# CSC411: Assignment 2
## [*Environment*] py3.6 with random.seed(1)

Due on Sunday, Feburary 18, 2018

**Yining Lin, Yufeng Li**

August 25, 2019

# Part 1

*Dataset description*

I picked 10 pictures of each digit from MNIST dataset. For every single digit, there are multiple ways to write. For example, people can write 1 in both handwritten and printed style. This provides a more variant collection of digits.

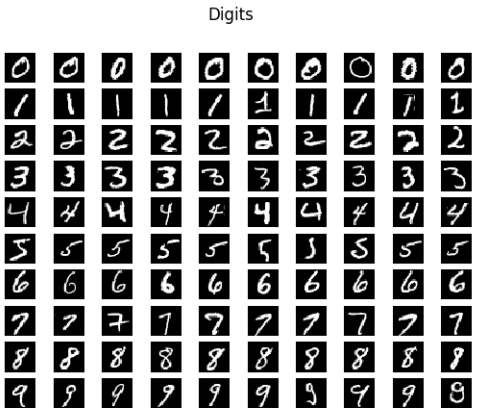



Figure 1: A selection of 10 picture of each digit from the MNIST

# Part 2

*Construct Neural network*

```
def softmax(y):
    return exp(y) / tile(sum(exp(y), 0), (len(y), 1))
def forward(x, w):
    L0 = dot(w.T, x)
    output = softmax(L0)
    return output
```

Suppose n is number of pictures, taking bias into account, x is 785*n input array, y is 10*n label array, w is 785*10 weight array.

# Part 3

*Construct Gradient Descend*

$$\frac{\delta C}{\delta w_{ji}} = \frac{\delta C}{\delta O_i} \cdot \frac{\delta O_i}{\delta w_{ji}}$$

$$= \sum_j \frac{\delta C}{\delta p_j} \frac{\delta p_j}{\delta O_i} \frac{\delta O_i}{\delta w_{ji}}$$

$$C = -\sum_j y_j \log p_j \quad \frac{\delta C}{\delta p_j} = -\sum_j \frac{y_i}{p_j}$$

$$p_i = \frac{e^{O_i}}{\sum_j e^{O_j}} = \frac{e^{O_i}}{\sum_{i \neq j} e^{O_j} + e^{O_i}}$$

since $\sum_j e^{O_j} = \frac{e^{O_i}}{p_i}$, $\sum_{j \neq i} e^{O_j} = \frac{e^{O_i}}{p_i} - e^{O_i} = e^{O_i}(\frac{1}{p_i} - 1)$

$$\frac{\delta p_i}{\delta O_i} = \frac{\sum_{j \neq i} O_j}{(\sum_{j \neq i} e^{O_j} + e^{O_i})^2} \cdot e^{O_i} \quad \text{since } \frac{d}{dx}(\frac{x}{x+a}) = \frac{x}{(a+x)^2}$$

$$= \frac{e^{O_i}(\frac{1}{p_i} - 1)}{(\frac{e^{O_i}}{p_i})^2} \cdot e^{O_i} = p_i(1 - p_i)$$

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

$$\frac{\delta O_i}{\delta w_{ji}} = \sum_j \frac{\delta O_i}{\delta w_{ji}} (w_{ji} x_j + b_i)$$

$$= \sum_j x_j$$

$$\therefore \frac{\delta C}{\delta w_{ji}} = \sum_j (p_i - y_i) x_j$$

Figure 2: Gradient function justification

Below is the vectorized code for gradient function.

```
def df(w, x, y):
    output = forward(x, w)
    return dot(x, (output - y).T)
```

I calculated the finite difference's output and my gradient function's output and compare the average error. Using the code above.

```
def finite_difference(x, y, theta, h):
    origin_t = f(theta, x, y)
    theta = theta + np.full((theta.shape[0], theta.shape[1]), h)
    after_t = f(theta, x, y)
    finite_diff = (after_t - origin_t) / h
```

```
        total_error = sum(finite_diff - df(theta, x, y))
        return abs(total_error) / (785 * 10 * 1.0)
    np.random.seed(1)
    W0 = np.random.randn(785, 10) / sqrt(785)
    x, y = generate_x_y('train', 200)
    print(finite_difference(x, y, W0, 1e-3))
```

Which has the output

```
finite_difference:2.21852260554e-15
```

This proves my gradient is indeed the true gradient of the cost function. Since the avg. error is rather insignificant.

