CSC411: Assignment #2

Due on Friday, February 23, 2018

Said Chehabeddine

March 12, 2018

Problem 1

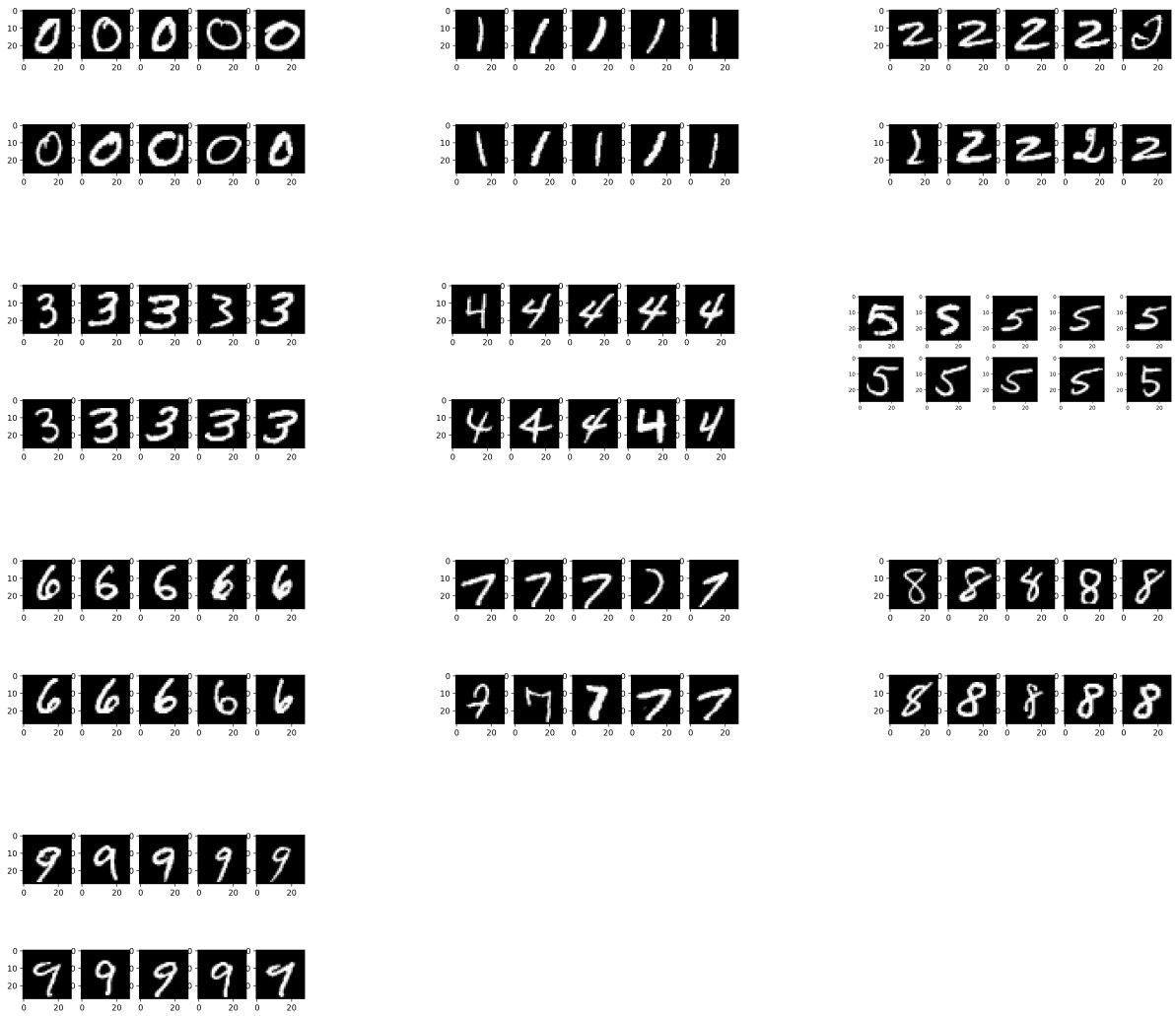


Figure 1: dataset samples

We can see that the data set contains a variety of script styles for each digit from 0-9. Each image has a 28x28 resolution. Some clear issues are apparent from the beginning, such as the similarities between the 8 digit, the 5 digit and the 6 digit, thus requiring a more sophisticated algorithm if we wish to achieve high accuracy.

