmax(h0.T[2])
    argmax(h0.T[3])
    argmax(h0.T[4])
    #We can see that for actor5 all of the max hidden layer value is at index 10 so
    plt.imshow(model[0].weight.data.numpy()[10, :].reshape((32, 32)), cmap=plt.cm.coolwarm)
    show()
```

Two visualization of the selected weights are as follows:

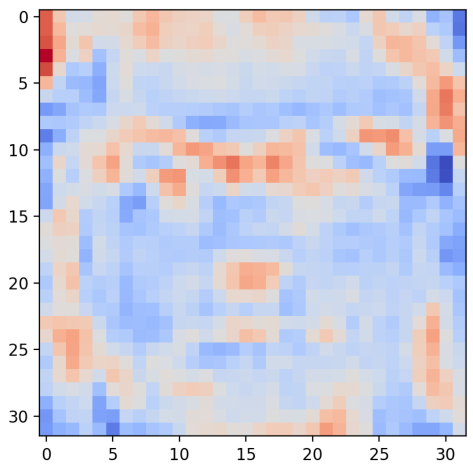


Figure 19: Weight visualization of Bracco

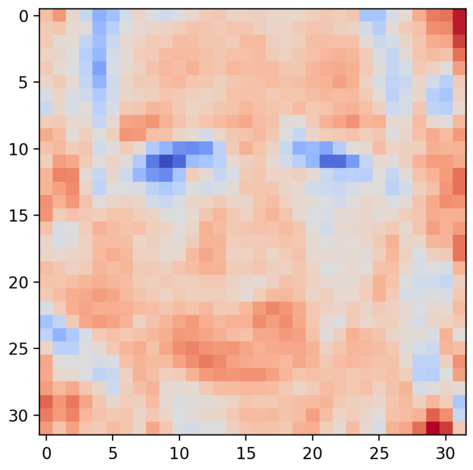


Figure 20: Weight visualization of Gilphin

Part 10

We modified the AlexNet code so that the forward function in AlexNet returns the activations. In the original AlexNet code, the model was splitted into two portions: features and classifier. The forward function ran foward pass on features first and then used the resulting activation in classifer. The output after features is the conv4 activations.

Below is how we modified the code:

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

We deleted the line "$x = self.classifier(x)$" which was originally in AlexNet class.

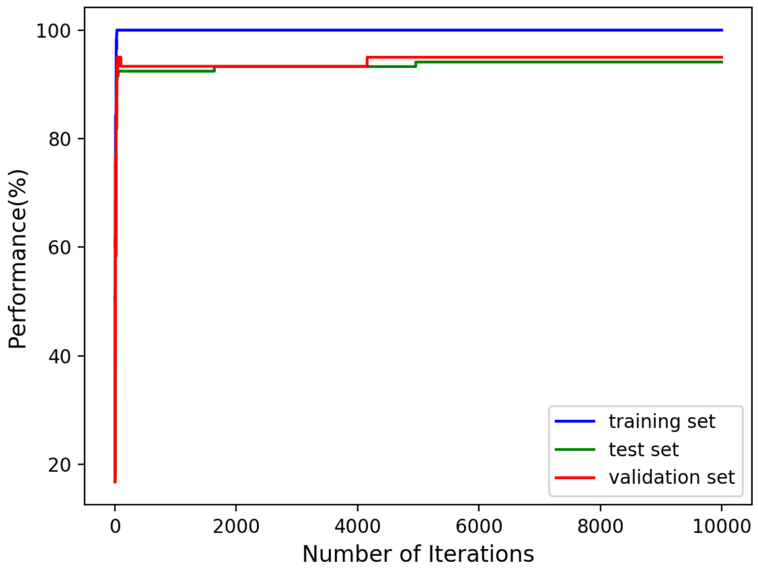
- x - is carved face with size (227*227*4). Then it is transformed to (227*227*3).
- y - is encoded to a vector of size 6 using one-hot coding. The output value are assigned as follows:

```
'bracco':  [1,0,0,0,0,0]
'gilpin':  [0,1,0,0,0,0]
'harmon':  [0,0,1,0,0,0]
'baldwin': [0,0,0,1,0,0]
'hader':   [0,0,0,0,1,0]
'carell':  [0,0,0,0,0,1]
```

- size of each set are generated as below:

```
training set: each actor has a size of 70 images, except "gilpin" has size 56 becaus
validation set: each actor has a size of 10 images
tet set: each actor has size of 20
```

To make better performance, we resized the cropped colour images into size $227 * 227 * 3$. The activation has size $n * (256 * 6 * 6) = n * 9216$ after flattened. n is the number of images.
We didn't use minibatch for this part. When we pass the activations into the pytorch model, we set alphas as 0.01 and hidden layer dimension as 30.
Below is the learning curve of training set, validation set and test set.



The performance of the training set is 100.0%
The performance of the validation set is 95.0%
The performance of the test set is 94.11764705882352%

By comparing the performances of part 8 and part 10, we could see that the performance of test set increased from 86% to 94%.