# Part 4

*Train the neural network using SDG*
Specifically, I like to show how I find the learning rate and how I initialized the weight.
I tried alpha in range of (0, 1e-1, 1e-2) and I get a plot of alpha against accuracy of test samples. See Figure 3 below. Clearly, 1e-2 was a good choice since it has the highest accuracy.

```
alpha = np.arange(0, 1e-1, 1e-2)
alpha_p = []
for a in alpha:
    W = sgd(x, y, W0, df, a, 2000)
    alpha_p.append(testing(W))
```

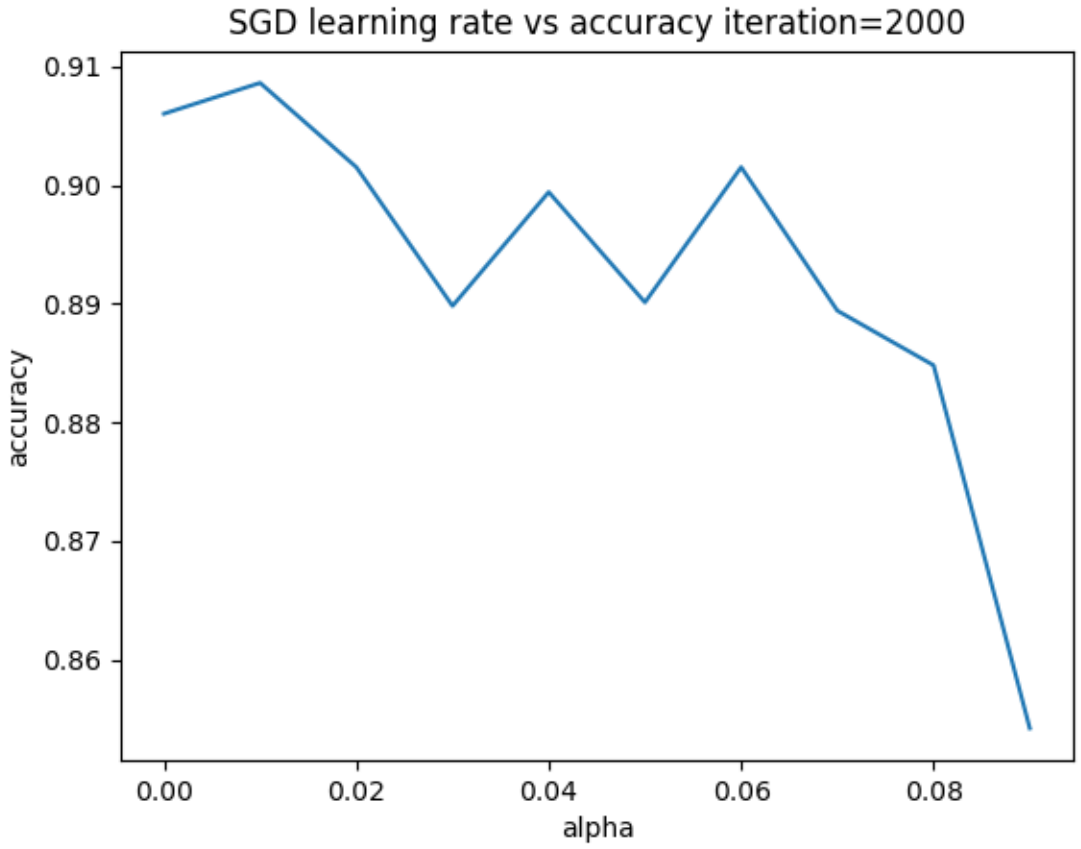
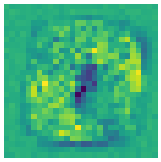



Figure 3: SGD alpha vs accuracy

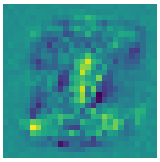
I initialized my weight using Gaussian and use np.random.randn() to generate weight with shape of (785,10) and the variance is $1/\text{size}_{in}$.

```
#Code for generating W0:
np.random.randn(785, 10) / sqrt(785)
```

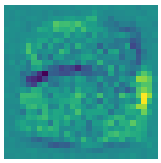
And I get the following weights below:



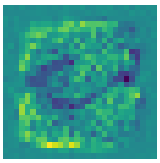
(a) 0's weight



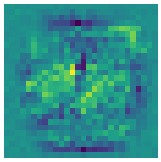
(b) 1's weight



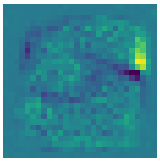
(c) 2's weight



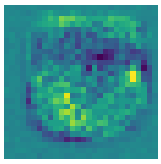
(d) 3's weight



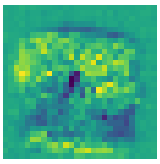
(e) 4's weight



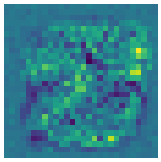
(f) 5's weight



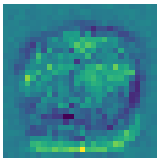
(g) 6's weight



(h) 7's weight



(i) 8's weight



(j) 9's weight

Since we uses mini-batch, We should expect it to be noisy since we divide whole batch into mini-batches and update weight each mini-batch.