Problem 2

we define the network as a linear combination of the weights per input, to an output layer which is then put through the softmax function. Thus the network can be defined with the following function:

```
def linCom(x,w,b):
    return softmax2(np.matmul(x,w.T) + b)
```

where we defined softmax as follows, due to some initial overflow issues with the given function.

```
def softmax2(y):
    return np.exp(y)/(np.array([np.sum(np.exp(y), 1)])).T
```

Problem 3

```
def neglog(p,y):
    return -np.sum(y*np.log(p))
```

(a)

$$\frac{\delta C}{\delta w_{ij}} = \frac{\delta C}{\delta o_i}\frac{\delta o_i}{\delta w_{ij}}$$

$$\frac{\delta C}{\delta o_{ij}} = p_i - y_i$$

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

therefore

$$\frac{\delta C}{\delta w_{ij}} = x_j(p_i - y_i)$$

(b)

```
def gradient(x,y,p):
    return np.matmul((p-y).T,x)
```

```
def finiteDifference(x,y,w):
    hl = [1,0.1, 0.01,0.001,1e-9,1e-10,1e-11]
    #b= (x.T[:10][:]).T
    b=0
    i=5
    for h in hl:
        print "h = ", h
        grad = np.zeros((10,784))

        for j in range(50):
            H = np.zeros((10,784))
            H[i][j]= h
            grad[i][j] = (neglog(linCom(x,w+H,b),y) - neglog(linCom(x,w,b),y))/h

        p = linCom(x,w,b)
        vGrad = gradient(x,y,p)
        diff = abs(grad[i][:50] - vGrad[i][:50])
        print "---Difference", np.sum(diff)
    return
```

digit 5	Difference
h = 1	55.37698077031473
h = 0.1	38.37164297320942
h = 0.01	37.10579468722517
h = 0.001	36.982648464320846
h = 1e-09	35.38344025368884
h = 1e-10	26.7687583339205

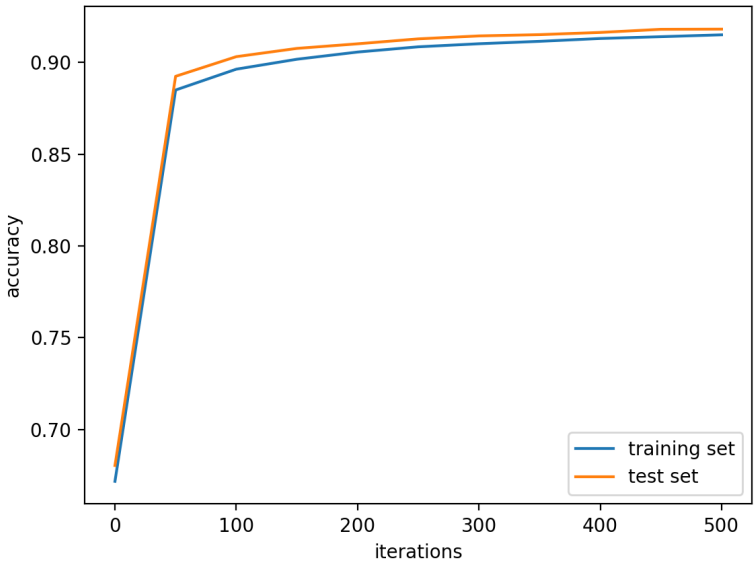
digit 8	Difference
h = 1	59.77008807466427
h = 0.1	41.41586717421251
h = 0.01	40.04961046709741
h = 0.001	39.916696998691805
h = 1e-09	39.341613091229455
h = 1e-10	27.933746715588537