And here's is the learning curve using SDG:

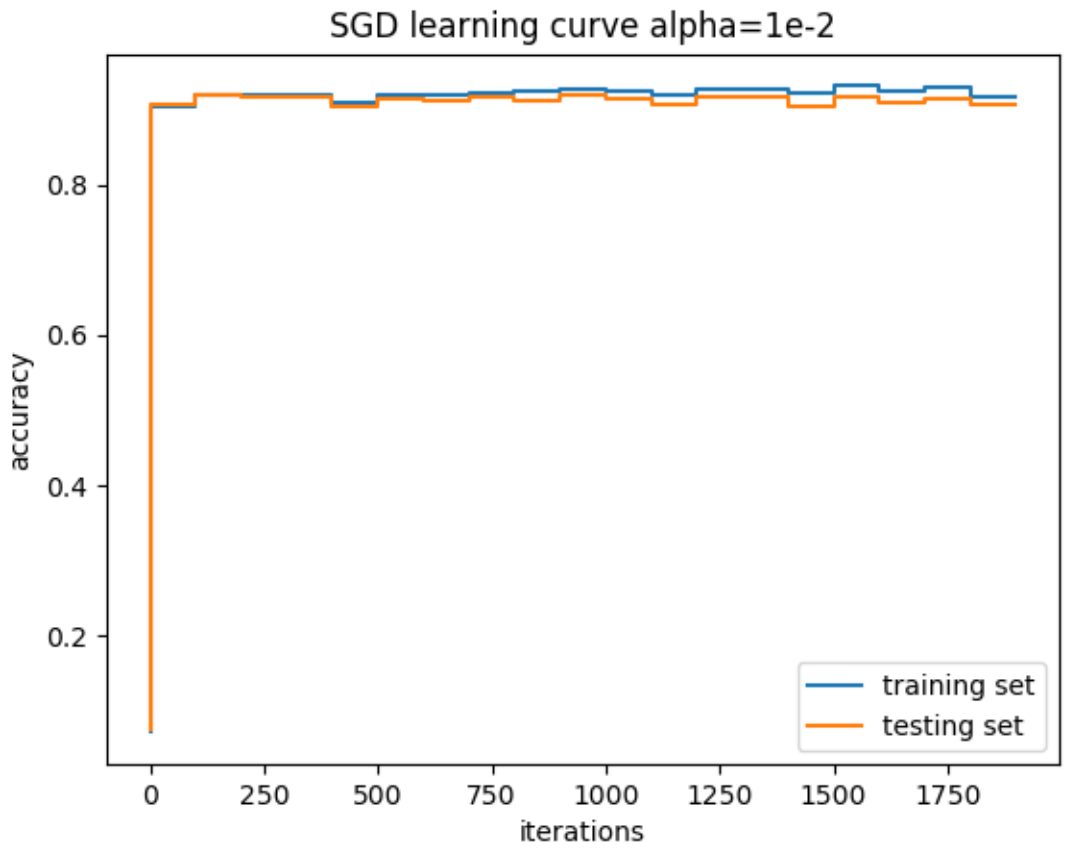



Figure 5: SGD learning curve

Since our training set is large enough so that many variations have taken count into the weight, the testing set's performance is very high at this point compared to training set. However, it starts to fluctuate at 750 iterations and there's no more improvement from there.
**Note**: There's no validating set in mnist_all.mat file

# Part 5

*Train the neural network using SDG with momentum*
Code I use to do SGD with momentum:

```
def next_batch(x, y, batchSize):
    for i in np.arange(0, x.shape[1], batchSize):
        if (i + batchSize) < x.shape[1]:
            yield (x[:, [i, i + batchSize]], y[:, [i, i + batchSize]])
        else:
            yield (x[:, [i, x.shape[1] - 1]], y[:, [i, y.shape[1] - 1]])

def sgd_momentum(x, y, w, df, alpha, epoch):
    it = 0
    velocity = zeros_like(w)
    while it < epoch:
        x, y = shuffle_input(x, y)
        for (batchX, batchY) in next_batch(x, y, 128):
            velocity = 0.9 * velocity + alpha * df(w, batchX, batchY)
            w -= velocity
        it += 1
    return w
```

And the learning curve using SDG momentum:

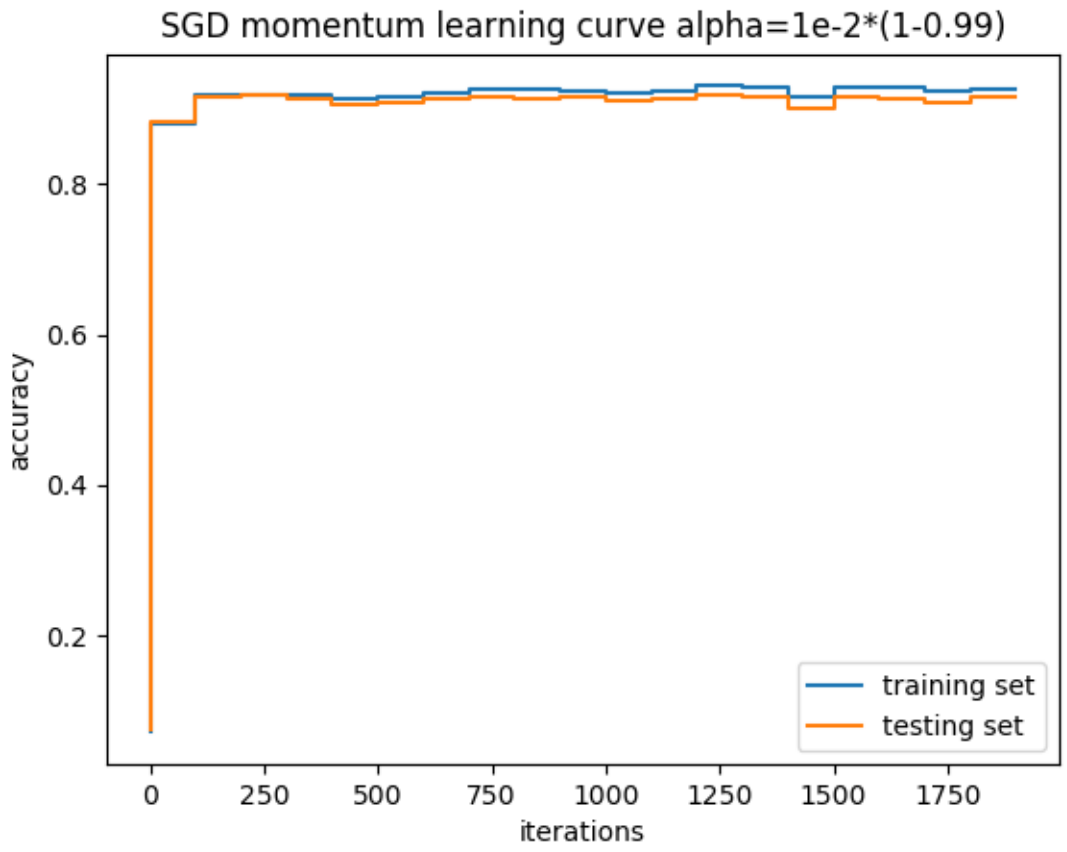



Figure 6: SGD with momentum learning curve

We can observe that the performance of testing set starts to fluctuate after 500 iterations and there's no more improvements from there compared to that of 750 iterations in SGD. The reason behind this is because with a smaller amount of iterations, by utilizing momentum, it descends faster, thus a closer approach to perfect weight which leads to the best performance. However, this effect dissipates after more epochs since we have already obtained a good weight and there's no point of accelerating the descend.
**Note**: There's no validating set in mnist_all.mat file

# Part 6

*Contour Plot*

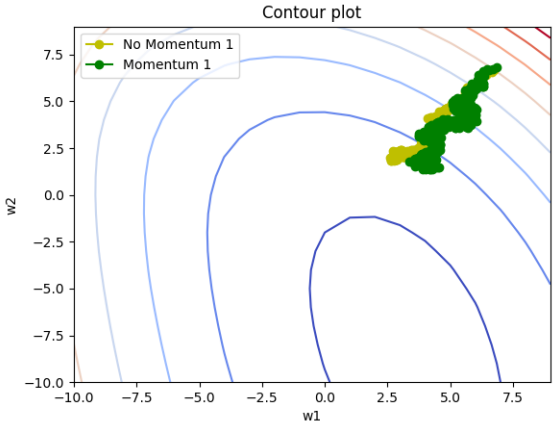



Figure 7: Contour plot with two trajectories at steeper position

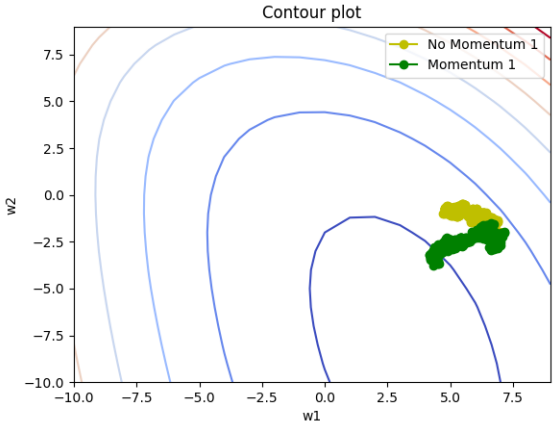



Figure 8: Contour plot with two trajectories at flatter position

As we can see from Figure 8, w1,w2 initialized at flatter position on contour plot,the trajectory of SGD with momentum descend faster than that of without momentum. However, in other scenarios, in Figure 7, I initialized the weights away from optimum and at steeper position in contour plot. Somehow, there's no big difference in both trajectories' descending speed. The reason behind this is that the velocity in momentum will be greater at flat position and smaller at steep position. Since at steeper positions, both momentum and without momentum methods can descend faster.
**Note**: I pick two weights that are in the centre of digits but not right close to each other.

# Part 7

*Show back propagation faster than update-each-weight method*
I construct a new gradient descend function but this time i will update each weight (i.e. not vectorize it). The code is below:

```
def update_every_weight(x, y, w, wi, wj, alpha, output):
    error = output[wj] - y[wj]
    w[wi, wj] -= alpha * np.sum(error * x[wi])


def gradient_descend(x, y, w, alpha, epoch):
    it = 0
    while it < epoch:
        x, y = shuffle_input(x, y)
        for (batchX, batchY) in next_batch(x, y, 128):
            for wi in range(w.shape[0]):
                for wj in range(w.shape[1]):
                    output = forward(x, w)
                    update_every_weight(batchX, batchY, w, wi, wj, alpha, output)
        it += 1
    return w
```

I calculate cost function w.r.t. each weight every time using ***update_every_weight*** and update that weight, it follows the not vectorized version of gradient of cost function obtained in **part3**.
I compared it with SGD (i.e. one back propagation method)w.r.t running time using:

```
from timeit import default_timer as timer
start=timer()
_=sgd(x, y, W0, df, 1e-2, 10)
backp_end=timer()-start
_=gradient_descend(x,y,W0,1e-2,10)
end=timer()-start
everyw_end=end-backp_end
```

And here I got the result:

```
back propagation finished in: 5.358921677048784
update-every-weight finished in: 1262.9500488219783
accelerating ratio: 235.67242160506785
```

Which is approx. power of 3.5 faster than update every weight.

# Part 8

*Faces Classification with NN*

[Part 8 Environment] Using Pytorch without GPU on Windows 10, All images using for datasets were checked with SHA256.

First, we try to determine which dimension of the hidden is the best of our dataset:

Our first attempt was done without the presence of the validation set, which was a mistake:

```
==================== Current Dimension of H: 500 ====================
==========TRAINING SET==========
[Performance - TRAINING SET]  100.0 %
==========TEST SET==========
[Performance - TEST SET]  90.0 %
==================== Results ====================
[training_performances]  [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[test_performances]  [0.8916666666666667, 0.9, 0.8916666666666667, 0.9083333333333333, 0.9, 0.8916666666666667, 0.9, 0.8
916666666666667, 0.9, 0.8916666666666667, 0.9, 0.9]
[Best dim_h]  100
[Best test_performance]  90.83333333333333 %
```

Figure 9: First/False Attempt: Training Size = 70, Testing Size = 20, max_iter=10000

Realizing that we still require the validation set, we decided to take 10 images out of the training set in the first attempt and dedicate those to our validation set:

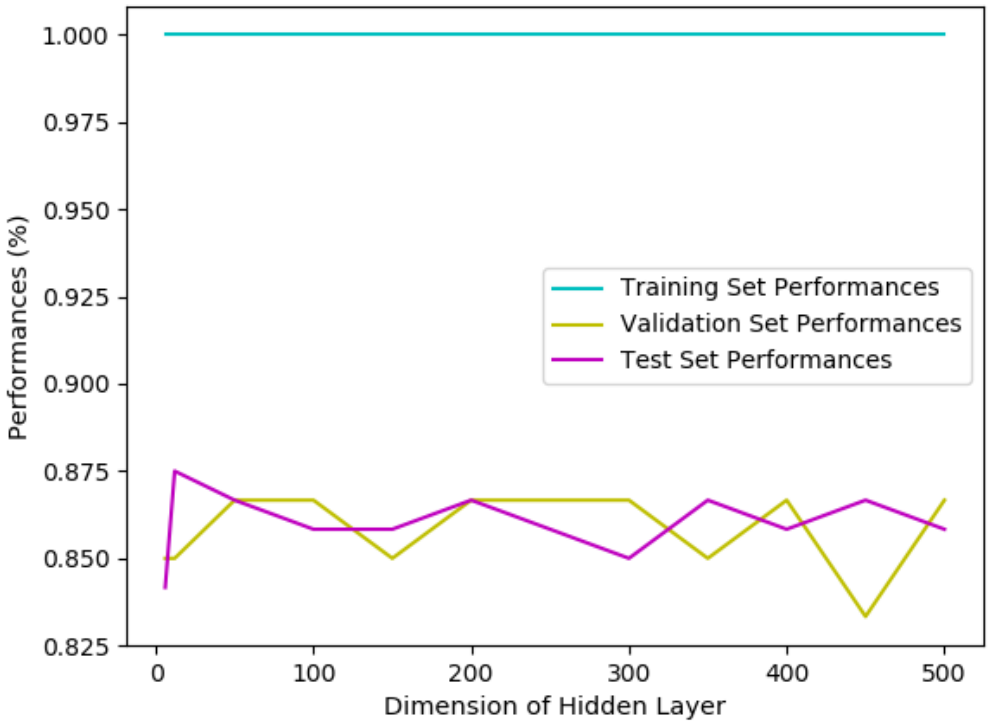



Figure 10: Second/Final Attempt: Training Size = 60, Validating Size = 10, Testing Size = 20, max_iter=10000

Now we got the best dimension of our hidden layer, which is 12 neurons.