Problem 4

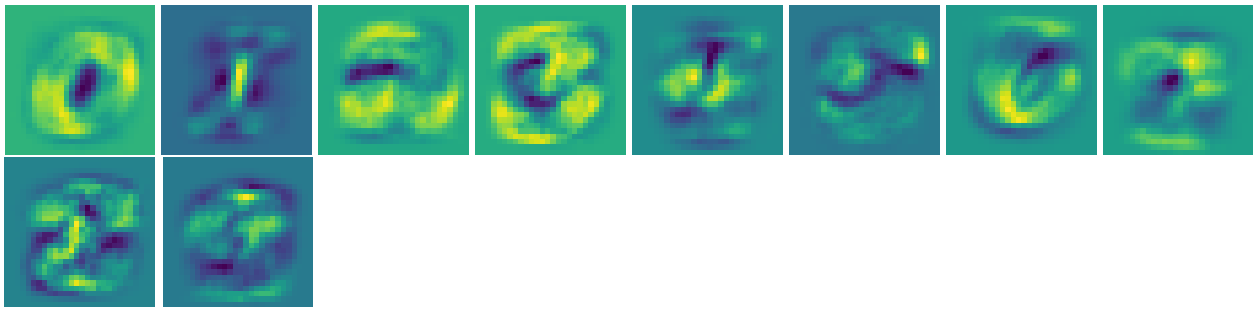
```
def accuracy(x,y,w):
    p=np.matmul(x,w.T)
    pf = np.array(np.isin(p,p.max(axis = 1)).flatten())
    check = map(int,np.array(map(bool,y.flatten())) & pf)
    correct = check.count(1)
    return correct/float(final_preds.shape[0]/10)

def gradientDescent(x,y,w,a):
    EPS = 1e-6
    prev_W = w - 10*EPS
    max_iter = 500
    iter = 0
    training = []
    test = []
    xt,yt = get_data_test(M)
    while np.linalg.norm(w-prev_W) > EPS and iter < max_iter:
        if iter != 0:
             prev_W = w.copy()
        p = linCom(x,w,0)
        w -= a*gradient(x, y, p)
        if iter % 50 == 0:
            training.append(accuracy(x, y, w))
            test.append(accuracy(xt, yt, w))
        iter += 1
    training.append(accuracy(x, y, w))
    test.append(accuracy(xt, yt, w))

    return w,training,test

def part4():
    w = np.ones((10,784))
    x,y = get_data(M)
    a = 0.00001
    iter =[]
    for i in range(0,550,50):
        iter.append(i)

    w,training,test = gradientDescent(x,y,w,a)

    plt.plot(iter, training)
    plt.plot(iter, test)
    plt.ylabel("accuracy")
    plt.xlabel("iterations")
    plt.legend(["training set","test set"])
    plt.show()
    for i in range(10):
        v=w[i,:]
        v=v.reshape([28,28])
        imsave("digit" + str(i)+".jpg",v)

    return
```

Employing gradient descent, we can then graph the learning curve as the number of iterations increases. running part 4 and plotting the data we get the following learning curves:

Learning Curve



The weights of the gradient descent can also be visualized by reshapining the weights to 28x28 images:

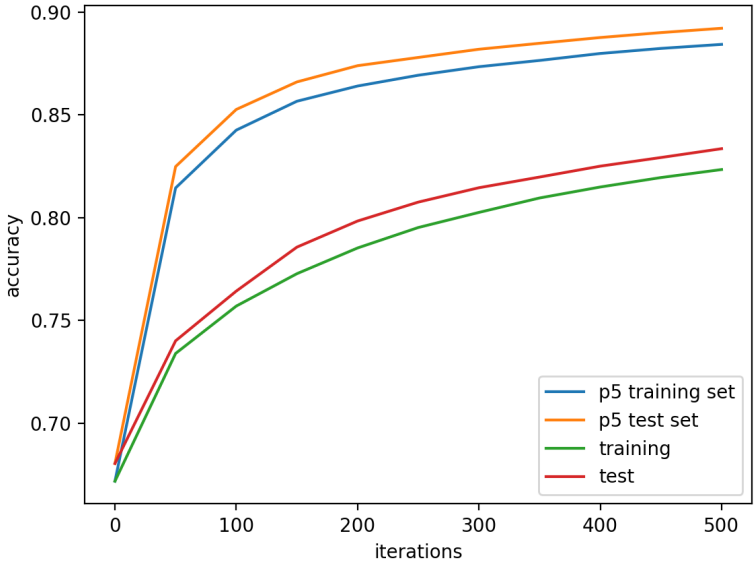


Problem 5

```
def gradientDescentMomentum(x,y,w,a):
    EPS = 1e-6
    prev_W = w - 10*EPS
    max_iter = 200
    iter = 0
    training = []
    test = []
    xt,yt = get_data_test(M)
    prevGradient = gradient(x, y, linCom(x,w,0))
    v = np.zeros((10,784))

    while np.linalg.norm(w-prev_W) > EPS and iter < max_iter:
        if iter != 0:
             prev_W = w.copy()
        p = linCom(x,w,0)
        v = 0.9*v + a*gradient(x, y, p)
        w -= v
        prevGradient = gradient(x, y, p)
        if iter % 50 == 0:
            training.append(accuracy(x, y, w))
            test.append(accuracy(xt, yt, w))
        iter += 1
    training.append(accuracy(x, y, w))
    test.append(accuracy(xt, yt, w))

    return w,training,test

def part5():
    w = np.ones((10,784))
    x,y = get_data(M)
    a = 0.0000001
    iter =[]
    for i in range(0,550,50):
        iter.append(i)

    w,training,test = gradientDescentMomentum(x,y,w,a)
    return w
```

using the velocity formula and picking gamma to be 0.9, we implemented gradient descent with momentum running part5 and graphing it against part4 we get the following learning curves:

Learning Curves



Problem 6

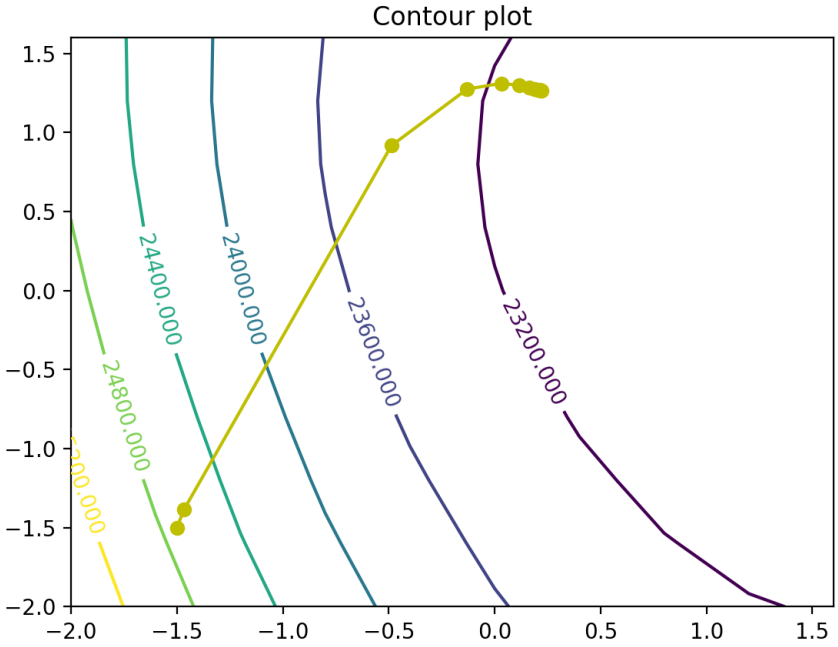
(a)
we define w1, w2 as 350 and 400 of digit 0. We use the following code to set up the contours, using a variation from -2 to 2 by 0.4 steps:

```
def part6():
    x,y = get_data(M)
    a = 0.0001
    w1 = np.arange(-2,2,0.4)
    w2 = np.arange(-2,2,0.4)
    gd = []
    mo = []
    # 6a produce contour plot of part 5
    w = part5()
    C = np.zeros([w1.size, w2.size])
    print "finished decsent"
    for i in range(len(w1)):
        for j in range(len(w2)):
            w[0][350] = w1[i]
            w[0][400] = w2[j]
            p = linCom(x,w,0)
            C[i][j] = neglog(p,y)
    w1s,w2s = meshgrid(w1,w2)
```