Using the best dim_h from the results of different trials (dim_h = 12), we can now get apply this to get our learning curve:

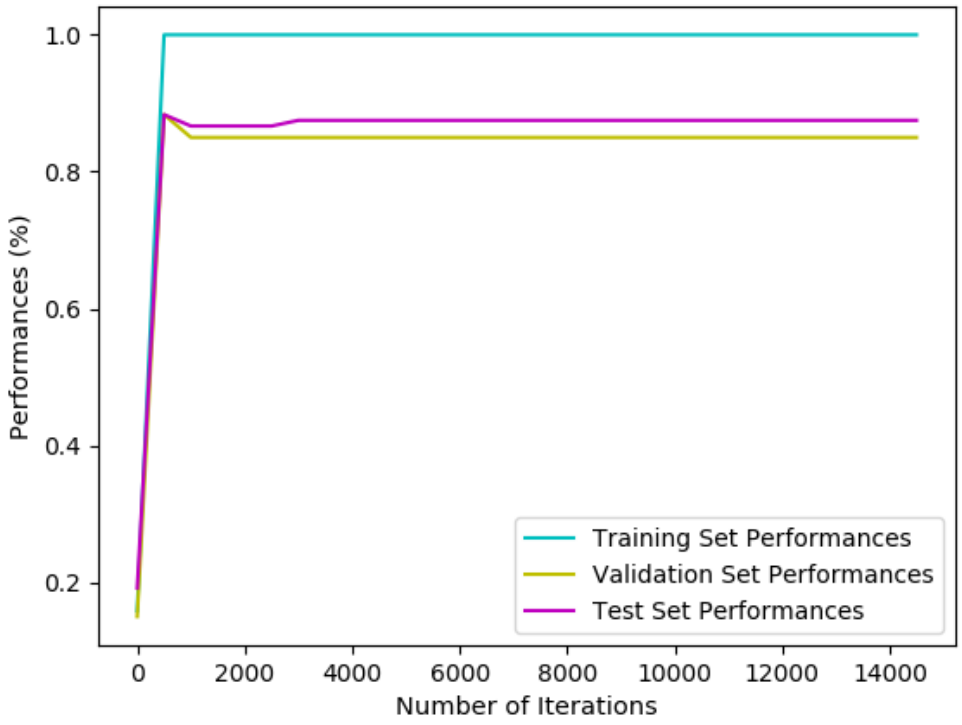


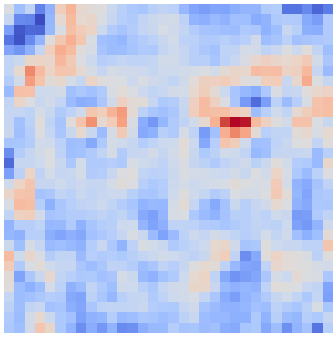

Figure 11: Learning Curve: Training Size = 60, Validating Size = 10, Testing Size = 20, dim_h = 12 (From Previous Figure.)

We now get the best number of iterations we should use to get the best outcome/performance.

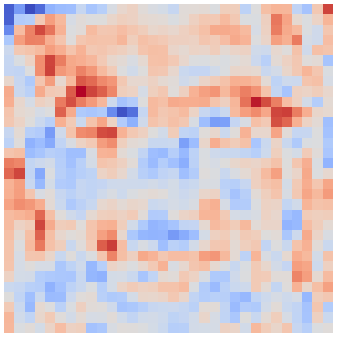
Note that we can see that our model is slightly over-fitting since the overall performance on the training set is higher than the other two.

# Part 9

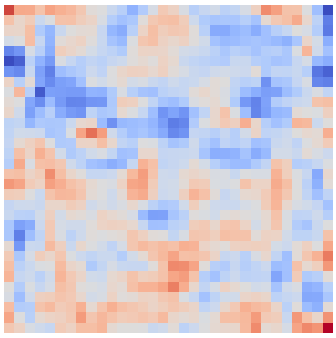
*Weights Visualizations*



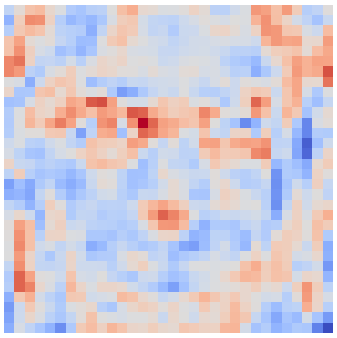
(a) Lorraine Bracco's weight



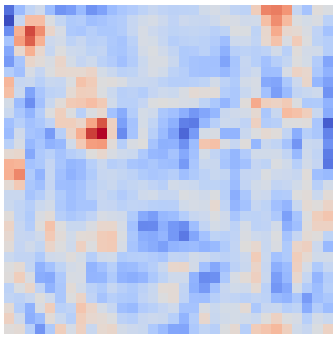
(b) Peri Gilpin's weight



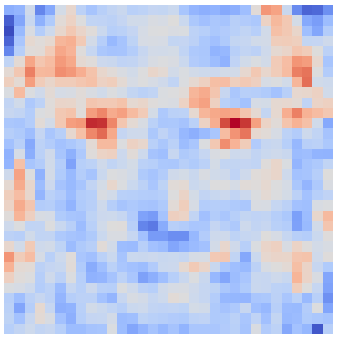
(c) Angie Harmon's weight



(d) Alec Baldwin's weight



(e) Bill Hader's weight



(f) Steve Carell's weight

# Part 10

*Transfer Learning with AlexNet*

[Part 10 Environment] Using Pytorch with CUDA 9.0 on Windows 10 with Images with RGB channels (using modified version of the Image Downloader from Project 1 using Python 2.7)

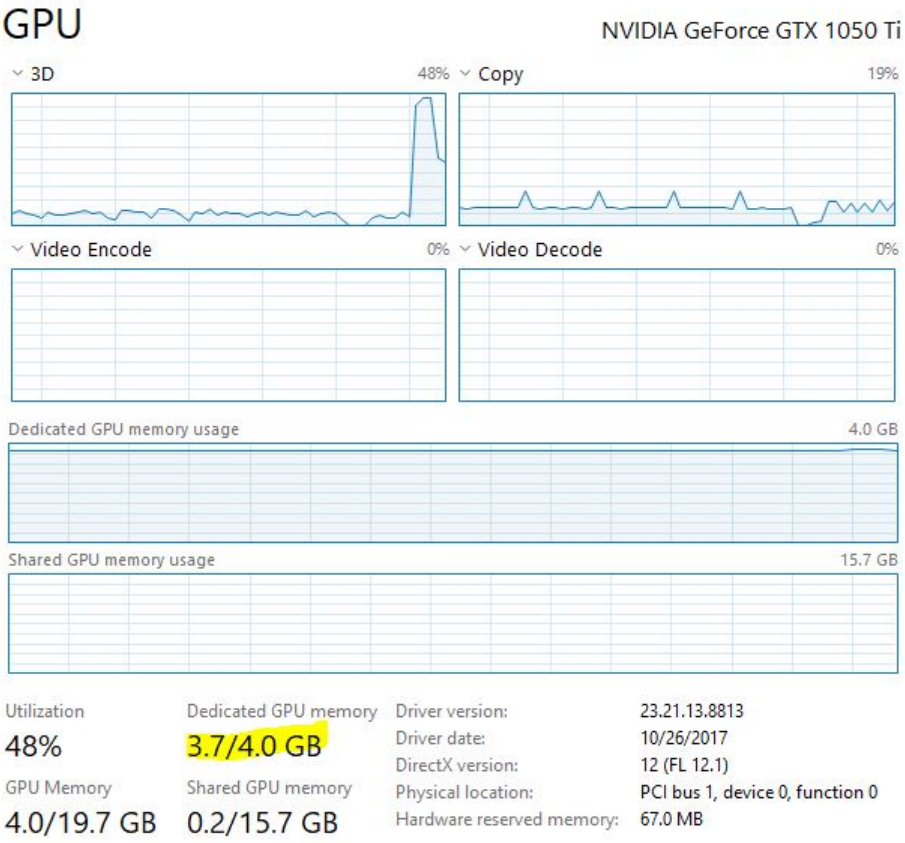



Figure 13: [Part 10] GPU Environment: NVIDIA GeForce GTX 1050Ti with 4.0GB of GRAM

[Part 10] NN Model

```
    def __init__(self, num_classes=1000):
        super(MyAlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            # nn.Conv2d(256, 256, kernel_size=3, padding=1),
            # nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.mynet = nn.Sequential(
            nn.Dropout(),
            torch.nn.Linear(256 * 6 * 6, 2048),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            torch.nn.Linear(2048, 12),
            nn.ReLU(inplace=True),
            torch.nn.Linear(12, 6),
        )
    ...
    ...
    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.mynet(x)
        return x
    ...
    ...
    x = Variable(torch.from_numpy(train_x), requires_grad=False).type(dtype_float)
    y_labels = Variable(torch.from_numpy(np.argmax(train_y, 1)),
                requires_grad=False).type(dtype_long)

    myANet = MyAlexNet()
    myANet.cuda()

    loss_fn = torch.nn.CrossEntropyLoss()

    ## TRAINING THE MODEL
    alpha = 1e-2
    optimizer = torch.optim.SGD(myANet.parameters(), lr=alpha, momentum=0.99)
```

```
for t in range(n):
    if t % 50 == 0:
        print('t: ', t)

    # forward
    y_pred = myANet.forward(x)
    ...
    ...
```

In order to get the Conv4 Layer, we commented out the fifth/last convolution layer. Then we apply this layer with our own classification layer (self.mynet(x)) to classify our 6 actors given the labels, whereas the original classification layer will output 1000 kinds of labels for different objects AlexNet has trained with.

The detail layers of our newly constructed NN is given above.