(b)
we now define "vanilla" gradient descent where only w1,w2 are varied while the rest of the weights remain constant:

```
def vanilla(x,y,w,a):
    EPS = 1e-6
    prev_W = w - 10*EPS
    max_iter = 500
    iter = 0
    w1 = []
    w2 =[]
    w1.append(w[0][350])
    w2.append(w[0][400])

    while np.linalg.norm(w-prev_W) > EPS and iter < max_iter:
        if iter != 0:
             prev_W = w.copy()
        p = linCom(x,w,0)
        w[0][350] -= a*gradient(x, y, p)[0][350]
        w[0][400] -= a*gradient(x, y, p)[0][400]
        if iter % 50 == 0:
            w1.append(w[0][350])
            w2.append(w[0][400])
        iter += 1
    w1.append(w[0][350])
    w2.append(w[0][400])

    return w1,w2
```

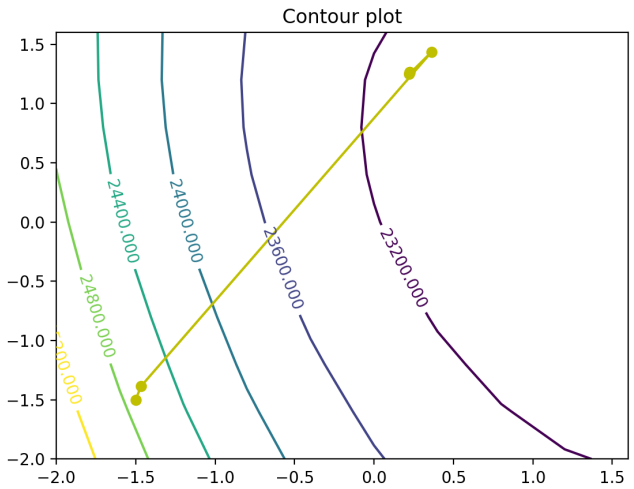
graphing the trajectory over the contour we get the following graph:



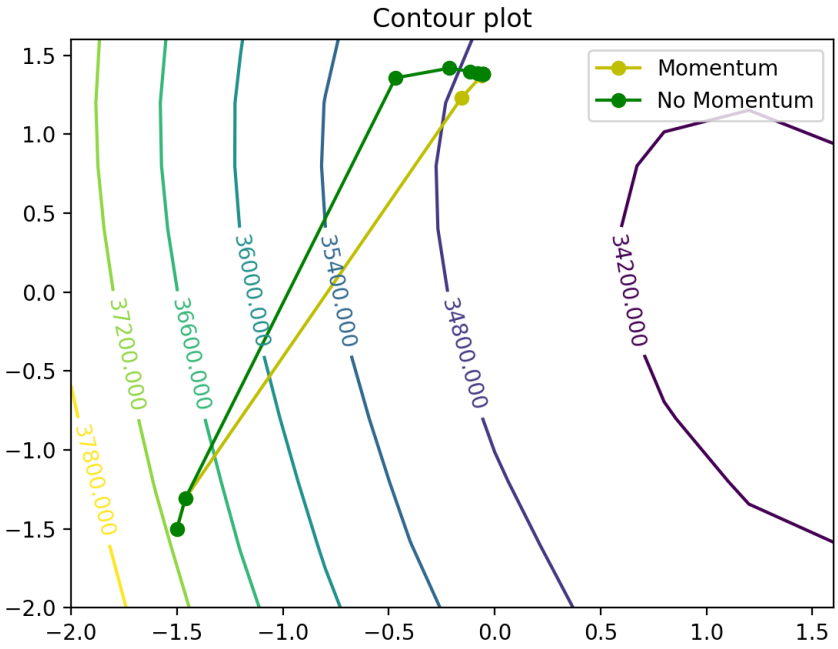
(c)
we now define an equivalent function to the above, but use momentum as an optimizer, thus

```
def rockyRoad(x,y,w,a):
    EPS = 1e-6
    prev_W = w - 10*EPS
    max_iter = 1000
    iter = 0
    prevGradient = gradient(x, y, linCom(x,w,0))
    v = np.zeros((10,784))
    w1 = []
    w2 =[]
    w1.append(w[0][350])
    w2.append(w[0][400])
    while np.linalg.norm(w-prev_W) > EPS and iter < max_iter:
        if iter != 0:
             prev_W = w.copy()
        p = linCom(x,w,0)
        v = 0.9*v + a*gradient(x, y, p)
        w[0][350] -= v[0][350]
        w[0][400] -= v[0][400]
        prevGradient = gradient(x, y, p)
        if iter % 50 == 0:
            w1.append(w[0][350])
            w2.append(w[0][400])
        iter += 1
    w1.append(w[0][350])
    w2.append(w[0][400])

    return w1,w2
```

graphing the trajectory over the contour we get the following graph:



(d)and(e)
clearly we see that gradient descent using momentum is significantly better than, gradient descent without momentum and reaches the local minimum in fewer steps. Graphing both trajectories we get the following graph:



Problem 7

Back Propagation

$$\frac{\delta C}{\delta w_{ijm}} = \frac{\delta C}{\delta h_{i_0 j}} \frac{\delta h_{i_0 j}}{\delta w_{ijm}} = \frac{\delta C}{\delta h_{i_0 j}} \frac{\delta h_{i_0 j}}{\delta h_{i_1 j}} \frac{\delta h_{i_1 j}}{\delta w_{ijm}}$$

from the chain rule we observe that as more hidden layers are added into the network, the more longer the operation will take. More specifically we know that i = 0...N, and j = 1...K and m = 1..K, considering that we can write the above more suggestively

$$\frac{\delta C}{\delta w_{ijm}} = \frac{\delta h_{i_0 j}}{\delta w_{ijm}} (\sum_{m=0}^{K} (\sum_{m=0}^{K} \frac{\delta C}{\delta h_{i_2 j}} \frac{\delta h_{i_2 j}}{\delta h_{i_1 j}}) \frac{\delta h_{i_1 j}}{\delta h_{i_0 j}})$$

extrapolating we see that $O(K^2)$ is the complexity for 2 hidden layers i.e. i = 2, so we can thus take i to N and find $O(K^N)$.

Matrix Multiplication

$$\frac{\delta C}{\delta \mathbf{W}_{ijm}^T} = \frac{\delta C}{\delta \mathbf{O}^T} \frac{\delta \mathbf{O}^T}{\delta \mathbf{H}_N} \cdots \frac{\delta \mathbf{H}_0}{\delta \mathbf{W}_{ijm}^T}$$

Looking at the multiplication we see that the complexity is related to the number of layers N, and the dimensions of the matrices multiplied, which is K^3 so $O(NK^3)$ which is clearly less than $O(K^N)$.