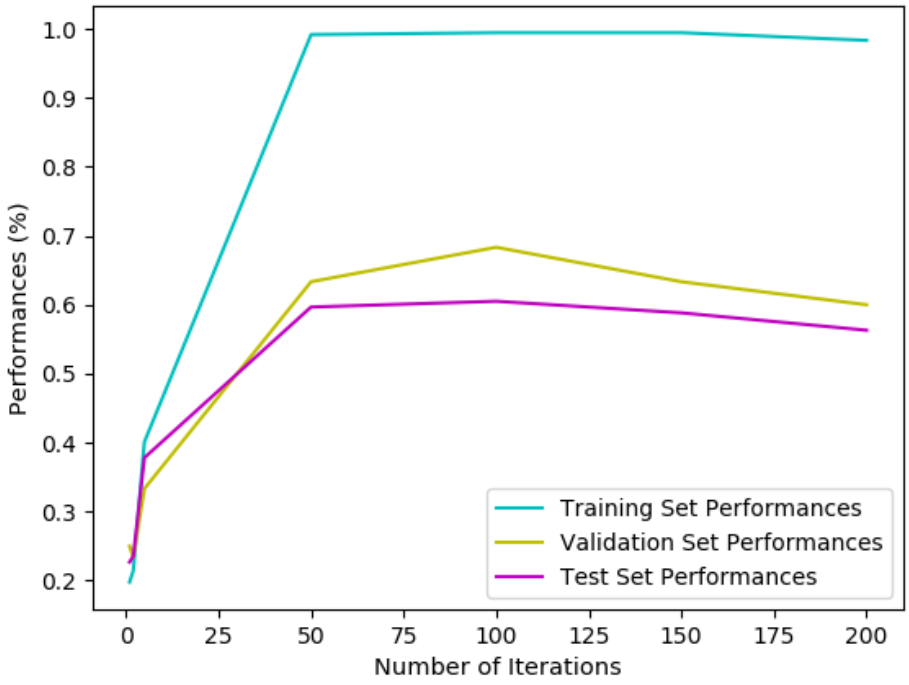



Figure 14: [Part 10] Learning Curve

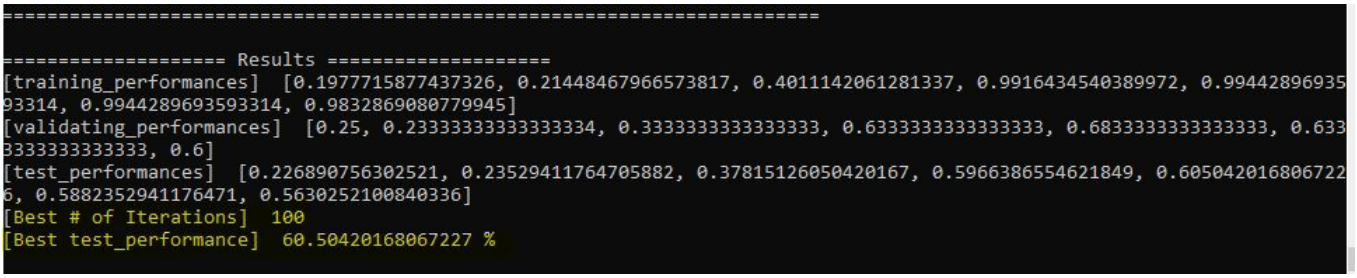



Figure 15: [Part 10] Find Best Iteration (max_iter = 100 from the Learning Curve Figure)

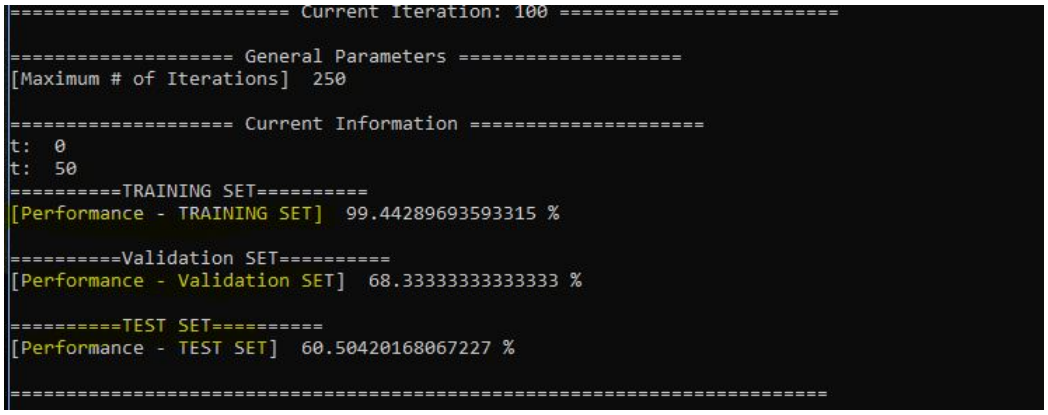



Figure 16: [Part 10] Use the Best of Iteration (=100) to get the Performances

From the learning curve, we can see that the best of iterations we should apply to minimize our cost is using max_iter = 100, shown in Figure 15.
The performance on our test set is 60.58% while training set is having its performance at almost 100% (similar to validation set). We can observe that the model is overfitting our training and test datasets.