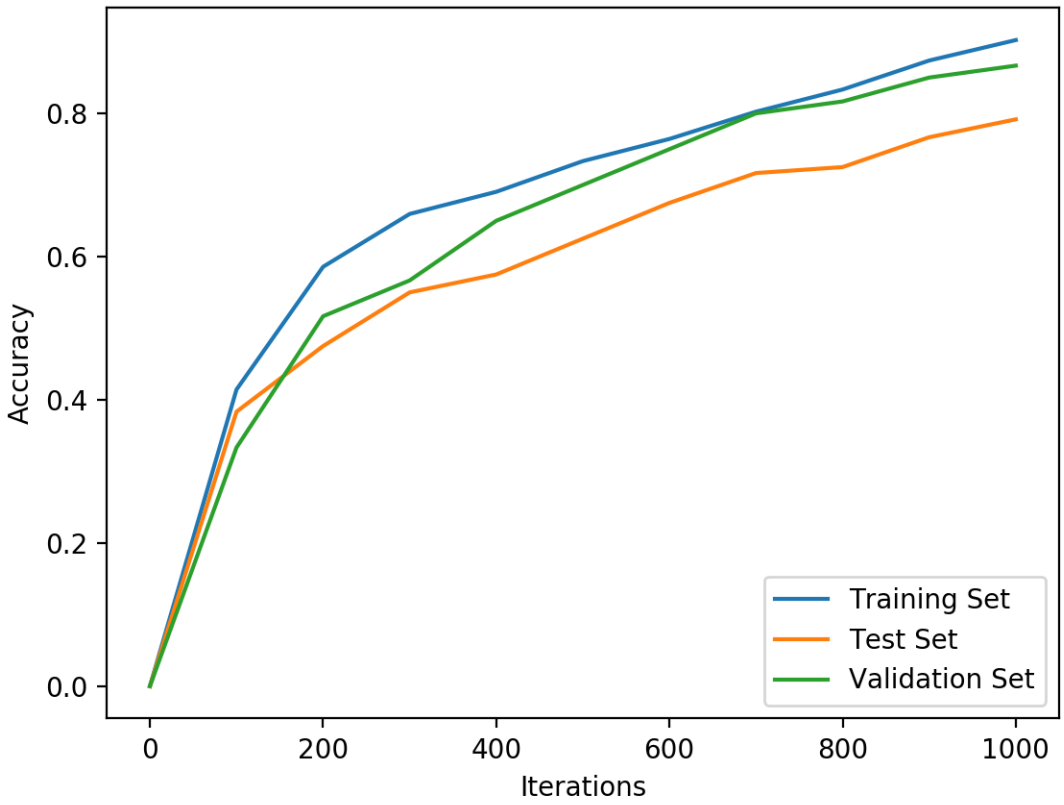
Problem 8

We run the following implementation of using Pytorch, implementing one-hot encoding on the face-scrub data set from project 1. I used a Hardtanh() middle hidden layer with an Adam optimizer as it achieved higher accuracy in less iterations as compared to ReLU, Tanh and Adagrad, Adadelta in various combinations.

```
dim_x = (64*64)*3 + 1
dim_h = 20
dim_out = 6
dtype_float = torch.FloatTensor
dtype_long = torch.LongTensor

training = [0]
test = [0]
valid =[0]
xaxis =[]
for i in range (0,1100,100):
    xaxis.append(i)

x = Variable(torch.from_numpy(train_x), requires_grad=False).type(dtype_float)
xt = Variable(torch.from_numpy(test_x), requires_grad=False).type(dtype_float)
xv = Variable(torch.from_numpy(valid_x), requires_grad=False).type(dtype_float)
y_classes = Variable(torch.from_numpy(np.argmax(train_y, 1)),
requires_grad=False).type(dtype_long)

model = torch.nn.Sequential(
    torch.nn.Linear(dim_x, dim_h),
    torch.nn.Hardtanh(),
    torch.nn.Linear(dim_h, dim_out),
)

loss_fn = torch.nn.CrossEntropyLoss()
model[0].weight.data.fill_(0)

learning_rate = 1e-6
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for i in range(100,1100,100):
    for t in range(i):
        y_pred = model(x)
        loss = loss_fn(y_pred, y_classes)

        optimizer.zero_grad() #model.zero_grad()
        loss.backward()    # Compute the gradient
        optimizer.step()   # Use the gradient information to
                    # make a step
    model.train()
    x = Variable(torch.from_numpy(train_x), requires_grad=False).type(dtype_float)
    y_pred = model(x).data.numpy()
    training.append(np.mean(np.argmax(y_pred, 1) == np.argmax(train_y, 1)))
    #test and validation accuracy
    model.eval()
    y_pred = model(xt).data.numpy()
    test.append(np.mean(np.argmax(y_pred, 1) == np.argmax(test_y, 1)))
    y_pred = model(xv).data.numpy()
```

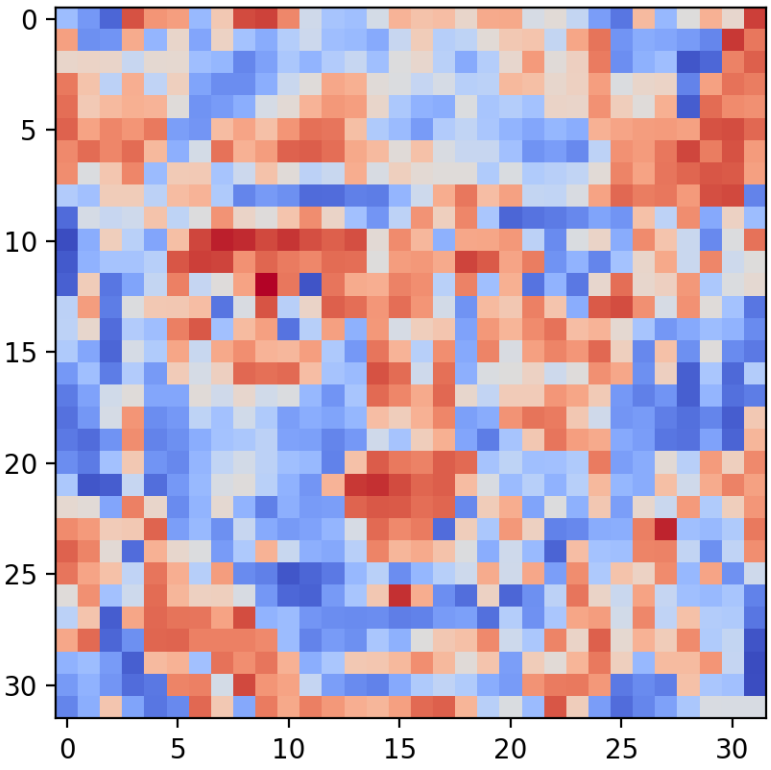
```
    valid.append(np.mean(np.argmax(y_pred, 1) == np.argmax(valid_y, 1)))
```

we can now graph the learning curves, and we see the significantly improved performance of this program versus and approximate 0.45ish accuracy, of the project 1 implementation.



Problem 9

using the images in gray-scale, and displaying the weights of one of the 6 actors we get the following image:



Problem 10

using the myAlexNet implementation we can train the neural network and graph the learning curves as the iterations increase.As before I used a Hardtanh() middle hidden layer with an Adam optimizer as it achieved higher accuracy in less iterations as compared to ReLU, Tanh and Adagrad, Adadelta in various combinations.

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

model = MyAlexNet()
model2 = torch.nn.Sequential(
    nn.Linear(256 *13 * 13, 20),
    nn.Hardtanh(),
    torch.nn.Linear(20,6)
)

model2[0].weight.data.fill_(0)
loss_fn = torch.nn.CrossEntropyLoss()

train_x, train_y = get_train(actors)
test_x, test_y = get_test(actors)
valid_x, valid_y = get_validation(actors)

train_x = np.asarray(train_x)
test_x = np.asarray(test_x)
valid_x = np.asarray(valid_x)

training = [0]
test = [0]
valid =[0]
xaxis =[]
for i in range (0,1100,100):
    xaxis.append(i)

im_v = Variable(torch.from_numpy(train_x), requires_grad=False)
im_vt = Variable(torch.from_numpy(test_x), requires_grad=False)
im_vv = Variable(torch.from_numpy(valid_x), requires_grad=False)

in_train = (model.forward(im_v)).data.numpy()
in_test = (model.forward(im_vt)).data.numpy()
in_valid = (model.forward(im_vv)).data.numpy()

in_train = Variable(torch.from_numpy(in_train), requires_grad = False).type(dtype_float)
in_test = Variable(torch.from_numpy(in_test), requires_grad = False).type(dtype_float)
in_valid = Variable(torch.from_numpy(in_valid), requires_grad = False).type(dtype_float)

y_classes = Variable(torch.from_numpy(np.argmax(train_y,1)),
     requires_grad=False).type(dtype_long)

learning_rate = 1e-6
optimizer = torch.optim.Adam(model2.parameters(), lr=learning_rate)
for i in range(100,1100,100):
    for t in range(i):
```

```
        y_pred = model2(in_train)
        loss = loss_fn(y_pred, y_classes)
        optimizer.zero_grad() #model.zero_grad()
        loss.backward()    # Compute the gradient
        optimizer.step()   # Use the gradient information to
    model2.train()
    y_pred = model2(in_train).data.numpy()
    training.append(np.mean(np.argmax(y_pred, 1) == np.argmax(train_y, 1)))

    #test and validation accuracy
    model2.eval()
    y_pred = model2(in_test).data.numpy()
    test.append(np.mean(np.argmax(y_pred, 1) == np.argmax(test_y, 1)))
    y_pred = model2(in_valid).data.numpy()
    valid.append(np.mean(np.argmax(y_pred, 1) == np.argmax(valid_y, 1)))
```

Graphing the learning curves we get the following figure, showing the low number of iterations needed to achieve a very high accuracy.